UNIVERSIDAD POLITÉCNICA DE CARTAGENA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

DESARROLLO DE APLICACIONES BASADOS EN *NETWORK PROCESSORS*

AUTOR: Zoraida Ros Jiménez
DIRECTOR: José Fernando Cerdán Cartagena

SEPTIEMBRE / 2007

Autor	**Zoraida Ros Jiménez**
E-mail del Autor	zoraidaros@gmail.com
Director(es)	**José Fernando Cerdán Cartagena**
E-mail del Director	fernando.cerdan@upct.es
Codirector(es)	
Título del PFC	**Desarrollo de Aplicaciones Basado en Network Processors**
Descriptores	

Resumen

El avance de la tecnología en las redes de comunicaciones ha provocado el desarrollo de distintos dispositivos de red que cubran las necesidades que demanda este fuerte crecimiento exponencial. Así es como nacen los *network processors*, para paliar los problemas que dispositivos anteriores no eran capaces de solventar debido a que este fuerte crecimiento hacía incapaces a los dispositivos de ese momento ajustarse a las nuevas tecnologías. En este proyecto se da a conocer las características de estos dispositivos y qué problemas nos puede solucionar.

El proyecto describe el concepto de *network processor* desde su definición hasta la implementación y simulación de aplicaciones, estudiando los distintos dispositivos de red y la descripción del entorno de desarrollo del mismo.

Se ha estudiado qué es un *network processor* y cuáles son las arquitecturas más destacables, seleccionando al final cuál se ajusta más a nuestras necesidades.

Posteriormente se habla del entorno de desarrollo que nos permite desarrollar nuestra aplicación y se ha desarrollado una pequeña implementación de código para comparar los distintos resultados y obsevar que se obtiene lo esperado.

Finalmente, en el apartado Apéndice A se podrá ver el código más significativo de dicha aplicación.

Titulación	**Ingeniería Técnica de Telecomunicación, esp.Telemática**
Intensificación	
Departamento	**Tecnologías de la Información y la Comunicación (TIC)**
Fecha de Presentación	**Septiembre 2007**

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO 1
INTRODUCCIÓN

1.1 INTRODUCCIÓN

La explosión del ancho de banda de finales del siglo XX ha impactado en nuestras vidas y este crecimiento exponencial continuará durante muchos años más.

La bajada en el coste del ancho de banda permite a las masas obtener una gran ventaja en la conectividad de los proveedores de Internet. Esto resultará hambriento en ancho de banda y en aplicaciones intensivas computacionalmente, como VoIP, flujo de audio y de video, aplicaciones P2P, VPNs,…

Para las redes que efectivamente manejan estas nuevas aplicaciones, necesitarán soportar nuevos protocolos que incluyen servicios diferenciados, seguridad, y varias funciones de mantenimiento de red. Mientras las redes demandan equipamiento con una tasa de transferencia alta, éstas necesitan también flexibilidad para soportar nuevos protocolos y aplicaciones. Además, los requerimientos que no cambian del equipamiento de red requieren soluciones que pueden persuadir al mercado rápidamente.

Las implementaciones de red de hoy se basan en FPGAs para niveles de procesamiento más bajos y GPPs para niveles de procesamiento más altos. Ninguna de estas soluciones reúne todos los requerimientos que demanda la red de procesamiento. Considerar las extensas categorías de alternativas para la implementación del sistema:

- **ASIC (Aplicación de Circuito Integrado Específico):** cualquier solución de cable duro.
- **ASIP (Aplicación de Procesador de Instrucción Específica):** un conjunto de instrucciones de procesador especializado para una aplicación de dominio particular.
- **FPGA (Puerta de Cadena de Campo Programable):** un dispositivo puede ser reprogramado en el nivel de entrada.
- **GPP (Procesador de Propósito General):** un procesador programable para computación de propósito general.

Los ASICs son los menos flexibles, pero proporcionan el mayor rendimiento. Los GPPs son los más flexibles en el coste del rendimiento más pequeño. En abstinencia de ASIPs o coprocesadores, los FPGAs tienen mayor rendimiento que los GPPs con más flexibilidad que los ASICs. El ASIP es la mejor propuesta para más implementaciones del sistema de interconexión. Un ASIP para interconexiones, o para *network processors*, proporciona el correcto balance de hardware y software para reunir todos los requerimientos indicados arriba:

- **Rendimiento:** para ejecutar la clave computacional de los *kernels* en hardware, los *network processors* pueden llevar a cabo muchas aplicaciones en la velocidad de cable.

- **Flexibilidad:** teniendo software como una mayor parte del sistema permite equipamiento de red para adaptar fácilmente los cambios de los estándares y de las aplicaciones.

- **TTM Rápido:** diseñar software es mucho más rápido (y más barato) que diseñar hardware de equivalente funcionalidad.

- **Potencia:** mientras los *network processors* no deben estar incrustados en dispositivos de energía sensible, su potencia de consumo es importante por razones de coste.

Los *network processors* son parte de un movimiento más extenso de ASICs para implementaciones de sistemas programables. Lo que más dificulta el diseño de ASICs son:

- Efectos de DSM con agravamiento de dificultades de diseño de circuito.

- Incremento exponencial del número de dispositivos en el chip.

- Integración en el chip del diverso incremento de los elementos.

- Disminución del tiempo de mercado.

La combinación de estas presiones ha resultado un cambio en implementaciones de sistema para soluciones más programables. La explosión reciente en arquitecturas de *network processors* soporta esta observación.

A principios de siglo, sólo había unos pocos *network processors* en desarrollo y sólo un producto transporte. Ahora, cada mes se anuncia un nuevo *network processor*. En un intento para aliviar el cuello de botella del ancho de banda, han emergido numerosas soluciones. Éstas varían grandemente en complejidad microarquitectónica y arquitectónico, arquitectura de memoria, soporte de software, e implementación física.

1.2 ESTRUCTURA DEL CONTENIDO

El Proyecto Fin de Carrera que se presenta aborda el estudio de los *network* processors y el desarrollo de aplicaciones basándose en ellos, llevando a cabo la búsqueda de información y documentación referente a estos dispositivos de red. Se proponen unas fases de desarrollo del proyecto:

1. Búsqueda de información y toma de contacto con los *network processors*.
2. Planteamiento de las necesidades de utilizar un *network processor* y propuesta de algunas arquitecturas.
3. Elección de la solución más viable y adecuada para las necesidades planteadas.
4. Estudio del entorno de desarrollo de aplicaciones basadas en *network processors.*
5. Estudio de una aplicación en concreto y modificación dicha aplicación.
6. Pruebas finales y optimización de la aplicación.

Estas fases se han descrito en la memoria de este proyecto bajo los siguientes capítulos:

En el capítulo segundo, *Network Processors*, se hace un estudio para saber qué son y qué problemas nos van a solucionar. Se trata de una fase de documentación y toma de contacto con estos dispositivos. Se hará una breve introducción de la evolución de las redes para comprender cuáles fueron las necesidades en cada momento, y cuáles son las necesidades actualmente. Se definirá *network processor* y se describirán sus funciones y sus paradigmas arquitectónicos. Se propondrán ejemplos de aplicaciones explicando sus problemas y demostrando que utilizando *network processors* estos problemas los solucionaría. Por último, se iniciará el estudio de distintos tipos de *network processors* y se elegirá uno de ellos.

El tercer capítulo está enfocado al manejo del entorno de desarrollo de los *network processors*, el Developer WorkBench, como herramienta de simulación de las aplicaciones desarrolladas para estos dispositivos de red. Es una especie de guía de usuario para entender como simular la aplicación y ver como se procesan los paquetes.

En el cuarto capítulo estudiaremos una aplicación (procesamiento de paquetes sobre un único hilo) y la modificaremos implementando varios contadores. Para ello simularemos la aplicación, recogeremos estadísticas, analizaremos el flujo de datos y observaremos que se obtienen los resultados esperados.

En el quinto y último capítulo, se presentan las conclusiones y líneas futuras que plantea este proyecto.

CAPÍTULO 2
NETWORK PROCESSORS

2.1 QUÉ SON LOS *NETWORK PROCESSORS*

Un *Network Processor* es un ASIP (*Aplication-Specific Processor*) para el dominio de aplicaciones de red: un dispositivo programable con características de arquitectura y/o trazados de circuitos para procesar paquetes de red.

Los *Network Processors*, también conocidos como unidades de *network processor* (NPUs), son circuitos integrados especializados (procesadores) de alta programación. Estos circuitos se utilizan en la industria de comunicaciones de alta velocidad. Son utilizados para optimizar el procesamiento del paquete que se está ejecutando en el desarrollo de la estructura funcional del ancho de banda del equipo de red.

Debido a la convergencia inconfundible de redes, el procesamiento de paquetes convierte la función primordial que es exigida para ser propiamente implementada en equipos de conexión de redes de alta velocidad tales como routers, conmutadores, etc. Obtener el rendimiento y la funcionalidad apropiados en los dispositivos de red es uno de los factores clave para determinar la utilidad, el atractivo, y el potencial económico de esos dispositivos dentro de las corporativas o de los mercados proveedores de servicios.

2.1.1 DÓNDE SE UTIILIZAN LOS NETWORK PROCESSORS

Un *network processor* es utilizado en un gestor de tráfico de red, que ocupa el espacio entre una interfaz de red y un *switch fabric* en un router/conmutador. El gestor de tráfico decide dónde, cuándo, y cómo los datos de entrada y salida serán enviados la siguiente vez. Deshace, añade, y modifica las cabeceras de los paquetes. También toma decisiones de encaminamiento y programación. El gestor de tráfico tiene interfaces para la red y para la *swich fabric*.

2.1.2 POR QUÉ NECESITAMOS LOS NETWORK PROCESSORS

Los *network processors* son dispositivos potentes y un desafío para las ingenierías de software. Entraron en una variedad desconcertante de arquitecturas, pero comparten unas características definidas: se colocan en rutas de datos de alta velocidad y manipulan datos de red en velocidades sostenidas de gigabits por segundo en software. Pero, ¿por qué de repente la velocidad es un problema cuando el silicio ha sido un conductor rápido? Y segundo, ¿por qué

un network processor tiene que implementarse en software? En la práctica se establece para perfeccionar la velocidad por la migración de funcionalidad del software al silicio. Los network processors van en contra de esta tendencia.

Velocidad: La velocidad es un problema porque el ancho de banda de la fibra óptica está creciendo en un radio más rápido que el silicio. El radio de crecimiento exponencial del ancho de banda de red se espera que continúe creciendo porque es un largo camino desde las fundamentales barreras.

Complejidad: Otra razón de porqué la velocidad es un problema es la cantidad de procesamiento que se ha hecho en los datos. Las decisiones de encaminamiento deben reemplazarse con una evaluación compleja de latencia, congestión, garantía del ancho de banda, y más. Esto es lo que se llama encaminamiento. Considera las posibilidades de cortafuegos, spam y detección de virus en el cable. Network processors pueden combatir las epidemias de virus detectando y eliminando los virus en tránsito. La complejidad de este procesamiento individual de paquetes es la razón por la que es necesario hacerlo en software. La nueva funcionalidad no es sólo compleja, pero evoluciona y está sujeta a cambios en el campo.

2.1.3 QUÉ HACEN LOS NETWORK PROCESSORS

Los *network processors* manipulan PDUs en la velocidad del cable para implementar una variedad de funciones incluyendo QoS, encriptación, corta fuegos,…. Estas características son especificadas como protocolos de red, así que son implementadas en pilas de protocolos. Pero los *network processors* no se ejecutan en las pilas de protocolos. Las pilas de protocolos son diseñadas para ejecutarse en GPPs y GPPs son diseñados entre otras cosas para ejecutar pilas de protocolos. El papel del *network processor* es implementar sólo estas partes de un protocolo que requiera acceso directo al flujo de datos. El complejo funcionamiento es dejado al GPP.

Muchos PDUs en tan poco tiempo: Debido a que la velocidad del cable de los datos llega al *network processor* rápidamente y debe ser enviada tan rápidamente que el *network processor* tiene muy poco tiempo para operar en un PDU, no puede llevar a cabo un procesamiento complejo.

Clasificación: Podemos hacer algunas generalizaciones sobre qué tipo de procesamiento es llevado a cabo en un PDU entre cuando es recibido y cuando es retransmitido. Primero se examina para determinar qué procesamiento se llevará a cabo. Esta examinación consiste en mirar el contenido de las PDUs para ver qué diseño de interés contiene. Este proceso es referido como clasificación y es utilizado en encaminamiento, cortafuegos, implementación de calidad de servicio, y aplicación de vigilancia.

Modificación: Un PDU debe ser modificado. Por ejemplo, en un paquete IP su contador de tiempo de vida se reducirá.

Encolamiento: La retransmisión de un PDU generalmente no es sencilla. Algunos PDUs deben tener prioridad sobre otros. Algunos deben ser descartados. Múltiples colas deben existir con diferentes prioridades.

Otras operaciones: Seguridad (encriptación, desencriptación, y autenticación), vigilancia, compresión, y métricas de tráfico.

2.1.4 FUNCIONES GENÉRICAS

- **Modelo a juego:** la habilidad para encontrar modelos específicos de bits o bytes dentro de paquetes en un flujo de paquetes.

- **Clave de búsqueda** por ejemplo, **dirección de búsqueda:** la habilidad para emprender rápidamente una búsqueda de base de datos utilizando una clave (típicamente una dirección en un paquete) para encontrar un resultado, normalmente información de asignación de ruta.

- **Cómputo o cálculo**.

- **Manipulación del campo de bit de datos:** habilidad para cambiar ciertos campos de datos contenidos en el paquete aunque esté siendo procesado.

- **Gestor de colas:** aunque los paquetes sean recibidos, procesados y programados para ser enviados, son almacenados en colas.

- **Control de procesamiento:** las microoperaciones para procesar un paquete son controladas en un macro nivel que implica comunicación con otros nodos.

2.1.5 PARADIGMAS ARQUITECTÓNICOS

- *Pipeline* de procesadores: cada etapa del *pipeline* consiste de un procesador que lleva a cabo una función.

- Procesamiento paralelo con múltiples procesadores, frecuentemente incluyendo múltiples hilos.

- Motores de microcódigo especializados para conseguir las tareas de manera más eficiente.

2.2 APLICACIONES GATEWAY

Las aplicaciones *Gateway* ocurren cerca de las afueras de la red. Las aplicaciones pueden alterar las cabeceras de los paquetes, redireccionar flujos de paquetes, o de caché de paquetes, pero mantienen las semánticas de la existencia de protocolos.

2.2.1 WIRELESS TCP/IP

El acceso a Internet sobre una *wireless* viola algunas de las claves asumidas del protocolo TCP/IP.

- Cuando el protocolo TCP no recibe un acuse de recibo para un paquete que ha sido enviado, entonces el paquete no alcanza el destino debido a la congestión de la red. Esto provoca al protocolo TCP iniciar un algoritmo exponencial de *back-off* que espera a reenviar paquetes.

- En el dominio *wireless* es más probable que la transmisión fracase debido a un error en la capa física. Como resultado, TCP espera mucho tiempo exponencialmente para enviar paquetes en ausencia de tráfico.

Soluciones

Cambiar las acciones del emisor debido a que el destino es un nodo *wireless* o tener como destino al *gateway* resuelve el problema. La forma propuesta parece bastante difícil de implementar, porque requiere que todos los nodos que envían al destino *wireless* sean afectados. La última propuesta es transparente a la red, así como el emisor no es consciente de que el nodo destino sea *wireless*.

Hay unas pocas propuestas principales para diseñar una estación-base que conecte terminales *wireless* a Internet:

1. La estación-base rompe la conexión desde el emisor al nodo *wireless*. Esto requiere que la estación-base procese todos los protocolos de capa superior y guarde los paquetes de la sesión. La estación-base enviará esos paquetes al auricular *wireless* (cualquiera usa TCP o algún otro protocolo). La estación-base mandará acuses de recibo al emisor, el emisor no sabrá que el destino es una *wireless*. Ésta propuesta puede provocar gran carga en la estación base y gastos generales.

2. La estación-base aligera los paquetes que todavía no han recibido su acuse de recibo por el auricular *wireless*. Cuando un paquete es descartado, la estación-base puede reenviar el paquete, en lugar de tener al emisor reenviando el paquete.

Ambas propuestas requieren que la estación base intercepte paquetes desde y hacia destinos *wireless*.

1. La primera propuesta requiere que la estación-base mantenga una pila del protocolo TCP para cada conexión pasando a través de ella y otra pila de protocolos para cada conexión al destino.

2. La segunda propuesta requiere que los paquetes de entrada se copien en una caché y que los paquetes de salida sean examinados mediante acuses de recibo para desahuciar estas entradas de la caché. Esta propuesta necesita mucho menos procesamiento pero hace que requiera temporizadores para cada conexión tiempos para determinar si se ha reenviado o no un paquete a un destino.

2.2.2 TRADUCCIÓN DE DIRECCIONES DE RED (NAT)

Permite que múltiples extremos se representen por una sola dirección IP (de los extremos detrás del *gateway*). Los extremos usan las direcciones IP reservadas para subredes locales. El *gateway* mapea la dirección TCP/IP y el puerto de todas las respuestas para *hosts* de fuera para direcciones y puertos del *gateway*. Para llevar a cabo el NAT, el extremo del router *gateway* necesita realizar las siguientes funciones:

1. Almacena una dirección IP local y un número de puerto en una tabla de traducción de dirección.

2. Modificar puerto origen del paquete según la tabla.

3. Cuando el paquete vuelve del nodo destino, el router chequea el puerto destino del paquete, entonces busca la traducción de la dirección en la tabla.

2.2.3 "CONMUTADOR" WEB

Es un dispositivo que usa información de los protocolos de capas más altas para tomar decisiones de encaminamiento/conmutación de capas más bajas.

Proporciona un punto de contacto en Internet para que los clientes puedan acceder a él. Las peticiones de los clientes son dirigidas selectivamente al servidor más apropiado. Esto hace al conmutador web más útil en una variedad de aplicaciones incluyendo caché de web distribuido y balance de carga.

Se puede implementar un balance de carga en un servidor TCP/IP utilizando un conmutador web como sigue:

1. El conmutador web reconoce una nueva conexión TCP identificando al paquete TCP SYN.

2. El conmutador determina el servidor más apropiado para maneja esta respuesta y atar la nueva sesión TCP a la dirección IP de ese servidor en una tabla.

3. Para todos los paquetes de entrada pertenecientes a una nueva sesión, el conmutador web sustituye el puerto TCP del conmutador, la dirección IP, y la dirección MAC para el servidor. Esto hace que los paquetes parezcan como si hubieran sido dirigidos al servidor por el cliente.

4. Asimismo, todos los paquetes de salida que pasan a través del conmutador de web son modificados como si pareciera que el conmutador web respondiera a la petición del cliente.

5. Cuando el conmutador web reconoce el paquete TCP FIN, el conmutador web borra la sesión del servidor atada desde la tabla.

2.2.4 APLICACIONES QUE PUEDEN IMPLEMENTARSE EN UN *NETWORK PROCESSOR*

- VOIP *gateway*
- Interconexión de redes - POS; AALS
- P2P
- SSL
- Diferentes algoritmos de encolamiento
- Ethernet
- SONET

2.3 ALGUNOS TIPOS DE *NETWORK PROCESSORS*

2.3.1 SISTEMAS AGERE – COMUNICACIONES OPTIMIZADAS. ARQUITECTURA DE *NETWORK PROCESSOR* PAYLOADPLUS

La arquitectura PayloadPlus es una solución completa al *network processor* OC-48c. Representa un avance tecnológico en la construcción de equipos de comunicación inteligente capaces de procesar las capas 2-7.

La familia de *network processors* PayloadPlus incluye:

- **FPP:** lleva a cabo la clasificación de la velocidad del cable y el análisis de múltiples protocolos de la capa 2 a la 7. FPP está programado con un lenguaje simple de procesamiento de protocolo (FPL): reconoce y clasifica los paquetes que le llegan basándose en millones de modelos de paquetes de datos (sin uso de CAMs caras) o en dispositivos de segmentación y reensamblaje (SAR).

- **RSP:** lleva a cabo algunos encolamientos, el mantenimiento del tráfico, la configuración del tráfico y las funciones de modificación de paquetes en el flujo de tráfico.

- **ASI:** proporciona interfaces PCI entre un host y procesadores Agere de alta velocidad para el control y el mantenimiento de funciones (tabla de encaminamiento y actualizaciones del circuito virtual, configuración de hardware y manejo de excepciones).

2.3.1.1 OBJETIVO DE LAS APLICACIONES

- Capas 2 y 3 para conmutación y encaminamiento.
- Creación de servicio de capa más alta.
- VPNs.
- Protocolo de inter-trabajo.
- Procesamiento de la lista de control de acceso.
- Procesamiento de paquetes del sistema de acceso y línea de servicios.
- Calidad de Servicio programable.
- Aprovisionamiento de ancho de banda rápido.

- Procesamiento de voz y datos (basados en IP, AAL5 o AAL2).

Los ASICs pueden manejar nuevos protocolos o cómo son desarrolladas las aplicaciones o cómo son requeridas nuevas funciones de red. La propuesta Agere también evita el uso de componentes caros como dispositivos CAM o SAR.

2.3.1.2 ARQUITECTURA HARDWARE BASADA EN PIPELINE CON PAYLOADPLUS OPTIMIZADO

La arquitectura PayloadPlus utiliza una tecnología llamada "pattern-matching optimization" para lograr un mayor rendimiento multiplicando por 5 la mejora sobre procesadores RISC. Este rendimiento alcanza el nivel de funciones ASICs fijadas mientras permite una flexibilidad equivalente y una programabilidad superior comparado con las soluciones RISC.

PayloadPlus hace ambas prestaciones y simplifica la programación usando un motor de "pattern-matching" y una arquitectura pipeline: cabeceras más bajas, ciclos de reloj más necesarios y más procesamiento de datos por transistor que otras soluciones.

El procesamiento de tareas se lleva a cabo en uno o dos dominios:

- **Ruta rápida:** procesamiento normal de cadena de datos.
- **Ruta lenta:** manejo de excepciones, mantenimiento, configuración y actualizaciones.

Atributos clave de FPP:

- Programabilidad de alto nivel/rendimiento detallado.
- Rápidas búsquedas detallistas.
- Uso de memoria estándar.
- Alto rendimiento en velocidades de reloj más bajas.
- SAR: combinación de FPP y de RSP.

Atributos clave de RSP:

- Transmite cola de datos (QoS y CoS).
- Parámetros QoS y CoS de cada cola programables.

- Descarta completamente las políticas programables.
- Modificación completa de datos que salen programables.
- Soporte para multicast.
- Soporte para tráfico en tiempo real.
- Soporte para programa independiente.
- Capacidades de segmentación para interfaz de fábricas basadas en celdas o interfaces de línea ATM.
- Generación de checksums requeridas y CRCs.
- Interfaces estándares de industria para entrada y salida.

Atributos clave de ASI:

- Interfaz de industria estándar para FPP y RSP para microprocesador host.
- Alta velocidad, mantenimiento de flujo de estado orientado para FPP.

Ejemplo de arquitectura PayloadPlus: procesado de paquetes de voz (VPP).

2.3.1.3 EJEMPLO DE APLICACIÓN GATEWAY 3G/MEDIA

El tráfico analógico es digitalizado por un subsistema DSP, que lo envía al *network processor* para procesamiento e interfaz del tráfico para redes ATM o IP. En este caso, PayloadPlus trabajaría junto con un sistema DSP. El tráfico desde el subsistema DSP sería clasificado, y después encaminado a la conexión apropiada. El VPP se utilizaría para manejar tráfico AAL2 ya que ese protocolo es usado en ambas aplicaciones.

Con esta configuración, varios paquetes, celdas y muestras de voz, una vez clasificado en el FPP, serían encolados y programados a la red apropiada usando las opciones de mantenimiento del tráfico RSP programable. Además, el procesamiento natural de la cadena de bits programable de la arquitectura PayloadPlus permite al sistema adaptar a los nuevos estándares, protocolos y entornos de red para manejar un array amplio de tráfico de red, incluyendo AAL2, AAL5, CPS, VoIP, IPv4, IPv6, MPLS, estructura de retransmisión y más.

2.3.1.4 ARQUITECTURA SOFTWARE Y VISIÓN GENERAL

Los *network processors* PayloadPlus son programados en lenguajes orientados a la aplicación de alto nivel. FPL es usado para clasificación de paquetes. Requiere pocas líneas de código tal

como el lenguaje C, que ofrece beneficios en el reducido desarrollo de tiempo y coste, mejora los defectos de radio y mantenimiento.

ASL (Lenguaje Script Agere) es utilizado para la modificación de paquetes, la vigilancia y las estadísticas. Puede ser mapeado eficientemente en los motores subyacentes que proporcionan funcionalidad asociada en los dispositivos PayloadPlus.

FPL requiere pocas líneas de código para programar una función. El código es intuitivo, pero hay que entender la construcción y sintaxis básica de FPL. Con pocas líneas de código es más fácil debutar, rehusar y modificar. Los programas suelen ser pequeños y no necesita optimizar el rendimiento en el montaje o los lenguajes de nivel más bajos.

El entorno de desarrollo será SDE: proporciona lenguajes de alto nivel para programación más rápida, como para el acceso al nivel API para configurar cuando sea necesario.

2.3.1.5 RENDIMIENTO

- **Precio:** $650
- **Voltajes requeridos:** 1.8 V núcleo / 3.3 V o 2.5 V E/S para FPP, RSP y ASI.
- **Potencia:** 12 W es la máxima potencia para los tres componentes (FPP, RSP, ASI).

2.3.1.6 ENLACES

Network processors:

http://www.lsi.com/networking_home/networking_products/network_processors/index.html

2.3.2 SISTEMAS CISCO – TOASTER2

Es un *network processor* de alto rendimiento capaz de enviar millones de paquetes por segundo. El ASIC está internamente organizado como un pipeline, multiprocesador, motor de procesamiento en paralelo.

2.3.2.1 OBJETIVO DE LAS APLICACIONES

El ASIC proporciona un envío rápido de paquetes basado en el procesamiento de la cabecera del paquete.

Un sistema basado en Toaster consistirá de uno o más ASICs Toaster, un buffer de paquete ASIC, interfaces de medios específicos, y un procesador de ruta (para ejecutar protocolos de encaminamiento, recoger estadísticas del mantenimiento de red, y mantener tablas de remitente Toaster). El *network processor* opera sobre las cabeceras de red y un buffer de paquete de ASIC para guardar el cuerpo del paquete mientras se está procesando la cabecera del paquete.

Las interfaces de especificación media adaptan múltiples interfaces físicas a un formato de paquete común usado por el buffer de paquete y los *network processors*. El sistema Toaster proporciona la capacidad para soportar diversos rangos de interfaces de ATM a Ethernet.

Toaster2 ha sido diseñado para soportar múltiples configuraciones divididas en 2 categorías: centralizadas y distribuidas. Se escoge según el requerimiento del objetivo de la plataforma delantera.

- **Centralizado:** para sistemas cuyas tasas de transferencia agregada es menor que OC-48. Sin él, todos los paquetes son enviados por un simple Toaster o array pipeline de los Toaster ASICs.

- **Distribuida:** usado para incrementar la capacidad delantera del sistema para permitir que múltiples Toasters operen en paralelo. Cada interfaz media podría tener un buffer ASIC dedicado y un *network processor*. La fábrica switch interconecta cada grupo de ASICs para incrementar la tasa de transferencia del sistema. Incrementa la tasa de transferencia del paquete que se envía basándose en la cantidad de paralelismo y el número de rutas no bloqueantes a través de la fábrica de switch.

2.3.2.2 EJEMPLO DE FLUJO DE PAQUETE PARA UN SISTEMA CENTRALIZADO

Las operaciones se aplican a un paquete como flujos de paquetes a través de un motor centralizado para su envío consistiendo de 2 Toasters y un buffer de paquete ASIC (las interfaces medias específicas no son parte del motor).

2.3.2.3 EJEMPLO DE FLUJO DE PAQUETE PARA UN SISTEMA DISTRIBUIDO

Las operaciones se aplican a un paquete como flujos de paquetes a través de un motor distribuido para su envío. Hay 4 motores, y cada uno está compuesto de un simple Toaster y un

buffer de paquetes distribuidos ASIC. Las tareas de Toaster están divididas en entrada y salida del motor para enviar.

Una implementación multicast simple puede lograrse debido a la entrada Toaster para producir múltiples respuestas de encolamiento para la entrada del buffer de paquete ASIC. El buffer de entrada del paquete crearía múltiples copias del paquete, cada una direccionada a una interfaz media distinta.

2.3.2.4 ARQUITECTURA SOFTWARE

Existen dos componentes asociados con la arquitectura software. **Parallel Express Forwarding** describe la porción de microcódigo de la arquitectura software que se ejecuta en el *network processor*. El procesador de ruta debe ser capaz de configurar y manejar la memoria del Toaster basada en la variedad de configuraciones, manejar paquetes para enviar los casos que no se soportan en el microcódigo y periódicamente coleccionar estadísticas de retransmisión. El driver del dispositivo Toaster es parecido al driver del dispositivo basado en hardware. La interfaz debe diseñarse de manera flexible para acomodar futuros cambios funcionales.

2.3.2.5 ENTORNO Y METODOLOGÍA DEL DESARROLLO DEL TOASTER

Cada microcódigo debe particionarse de manera que encaje en una o más estancias de pipeline. Sin el Toaster, cada columna de procesadores representa una estancia pipeline.

El número de competencia del procesador reduce el número permitido de accesos a memoria permitida para un procesador dado.

El tamaño de operaciones de memoria también afecta al número de accesos permitidos.

La “memoria de programa”: técnica microcódigo para proporcionar memoria eficiente.

Cuando la función microcódigo se ha particionado para encajar en una estancia de software, hay herramientas para simular el funcionamiento del microcódigo. El microcódigo para que pueda ejecutarse en Toaster2 debe escribirse en código ensamblador TMC. Tiene un macro ensamblador para generar código máquina y chequearlo.

El ciclo de simulación exacto para ejecutar microcódigo emula un Toaster2 o un chipset (modelos basados en C para múltiples ASICs Toaster2, buffer de paquetes y dispositivos de interfaz media).

La memoria de programa sirve para aplicaciones que requieran incrementar el rendimiento asociado con la memoria de programa.

2.3.2.6 ENLACES

La página en España:
http://www.cisco.com/global/ES/index.shtml

Para poder bajar software, es necesario registrarse:
http://tools.cisco.com/RPF/register/register.do

El software que se puede adquirir es el siguiente: http://www.cisco.com/public/sw-center/index.shtml

2.3.3 PMC-SIERRA, INC. – CLASSIPI

El dispositivo ClassiPI (Clasificación por Inspección de Paquetes) asiste a los *network processors* descargando la tarea compleja y de tiempo de consumo de la compleja clasificación de paquetes. Es compatible con interfaces sin ninguna pega lógica para una variedad de comercialización disponible de *network processors*. Cuando se usa junto con ClassiPI, la generación común de *network processors* puede entregar el OC-48 llevándose a cabo en soluciones de características ricas como opuestas a unos simples tipos de aplicaciones que se llevan a cabo. La arquitectura ClassiPI permite una fácil interfaz de software para arquitectura de *network processor* multinúcleo. También permite una representación eficiente de los tipos de reglas de la lista de control de acceso (ACL) comparadas con los dispositivos CAM, llevando a una regla más pequeña del tamaño de memoria, que vuelve a conducir a una potencia de consumo más baja. ClassiPI tiene una función especial: acelera las aplicaciones que requiere procesamiento de carga explosiva (capa 7).

2.3.3.1 OBJETIVO DE LAS APLICACIONES

El objetivo de las aplicaciones va a determinar los requisitos de clasificación de equipamiento de red.

- En las aplicaciones de conmutación y retransmisión, se requieren la combinación exacta de la dirección MAC L2 (o etiqueta como en MPLS LSR) y mayor longitud del prefijo de combinación de la dirección de la capa 3 (para MPLS LER y retransmisión).

- Aplicaciones más complejas de filtrado de paquetes (firewall, VPN y equipamiento de conexión de redes que aporte capacidad de QoS): requieren una combinación exacta, prefijo y chequeo de rango para capas 2 y 3, y direcciones de capa 4.

- Aplicaciones de capas más altas (balance de carga del servidor, alojamiento Web, y detección de intrusiones): búsquedas en la porción de carga explosiva (capa 7) del paquete.

Los *network processors* tienen un tamaño de instrucción de memoria limitado y una incapacidad general para procesar carga explosiva de datos. ClassiPI direcciona esta limitación para rendimiento de una profunda clasificación del paquete. Este dispositivo (diseñado para una interfaz fácil con una variedad de red o procesadores ASIC a medida), puede descargar la tarea basada en la clasificación del contenido del paquete desde el procesador. De hecho, sus características de búsqueda de contenido de la capa 7 puede encontrar ocurrencias de una variedad de cadenas de longitud variable donde quiera sin el paquete. Tiene regla de memoria en el chip y baja potencia de consumición.

2.3.3.2 IMPLEMENTACIÓN CLASSIPI

- 3.3 V para interfaces E/S y 1.5 V para voltaje de núcleo.

- Reloj en la interfaz de entrada con frecuencia superior a 116 MHz (2 PLLs), 2 por interfaz de reloj (232 MHz).

2.3.3.3 ARQUITECTURA SOFTWARE Y KIT DE DESARROLLO

Herramienta de desarrollo y depurador para ClassiPI: SDK. Incluye un modelo API y drivers, un marco de modelo para varias aplicaciones software, y significados para desarrollo y depuración de software.

2.3.3.4 PLATAFORMAS

La arquitectura SDK de ClassiPI soporta variedad de plataformas.

Modos de desarrollo para utilizar el SDK de ClassiPI:

- **Software:** simulador ClassiPI (modelo C) se usa en un entorno de PC/estación de trabajo para simular la función del dispositivo (antes de que la plataforma hardware esté lista).

- **Hardware:** el dispositivo ClassiPI se integra en el sistema hardware cuando la plataforma hardware está lista.

Plataformas: entorno Linux/WinNT PC y algunas plataformas de *network processors*.

Tarjetas para las plataformas soportadas:

- **PCI:** tapa a un PCI backplane.

- **Tarjeta hija:** permite conectores en varias plataformas de *network processors*.

2.3.3.5 MÓDULOS

- Drivers del software, APIs y varias aplicaciones simples.
- Simulador de funcionalidad exacta.
- Dispositivo depurador.

2.3.3.6 DESARROLLO SOFTWARE

Módulos de "Ruta de Control" del SDK ClassiPI: plasma la regla y el recurso asociado al mantenimiento y configuración del ClassiPI para varias aplicaciones de red. Los componentes del núcleo comprimidos de drivers y diferentes rutinas de función exportan una serie de interfaces de aplicación programables (APIs) que pueden usarse para mantener el ClassiPI por un código de aplicación. Las aplicaciones del software incluidas en ClassiPI SDK presentan una estructura de programación para implementación real de la meta de la función de aplicación de red en una plataforma de equipamiento específico utilizando ClassiPI como un procesador de clasificación. Las aplicaciones incluyen:

- Búsqueda de la cadena de carga explosiva del paquete para procesar el contenido del paquete.
- Expresión regular para aplicación de seguridad de alto nivel.
- Encaminamiento IP.
- Filtrado de capa 3 a través de capa 7.

Escrito en código C y compilado bajo Linux o VxWorks.

Módulos de "Ruta de Dato": compuesta principalmente por procesamiento de paquete y retransmisión de paquetes. El código es dependiente de las aplicaciones específicas de red y del *network processor* (CPU) usado en una plataforma específica. El código escrito está basado en la sintaxis del microcódigo del *network processor* involucrado.

2.3.3.7 SIMULADOR

Es una implementación totalmente funcional del software (modelo C) del ClassiPI. Es un programa ejecutable que se comunica por una interfaz de socket.

No es un ciclo exacto modelo C. Pero se puede permitir un retardo controlado desde la petición al resultado. Se puede estimar el rendimiento, la latencia, y los retardos cuando se usa ClassiPI en distintos escenarios.

2.3.3.8 DEPURADOR

Se comunica con el simulador o dispositivo ClassiPI. Aporta facilidades para depurar en una plataforma o aplicación de red dada previendo una interfaz de línea de comando que se integra en la plataforma de interfaz shell de usuario o conectado remotamente por socket TCP/IP utilizando una PC/estación de trabajo a distancia.

El usuario interactúa mediante interfaz de línea de comando (CLI) o una interfaz gráfica de usuario (GUI).

Operaciones en el depurador:

- Lectura/Escritura del conjunto de registros locales y globales.
- Lectura/Escritura de la base de datos en varias plataformas.
- Lectura/Escritura de datos asociados (estadísticos y definidos por el usuario).
- Escritura fuera de línea de paquete de entrada (cadena de datos).
- Lectura fuera de línea del resultado de la clasificación.

2.3.3.9 RENDIMIENTO

- **Precio:** 200$
- **Tecnología:** 0.18 micron
- **Empaquetado:** 352-pin TEBGA.
- **Voltajes requeridos:** 1.5-1.6 V núcleo, 3.3 V E/S.
- **Potencia:** por debajo de 4.0 W máximo en el peor de los casos.

2.3.3.10 ENLACES

Página principal:
http://www.pmc-sierra.com/

SDK ClassiPI:
https://www.pmc-sierra.com/products/details/pm2329/

2.3.4 TRANSWITCH-ASPEN: PROCESAMIENTO DE RED FLEXIBLE PARA SOLUCIONES DE ACCESO

ASPEN combina las capacidades del *network processor*, procesando cada célula ATM o paquetes ATM.

Principales aplicaciones:

- Acceso de segmento WAN.
- Accesos específicamente multiplexados DSL (DSLAM).
- Conmutar voz sobre IP.

2.3.4.1 APLICACIONES

Las redes de telecomunicaciones se están transformándose en servicios de voz basados principalmente en paquetes, soportando una variedad de servicios orientados a datos. Requiere unas pocas capas de mantenimiento, conversión de protocolo, y multiplexación. El objetivo principal es concentrador y multiplexar el acceso a multiservicio.

Arquitectura de Red DSL con un splitter POTS:

DSL ventila el ancho de banda de una línea POTS usando un espectro más largo que se usa por voz, para empaquetar más bits de datos en cada banda de frecuencia, y por minimizar el cruce de conversación con cancelación del eco o con división en frecuencia.

DSLAMs basado en ADSL usa ATM como su principal mecanismo de transporte: procesamiento de celdas, mantenimiento del tráfico, OAM&P y conmutación.

Funciones DSLAM: proporciona una capacidad de conmutación entre cadenas de datos de entrada/salida (cellbus lo hace posible).

Cellbus: puede implementar en una tarjeta de circuito o en una configuración de backplane entre múltiples tarjetas de circuitos. Hay una función de arbitración central que se usa para resolver contención de acceso al bus.

Varias estructuras de conmutación de paquetes o multiplexación de paquete puede formarse interconectando varios ASPEN sobre cada 2 buses CellBus. Un paquete en un cellbus puede encaminarse a otro cellbus o a múltiples cellbus.

En la arquitectura típica de un cellbus para acceso a nodo multiservicio, PHAST es un dispositivo de línea Terminal SONET/SDH que lleva a cabo el procesamiento de la capa física ATM y PPP.

Los paquetes de longitud variable como IP y Ethernet también pueden ser conmutados y transportados usando segmentación y reensamblaje de ASPEN.

ASPEN permite una agregación y concentración de bajo coste para procesamiento de distintos tipos de radio de acceso al tráfico.

2.3.4.2 ARQUITECTURA Y OPERACIONES ASPEN

2.3.4.2.1 PROCESADORES Y COPROCESADORES

La arquitectura ASPEN está basada en 3 procesadores RISC, un conjunto de procesadores, un sistema de interconexión bus/switch y varias lógicas de soporte.

2.3.4.2.2 CONTROL Y FLUJO DE DATOS

Los movimientos de las celdas a través de ASPEN pueden describirse en términos de las siguientes acciones:

1. **Celdas de llegada son recibidas desde la entrada UTOPIA (controlado por la entrada del procesador ACE):**

   El procesador ACE controla la clasificación de la celda (celda usuario o celda OAM). Si la celda pertenece a una conexión virtual activa válida, devuelve el índice de conexión de la celda de entrada. Si la celda cumple los parámetros sobre el acuerdo del tráfico contratado, se traduce la celda como es requerido y se pasa al DMA para encolamiento. Las celdas no conformes son etiquetadas o descartadas. Si la celda recibida es OAM, se pasa a la unidad de procesamiento OAM. Después, son pasadas al DMA para encolamiento.

2. **Las celdas que salen van a la salida UTOPIA (controlado por la salida del procesador ACE):**

   Las celdas de entrada son recibidas por un bloque de interfaz CellBus y almacenadas en colas externas. El ACE de salida monitoriza los buffer de datos para espacio de celda disponible, y cuando se obtienen, devuelve una celda.

   BICOP recibe las celdas del DMA y son encoladas en un buffer interno. El router de datos pasa la cabecera al compilador de mensaje y todo el buffer al controlador de datos. El compilador del mensaje manda una cabecera de búsqueda del mensaje al DMA y se realiza la búsqueda a la tabla de conexión.

   Todas las colas de celdas (tanto de entrada como de salida) son contenidas en la SRAM. Las celdas de entrada del UTOPIA de entrada o del CellBus son encoladas en la SRAM.

3. **Movimiento de celdas desde/a el CellBus (controlado por la velocidad del procesador ACE):**

   QMCOP tiene una ruta especial para comunicaciones con SRAM y una lógica para obtener las posibles latencias más pequeñas para determinar las localizaciones del buffer de datos y actualización de los punteros a las tablas. También maneja la generación CRC-32 para flujos AAL5.

Destinos del flujo de datos desde/a las interfaces CellBus: controlador DMA, y procesadores ACE de entrada, salida y velocidad. El dato se pasa al DMA para ser encolado externamente en la SRAM. El dato puede encaminarse a cada uno de los 3 procesadores ACE.

Hay 4 colas de tráfico de entrada y salida que soportan 4 niveles de prioridad (CoS). Estas colas son almacenadas en buffer en SRAMs externas. Los niveles de prioridad son de mayor a menor. Las celdas de entrada del UTOPIA con escritas en una simple cola. Las celdas que entran al CellBus son escritas en puertos de colas donde son programadas para la interfaz UTOPIA. En la salida, los puertos de colas UTOPIA sin prioridad son servidos usando una disciplina de torneo.

2.3.4.3 ENTORNO DE PROGRAMACIÓN

El entorno de programación está compuesto por:

- **Compilador C:** compilador estándar ANSI para procesadores Transwitch ACE, como es el ensamblador/depurador.
- **Simulador:** simulador de nivel de instrucción adaptado hasta dar resultados exactos de una manera muy oportuna.
- **API:** sirve como interfaz funcional entre el host y el ASPEN en un entorno de cliente. Está en lenguaje C. Utilidades: inicialización, encolamiento, y configuración del ASPEN. De esta manera se esconde del host la complejidad de comunicación con ASPEN. Facilita al cliente modificar su sistema software para acomodarlo al nuevo dispositivo. Las funciones extendidas requerían nuevo software.
- **Ensamblador.**
- **Depurador.**

2.3.4.4 APLICACIONES

- Comunicaciones del segmento de acceso a datos WAN (DSLAMs y conmutación de VoIP).
- ASPEN ha sido extendido para incluir aplicaciones para MTU y AAL2 para voz ATM mediante ASPEN-PX.

2.3.4.5 ENLACES

Distribuidores en España:
http://server-die.alc.upv.es/electroweb/fabricantes/empresas/transwitch.htm

2.3.5 INTEL S.A. – *NETWORK PROCESSOR* INTEL IXP2400: UNA SEGUNDA GENERACIÓN DE NPU INTEL

2.3.5.1 OBJETIVO DE LAS APLICACIONES

Requerimientos del procesamiento de paquetes**:**

El acceso al equipo de red debe soportar múltiples interfaces y protocolos. Este equipo necesita reunir una fuerte potencia y requerimientos del estado real dedicados por contrastes de espacio en los armarios del cableado. El equipo en el extremo de la red debe soportar rápido aprovisionamiento de servicios, ejecución escalabre para proporcionar soporte para servicios emergentes en la velocidad del cable, y migración sin problemas a estándares emergentes. La solución es IXP2400.

Aplicaciones IXP2400:

- Conmutación multiservicio.
- DSLAMs.
- CMTS.
- Infraestructura wireless 2.5 G y 3 G y conmutaciones de las capas 4-7 (firewalls…), gateways VoIP, acceso a plataformas multiservicio, routers extremo, concentradores de acceso remoto y gateway VPN.

Objetivos para el mercado:

- Agrupación, QoS, funciones ATM SAR, configuración del tráfico, vigilancia, rendimiento, y conversión de protocolo en el equipamiento DSLAM.
- Agrupación, QoS, rendimiento, y conversión de protocolo en equipamiento CMTS.
- ATM SAR, encriptación y rendimiento en los controladores de red de la estación base de radio control (BSC/RNC).
- Protocolo de tunelado GPRS, tunelado, e IPv6 e infraestructura inalámbrica.
- ATM SAR, etiqueta multiprotocolo de conmutación (MPLS), QoS, configuración del tráfico, vigilancia, conversión de protocolo, y agrupación para conmutadores multiservicio.
- Balance de carga del contenido consciente, rendimiento, y vigilancia para descarga del servidor del extremo.

2.3.5.2 ARQUITECTURA HARDWARE

IXP2400 está compuesto por dos tipos de interfaces:

- Interna.
- Externa

Características del núcleo de Intel XScale:

- Procesador RISC de 32 bits.
- Inicializa y maneja el *network processor*, maneja excepciones, y lleva a cabo las rutas lentas de procesamiento y otras tareas del plano de control.

Micromotor:

Realiza muchos de los procesamientos de paquetes programables.

Proporciona soporte para operaciones multi hilo de software controlado (un hilo bloqueará las operaciones de espera para memoria).

SRAM:

Consta de dos controladoras SRAM independientes. Cada una soporta RAM estática síncrona QDR pipeline y/o coprocesador que adhiere señalización QDR.

Unidad Scratchpad y Hash:

- Scratchpad: proporcionan una memoria SRAM en un chip de 16 KB que puede usarse para opciones de propósito general por el núcleo XScale y el micromotor. También proporciona 16 alarmas de hardware para usarse en comunicaciones entre los micromotores y el núcleo.
- Hash: proporciona un acelerador hash de polinomios (búsqueda ATM VC/VP y clasificación IP tupla-5).

PCI, periféricos Intel XScale y rendimiento de los monitores:

- **PCI:** 64 bits. Puede usarse para conectar host o dejar caer atados los periféricos. Trabaja a 66 MHz.
- **XPI:** conectar IXP2400 al controlador de interrupciones, timers, UART, GPIO, y flash ROM.
- **MP:** contadores para contar eventos seleccionados del hardware interno del chip.

2.3.5.3 MEDIO DE DESARROLLO SOFTWARE

Intel IXP2400 proporciona herramientas en varias áreas clave: desarrollo del código del plano de control, plano de datos y desarrollo de herramientas del sistema.

SDK 3.0 permite desarrollo hardware y software para proceder en paralelo. El entorno gráfico de desarrollo es fácil de usar para desarrollo, depuración, optimización y simulación del código.

Herramientas de desarrollo:

- **MDE:** desarrollo del banco de trabajo, compilador y transactor. Proporciona a los desarrolladores dos lenguajes de programación:
- **C:** más rápido al mercado, portabilidad óptima de código y beneficios de un lenguaje de programación familiar de alto nivel que proporciona aislamiento de hardware específico.
- **Microcódigo:** maximiza el rendimiento de la aplicación y minimiza el espacio de utilización.
- **SDK 3.0:** soporta WindRiver VxWorks 54 (big-endian) y Linux (BE) en el núcleo XScale.

Modelo de programación:

Los micromotores emplean un modelo software pipeline en la ruta más rápida de procesamiento de los paquetes. Cada pipeline está compuesto de elementos más pequeños (“plataforma pipeline”).

Hay dos tipos de pipeline: pipeline de contexto y pipeline funcional.

Estructura de la portabilidad:

Proporciona un desarrollo rápido y de coste eficiente de código, mientras invierte en la protección de software a través de portabilidad de software y el rehúso de código para micromotores y núcleo Intel XScale.

Permite el desarrollo de bloques de código modular y portátil, y la integración de software de larga vida y fácil mantenimiento, mientras elimina el tiempo de consumición desarrollado de software de infraestructura funcional.

2.3.5.4 CONFIGURACIONES DEL SISTEMA IXP2400 Y ANÁLISIS DEL RENDIMIENTO

Las funciones clave que se llevan a cabo por el equipamiento en el objetivo del espacio de mercado para IXP2400 son:

- Encaminamiento IPv4 e IPv6.
- IP DiffServ y QoS.
- ATM AAL5 y AAL2 SAR.
- Vigilancia y forma de tráfico ATM.
- Compresión de cabecera IP/UDP/RTP.
- Encapsulación/Desencapsulación.
- Tunelado.

2.3.5.5 RENDIMIENTO

- **Tecnología:** procesador Intel 0.18 micron.
- **Voltaje requerido:** 1.3 V núcleo, 1.5 V QDR, 2.5 V DDR, y 3.3 V E/S.
- **Potencia:** típico 10 W; máximo 12.4 W.

2.3.5.6 ENLACES

Procesador IXP2400:
http://www.intel.com/design/network/products/npfamily/ixp2400.htm

Para descargar el SDK:
http://www.intel.com/design/network/products/npfamily/sdk_download.htm

Premier Members en Madrid:
http://premierlocator.intel.com/select_profile.asp

Contacto:
http://www.intel.com/cd/corporate/contact/emea/spa/254122.htm

Distribuidores autorizados:
http://indigo.cps.intel.com/dlt/distributorList.aspx?country=Spain

Dónde comprar:
http://www.intel.com/buy/networking/design.htm?matrix=npfamily&prod=IXP2400

2.3.6 IBM – *NETWORK PROCESSOR* POWERNP

PowerNP proporciona procesamiento de paquetes a velocidad de cable y capacidad de forma a través de un conjunto de procesadores incrustados y coprocesadores programables con una multiplicidad de ancho de banda incrustada y memoria externa. Los coprocesadores operan en paralelo con procesadores y ejecutan funciones. PowerNP soporta múltiples interfaces de red a través de MACs integradas. Tiene un procesador IBM PowerNP incrustado que puede funcionar como el Punto de Control (CP) para el sistema. Proporciona una interfaz de bus PCI que puede usarse para comunicarse con el PowerNP interno, así como otros procesadores dentro de PowerNP.

PowerNP puede utilizarse en equipos routers/conmutadores tradicionales así como en aplicaciones emergentes para equipamiento del servidor y almacenamiento. También proporciona funciones de procesamiento del plano de control y mantenimiento del sistema y actúa como el CP para el sistema entero. CP puede adjuntarse a través de la interfaz PCI.

Para aplicaciones emergentes en el extremo de red, PowerNP puede utilizarse para soportar funciones de monitorización de funciones escalables o rasgos de seguridad.

2.3.6.1 SOFTWARE

Ofrece un modelo de programación paralelo “run-to-completion” (ejecución-a-finalización) que permite al programador ver una imagen simple que procesa paquetes desde llegada a partida. Permite al programador acceder al espacio de instrucciones de memoria y a todas las partes de los recursos.

2.3.6.1.1 ARQUITECTURA SOFTWARE

- **Plano de Control**: Los protocolos están mejor apropiados para GPPs ya que poseen rutas largas de código y paralelismo exhibido de datos pequeños. El plano de control puede ser el PowerNP apropiado o un procesador externo. El plano de control tiene la responsabilidad de inicializar el sistema y sobre todo ocuparse del mantenimiento del sistema. Hay dos tipos de APIs: servicios de protocolos APIs (maneja objetos de protocolos de independiente hardware como IP y MPLS) y servicio de mantenimiento de APIs (maneja recursos hardware PowerNP y es usado para servicios como inicialización PowerNP y depurador).

- **Plano de Datos:** Los protocolos son los responsables de la ejecución de los paquetes. Reúne requerimientos de QoS. El plano de datos es más apropiado para procesadores en paralelo ya que tienen rutas cortas de código y paralelismo de datos largo debido a la independencia entre paquetes. El software del plano de datos tiene dos componentes: librería del sistema (funciones como mantenimiento de la memoria, depurador y árbol de servicios) y software de ejecución. Estas funciones proporcionan una capa de abstracción que puede usarse desde el plano de control, usando una interfaz de paso de mensaje o desde el software del plano de datos usando llamadas a API. El software de ejecución es el responsable para la ejecución del procesamiento de paquetes de alta velocidad que maneja la porción de estado seguro del plano de datos de aplicaciones de red (protocolos de ejecución de paquete IP).

2.3.6.1.2 KIT DE HERRAMIENTAS DE DESARROLLO SOFTWARE

Son herramientas que direccionan cada fase del proceso de desarrollo software. Son diseñados para ejecutarse en la parte superior de cualquier arquitectura hardware o el modelo de simulación del nivel de chip.

- **Sistemas:** Windows, Sun Solaris y Linux.
- **Componentes:** ensamblador, depurador, generador del caso de test y perfil de ejecución.
- **Ensamblador:** genera imágenes desde el pico código que ejecuta en un modelo de simulación de nivel de chip o PowerNP.
- **Simulador:** proporciona un intérprete TCL que permite a un programa ejecutar pico código en un entorno simulado.
- **Depurador:** genera archivos que el programa puede usar para depurar pico código.
- **Generador del caso de test:** proporciona una interfaz TCL para el testeo inicial de pico código y para regresión del caso del test.
- **Perfil de ejecución:** genera ejecución de pico código usando un modelo de simulación basado en eventos.

2.3.6.2 PRESTACIONES

Muchos coprocesadores utilizados junto con otros coprocesadores para llevar a cabo el procesamiento de paquetes. Medidas de prestaciones:

- Ciclo de presupuesto abierto por paquetes para POS y ethernet.
- Espacio para la cabeza disponible sobre ejecución de paquetes IP típicos.

2.3.6.3 ENLACES

Página principal de IBM:
http://www.ibm.com/es/

2.3.7 MOTOROLA – *NETWORK PROCESSOR* C-5E

- Está altamente integrado, es flexible y es funcionalmente un rico procesador para desarrollo y despliegue de servicios avanzados para las redes de próxima generación.
- Gbps de ancho de banda de línea y más de 4500 MIPS de potencia de computación.

- Objeto de demanda.
- Soporta dispositivos externos para proporcionar extensiones en procesamiento y ancho de banda necesitado.
- *Network processor* C-Port tiene protocolo agnóstico, soportando IP, ATM, e interfaz de protocolo personalizado.
- API robusta.

2.3.7.1 OBJETIVO DE LAS APLICACIONES

Su objetivo son el acceso y las aplicaciones de red del extremo. La clave de acceso al extremo que refleja estos requisitos incluye:

- Plataforma multiservicio de acceso al extremo.
- Acceso a banda ancha.
- Gateway media.
- Alta función de encaminamiento.

2.3.7.1.1 EJEMPLO DE APLICACIÓN: CMTS

El diseño del equipo CMTS puede ser cuestionado debido a la combinación de la evolución y de la diversidad de protocolos, alto rendimiento, y avanzados requerimientos de QoS que ocurren en el espacio de cable de la banda ancha. Con una tendencia hacia implementaciones más distribuidas, una plataforma de acceso al diseño, donde el mismo software puede ser apalancado en cualquier lugar desde los puntos de distribución hacia el final de la cabeza del cable, ofreciendo beneficios significantes. La familia de *network processors* C-Port trae alto servicio, altos requerimientos de funcionalidad junto en una solución integrada de silicio. La programabilidad de la línea de las interfaces C-Port permite al procesador adaptarse flexiblemente a los requerimientos de la capa 2.

2.3.7.2 ARQUITECTURA HARDWARE

El *network processor* C-5e también consta de varios coprocesadores especializados que soportan CPs (Procesadores de Canal) y XP:

- Unidad de tabla de búsqueda (TLU).
- Unidad de mantenimiento de buffer (BMU).
- Unidad de mantenimiento de cola (QMU).
- Fábrica de procesador (FP).

2.3.7.3 ARQUITECTURA SOFTWARE

La arquitectura de los *network processors* C-Port fue diseñada para soportar un modelo simple de programación, que incluye la habilidad de programar *network processors* C-Port en lenguaje C usando una capa API de abstracción. La API C-Ware define las actividades más comunes de encaminamiento, así como el mantenimiento de la interfaz física, enviar datos, búsquedas de tabla, mantenimiento del buffer, y operaciones de encolamiento.

API C-Ware está formado por servicios que direccionan cada uno de los chips a bloques funcionales: protocolo y PDU, servicios de tabla, servicios de buffer, servicios de colas, servicios de fábricas, servicios kernel, servicios de diagnóstico.

Programar a las APIs C-Ware asegura compatibilidad de software y escalabilidad de generación en generación de la familia C-Port NP y entre emisión CST.

2.3.7.3.1 REFERENCIA DE LAS APLICACIONES

La Librería de Aplicaciones C-Ware (CAL) proporciona un rico conjunto de soluciones funcionales a las comunicaciones que pueden usarse como modelos de referencia o integrados en productos o servicios de alto nivel.

CAL incluye 10/100 Ethernet, Gigabit Ethernet y conmutadores y routers de Canal de Fibra; un gateway de canal de fibra; ATM SAR; AAL2 y conmutador de celda ATM; y gateway media de VoIP a VoATM. Una aplicación Ethernet soporta puente, encaminamiento, VLAN, y prioridad de encolamiento. Los clientes que tienen licencia para el conjunto de herramientas de software para el C-Ware reciben el código fuente entero de esas aplicaciones y pueden usarlas en sus productos basado en unos derechos de autor gratis bajo los términos de licencia del CST.

2.3.7.3.2 HERRAMIENTAS DEL SOFTWARE C-WARE (CST)

Consta de un compilador C y un depurador basado en GNU, un entorno de simulación funcional y de ejecución, herramientas de análisis del rendimiento basado en GUI, y herramientas de escritura del tráfico.

El entorno de simulación da un modelo seguro del ciclo del chip. Todos los registros, memoria, núcleos RISC, buses, y otros componentes de hardware especializados son simulados.

Después de que el programa es escrito y cargado en el simulador, el tráfico puede alimentarse en el simulador para verificar el propio funcionamiento de la aplicación. El simulador recibe el tráfico en formato binario que lo reensambla.

El depurador permite al usuario pararse a través del código, añadir vigilancia, insertar puntos de parada, etc.

Se puede insertar etiquetas en el código para medir el tiempo que va desde una línea del código a otra.

2.3.7.3.3 SISTEMA DE DESARROLLO C-WARE

Motorola ofrece un sistema que incluye potencia de suministro, módulos de conmutación de *network processor* (SMs), y módulos de interfaz física (PIMs) de manera que los clientes pueden mezclar y encajar interfaces para crear una configuración que encaja con su sistema de objetivo final. Esto permite a los clientes hacer un prototipo de un completo sistema previo al sistema objetivo final disponible.

2.3.7.4 RENDIMIENTO

- **Reloj:** 266 MHz
- **Precio:** $450
- **Tecnología:** 0.15 micron
- **Potencia:** típica 9.2 W; máxima 13.0 W

2.3.7.5 ENLACES

Network processors en la página de motorola:
http://www.motorola.com/mediacenter/news/detail.jsp?globalObjectId=3332_2752_23

2.4 ELECCIÓN DEL *NETWORK PROCESSOR*

A la hora de seleccionar un *network processor* para utilizarlo en el desarrollo de una aplicación, debemos tener en cuenta una serie de características:

- La aplicación que vamos a desarrollar.
- Arquitectura.
- El lenguaje de programación.
- Las herramientas para el desarrollo del software.
- Las plataformas para su desarrollo.
- La facilidad de adquisición del producto, y el soporte que nos ofrece el fabricante.
- El equilibrio entre el coste del producto y su calidad.

Para este proyecto, buscaremos un *network processor* que cumpla en gran medida las características mencionadas anteriormente.

El objetivo de las aplicaciones del *network processor* Agere son, entre otras, el procesamiento de voz y de datos, una calidad de servicio programable, etc. Su arquitectura está dividida en 3 módulos: FPP (lleva a cabo la clasificación de los paquetes), RSP (se encarga de las colas y del mantenimiento del tráfico), y ASI (proporciona interfaz PCI entre un host y procesadores Agere). Los paquetes son programados en lenguajes de alto nivel (FPL) que requiere pocas líneas de código, lo que reduce el desarrollo del tiempo y del coste, mientras que la modificación de los paquetes, vigilancia, etc. son programadas en ASL. Como herramienta de desarrollo, está el SDE, que proporciona una programación más rápida. La propuesta Agere también evita el uso de componentes caros como dispositivos CAM o SAR. Su precio oscila sobre los $650.

Intel en cambio está más abierto a otro tipo de aplicaciones: encaminamiento IPv4 e IPv6, IP DiffServ y QoS, ATM AAL5 y AAL2 SAR, vigilancia y forma de tráfico ATM, compresión de cabecera IP/UDP/RTP, encapsulación/desencapsulación. Su arquitectura se divide en varios bloques, cada uno de ellos aportando distintas funcionalidades. Las aplicaciones se pueden escribir en lenguaje de programación C o microcódigo, y proporciona una herramienta de desarrollo llamada SDK, que permite el desarrollo tanto hardware como software para proceder en paralelo. Contiene un compilador, depurador, simulador y transactor. La plataforma de desarrollo puede ser Windows o Linux.

El procesador de red Motorola C-5e está formado por distintos coprocesadores. El lenguaje de programación para sus aplicaciones es C. Ofrece código fuente gratuito basados en derechos de autor. Su herramienta de desarrollo contiene un compilador, un depurador y un simulador. Podemos ejecutar la aplicación y analizar el rendimiento del *network processor* y el tráfico que genera.

Teniendo en cuenta el análisis anterior, Intel ofrece más variedad de desarrollo de aplicaciones, además de dar mayor soporte, incluyendo en España.

2.5 OPTIMIZACIÓN DEL DESARROLLO DE APLICACIONES PARA *NETWORK PROCESSORS* INTEL

2.5.1 DESCRIPCIÓN

Los fabricantes de equipos de telecomunicaciones se enfrentan al desafío cada vez mayor de ofrecer al mercado productos únicos de valor agregado, con rapidez y de forma rentable. Los *network processors* deben superar este desafío y ofrecer una plataforma de procesamiento de alto desempeño con capacidad de programación de software de fácil implementación y reutilización a través de varios productos. La línea de productos de *network processors* de Intel supera estos retos.

Al combinar el alto desempeño y la modularidad de la Arquitectura Intel® IXA (Intel® Internet Exchange) con un paquete de herramientas avanzadas de desarrollo de software que simplifican y automatizan las porciones críticas del proceso de desarrollo, Intel proporciona a los fabricantes de equipos de telecomunicaciones (TEM) la velocidad, flexibilidad y facilidad de uso y reutilización que necesitan. Resulta ser una combinación de éxito que provee un plazo de comercialización (TTM) más rápido, mayor duración en el mercado y una portabilidad superior de los diseños de un producto a otro.

2.5.2 MAXIMIZACIÓN DE LAS VENTAJAS DE DISEÑO DE LOS *NETWORK PROCESSOR* INTEL®

Los *network processors* Intel® IXP2400, IXP2800 e IXP2850 logran el desempeño y la flexibilidad gracias al uso de paralelismo en una arquitectura completamente programable. Con este método, las tareas de procesamiento se difunden a través de una matriz de elementos de procesamiento independientes, o micromotores, de modo que cada uno de ellos ejecute hasta ocho subprocesos cooperativos. Como resultado, los *network processors* Intel® tienen la capacidad de proveer el desempeño robusto necesario para admitir servicios de red de valor agregado a velocidades de línea, a la vez que brindan la flexibilidad de diseño que satisface los requisitos de aplicación de los productos individuales.

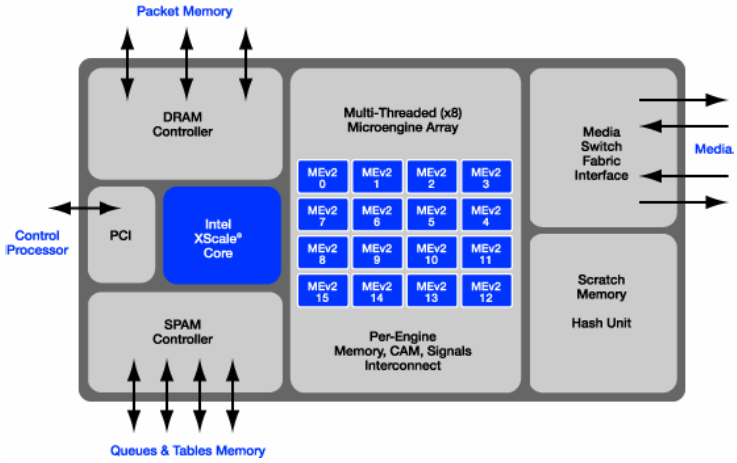


Figura 2.1- Diagrama de bloques del *network processor* Intel® IXP2800

¿Es decir que ahora los desarrolladores pueden tomar ventaja de esta multitud de elementos de procesamiento con facilidad? Intel responde con un paquete de herramientas de desarrollo de software complementarias, implementadas dentro de un marco de programación modular de fácil uso, la Estructura de portabilidad Intel® IXA. El componente clave de la arquitectura Intel IXA facilita la portabilidad y la reutilización del código entre varios proyectos basados en *network processors*. También ofrece una ruta directa para el aprovechamiento de capacidades de futuras generaciones de *network processors* Intel. Al maximizar la eficacia de los desarrolladores en este entorno flexible, Intel permite que los TEM aceleren el ciclo de diseño y ofrezcan productos nuevos al mercado con rapidez, a la vez que protegen las inversiones en desarrollo existentes.

2.5.3 SIMPLIFICACIÓN DEL DESARROLLO Y LA REUTILIZACIÓN DEL CÓDIGO

El paquete de herramientas avanzadas de Intel satisface necesidades importantes en cada fase del proceso de desarrollo. Por ejemplo, Intel elimina la tarea normalmente lenta de estimar el desempeño y la utilización de recursos en hardware objetivo con una herramienta de arquitectura que permite que los arquitectos de sistemas determinen las características de desempeño de una aplicación particular antes de que se complete el desarrollo del software. Además, para optimizar los prototipos, los perfiles y la depuración de aplicaciones, Intel ha introducido mejoras de simulación de paquetes avanzadas en la mesa de trabajo de los desarrolladores, entre las que se incluyen un generador de tráfico que puede personalizarse y un creador de perfiles gráfico centrado en paquetes.

No obstante, quizá la etapa más crucial del desarrollo sea la creación de código para micromotores y la asignación de dicho código a los recursos del procesador. Al aumentar la productividad y mejorar la reutilización del código a través de este proceso, los programadores pueden maximizar las ventajas del desempeño y la flexibilidad de los *network processors* Intel, a la vez que reducen considerablemente el tiempo general del proceso de desarrollo. La solución que Intel ofrece es un modelo de programación con particiones automáticas, el cual automatiza el análisis a nivel bajo y la mecánica de los procesos, para permitir que los programadores se enfoquen en las capacidades de valor agregado.

2.5.3.1 AUTOMATIZACIÓN DEL PROCESO DE PROGRAMACIÓN

El modelo de programación con particiones automáticas, el cual Intel planea incorporar en la siguiente generación de compiladores C dentro del Kit de desarrollo de software (SDK) Intel® IXA, permite que los desarrolladores aprovechen al máximo el paralelismo de los *network processors* Intel, a la vez que mantienen un método familiar y secuencial en el procesamiento de aplicaciones. El modelo de programación con particiones automáticas permite que una aplicación de *network processor* completa se exprese como un conjunto de etapas de procesamiento de paquetes (PPS), las cuales comunican procesos secuenciales que se ejecutan de forma simultánea.

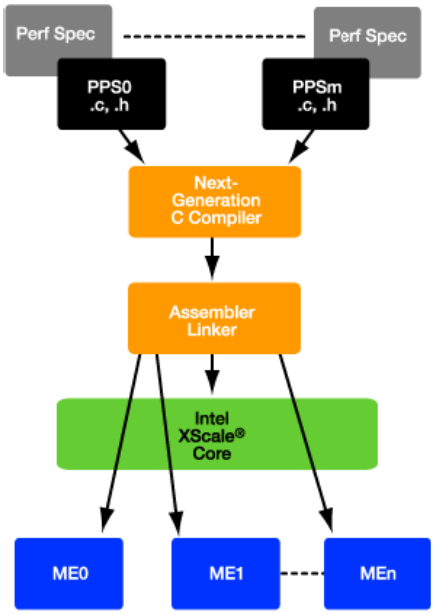


Figure 2. Auto-partitioning programming model

Figura 2.2– Modelo de programación de auto partición

Con la familiar sintaxis de las funciones C, los desarrolladores simplemente definen un conjunto de PPS de interacción en las cuales cada fase de procesamiento sigue lo parámetros lógicos de un diseño de aplicación, independientemente de cualquier limitación en los micromotores individuales. Como consecuencia, la siguiente generación de compiladores C presentará las PPS a través de una cantidad de subprocesos de micromotor y gestionará todos los requisitos de sincronización. Los programadores pueden gestionar de forma explícita la comunicación entre las PPS, a través de "conductos" o canales de comunicación; los cuales son utilizadas automáticamente por el compilador para permitir la comunicación de datos entre las PPS. Cuando una sola PPS se fragmenta a través de varios micromotores, el compilador se hace responsable de la comunicación entre las PPS.

El modelo de programación con particiones automáticas garantiza que se satisfagan los requisitos de desempeño mediante la introducción de conceptos de desempeño directamente en el proceso de compilación. Por ejemplo, cada PPS tiene una especificación de desempeño asociada, la cual define el presupuesto de conteo de ciclos para las rutas críticas. En base a dichas especificaciones de desempeño, el compilador evalúa la cantidad de micromotores

necesarios para satisfacer el objetivo de desempeño y elige automáticamente la partición adecuada.

De forma más notable, las funciones de multiprocesamiento, de subproceso múltiples y de E/S asíncrona de los *network processors* Intel permanecen completamente invisibles para el programador, a la vez que el compilador aprovecha estas capacidades de ejecución simultánea para generar programas de alto desempeño.

2.5.3.2 AMPLIACIÓN DEL VALOR DE LAS INVERSIONES EN EL DESARROLLO

La protección de la inversión en el código existente y la reutilización del código a través de varios productos es un factor crucial en la entrega de un valor total para los TEM. Con el modelo de programación con particiones automáticas, puede volver a emplearse el código existente con el compilador C de la siguiente generación, para permitir que coexista el código existente con el nuevo.

Además, el código de aplicación desarrollado con el modelo de programación con particiones automáticas puede volver a emplearse con facilidad a través de varios *network processors* Intel, los cuales admiten una amplia variedad de productos. Al abstraer las diferencias en la cantidad de micromotores y otras unidades funcionales entre los *network processors* Intel, el modelo de programación con particiones automáticas permite que los desarrolladores se enfoquen en la creación de una base de código reutilizable que contenga funciones de red comunes e incorporen dicho código de forma transparente en varias aplicaciones que se ejecuten en los distintos *network processors* Intel, todo ello sin necesidad de reoptimizar ni reconfigurar.

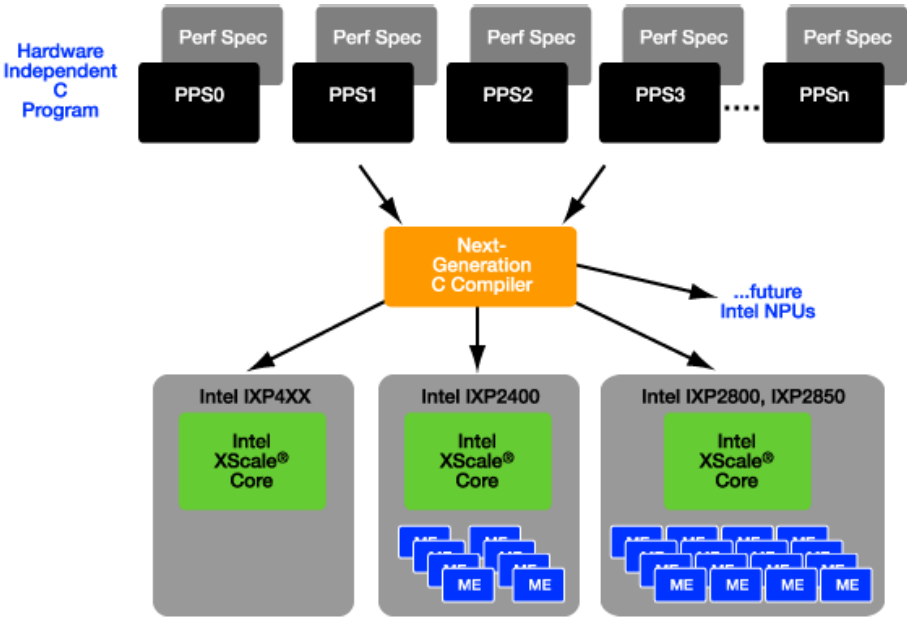


Figure 3. Cross network processor development model

Figura 2.3– Modelo de desarrollo cruzado del *network processor*

2.5.4 CONCLUSIONES

Con la arquitectura de alto desempeño y totalmente programable y las herramientas avanzadas que aceleran el desarrollo de aplicaciones en dicha arquitectura, Intel está cumpliendo la promesa de brindar *network processors* que ofrecen velocidad, flexibilidad y facilidad de uso.

Como resultado, los fabricantes de equipos de telecomunicaciones obtienen una mayor ventaja comercial cuando utilizan los *network processors* Intel: la capacidad para desarrollar productos de red que se distinguen de la competencia y para llevar dichos productos al mercado de forma rápida y rentable. Como parte central de esta ventaja está el modelo de programación con particiones automáticas que Intel planea incorporar en la siguiente generación de compiladores C. Con esta herramienta única, los desarrolladores pueden aprovechar al máximo el desempeño y la flexibilidad inherentes en los *network processors* Intel con un esfuerzo de programación mínimo.

Este método sofisticado de programación facilita y acelera de forma dramática el desarrollo de aplicaciones basadas en *network processors*, para aislar a los programadores de la administración de hardware de bajo nivel y permitirles que se enfoquen en el desarrollo de capacidades de valor agregado. Es más, el modelo de programación con particiones automáticas facilita la reutilización del código para ampliar el valor de las inversiones en el desarrollo de varios productos.

En conclusión, el alto desempeño, la capacidad de programación y la facilidad de uso de los *network processors* Intel brindan a los fabricantes de equipos de telecomunicaciones una poderosa ventaja competitiva y una mayor rentabilidad de la inversión a largo plazo.

CAPÍTULO 3
DEVELOPER WORKBENCH

3.1 INTRODUCCIÓN

El Developer Workbench es un entorno de desarrollo integrado para ensamblar, compilar, enlazar y depurar microcódigo que se está ejecutando en los micromotores de los *Network Processors* IXP2400 y IXP2800. Las características más importantes del Workbench incluyen:

- Depuración del nivel de fuente.
- Modo de creación de proyecto para sólo depurar.
- Ejecución de historia.
- Estadísticas.
- Dispositivo de bus media y simulación del tráfico de red para *Network Processors*.
- Interfaz de línea de comandos para los simuladores de *Network Processors* (Transactores).
- Componentes de GUI customizables.

El Workbench soporta depuración en cuatro configuraciones diferentes:

- **Simulación local sin modelo externo:** el Workbench y el Transactor se ejecutan en la misma plataforma de Microsoft Windows.
- **Simulación local con modelos locales externos:** el Workbench, el Transactor y uno o más librerías de enlace dinámico de modelo externo se ejecutan en la misma plataforma de Windows.
- **Simulación local con un modelo remoto externo:** el Workbench y el Transactor se ejecutan en la misma plataforma Windows y se comunican sobre la red con un modelo externo ejecutándose en un sistema remoto.
- **Hardware:** el Workbench se ejecuta en un host Windows y se comunica sobre una red o un puerto serie con un subsistema contando los *Network Processors* actuales.

Se puede obtener ayuda sobre el Workbench y los *Network Processors*.

La Interfaz Gráfica de Usuario del Workbench cumple la vista estándar de Windows. Se puede hacer lo siguiente:

- Acoplar y desacoplar ventanas, barras de menús, y barras de herramientas.

- Ocultar y mostrar ventanas y barras de herramientas.
- Customizar barras de herramientas y barras de menús.
- Salvar y restaurar personalizaciones GUI.

Los proyectos pueden crearse de dos maneras diferentes:

- **Modelo Estándar:** contiene uno o más *Network Processors*, archivos fuente de microcódigo, archivos script de depurado, un ensamblador, un compilador, y unos ajustes de enlazador utilizados para construir los archivos imagen del microcódigo. Esta información de configuración del proyecto está guardada en un archivo de proyecto del Developer Workbench (.dwp).
- **Modelo de depuración:** el usuario especifica un archivo uof construido externamente para cada chip en el proyecto. El usuario no tiene que especificar los archivos fuente de ensamblador y compilador, ni el control de ajustes construidos, o llevar a cabo archivos uof construidos utilizando el GUI de Workbench.

Al arrancar el Workbench, se puede: crear un nuevo proyecto, abrir un proyecto existente, salvar un proyecto, cerrar un proyecto, y especificar una ruta por defecto para crear y abrir proyectos.

El tipo de procesador, que se selecciona cuando se crea un nuevo proyecto, determina qué Transactor es el que se utiliza para la simulación. El Workbench mostrará sólo los componenetes GUI que son relevantes para el tipo de procesador seleccionado. La familia de procesadores no puede cambiarse una vez que se haya creado el proyecto. Los tipos de procesador soportados son: IXP2800 A1, IXP2800 A2, IXP2800 B0, IXP2400 B0.

El Workbench permite: crear archivos, abrir archivos, cerrar archivos, guardar archivos, guardar todos los archivos al mismo tiempo, imprimir archivos, insertar archivos en un proyecto, borrar archivos desde un proyecto, editar un archivo, etiquetas de error y de marcas de libro.

En este capítulo se explicarán algunas de las funciones que soporta el Developer Workbench que son útiles para manejar cualquier aplicación. En los apartados 3.5, 3.6 y 3.8 se explicarán la configuración del entorno de desarrollo Workbench para nuestra aplicación en concreto, a parte de detallar qué es lo que hace cada función del programa.

3.2 EL ENSAMBLADOR

Invocar el ensamblador es un proceso que se lleva a cabo en dos pasos:

- **Preprocesador:** toma un archivo .uc y crea un archivo .ucp para el ensamblador.

- El **ensamblador** toma un archivo .ucp y crea un archivo intermedio con el archivo de nombre de extensión .uci. El archivo .uci es utilizado por el ensamblador para crear un archivo .list y proporciona información de errores que debe utilizarse para resolver problemas de semántica (como conflictos de registros) en el archivo de entrada.

El archivo .uc contiene tres tipos de elementos: micropalabras, directivas, y comentarios. Las micropalabras contienen un código op y argumentos y genera una micropalabra en un archivo .list. Las directivas pasan cualquier información a los preprocesadores, ensamblador, o componentes más bajos (como el enlazador) y generalmente no genera micropalabras. Los comentarios son ignorados en el proceso de ensamblaje. El ensamblador lleva a cabo las siguientes funciones en convertir el archivo .uc a un archivo .list:

- Chequea las restricciones del microcódigo.

- Resuelve nombres de registro simbólico para localizaciones físicas.

- Lleva a cabo las optimizaciones.

- Resuelve etiquetas de direcciones.

- Traduce códigos op simbólicos en modelos de bits.

3.3 EL MICROMOTOR DEL COMPILADOR C

El Workbench contiene un compilador C para compilar código fuente C en microcódigo para los micromotores. EEl micromotor del compilador C es un compilador de propósito general pero el lenguaje C utilizado por los micromotores es limitado. El compilador C puede compilar un archivo fuente (.c, .h) o un archivo objeto (.obj).

Podremos:

- Añadir archivos fuente en C al proyecto.
- Seleccionar el archivo .list objetivo.
- Selecciones el archivo .obj objetivo para compilar.
- Borrar un archivo .list objetivo.
- Seleccionar archivos fuente C para compilar.
- Seleccionar archivos objeto C para compilar.
- Borrar archivos fuente C del proyecto.
- Seleccionar parámetros de compilación.

El compilador lleva a cabo las siguientes funciones para convertir el archivo .c a un archivo .list:

- Acepta el estándar V con __declspec() para especificar segmentos de memoria y propiedades y registro de uso {*signal xfer nearest-neighbor remote*}.
- Acepta ensamblajes restrictivos via __*asm*{ }.
- Optimiza el programa en “modo de programa completo” donde cada función es analizada.

Genera el archivo .list para ejecutarlo en un único micromotor.

3.4 LINKER

El *Linker* es utilizado para enlazar imágenes microcódigo. Las imágenes microcódigo son generadas por el compilador de microcódigo o por el ensamblador. Los objetos de la aplicación son generadas por el procesador Intel XScale compilador C/C++. El método del compilador C/C++ es independiente.

La memoria es compartida entre el procesador Intel XScale y los micromotores. Una manera común de diseño sería que el procesador Intel XScale generara y mantuviera las estructuras de los datos, mientras el micromotor lee los datos. Los punteros a direcciones comunes serían utilizados para estas estructuras de datos.

Varias configuraciones de estructuras de datos de los micromotores existirían en el procesador Intel XScale. Éstas son utilizadas para mantener información sobre los micromotores.

Para actualizar la dirección compartida:

- Desde el ensamblador, obtiene una lista de “externos”. Obtiene las localizaciones microPC donde estas variables son utilizadas como inmediatas. Obtiene el formato micropalabra para saber el tamaño del “inmediato”. Formatea ésto en el microcódigo del objeto enlazado.

- En la aplicación del procesador Intel XScale, carga o mapea el microcódigo del objeto enlazado

- En la aplicación del procesador Intel XScale, añade la variable externa con un valor llamando a la función apropiada en la librería de carga.

- La librería de carga descarga los “inmediatos” con el valor añadido para todos los acontecimientos de la variable en la microimagen.

- En la aplicación del procesador Intel XScale, carga las imágenes para los micromotores apropiados vía función de librería cargada.

3.5 CONFIGURACIÓN DEL ENTORNO DE SIMULACIÓN DEL IXP2400

Para configurar el entorno de simulación para el proyecto IXP2400, ir a ***Simulation*** -> ***System Configuration***. Podremos establecer o cambiar los valores de configuración dependiendo de la familia chip seleccionado: Frecuencias de reloj, Memoria, Dispositivos MSF, Conexiones de red, Conexiones CBUS.

El contenido de cada diálogo depende del tipo de procesador definido para el proyecto. Esta configuración de datos se pasa al Transactor y a los modelos de dispositivos a través de la interfaz de línea de comando cuando se comienza a depurar.

3.5.1 FRECUENCIAS DE RELOJ

Dependiendo del tipo de procesador definido para el proyecto, la página de propiedades de Frecuencias de Reloj mostrará diferentes valores por defecto. Los valores que se especifiquen en esta página pasarán al Transactor utilizando la función *set_clocks()*.

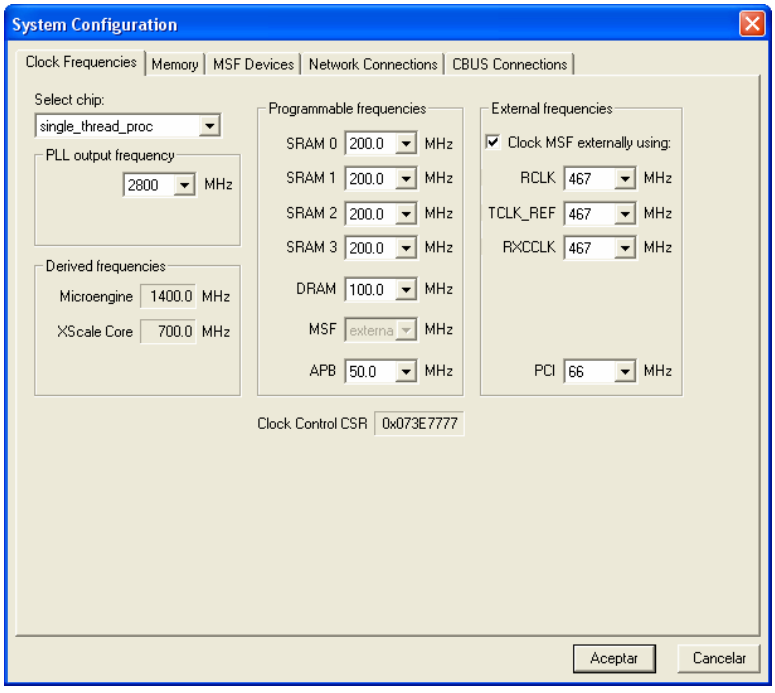


Figura 3.1– Configuración de las Frecuencias de Reloj

3.5.2 MEMORIA

La etiqueta ***Memory*** dentro del cuadro de diálogo de ***System Configuration*** soporta la configuración del simulador de memoria. Hay algunas variaciones en la pantalla dependiendo de cual *Network Processor* se está configurando.

- El canal SRAM no puede exceder de 64 MB, de manera que la opción ***Part count*** de 2 está deshabilitada si el ***Part size*** es 64.

- El simulador debe tener un canal SRAM poblado. La memoria cero no puede configurarse, por eso no hay ninguna opción para un ***Part size cero*** o un ***Part size coun to cero***.

Hay cuatro canales disponibles para configurar la memoria SRAM:

- El valor ***Chan size*** del DRAM multiplicado por el valor de ***Chan count*** produce la memoria DRAM total disponible en el simulador. Los canales DRAM posibles están entre 1 y 3. El tamaño DRAM está limitado a 2048 MB, así que las opciones del valor del contador del canal pasa a estar limitado en tamaños de canal más altos.

- Cada uno de los cuatro canales SRAM deben configurarse independientemente.

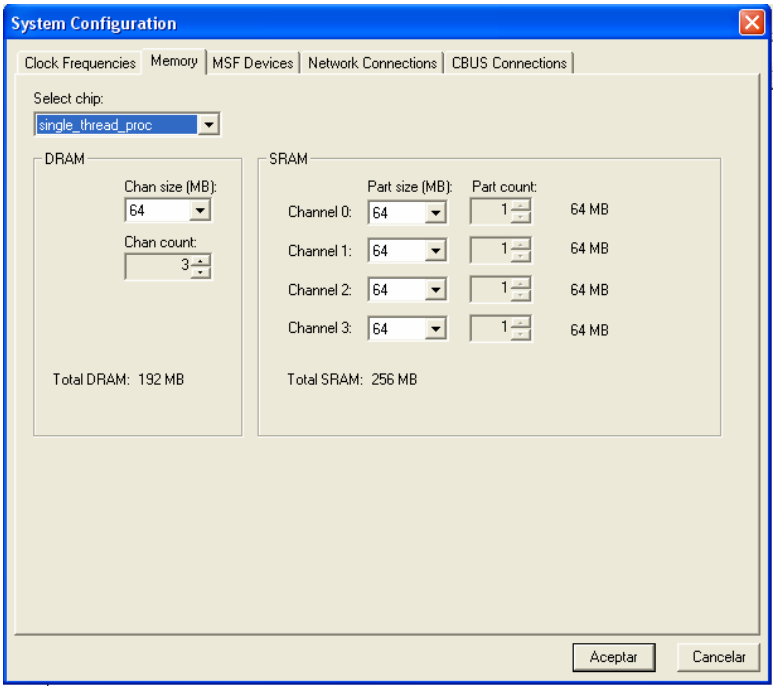


Figura 3.2– Configuración de la Memoria

3.5.3 CONFIGURACIÓN DEL DISPOSITIVO MSF

Esta opción soporta la configuración de las interfaces medias y del *switch fabric*.

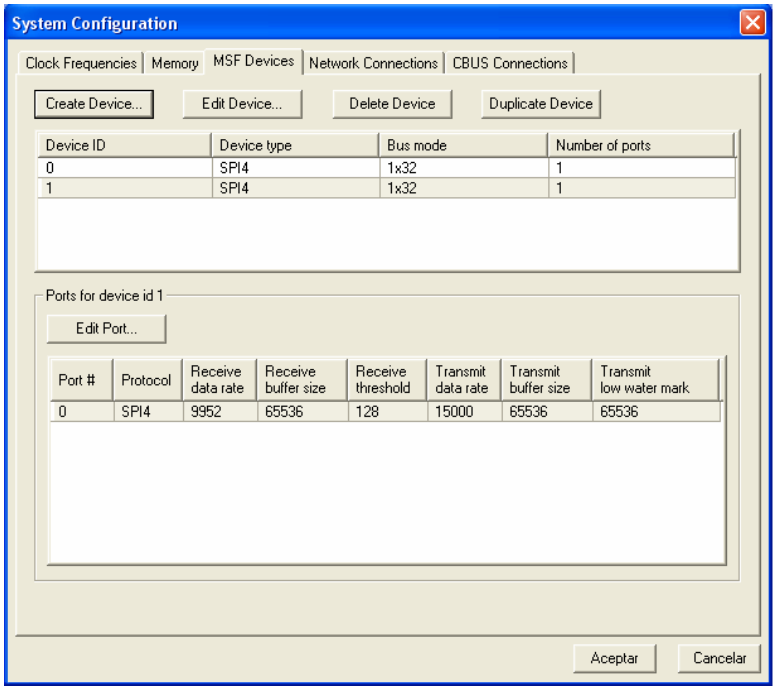


Figura 3.3– Configuración de los dispositivos MSF

Para crear un dispositivo haremos clic sobre ***Create Device…***. Aparecerá el cuadro de diálogo ***Create Media Bus Device***. Aquí es donde escogeremos el tipo de dispositivo en ***Select device type…***. Los tipos soportados para el IXP2800 son SPI4 y CSIX.

- **CSIX:** especifica el interfaz de interconexión hardware entre la *switching fabric* y las unidades de procesado superiores.

- **SPI4:** es una interfaz para paquetes y celdas de transferencia entre un dispositivo PHY y un dispositivo de capa de enlace, para añadir anchos de banda de OC-192 ATM y Paquetes sobre SONET/SDH (POS), y aplicaciones Ethernet de 10 Gb/s.

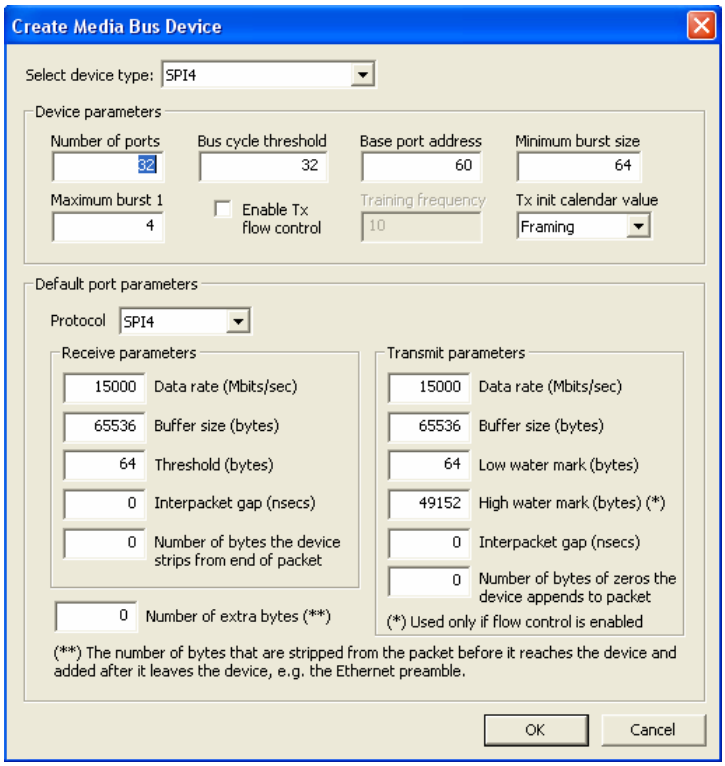


Figura 3.4– Cuadro de diálogo Create Media Bus Device para SPI4

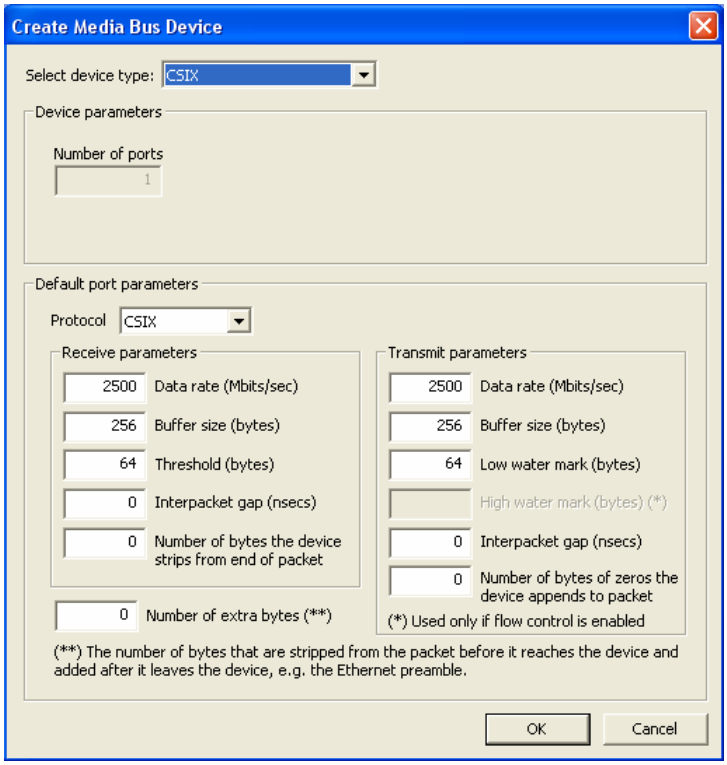


Figura 3.5– Cuadro de diálogo Create Media Bus Device para CSIX

El tipo de dispositivo seleccionado será SPI4 y será configurado como muestra la figura.

- ***Bus cycle threshold*:** este valor determina el # máximo por defecto de ciclos MSF por ráfaga.

- ***Base port address*:** SPI4 soporta 8 bits de dirección, así que para un dispositivo con 10 puertos, la dirección base del puerto sería entre 0 y 245. El dispositivo SPI4 asigna direcciones consecutivas a los puertos del mismo dispositivo.

- ***Minium burst size*:** determina el número de bytes mínimo que el dispositivo SPI4 transferirá en una transacción, a no ser que se alcance EOP.

- ***Maximum burst 1*:** número máximo de bloques de 16 bytes que la cola de transmisión FIFO puede aceptar cuando el canal de estado FIFO indica una condición "*Starvating*".

- ***Enable Tx flow control*:** permite al flujo de control de transmisión apuntar entre el dispositivo SPI4 y el *network processor*.

- ***Training frequency*:** indica el intervalo máximo entre *scheduling* del control de flujo apuntando secuencias.

- ***Tx Inicial Calendar Value***

- ***Data rate*:** especifica el radio en el que los datos son llevados desde la red e insertados en el buffer receptor del puerto y el radio en que los datos son llevados desde el buffer transmisor del puerto y puestos en la red.

- ***Receive buffer size*:** especifica el número de bytes en el buffer de recepción. El buffer de recepción lleva el dato recibido desde la red hasta el *Network Processor* que lo lee desde el puerto.

- ***Receive threshold*:** especifica el número de bytes que deben haber en el buffer receptor del puerto en regla para el puerto para señalar el *Network Processor* que puede seleccionar el puerto y el dato de petición desde él.

- ***Transmit buffer size*:** especifica el número de bytes en el buffer de transmisión. El buffer de transmisión lleva el dato transmitido por el *Network Processor* hasta que sea transmitido en la red.

- ***Low water mark y High water mark*:** determina si el dispositivo está "Hambriento" o "Satisfecho". Estará "Hambriento" si el número de bytes en el buffer de transmisión está entre *low and high water marks*. Si no es así, estará "Satisfecho".

- ***Interpacket gap*:** especifica la cantidad de tiempo entre paquetes cuando recibe paquetes desde la red y transmite paquetes a la red.

- ***Number of bytes the device strips from end of packet*:** especifica el número de bytes que el dispositivo debe deshacer desde el final de cada paquete recibido antes de que el paquete sea pasado al *Network Processor*.

- ***Number of bytes of zeros the device appends to packet*:** especifica el número de bytes de ceros que el dispositivo añade al paquete antes de que sea transmitido por el *Network Processor*.

- ***Number of extra bytes*:** especifica el número de bytes que son deshechos desde el principio del paquete antes de que alcance el dispositivo y añada al principio del paquete después de que deje el dispositivo.

3.5.4 CONEXIONES DE RED

Después de haber configurado la simulación de paquetes con dispositivos y puertos y haber creado o importado flujos de datos, hay que especificar las conexiones para el *switch fabric/media* para cada chip en el proyecto.

Hacemos clic en la pestaña ***Network Connections***, y configuraremos el cuadro de diálogo de la siguiente forma:

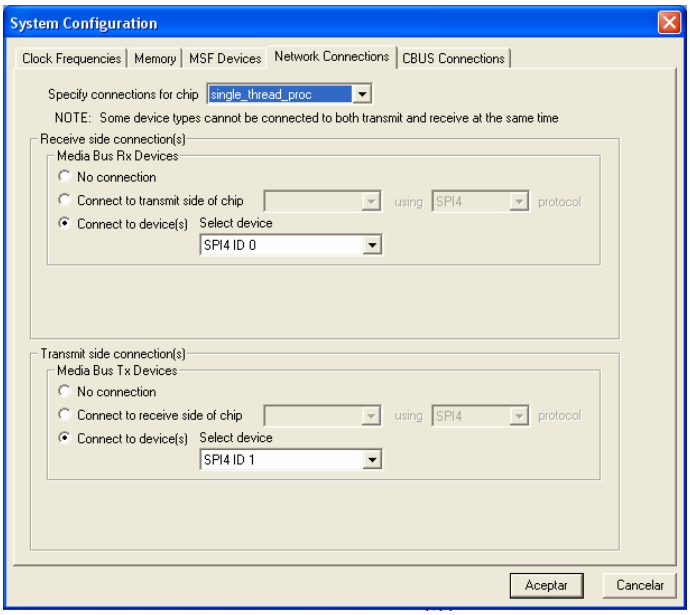


Figura 3.6- Configuración de las Conexiones de Red

- ***No connection*:** la simulación se ejecuta sin nada conectado a ese lado del MSF.

- ***Connect to transmit side of chip*:** en multiples chips de proyectos, el lado del receptor puede conectarse al lado del transmisor de cualquier chip en el proyecto seleccionando esta opción. Se deberá seleccionar qué chip y qué protocolo se usará para la conexión. Para IXP2800, el único protocolo soportado es SPI4.

- ***Connect to device(s)*:** el lado del transmisor puede conectarse al lado del receptor de cualquier chip del proyecto seleccionando esta opción. Esta opción será la que seleccionaremos.

3.5.5 CONEXIONES CBUS

Por último, hay que especificar las conexiones al *fabric switch/media* para cada chip en el proyecto.

Seleccionamos la etiqueta ***CBUS Connections***, y configuraremos de la siguiente forma:

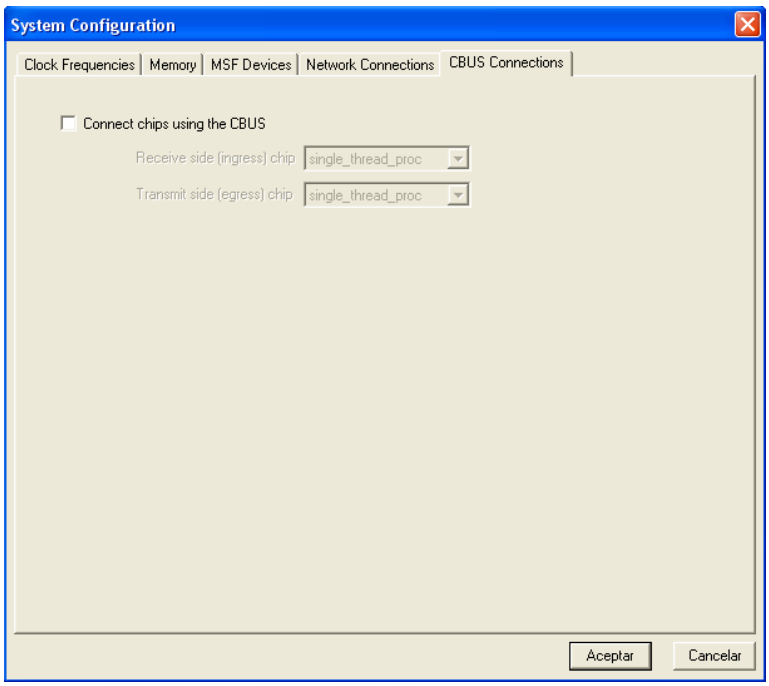


Figura 3.7- Configuración de las Conexiones CBUS

3.6 SIMULACIÓN DE PAQUETES

El Workbench proporciona simulación de paquetes de los dispositivos de bus Media también como simulación del tráfico de red. Para simular dispositivos y tráfico de red hay que:

- Configurar los dispositivos en el bus media utilizando el menú ***System Configuration***. Esto implica especificar cuántos dispositivos están en el bus así como las características de cada dispositivo.

- Definir el tráfico de red, que puede consistir de tramas Ethernet o celdas ATM.

- Asignar tráfico de red para cada puerto del dispositivo por donde se quiera recibir paquetes.

- Especificar el registro y parar las opciones de control de parada bajo que se quiera la simulación de tráfico para operar utilizando el menú ***Packet Simulation Options....***

Mientras se está ejecutando la simulación, se pueden observar las conexiones que se han asignado a los dispositivos que se han creado.

Los paquetes no se recibirán hasta que se ejecuta la función *ps_start_packet_receive();* en la línea de comandos o se añada en el script *startup* en el punto donde se quiere que comience la recepción de paquetes.

3.6.1 OPCIONES GENERALES

- ***Run unbounded (infinite wire speed)*:** permite tener datos siempre listos para ser recibidos por el *Network Processor* y tener los puertos siempre listos para recibir datos desde el *Network Processor*. Si esta opción no está marcada, los datos son recibidos y transmitidos desde y hacia la red en el radio de datos especificado con un intervalo de interpaquete.

- ***Select units for transmit/receive rates*:** muestra el radio de datos recibidos y transmitidos.

- ***Update status window every xxx microengine cycles*:** actualiza el estado en el intervalo especificado mientras la simulación se está ejecutando.

- ***Use this seed for random selection …Previous value xxxxxxxx*:** fuerza el generador de selección aleatoria para utilizar el número usado en la última simulación, o escribir un nuevo número.

3.6.2 REGISTRO DE LA INTERFAZ DE TRÁFICO

Los paquetes que están completos son los únicos que se registran. Si se permite el registro durante la simulación, el registro de un puerto comenzará cuando ocurra el siguiente SOP.

Si no se permite el registro, cualquier paquete que se encuentre en el proceso de ser recibido o transmitido no se registrará. La simulación de paquetes registra archivos que son cerrados automáticamente cuando la simulación termina. El archivo de registro se abre y es añadido cuando la simulación se reanuda. Existen archivos de registro que se borran automáticamente cuando el primer paquete es registrado después de que haya comenzado el proceso de depurado.

- ***Enable Logging*:** cambia de estado si el registro del paquete ocurre o no.

- ***Log frame numbers*:** cambia de estado si los números de trama son registrados o no junto con los datos del paquete. La numeración de trama comienza cuando comienza la depuración, con el primer paquete recibido en un puerto y el primer paquete transmitido a un puerto siendo el número 1.

- ***Log media bus cycle times for SOP and EOP*:** cambia de estado si registra o no los tiempos de ciclo en el que el primer byte de un paquete es recibido o transmitido.

- ***Traffic interface logging options*:**

    - ***Log the packets received by the chip thru this traffic interface to file*:** permite obtener un registro de los paquetes recibidos por el *Network Processor* a un puerto. Para ello seleccionamos un puerto, marcamos esta opción y escribimos la ruta de archivo.

    - ***Log the packets transmitted by the chip thru this traffic interface to file*:** permite obtener un registro de los paquetes transmitidos por el *Network Processor* a un puerto. Para ello seleccionamos un puerto, marcamos esta opción y escribimos la ruta de archivo.

3.6.3 CONTROL DE PARADA

Nos permite especificar si y cuando se quiere que la simulación o la recepción de paquetes finalice.

- ***General options*:**

    - ***Stop simulation if a receive overflow occurs*:** controla si el Workbench para o no la simulación cuando ocurre la recepción de un desbordamiento.

    - ***Stop simulation if a transmit overflow occurs*:** controla si el Workbench para o no la simulación cuando ocurre la transmisión de un *underflow*.

- ***Device options*:**

    - ***Stop the simulation alter the chip receives the next nnn packets from this device*:** para la simulación después de que el *Network Processor* reciba un número de paquetes específico.

    - ***Stop the simulation alter the chip transmit the next nnn packets to the device*:** para la simulación después de que el *Network Processor* transmita un número de paquetes específico.

- ***Traffic interface*:**

- ***Send packets to the chip thru this traffic interface*:** permite al puerto recibir paquetes desde la red.

- ***After the chip receives the next nnn packets thru this traffic interface to the chip or stop the simulation*:** permite tomar acción después de recibir un número específico de paquetes por el *Network Processor* desde el puerto.

  - ***Stop sending packets thru this traffic interface to the chip*.**

  - ***Stop simulation*.**

- ***After the chip transmits the next nnn packets thru this traffic interface to the chip, stop the simulation*:** permite tomar acción después de que sean transmitidos un número de paquetes desde el *Network Processor* al puerto.

- ***Stop the simulation if a packet validation error occurs for this traffic inter*face:** permite parar la simulación si un error de validación de paquetes es detectado por el generador de paquetes.

3.6.4 ASIGNACIÓN DE TRÁFICO

Para simular las operaciones en los puertos, también hay que simular el tráfico de red. El componente que proporciona este tráfico es el llamado NTS (Simulador de Tráfico de Red). El Workbench soporta dos NTSs; PacketGen y DataStream. También se puede crear un NTS.

Al conectar un dispositivo en el lado de recepción del chip, se puede especificar el tipo de datos que se va a recibir en cada puerto en ese dispositivo. Cada puerto está conectado al *network processor* por un lado y por el otro lado está conectado a la red. Los datos llegan al puerto desde la red y son trasladados al buffer del puerto de recepción.

En la lista de puertos de recepción, el Workbench muestra la entrada que ha sido asignada a cada puerto configurado para el dispositivo conectado. Muestra el nombre del plug-in del NTS.

Cuando se conecta un dispositivo al lado de transmisión del chip, se puede especificar que se tiene que hacer con el dato enviado a la red por cada puerto en ese dispositivo. La salida a la red puede tirarse o puede dirigirse hacia el PackGen que incorpora el NTS o un plug-in DLL para NTS que proporciona el usuario. En la lista de puertos que transmiten, el Workbench muestra el nombre de los plug-in NTS, el DLL que se asignó al receptor de salida desde el puerto.

3.6.5 PLUG-INS NTS

Sirven para definir, generar y controlar el tráfico de red.

- ***Select Network Traffic Simulator Plug-in*:** lista todos los plug-ins del NTS.

- ***Description*:** información acerca del plug-in seleccionado.

- ***Run time DLL***

- ***Configuration GUI DLL***
- ***Run time GUI DLL***

3.7 PACKETGEN NTS

3.7.1 DEFINIR TRÁFICO DE RED UTILIZANDO PACKETGEN

Para crear configuraciones de tráfico, haremos clic en **Simulation** -> **Define Network Traffic** -> **PacketGen…**.

3.7.1.1 PMD FILE MANAGEMENT

Define el conjunto de archivos PMD que contiene las definiciones de protocolo necesarias que serán usadas para configurar pilas de protocolos durante la creación de un flujo.

1. Clic en el botón ***Import*** para importar el archivo .pmd.

2. La ventana ***Protocols*** mostrará los protocolos contenidos dentro del archivo .pmd importado.

3.7.1.2 FLOW SPECIFICATION

Sirve para definir la pila de protocolos, el algoritmo de radio de flujo, carga explosiva de datos, y otros atributos asociados con el flujo del tráfico de red.

1. Clic en ***New…*** para crear una especificación de flujo nuevo. La llamaremos **Test.flw**.

2. Una vez que se ha creado un nuevo flujo, pueden especificarse o cambiarse los distintos atributos de tráfico de red. La dirección del flujo se puede especificar como ***Input Flow*** (circulación de paquetes en el Transactor), ***Output Flow*** (paquetes que salen del Transactor) y ***Bidirectional Flow***.

3. ***Protocol Snack Top Layer Data Payload*** muestra el estado de la pila de protocolos con el flujo. Define la carga explosiva de datos asociados con la cumbre de la capa en la pila de protocolos.

    a. ***Hierarchical Payload*****:** permite al usuario seleccionar una carga explosiva jerárquica (es una configuración de tráfico que generará paquetes que son después pasados a este flujo como una carga explosiva durante la generación de paquetes).

    b. ***Simple Payload*****:** permite al usuario configurar una sencilla carga explosiva.

    c. ***Configure Payload*****: *Capture Validation*** permite al usuario escoger para validar el tamaño de la carga explosiva y/o carga explosiva de datos ara PDUs capturadas de este flujo.

4. Haciendo clic sobre el botón ***Edit Snack*** podemos editar las pilas de protocolos soportadas. Aparecerá la ventana ***Edit Protocol Snack***.

    a. Seleccionamos los protocolos de la lista ***Available Protocols*** y con el botón ***Add*** los añadimos a la lista ***Current Snack***.

5. Podemos cambiar los atributos de los campos específicos de los protocolos seleccionados haciendo doble clic en cualquier del ***Field Name*** en el área ***Protocol Fields Definitions***.

    a. En el cuadro de diálogo ***Protocol Field Settings***, podemos especificar una clave de campo a un conjunto de campos en la pila de protocolos, marcando la opción ***Key Field for PDU Identification***. Esta opción se utiliza para identificar qué campos en una capa de protocolos en un flujo debería de usarse para encajar con paquetes capturados en contra de la capa de protocolos.

    b. Con ***Validate Field during Capture***, se pueden validar un conjunto de campos en la pila de protocolos.

6. Para configurar las propiedades de la capa de protocolo, hacemos clic sobre el botón ***Configure Layer***. Se abre el cuadro de diálogo ***Protocol Layer Settings***, que muestra los atributos de la capa de pila de protocolos incluyendo el nombre de la capa de protocolo y una lista de alias para la capa de protocolo. También se puede permitir el registro para capa individual. Este cuadro de diálogo contiene configuraciones utilizadas por el generador y el capturador de paquetes para esta capa de protocolo. Se puede activar ***Enable logging***. Si esta opción está marcada y además el flujo contiene el registro de la capa de pila permitida, los PDUs se escribirán en el archivo de registro de flujo.

7. ***Logging*****:** permite activar o desactivar el registro para el flujo. También se puede especificar el directorio de registro para el flujo.

3.7.1.3 CONFIGURACIONES DE TRÁFICO

Asocia un grupo de flujos juntoss en un conjunto llamado configuración de tráfico. También asocia un radio de flujo con el flujo. La configuración de tráfico puede asignarse a uno o más puertos utillizando Traffic Assignment del cuadro de diálogo Packet Simulation Options. El puerto de recepción es el puerto que recibe paquetes que van al Transactor y el puerto de transmisión es el puerto que transmite paquetes fuera del Transactor.

La configuración de tráfico puede también asignarse a uno o más flujos como Hierarchical Payload en la pestaña Traffic Configuration.

1. Clic en **New…** para crear una nueva configuración de tráfico.

2. **Flow Selection:**

    a. **Flow Selection Algorithm:** despliega varios algoritmos disponibles para especificar mecanismos de selección de flujos de paquetes para la generación de tráfico desde configuraciones de tráfico recibido.

    b. **Flow Selection Algorith Parameters:** muestra los atributos asociados con el algoritmo de selección de flujo seleccionado. Puede modificarse haciendo doble clic sobre el campo seleccionado. Los algoritmos de selección de flujo son aplicables al puerto de entrada y proporcionan un mecanismo que determina que flujo proporciona la próxima estructura para ser generada y la entrada al Transactor.

3. Los paquetes recibidos o transmitidos desde el puerto pueden registrarse en un archivo especificado.

4. **Log Directory:** Los registros de salida desde la Simulación de Paquetes Avanzada son almacenados en la carpeta específica.

5. **Direction:** la dirección de la configuración del tráfico puede ser Receive (añade un flujo específico por **Input Flow**), Transmit Receive (añade un flujo específico por **Output Flow**) o Bidirectional.

6. **Add…:** asigna un flujo a la configuración del tráfico.

3.7.2 RUN-TIME PACKET SIMULATION CONTROL

Accedemos yendo al menu **View**-> **Debug Windows**->**Network Traffic Control**.

3.7.2.1 SIMULACIÓN DE PAQUETES VIA INTERFAZ DE LÍNEA DE COMANDOS

Se puede acceder a la interfaz de línea de comandos mediante los archivos script IND o mediante la línea de comandos del Transactor.

Para asignar tráfico mediante la línea de comandos a un puerto, seleccionaremos <command line> como ilustra la figura.

3.8 UTILIZACIÓN DEL FLUJO DE DATOS NT

Los archivos llamados “data streams” son utilizados para definir el tráfico de red. Para crear y editar flujos de datos:

1. Menú **Simulation->Define Network Traffic…** Aparecerá el cuadro de diálogo **Data Stream**.

2. Clic en **Create Stream…**

Crearemos los siguientes flujos de datos (16 paquetes por cada flujo de datos):

3.8.1 FLUJO DE DATOS ETHERNET TCP/IP

Para ello tendremos que crear una trama Ethernet TCP/IP. Hacemos clic sobre **Create frame(s)**, seleccionamos **Ethernet TCP/IP** y le ponemos un nombre (**Packet Flow 1.strm**). Aparecerá el cuadro de diálogo **Ethernet TCP/IP Data Stream**, y configuraremos las cabeceras tal y como muestran las figuras.

Al hacer clic en **Advanced…** se abrirá un cuadro de diálogo donde se puede especificar cómo se quiere la dirección MAC destino y la dirección MAC fuente. Se marcará **From within range** para que estén comprendidas entre el rango que se le especifique, éste estará comprendido entre 0x000000000000 y 0xffffffffffff.

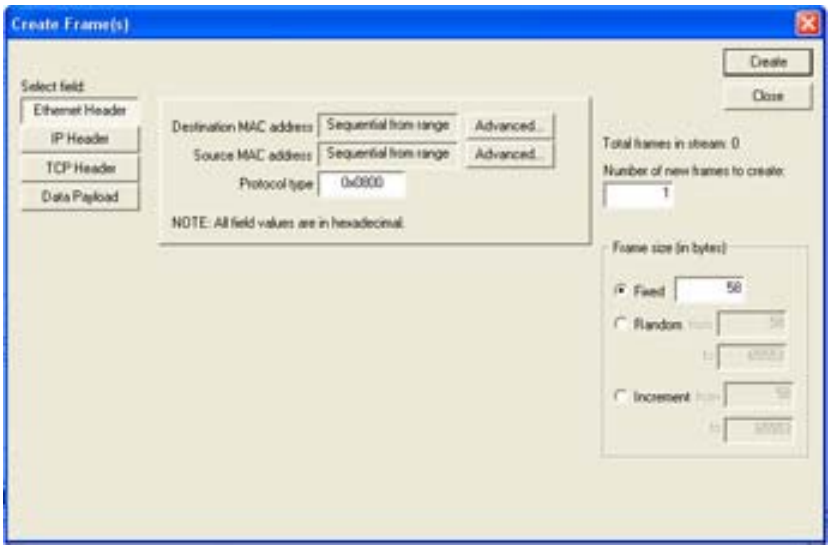


Figura 3.8– Configuración de la cabecera Ethernet para una trama Ethernet TCP/IP

Seleccionamos **Computed** en **Packet length** y en **Header checksum**, para que la longitud del paquete y el checksum de la cabecera se calculen automáticamente. Además, como se van a crear múltiples tramas y se quiere que la dirección IP origen y la dirección IP destino tengan distinto valor, haremos clic en **Advanced…** y seleccionaremos un rango de direcciones para que ambas direcciones estén comprendidas entre 0.0.0.0 y 255.255.255.255.

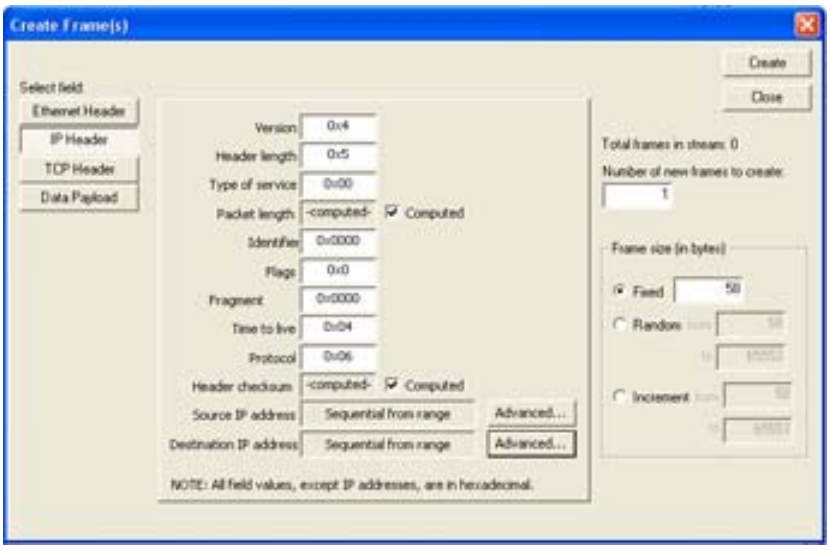


Figura 3.9– Configuración de la cabecera IP para una trama Ethernet TCP/IP

Marcamos la opción **Computed** para que el checksum sea calculado automáticamente.

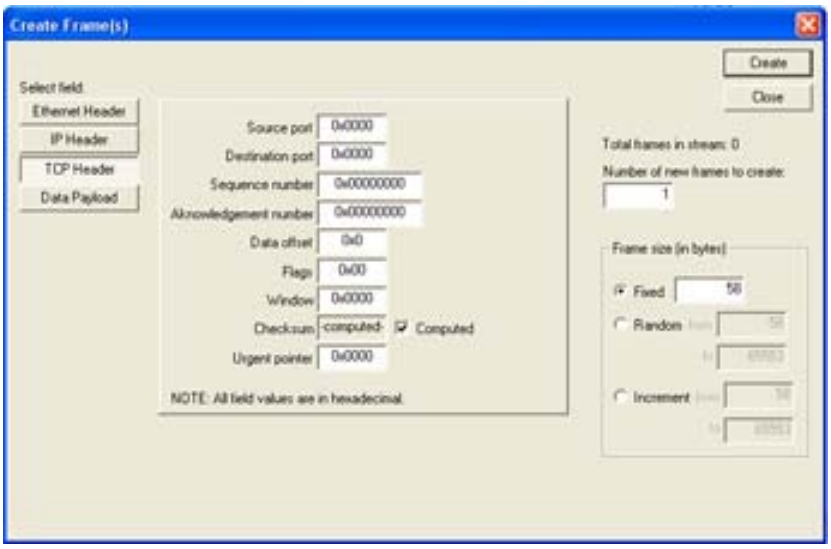


Figura 3.10– Configuración de la cabecera TCP para una trama Ethernet TCP/IP

Seleccionaremos **increment byte** en **Fill pattern**.

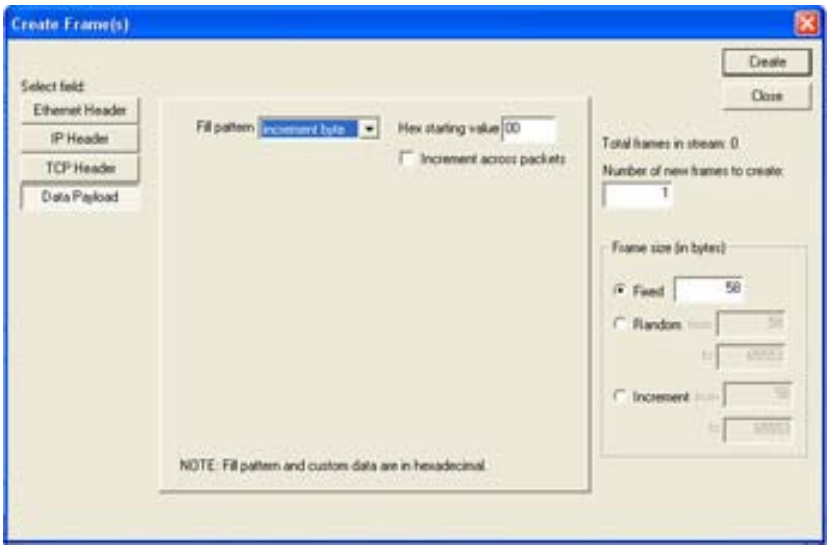


Figura 3.11– Configuración de la carga útil de datos para una trama Ethernet TCP/IP

3.8.2 FLUJO DE DATOS POS IP

Para ello tendremos que crear una trama POS IP. Le llamaremos **Packet Flow 2.strm**. Aparecerá el cuadro de diálogo **POS IP Data Stream**.

Marcamos **Include Address** e **Include Control** para incluir la dirección y el control en la cabecera PPP.

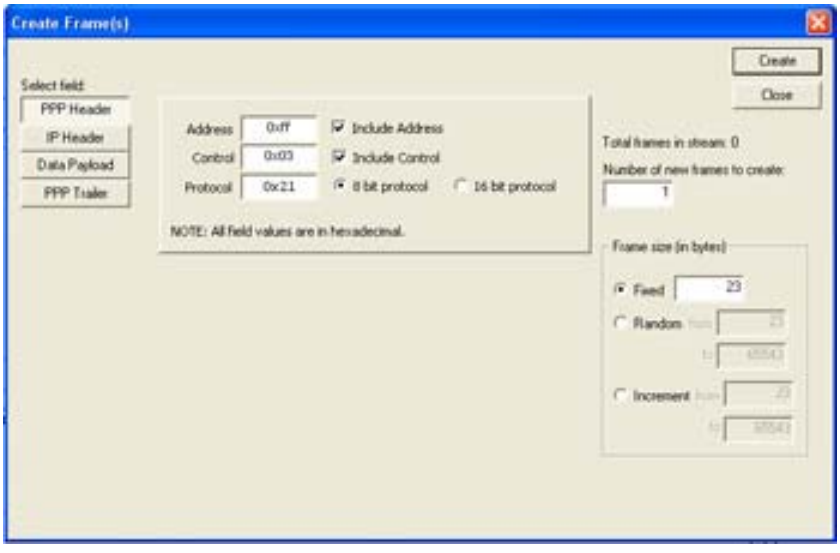


Figura 3.12– Configuración de la cabecera PPP para una trama POS IP

Seleccionamos **Computed** en **Packet length** y en **Header checksum.** Haremos clic en **Advanced…** y seleccionaremos un rango de direcciones entre 0.0.0.0 y 255.255.255.255.

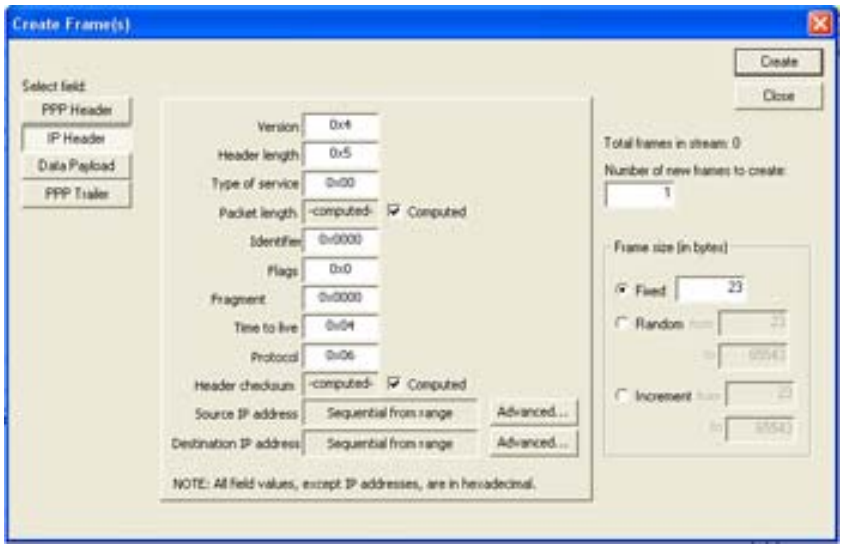


Figura 3.13– Configuración de la cabecera IP para una trama POS IP

Seleccionamos **increment byte** en **Fill pattern**.

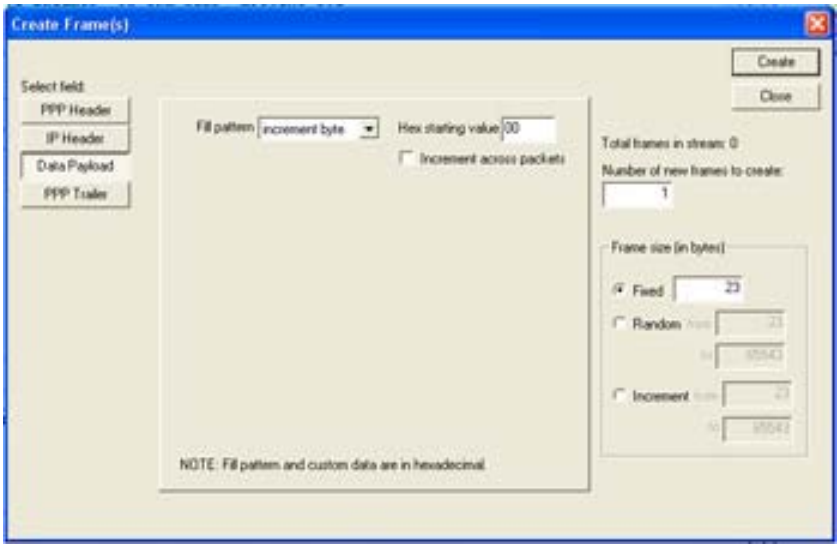


Figura 3.14– Configuración de la carga útil de datos para una trama POS IP

Marcamos **Computed** en **Checksum** para que el checksum se calcule automáticamente.

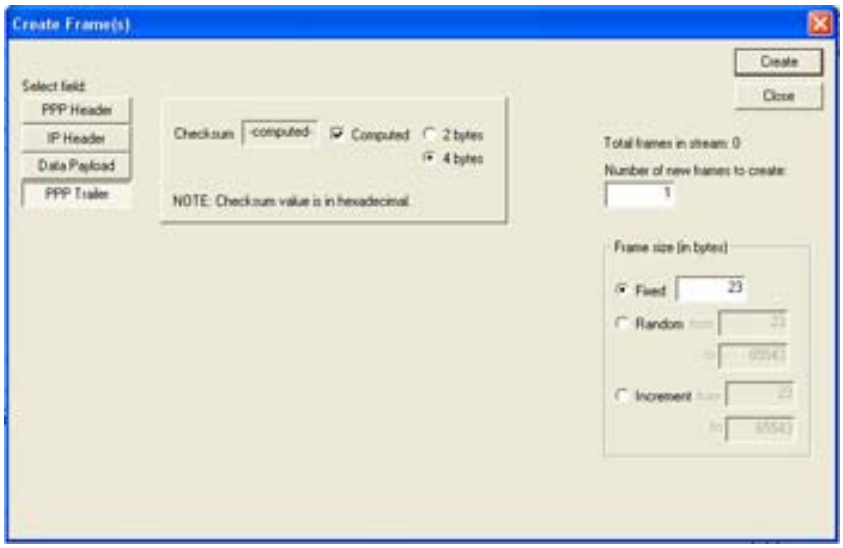


Figura 3.15– Configuración de PPP Trailer para una trama POS IP

3.8.3 FLUJO DE DATOS ATM

Para ello tendremos que crear una trama ATM. Le llamaremos **Packet Flow 3.strm**. Aparecerá el cuadro de diálogo **ATM Stream**.

En el cuadro de diálogo se puede elegir entre crear un paquete o múltiples paquetes. Marcaremos la opción **Multiple packets from pool** y haciendo clic en **Create Pool…**crearemos paquetes y configuraremos las cabeceras.

Seleccionamos **Computed** en **Packet length** y en **Header** checksum. Haremos clic en **Advanced…** y se selecciona un rango entre 0.0.0.0 y 255.255.255.255.

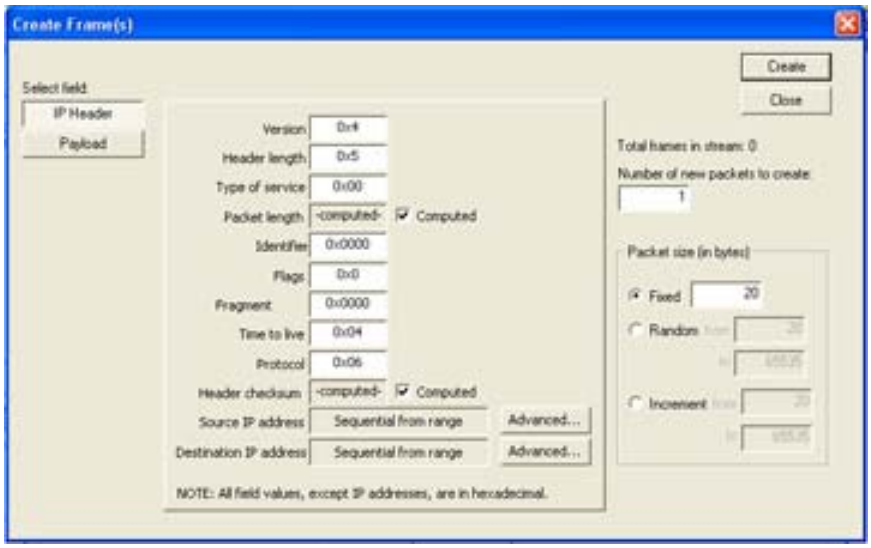


Figura 3.16– Configuración de la cabecera IP para una trama ATM AAL5

Seleccionamos **increment byte** en **Fill pattern**.

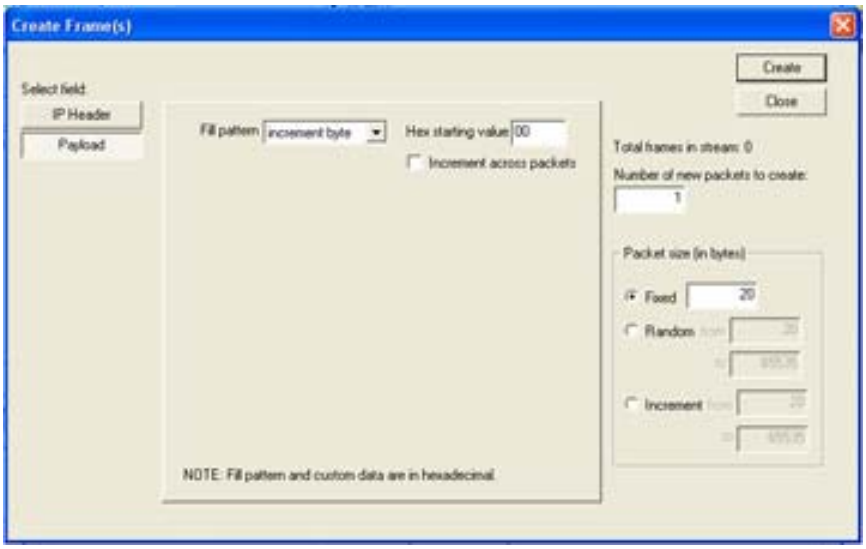


Figura 3.17– Configuración de la carga útil para una trama ATM AAL5

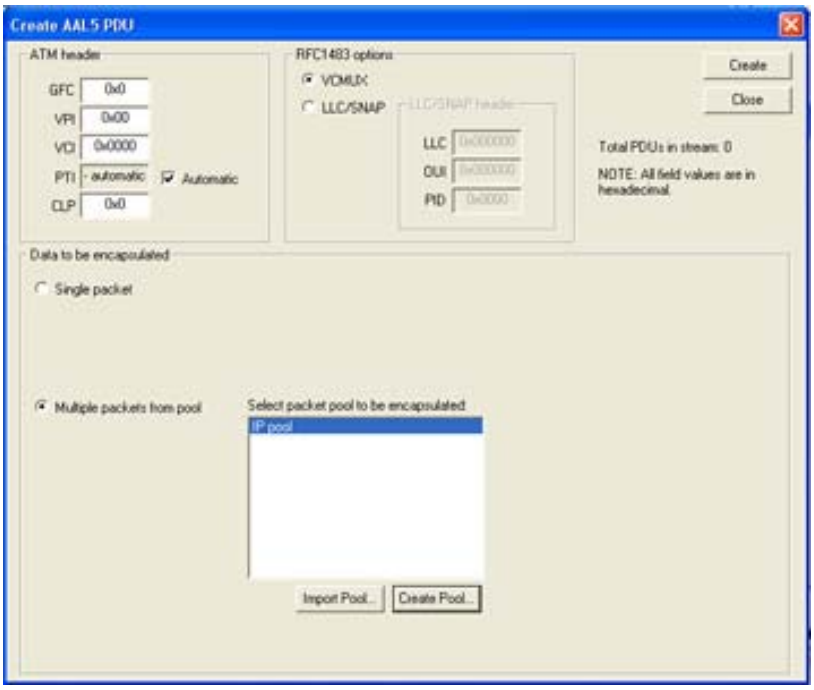


Figura 3.18– Configuración final de la trama ATM AAL5

3.8.4 FLUJO DE DATOS ETHERNET TCP/IP A MEDIDA

Para ello tendremos que crear una trama Custom Ethernet IP. Le llamaremos **Packet Flow 4.strm**. Aparecerá el cuadro de diálogo **Custom Ethernet IP Data Stream**.
Se deja el valor de la cabecera por defecto.

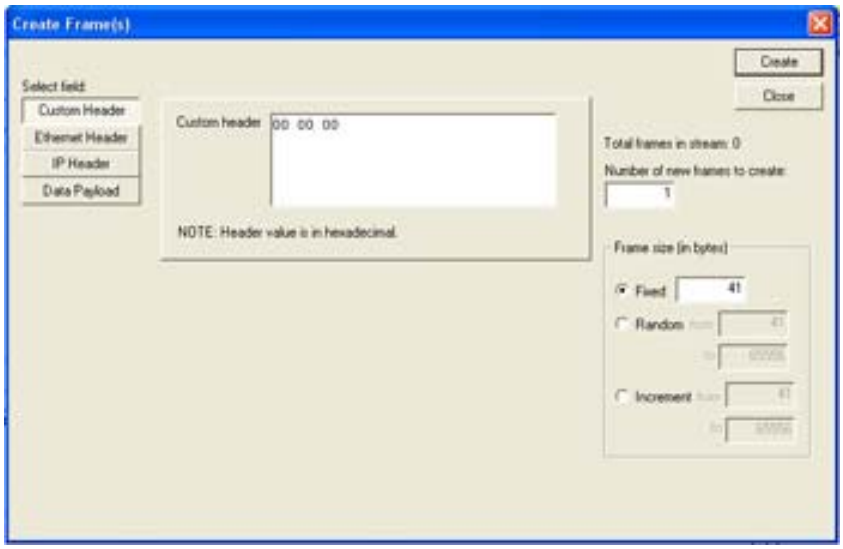


Figura 3.19– Configuración de la cabecera a medida para una trama Ethernet TCP/IP a medida

Hacemos clic en **Advanced…** y marcamos **From within range** para que las direcciones estén comprendidas entre 0x000000000000 y 0xffffffffffff.

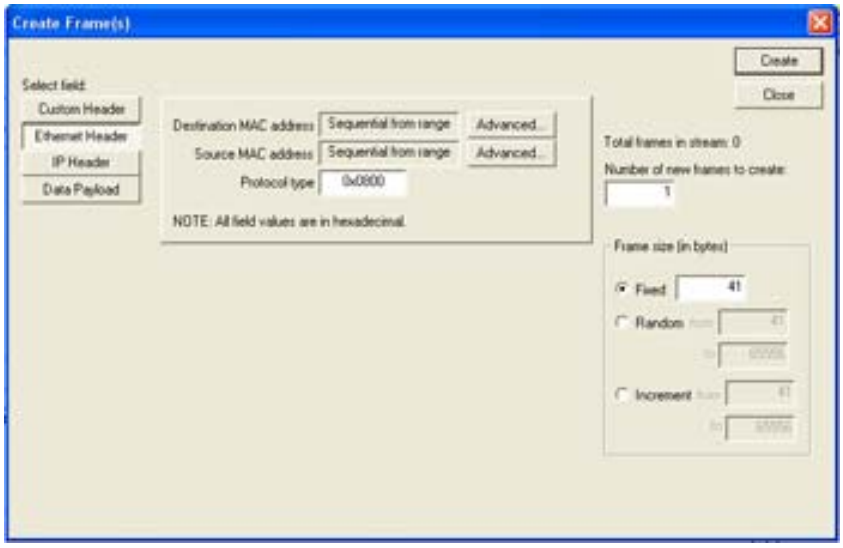


Figura 3.20– Configuración de la cabecera Ethernet para una trama Ethernet TCP/IP a medida

Seleccionamos **Computed** en **Packet length** y en **Header checksum**. Haremos en **Advanced…** y se selecciona un rango de direcciones entre 0.0.0.0 y 255.255.255.255.

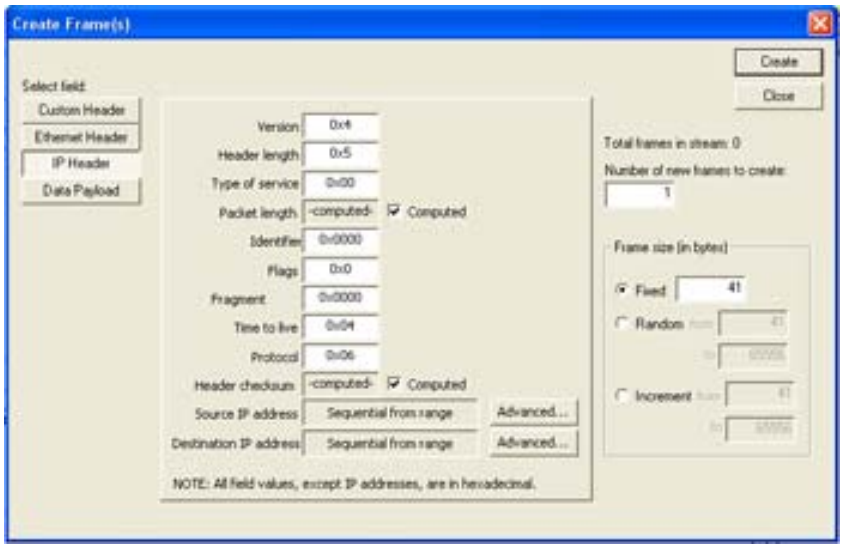


Figura 3.21– Configuración de la cabecera IP para una trama Ethernet TCP/IP a medida

Seleccionaremos **increment byte** en **Fill pattern**.

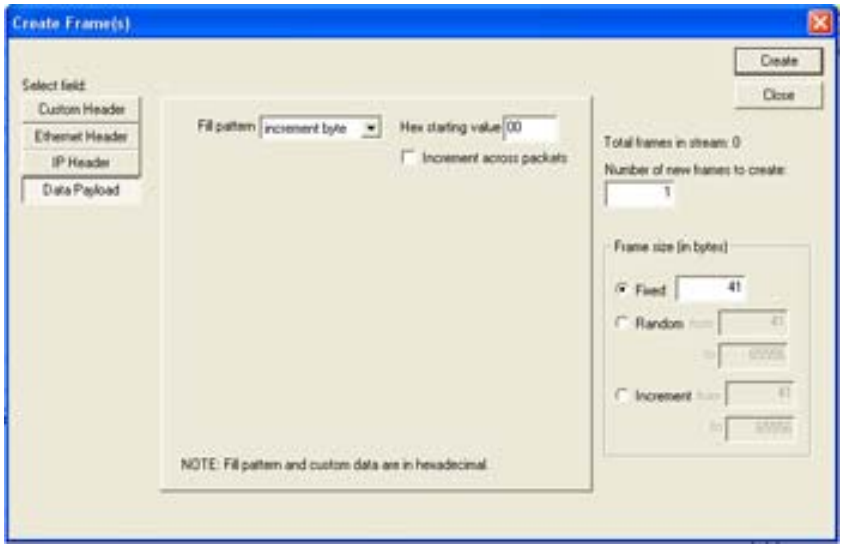


Figura 3.22– Configuración de la carga útil de datos para una trama Ethernet TCP/IP a medida

3.8.5 FLUJO DE DATOS PPP TCP/IP

Para ello tendremos que crear una trama PPP TCP/IP. Le llamaremos **Packet Flow 5.strm**. Aparecerá el cuadro de diálogo **PPP TCP/IP Data Stream**.

Marcamos **Include Address** e **Include Control**.

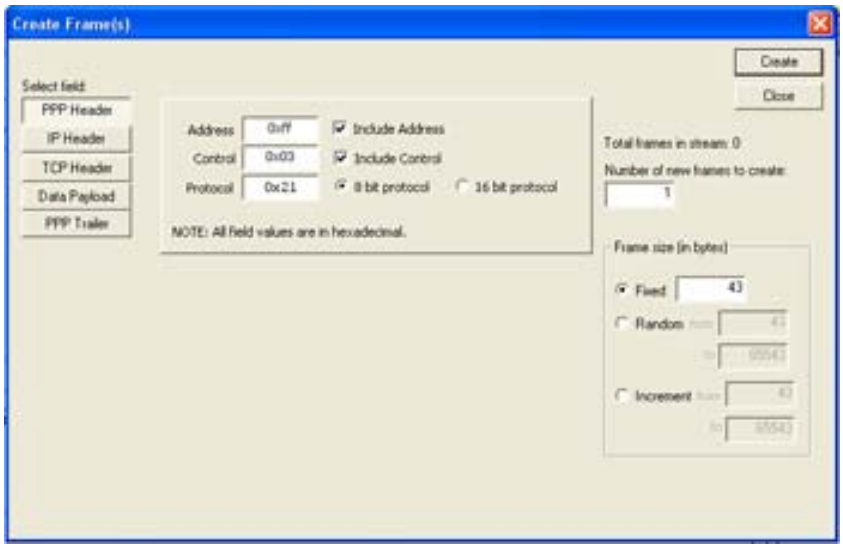


Figura 3.23– Configuración de la cabecera PPP para una trama PPP TCP/IP

Seleccionamos **Computed** en **Packet length** y en **Header checksum**. Haremos clic en **Advanced…** y se selecciona un rango de direcciones entre 0.0.0.0 y 255.255.255.255.

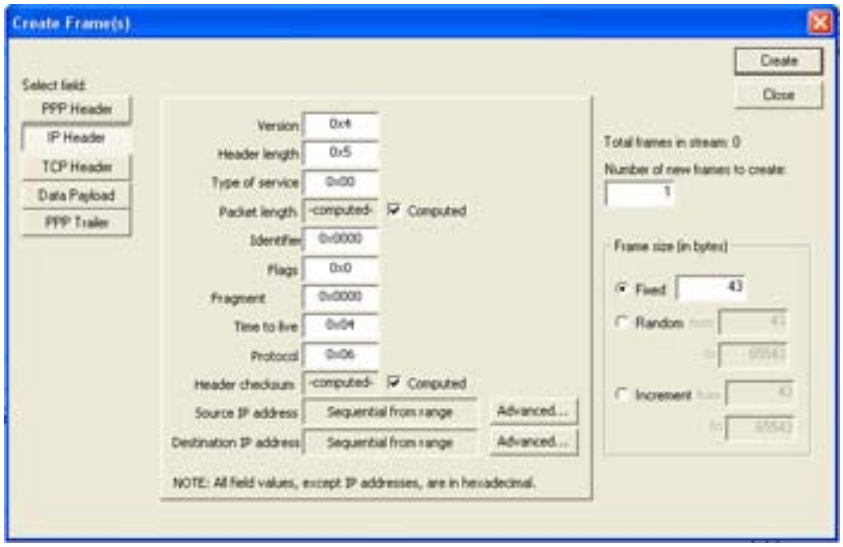


Figura 3.24– Configuración de la cabecera IP para una trama PPP TCP/IP

Marcamos la opción **Computed**.

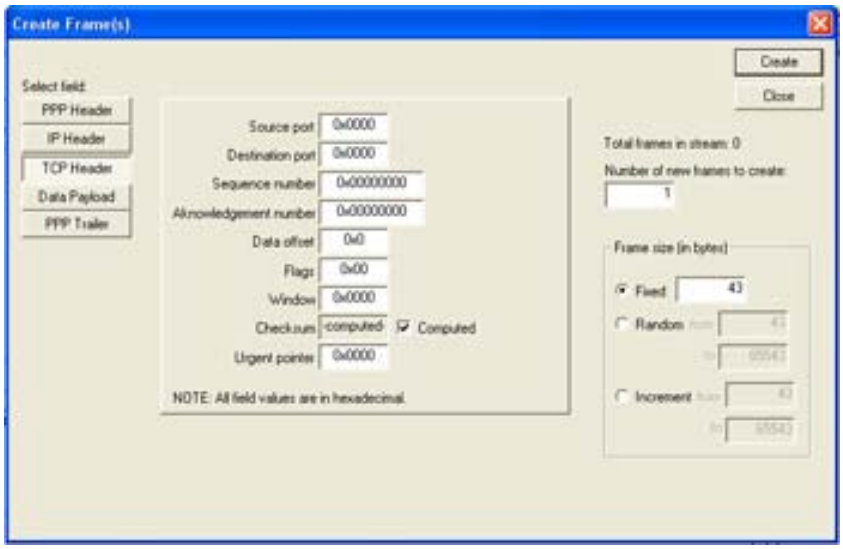


Figura 3.25– Configuración de la cabecera TCP para una trama PPP TCP/IP

Seleccionaremos **increment byte** en **Fill pattern**.

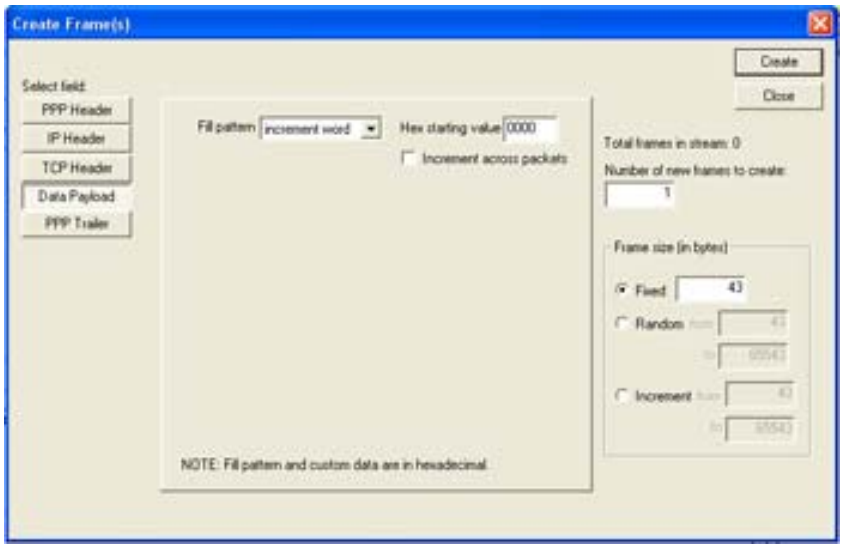


Figura 3.26– Configuración de la carga útil de datos para una trama PPP TCP/IP

Marcaremos la opción **Computed**.

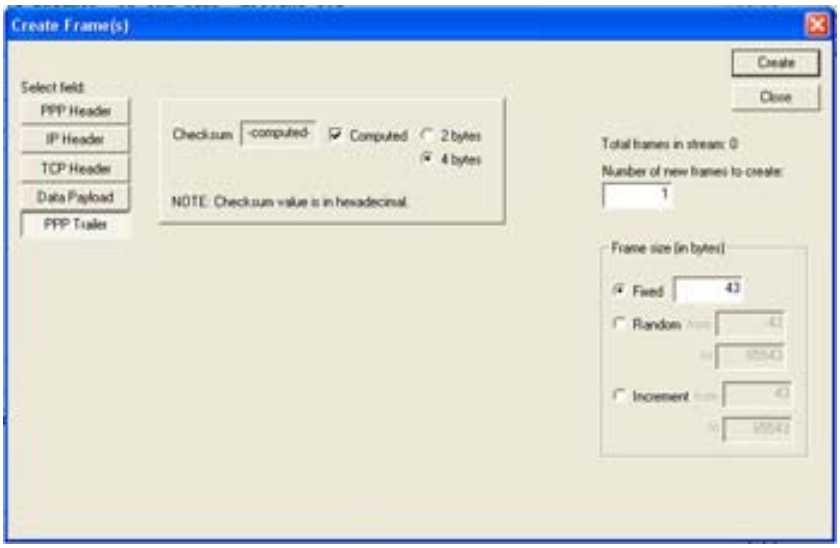


Figura 3.27– Configuración de PPP trailer para una trama PPP TCP/IP

3.8.6 FLUJO DE DATOS ETHERNET IP

Para ello tendremos que crear una trama Ethernet IP. Le llamaremos **Packet Flow 6.strm**. Aparecerá el cuadro de diálogo **Ethernet IP Data Stream**.

Hacemos clic en **Advanced…** y marcamos **From within range** para que las direcciones estén comprendidas entre el rango que se le especifique, éste estará comprendido entre 0x000000000000 y 0xffffffffffff.

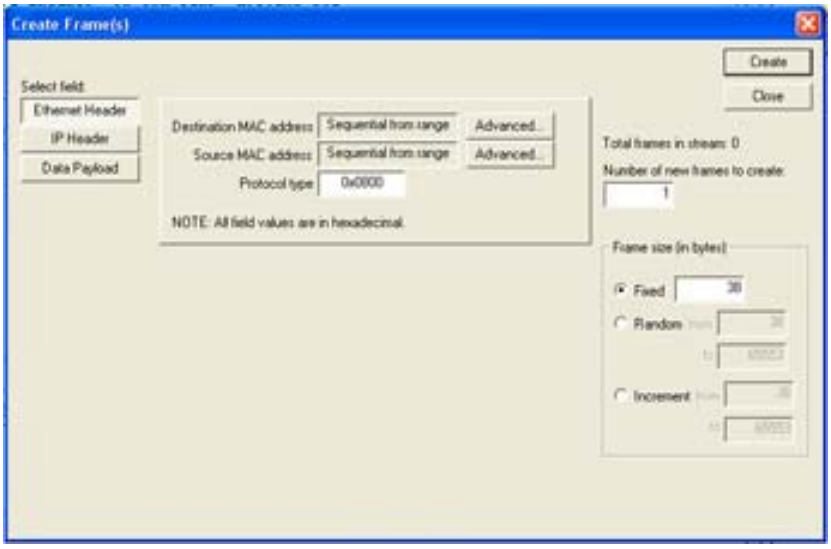


Figura 3.28– Configuración de la cabecera Ethernet para una trama Ethernet IP

Seleccionamos **Computed** en **Packet length** y en **Header checksum.** Hacemos clic en **Advanced…** y se selecciona un rango de direcciones entre 0.0.0.0 y 255.255.255.255.

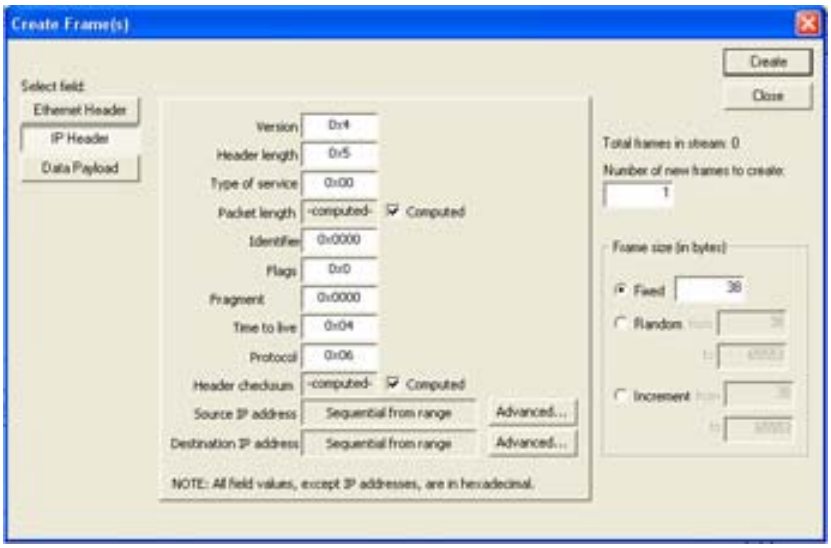


Figura 3.29– Configuración de la cabecera IP para una trama Ethernet IP

Seleccionamos **increment byte** en **Fill pattern**.

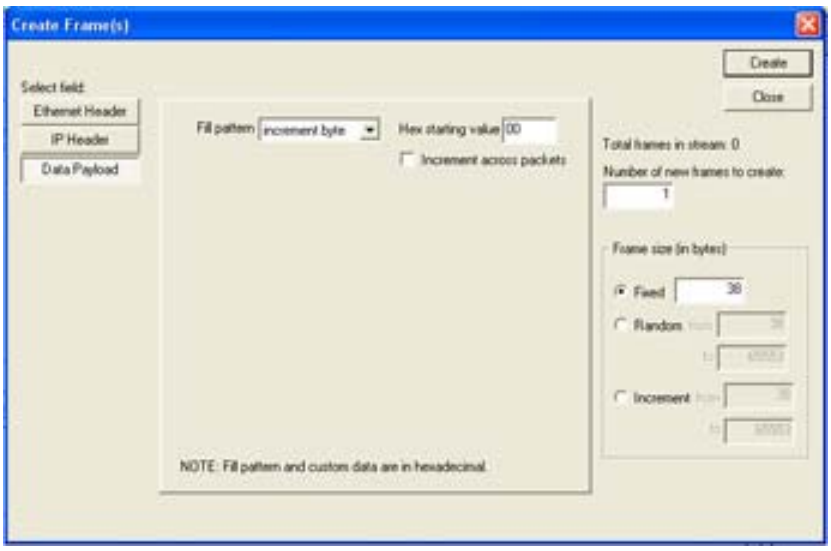


Figura 3.30– Configuración de la carga útil de datos para una trama Ethernet IP

3.9 EL DEPURADOR

Utilizando el WorkBench se puede depurar microcódigo en modo Simulación o en modo Hardware:

- **Modo Simulación:** el Transactor proporciona soporte de depurado para el WorkBench.
- **Modo Hardware:** *Microengine Debug Library* se ejecuta como parte de un programa de aplicación del procesador Intel XScale. Se comunica con el WorkBench y retransmite depurando operaciones entre el WorkBench y los micromotores.

La aplicación debe ser una que sea suministrada con la Plataforma de Desarrollo o una que sea de desarrollo independiente.

Los menús del WorkBench y las barras de herramientas de selección proporcionan las siguientes características:

- Breakpoints y control de ejecución del microcódigo.
- Vista del código fuente en un per hilo básico.
- Muestra el estado y la historia de los micromotores, hilos y colas.
- Vista y breakpoints en datos, registros y pins.

3.9.1 NÚCLEO DE EJECUCIÓN

Para un micromotor que contiene código C, se puede seleccionar una vista fuente o una vista lista. En la vista fuente se puede seleccionar la fuente a mostrar. Las líneas fuente muestran la ejecución máxima que cuenta la cantidad de todas las instrucciones generadas.

Para funciones en línea, el contador para una línea fuente es la suma de los contadores máximos para todas las instancias. Las unidades para el eje horizontal en la gráfica de barras recuerdan las direcciones de instrucción.

Para mostrar el núcleo de ejecución, hacer clic en **Simulation** -> **Execution Coverage**. Si el proyecto contiene múltiples chips, seleccionamos el chip cuyo núcleo de datos deseamos ver. También habrá que seleccionar un micromotor y un módulo/archivo de lista.

Aparecerá un número a la izquierda de cada línea que indicará el número de veces que esa instrucción fue ejecutada.

La gráfica de barras muestra las direcciones de las instrucciones y los contadores de ejecución.

Los contadores de ejecución son el total para todos los contextos en el micromotor. El contador de ejecución para la referencia de un macro contraído es la suma de los contadores de ejecución de todas las instrucciones generadas por el macro.

Cada color indica las veces que se ejecutó la instrucción. Se puede sincronizar la vista entre la ventana de código y el gráfico de barras.

Por defecto, el contador de ejecución comienza en el primer ciclo de simulación. Se podrá hacer para que los contadores se reseteen cuando se resetee la simulación.

Se puede reportar los contadores de ejecución:

- Para el micromotor y contextos seleccionados.
- Por la dirección *microstore* para todos los micromotores y contextos en uso.
- Por línea para el archivo fuente.

El WorkBench direcciona funciones de consola que relatan el núcleo de ejecución. Están disponibles por la línea de comandos del Transactor sólo cuando el WorkBench está depurando.

3.9.2 BREAKPOINTS

Permite al usuario mirar el conjunto de *breakpoints* en el proyecto actualmente abierto. Permite al usuario modificar el estado de los *breakpoints*, borrarlos, y clasificarlos siguiendo un cierto criterio para mostrar un subconjunto de *breakpoints*. Para visualizarlos, hacer clic en **View** -> **Debug Windows** -> **Breakpoints**.

Operaciones de filtro:

- Filtrar *breakpoints* por grupo o por chip.
- Mostrar el código, los datos y la memoria de los *breakpoints*.

Para ir a un *breakpoin*t, se hará de tres maneras:

- Si lo que está seleccionado es el código de *breakpoin*t, el cursor va a la línea en la Ventana de Hilos donde está el *breakpoint*.
- *Data Watch*.
- *Memory Watch*.

3.9.3 CÓDIGO DE BREAKPOINTS

Cuando un *breakpoint* es alcanzado durante la ejecución:

- Se activa la ventana de hilos que alcanzó el *breakpoint*.
- Se muestra y se marca la instrucción apropiada.
- Aparecerá un mensaje indicando que el *breakpoint* fue alcanzado.

Se puede configurar en código de *breakpoint* desde la línea de comandos del Transactor. También es posible borrarlo, activarlo o desactivarlo desde el WorkBench, pero está sujeto a varias limitaciones.

Hay distintos marcadores.

Para activar o desactivar los *breakpoints*:

1. Situar el cursor en la línea donde se quiere activar/desactivar el *breakpoint*.
2. **Debug** -> **Code Breakpoint** -> **Enable/Disable**, **Disable All**, **Enable All**.
3. Para borrar: **Remove All**.

Implementa soporte para la habilidad de manipular múltiples micromotores simultáneamente.

El usuario puede definir grupos de códigos de *breakpoint* que pueden ser activados o desactivados. De esta manera, podemos activar un conjunto dependiendo de lo que se quiera chequear.

Los códigos de *breakpoint* sólo pueden asignarse a un grupo: **Debug** -> **Code Breakpoint** -> **Manager Breakpoint Groups**.

Una vez creado, podemos asignar el *breakpoint* al Grupo *Breakpoint*: **Debug** -> **Code Breakpoint** -> **Assign to Breakpoint Group…**

3.9.4 DATA WATCHES

Se pueden monitorizar los valores de los estados de simulación en modo Depurado utilizando la ventana *Data Watches*. El WorkBench reconoce los Registros de Control y Estado (CSR), Memoria micromotor, búffers MSF y nombres de pines.

View -> **Debug Windows** -> **Data Watch**.

Categorías *Data Watch*: CAP CSRs, micromotores CSRs, memoria CSRs, MSF CSRs, MSF CSRs, PCI CSRs (sólo en depuración en modo hardware), procesador Intel Xscale CSRs, memoria de micromotores, buffers MSF, pins.

Operaciones:

- Añadir un *Data Watch* haciendo clic en el botón **Add Watch…**, introducir uno nuevo, borrarlo, cambiarle el nombre y cambiar la notación a mostrar (direcciones en hexadecimal y datos en hexadecimal).
- Visualizar *Data Watch* en la ventana de hilos C.
- Ver los registros micromotores.
- *Pipeline Data Watch*: el WorkBench soporta *Data Watch* de Variables de Próximo Vecino y *Scratch Pipe* en aplicaciones de Modo de Autoparticionado.
- Cambiar el valor de un estado de simulación.
- *Breakpoint* en cambios y escrituras de datos (valor de estado). No es posible en modo hardware.

3.9.5 VENTANA DE HILOS

Configurar y desbloquear *breakpoints*, mostrar registros o contenidos de las variables, y ver las instrucciones que actualmente se están ejecutando.

Operaciones:

- **Controlar la Activación de la ventana de hilos**. **Debug** -> **Thread Window Options**: utilizar una ventana de hilos sólo, por chip, por micromotor, o separada por cada hilo.

- **Seguirle la pista al hilo activo**.

- **Ejecutar hacia el Cursor**. **Debug** -> **Run Control**. Si la línea está en la vista fuente de un hilo compilado o si contiene a una referencia de macro contraído en un hilo ensamblado, entonces la simulación corre hasta que la primera instrucción generada sea alcanzada.

- **Vista de palanca de conmutación**. Cuando se está depurando un proyecto micromotor C o una mezcla de C y ensamblador, la ventana de hilo puede conmutar entre la vista fuente y la vista lista.

- **Activar la ventana de hilos**. Se puede acceder directamente al estado de ejecución de todos los hilos en el proyecto una vez que el microcódigo se haya cargado.

- **Mostrar, expandir, y contraer macros** (sólo hilos ensamblados).

- **Mostrar y Ocultar las direcciones de Instrucción**.

- **Marcadores de Instrucción**.

- **Vista lista y fuente de los marcadores de microdirecciones**, mostradas en distintos colores.

- **Vista de la ejecuicón de instrucción en Thread Windows**: durante una sesión de simulación, el marcador de Etapa de *Pipeline* permite ver qué instrucción está dentro de las seis etapas del *pipeline*.

- **Documento e Historia** del *Thread Window*.

- **Localización de la Ejecución de Instrucción**. Permite navegar rápidamente hacia atrás y hacia delante a través de la historia hilo/PC desde el actual punto en el tiempo a algún otro punto en el tiempo cuando una instrucción designada fue ejecutada por el micromotor o por el hilo asociado con el *Thread Window* centrado.

- **Localización del operando de instrucción**. Proporciona al usuario la habilidad para mover hacia atrás y hacia delante a través de la historia de ejecución de instrucción, siguiendo las dependencias entre operandos fuente y destino. Esto permite al usuario moverse rápidamente a la instrucción donde un operando fuente particular fue anteriormente escrito, permitiendo al usuario seguir fácilmente las decisiones del procesamiento de paquetes. Los tipos de registros soportados son: GPRs, registros vecinos, registros de transferencia, memoria local.

3.9.6 MEMORY WATCH

En modo depuración, se puede observar los valores DRAM, SRAM, y las localizaciones de la memoria *scratchpad* utilizando la ventana de *Memory Watch*. La dirección de memoria de los *network processors* en bytes, de esta manera la ventana de la *Memory Watch* interpreta la dirección para ser visualizada como byte alineado. Un byte de dirección SRAM o SCRATCHPAD es redondeado para la próxima *longword* más baja y el dato es mostrado en *longwords*. Un byte de dirección DRAM es redondeada a la próxima *quadword* más baja y el dato es mostrado en *quadwords*.

Para visualizar la ventana de Memory Watch: **View** -> **Debug Windows** -> **Memory Watch**.

Seleccionamos el chip de memoria que queremos visualizar. También podemos controlar la visibilidad de las subventanas SDRAM, SRAM, y SCRATCHPAD.

Las operaciones que podemos realizar son:

- Introducir un nuevo Memory Watch.
- Añadir un Memory Watch.
- Borrar un Memory Watch.
- Cambiar un Memory Watch.
- Cambiar la notación a mostrar de un Memory Watch.
- Poner un breakpoint al cambiar o al escribir la memoria.

3.9.7 ESTADÍSTICAS DE RENDIMIENTO

Para ver las estadísticas del rendimiento, seleccionamos **Performance Statistics** del menú **Performance**. Se mostrará un cuadro diálogo con tres pestañas:

- **Sumary**: muestra el porcentaje de tiempo que está activo cada micromotor y cada unidad de memoria, y el radio que esta actividad representa.
- **MicroEngine**: contiene un árbol jerárquico en varias columnas que muestra las estadísticas en porcentajes de tiempo que se estuvo ejecutando cada componente.
- **All**: muestra todas las estadísticas acumuladas por el Transactor.

3.9.8 HISTORIA DE HILOS Y DE COLA

Permite ver el estado de todos los hilos y numerosas colas en un chip al mismo tiempo.

Para verlo, se selecciona **View** -> **Debug Windows** -> **History**. En modo hardware no es posible mostrar la historia de los hilos y de las colas.

En el botón **Customize…** se pueden seleccionar opciones para que la gráfica nos lo muestre. Se puede observar cinco grupos de colas: DRAM, SRAM, MicroEngines, SHaC y MSF.

3.9.9 ESTADO DE LA COLA

Proporciona información actual e histórica del contenido de los 5 grupos de colas.

Para activar esta opción, habrá que seleccionar **View** -> **Debug Windows** -> **Queu Status**.

3.9.10 LISTADO DE PAQUETES

Muestra una lista de todos los paquetes conocidos en el **WorkBench**. Incluye paquetes que se están procesando actualmente por la aplicación microcódigo y los paquetes que han sido transmitidos previamente.

Para poder visualizarlo, se selecciona **View** -> **Debug Windows** -> **Packet List**.

3.9.11 LISTADO DE EVENTOS

Muestra una lista filtrada de eventos de referencia de memoria y de paquetes clasificados por ciclo de tiempo.

Seleccionando **View** -> **Debug Windows** -> **Event List** se podrá visualizar la lista de eventos.

Los eventos de referencia de memoria son registrados automáticamente por el WorkBench. Los eventos de los paquetes son registrados manualmente "instrumentando" la aplicación microcódigo utilizando *breakpoints* condicionales y el paquete que por el WorkBench observando la interacción de la aplicación microcódigo con los búffers MSF y CSRs.

3.9.12 ESTADO DEL HILO

Se puede ver el estado de los hilos seleccionando **View** -> **Debug Windows** -> **Thread Status**.

El estado del hilo proporciona información estática o instantánea del estado de cada hilo en un chip seleccionado en el proyecto. Para cada hilo, se muestra la siguiente información:

- Dirección de la instrucción actual.
- Lista de eventos por el cual el hilo está esperando.
- Lista de eventos que han sido señalados para el hilo.

3.9.13 ESTADO DE LA SIMULACIÓN DE PAQUETES

Proporciona información estática o instantánea en el estado del tráfico del paquete en el proyecto.

Proporciona información en la interfaz de tráfico configurada, el estado de los búffers receptor y transmisor, protocolos y paquetes.

CAPÍTULO 4
SIMULACIÓN DE LA APLICACIÓN

4.1 PROCESAMIENTO DE PAQUETES SOBRE UN ÚNICO HILO

Lleva a cabo la clasificación de los paquetes que entran en el *network processor*.

Por un lado, la aplicación chequea la longitud del paquete para determinar si es un paquete Ethernet II o no. Si no lo es, lo clasifica como inválido. Si lo es, coge la cabecera del paquete para averiguar si es una dirección origen de tipo *broadcast* o *multicast*. Si la dirección supera los 8 bits, lo clasifica también como paquete inválido, sino chequea la dirección destino para clasificarlo como *broadcast*, *multicast*, local u otra.

Por otro lado, lleva a cabo la clasificación de los paquetes basándose en la clasificación *5-tuple*: dirección IP origen, dirección IP destino, protocolo IP, puerto origen (TCP o UDP) y puerto destino (TCP o UDP). Basándose en esta clasificación, el anillo *scratch* de salida se configura para el paquete.

Para que el clasificador *5-tuple* opere, se inicializan los CSRs y las estructuras de datos necesarios.

El resultado de esta clasificación es un indicador de descarte, un número de anillo, una dirección IP de siguiente salto:

- Si *5-tuple* no está en la tabla, el paquete se descarta.
- Si el paquete no es lo suficientemente grande para ser un paquete IP:
    - Si TTL<1 o no es ni TCP ni UDP, el paquete se descarta.
    - Si no, será un paquete aceptable. Se lleva a cabo una búsqueda hash para ver si la tabla coincide con la clave.

A través de la cabecera del paquete, se obtiene la longitud del mismo. El diagrama de bloques podemos verlo en la figura 4.1.

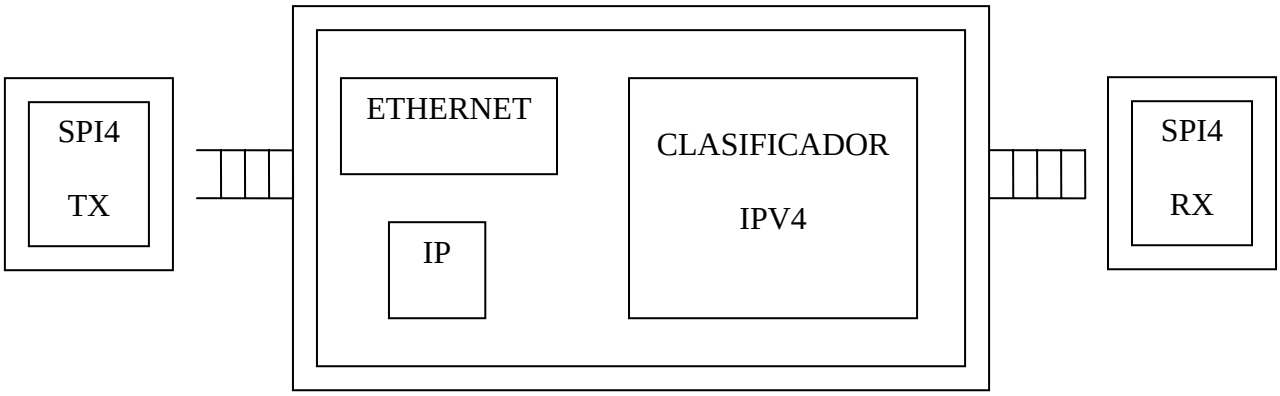


Figura 4.1– Diagrama de bloques. Procesamiento de Paquetes en un único Hilo

Como muestra la figura 4.1, el programa desencola un paquete de la tarea de recepción y se lo manda al microbloque Ethernet donde se comprobará si es un paquete Ethernet válido y si está diseccionado localmente.

En este punto sabremos que tenemos un paquete Ethernet II. Pero antes de mandarlo al clasificar IPv4, habrá que asegurarse de que es un paquete IP.

Ahora que se tiene un paquete IP, será mandado al clasificador IPv4 *5-tuple*. El clasificador asignará al paquete un anillo de salida, y una dirección IP de próximo salto.

Una vez asignado el anillo de salida, el paquete será mandado al buffer gestor RED para encolarlo o descartarlo. Después, el paquete será movido al anillo para poder ser transmitido.

1. Primero, asignaremos varios tipos de flujo de tráfico. Hacemos clic en **Simulation** -> **Packet Simulation Options** -> **Traffic Assignment** -> **Assign Input…** y añadiremos el siguiente tráfico: Packet flow 1.strm, Packet flow 2.strm, Packet flow 3.strm, Packet flow 4.strm, Packet flow 5.strm, Packet flow 6.strm.

2. Comprobamos que la aplicación no produce ningún error haciendo clic en **Build** -> **Rebuild**.

3. Clic en **Debug** -> **Start Debugging** para empezar a depurar la aplicación. Dejamos que la aplicación se ejecute unos 100 000 micromotores.

Estadísticas del rendimiento:

En la figura 4.2 se detalla el rendimiento de los micromotores que utilizamos para la aplicación. Estos micromotores son: 0:0, 0:1 y 1:7.

	Active	Rate
Chip [single_thread_proc]		
Microengine 0:0	16.91%	236.73 Mips
Microengine 0:1	11.64%	162.90 Mips
Microengine 0:2	0.00%	0.00 Mips
Microengine 0:3	0.00%	0.00 Mips
Microengine 0:4	0.00%	0.00 Mips
Microengine 0:5	0.00%	0.00 Mips
Microengine 0:6	0.00%	0.00 Mips
Microengine 0:7	0.00%	0.00 Mips
Microengine 1:0	0.00%	0.00 Mips
Microengine 1:1	0.00%	0.00 Mips
Microengine 1:2	0.00%	0.00 Mips
Microengine 1:3	0.00%	0.00 Mips
Microengine 1:4	0.00%	0.00 Mips
Microengine 1:5	0.00%	0.00 Mips
Microengine 1:6	0.00%	0.00 Mips
Microengine 1:7	5.20%	72.87 Mips
Total		472.50 Mips

Figura 4.2– Estadísticas del rendimiento

Estado de la simulación de paquetes:

single_thread_p | Options... | Save Stats To File...

Packets received 421 | Receive rate 9468.61 Mbps (at network)

Packets transmitted 19 | Transmit rate 187.33 Mbps (at network)

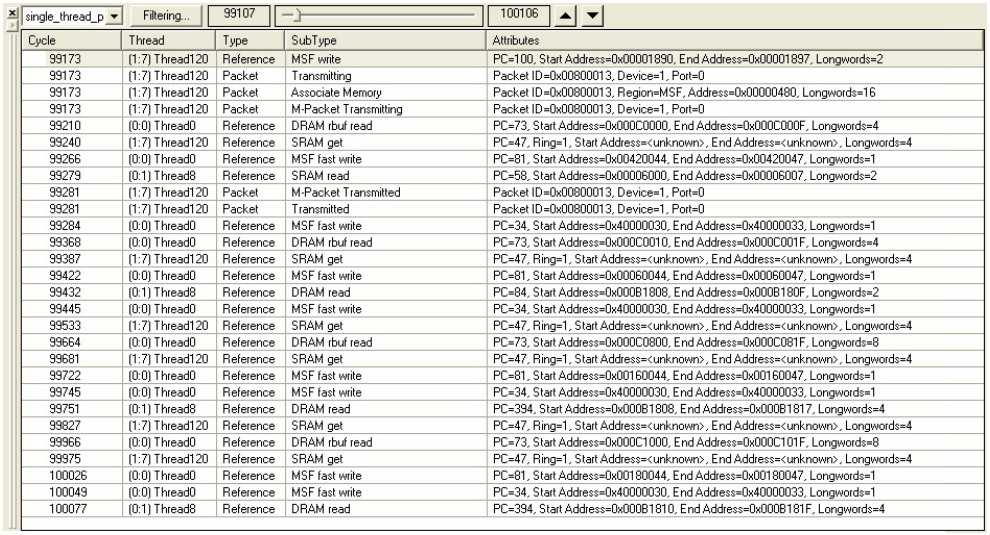
Traffic Interface	Rx buffer fullness	Tx buffer fullness	Packets received	Receive rate	Packets sent	Transmit rate
Device ID 0 (SPI4 Rx)			421	9468.611	n/a	n/a
Port 0	Unboun...	Unboun...	421	9468.611	n/a	n/a
Device ID 1 (SPI4 Tx)			n/a	n/a	19	187.3320
Port 0	Unboun...	Unboun...	n/a	n/a	19	187.3320

Figura 4.3– Estado de la simulación de paquetes

Como muestra la figura 4.3, el radio de transmisión de paquetes es menor que el radio de recepción de paquetes. Los paquetes que no se transmiten son porque no cumplen las condiciones de ser Ethernet, no están direccionados localmente, y no cumplen las características de ser un paquete IP. Como el tráfico que se está generando es ATM AAL5, POS IP, etc, es por ello por lo que se rechazan tantos paquetes por lo cual se ve reflejado a la hora de transmitirlos.

Lista de eventos:

En la figura 4.4 se puede apreciar los eventos que transcurren en cada hilo, ya sea una referencia o un paquete, y qué valor tiene el PC en cada ciclo de procesamiento.



Cycle	Thread	Type	SubType	Attributes
99173	(1:7) Thread120	Reference	MSF write	PC=100, Start Address=0x00001890, End Address=0x00001897, Longwords=2
99173	(1:7) Thread120	Packet	Transmitting	Packet ID=0x00800013, Device=1, Port=0
99173	(1:7) Thread120	Packet	Associate Memory	Packet ID=0x00800013, Region=MSF, Address=0x00000480, Longwords=16
99173	(1:7) Thread120	Packet	M-Packet Transmitting	Packet ID=0x00800013, Device=1, Port=0
99210	(0:0) Thread0	Reference	DRAM rbuf read	PC=73, Start Address=0x000C0000, End Address=0x000C000F, Longwords=4
99240	(1:7) Thread120	Reference	SRAM get	PC=47, Ring=1, Start Address=<unknown>, End Address=<unknown>, Longwords=4
99266	(0:0) Thread0	Reference	MSF fast write	PC=81, Start Address=0x00420044, End Address=0x00420047, Longwords=1
99279	(0:1) Thread8	Reference	SRAM read	PC=58, Start Address=0x00006000, End Address=0x00006007, Longwords=2
99281	(1:7) Thread120	Packet	M-Packet Transmitted	Packet ID=0x00800013, Device=1, Port=0
99281	(1:7) Thread120	Packet	Transmitted	Packet ID=0x00800013, Device=1, Port=0
99284	(0:0) Thread0	Reference	MSF fast write	PC=34, Start Address=0x40000030, End Address=0x40000033, Longwords=1
99368	(0:0) Thread0	Reference	DRAM rbuf read	PC=73, Start Address=0x000C0010, End Address=0x000C001F, Longwords=4
99387	(1:7) Thread120	Reference	SRAM get	PC=47, Ring=1, Start Address=<unknown>, End Address=<unknown>, Longwords=4
99422	(0:0) Thread0	Reference	MSF fast write	PC=81, Start Address=0x00060044, End Address=0x00060047, Longwords=1
99432	(0:1) Thread8	Reference	DRAM read	PC=84, Start Address=0x000B1808, End Address=0x000B180F, Longwords=2
99445	(0:0) Thread0	Reference	MSF fast write	PC=34, Start Address=0x40000030, End Address=0x40000033, Longwords=1
99533	(1:7) Thread120	Reference	SRAM get	PC=47, Ring=1, Start Address=<unknown>, End Address=<unknown>, Longwords=4
99664	(0:0) Thread0	Reference	DRAM rbuf read	PC=73, Start Address=0x000C0800, End Address=0x000C081F, Longwords=8
99681	(1:7) Thread120	Reference	SRAM get	PC=47, Ring=1, Start Address=<unknown>, End Address=<unknown>, Longwords=4
99722	(0:0) Thread0	Reference	MSF fast write	PC=81, Start Address=0x00160044, End Address=0x00160047, Longwords=1
99745	(0:0) Thread0	Reference	MSF fast write	PC=34, Start Address=0x40000030, End Address=0x40000033, Longwords=1
99751	(0:1) Thread8	Reference	DRAM read	PC=394, Start Address=0x000B1808, End Address=0x000B1817, Longwords=4
99827	(1:7) Thread120	Reference	SRAM get	PC=47, Ring=1, Start Address=<unknown>, End Address=<unknown>, Longwords=4
99966	(0:0) Thread0	Reference	DRAM rbuf read	PC=73, Start Address=0x000C1000, End Address=0x000C101F, Longwords=8
99975	(1:7) Thread120	Reference	SRAM get	PC=47, Ring=1, Start Address=<unknown>, End Address=<unknown>, Longwords=4
100026	(0:0) Thread0	Reference	MSF fast write	PC=81, Start Address=0x00180044, End Address=0x00180047, Longwords=1
100049	(0:0) Thread0	Reference	MSF fast write	PC=34, Start Address=0x40000030, End Address=0x40000033, Longwords=1
100077	(0:1) Thread8	Reference	DRAM read	PC=394, Start Address=0x000B1810, End Address=0x000B181F, Longwords=4

Figura 4.4– Lista de eventos

Lista de paquetes:

Como ejemplo, mostramos en la figura 4.5 uno de los paquetes cuando ha sido validado y transmitido.

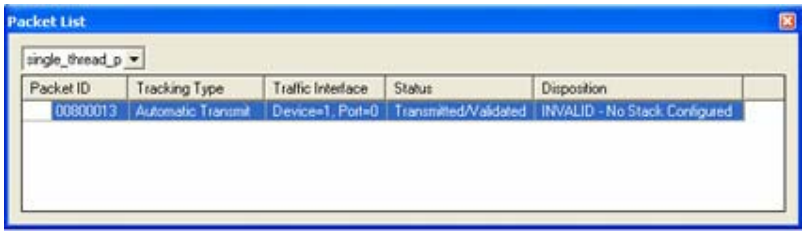


Figura 4.5– Paquete validado y transmitido

Historia:

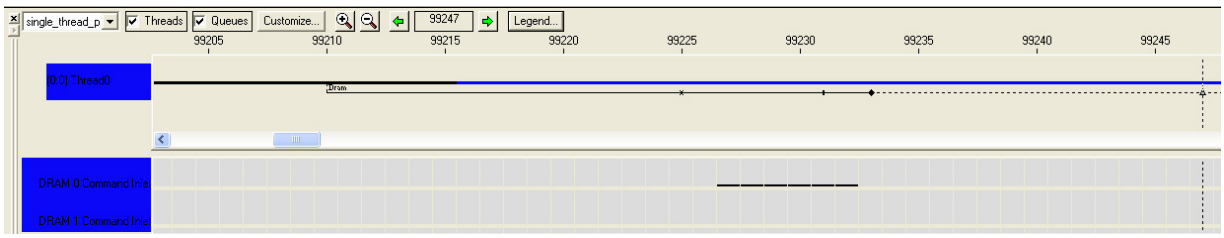


Figura 4.6– Encolamiento de un paquete y transmisión del mismo

Al entrar en la SRAM, se encola un paquete en un buffer de tamaño 17. Desde que el paquete es encolado hasta que el paquete es borrado de la cola transcurren 6 micromotores. El proceso de

encolamiento termina pasados 2 micromotores más. Así es igual en todos los encolamientos. Podemos apreciarlo en la figura 4.6.

4.2 CONTADOR DE PAQUETES ETHERNET

Se implementará un contador de paquetes Ethernet. Este contador contará el número de paquetes ethernet válidos y direccionados localmente.

Para ello, habrá que añadir un microbloque que reciba los paquetes del microbloque Ethernet, incremente un contador, y envíe el paquete al microbloque IP. El nuevo microbloque se llamará CountEth y la nueva aplicación será la mostrada en la figura 4.7:

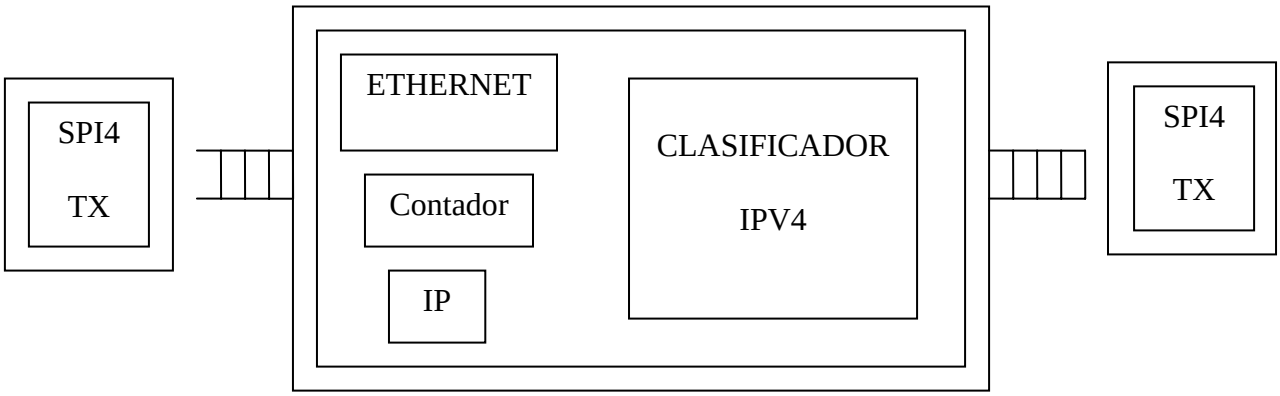


Figura 4.7– Diagrama de bloques. Procesamiento de Paquetes en un único Hilo implementando contador de paquetes Ethernet

1. Habrá que crear una carpeta llamada "counteth" bajo la dirección C:\IXA_SDK_4.3\src\building_blocks. Dentro de la carpeta "counteth", se crearán los subdirectorios "incluye" y "microc". Los archivos *.c estarán en la carpeta "microc" y los archivos *.h, en la carpeta "incluye", aunque en este caso bastará con crear sólo un archivo.

2. Creamos el archivo **counteth.c** dentro de la carpeta microc:

```
extern __declspec(gp_reg) int    dlNextBlock;

INLINE void counteth ( void )

{
//comprobamos si este paquete es para nosotros
if ( dlNextBlock != BID_COUNT )

return ;

sram_incr( ( volatile void _declspec(sram) * )( COUNT_SRAM_ADDR ) ) ;

//enviamos el paquete al siguiente bloque
dlNextBlock = ETHERNET_VALIDATE_LOCAL ;
return ;
}
```

3. Hay que añadir un ID para el contador. En el archivo **dl_system.h** se añadirá para el microbloque contador:

```
/* Contador */
#define BID_COUNT     0x2A
/* Contador */
```

4. También hay que especificar la dirección SRAM donde el microbloque incrementará el contador de paquetes. Se añadirá las siguientes líneas en el archivo **dl_system.h**:

```
/* Contador */
#define COUNT_SRAM_ADDR     0x40300200
/* Contador */
```

5. En el proyecto original, el microbloque Ethernet enviaba los paquetes al microbloque IP. Ahora los paquetes deberían enviarse al microbloque CountEth. Para ello se modificará el archivo **ethernet.c** como sigue:

```
void ethernet_validate()
{
…

// Check to see if the packet is locally addressed
if (header.destination_addr_hi32 !=                    control-
>device_addr_hi32 &&
        header.destination_addr_lo16 !=                control-
>device_addr_lo16)
{
        dlNextBlock = ETHERNET_VALIDATE_OTHER;
                return;
}

//dlNextBlock = ETHERNET_VALIDATE_LOCAL;
 /* Contador */
 dlNextBlock = BID_COUNT;
  /* Contador */
return;
}
```

6. Para llamar a la función **counteth()**, habrá que modificar el dispatch loop **process_dl.c** para que la aplicación llame a esta función después de evaluar que el paquete es un paquete Ethernet válido y que está direccionado localmente y antes de llamar al microbloque IP:

```
while (1)
{
// Dequeue a packet from the rx task
dl_source();
if (dlBufHandle.value == 0)
{
        continue;
}

// Verify that this packet is an acceptable
// Ethernet packet and that it is locally
```

```
// addressed.
ethernet_validate();

 /* contador */
counteth();
 /* contador */

…
}
```

7. Por último, hay que incluir el archivo **counteth.c** en el archivo **process_dl.c**:

```
#include        "dl_system.h"
#include        "system_init.h"
#include        "dispatch_loop.h"
#include        "dl_source.h"
#include        "dl_buf.c"
#include        "dl_meta.h"
#include        "ethernet.h"
#include        "ipv4_five_tuple_class.h"
#include        "red.h"

/* Contador */
#include "counteth.c"
/* Contador */
```

8. Para poder construir el proyecto, habrá que añadir el archivo **counteth.c** al proyecto haciendo clic en **Project** -> **Insert Compiler Source Files….**

9. Hacer clic en **Build** -> **Settings…** y seleccionar la pestaña **General** para añadir la ruta para incluir counteth.c.

10. **Build** -> **Rebuild** para construir la aplicación y ver que no produce ningún error.

4.2.1 DEPURAR LA APLICACIÓN

Hay que modificar el script **single_thread_proc.ind** añadiendo la siguiente línea para que la dirección SRAM para el microbloque contador se inicialice:

```
ps_start_packet_receive();
init_sram(0x0,0x40300200,0x40300300);
…
```

1. Primero, asignaremos el tráfico. Hacemos clic en **Simulation** -> **Packet Simulation Options** -> **Traffic Assignment** -> **Assign Input…** y añadiremos como único flujo de tráfico Packet flow 1.strm.

2. Depurar la aplicación, bien haciendo clic sobre el botón de depurar o desde **Debug** -> **Start Debugging**

3. Configurar un *watch point* en la dirección SRAM 0x40300200 para ver que el contador se está incrementando. Para ello, primero habrá que visualizar la pantalla de Memory

Watch haciendo clic en **View** -> **Debug Windows** -> **Memory Watch**. Haciendo clic en **Add Watch…** insertaremos el *watch point* en dicha dirección. Podemos verlo en la figura 4.8.

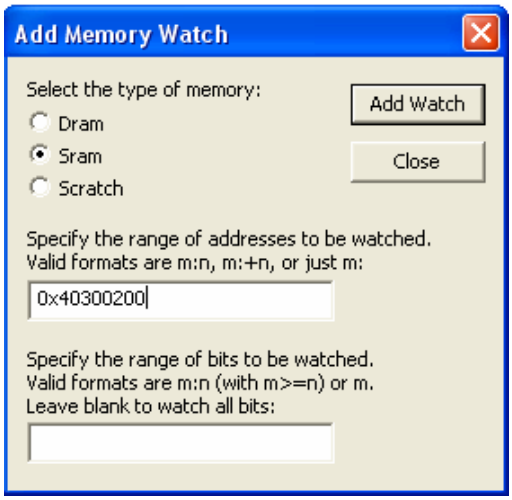


Figura 4.8– Añadir Watch Point en la dirección 0x40300200 de la SRAM

4. Buscamos esa dirección e insertamos un *breakpoint* haciendo clic con el botón derecho sobre la dirección y seleccionando **Set Break on Any Change**. De esta manera, cada vez que se modifique el valor guardado en la dirección 0x40300200 de la SRAM, el WorkBench nos avisará.

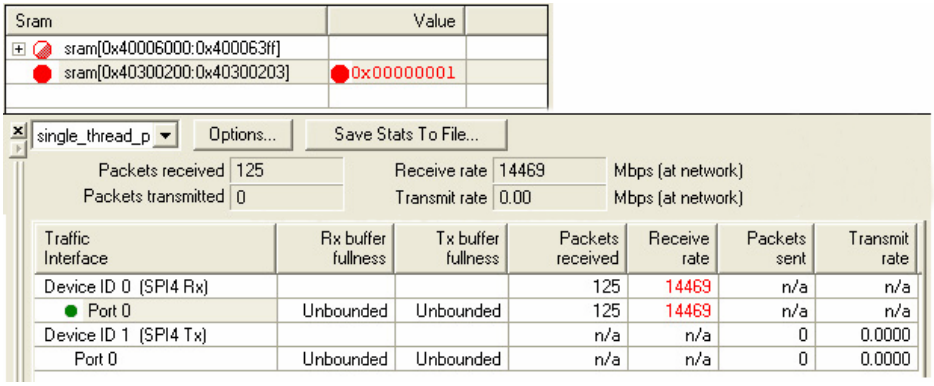
5. Hacer clic en **Debug** -> **Run Control** -> **Go**.

Estado de simulación de paquetes:

La tabla 4.1 recoge los valores tomados durante siete simulaciones. La figura 4.9 muestra un ejemplo del estado de simulación de paquetes recogida en la simulación 1.

Simulación	Micromotores	Paquetes recibidos	Contador de paquetes Ethernet	Paquetes transmitidos
1	6567	125	0x00000001	0
2	10389	145	0x00000002	1
3	11211	152	0x00000003	2
4	18117	160	0x00000004	3
5	70031	270	0x00000011	16
6	98188	330	0x00000018	23
7	98463	331	0x00000019	44

Tabla 4.1– Estado de Simulación de paquetes



Sram	Value
sram[0x40006000:0x400063ff]	
sram[0x40300200:0x40300203]	0x00000001

single_thread_p | Options... | Save Stats To File...

Packets received 125 | Receive rate 14469 Mbps (at network)

Packets transmitted 0 | Transmit rate 0.00 Mbps (at network)

Traffic Interface	Rx buffer fullness	Tx buffer fullness	Packets received	Receive rate	Packets sent	Transmit rate
Device ID 0 (SPI4 Rx)			125	14469	n/a	n/a
Port 0	Unbounded	Unbounded	125	14469	n/a	n/a
Device ID 1 (SPI4 Tx)			n/a	n/a	0	0.0000
Port 0	Unbounded	Unbounded	n/a	n/a	0	0.0000

Figura 4.9– Estado de Simulación de paquetes de la simulación 1

Estadísticas del rendimiento:

La figura 4.10 recoge las estadísticas del rendimiento de los tres micromotores que participan en la depuración de la aplicación.

	Active	Rate
Chip [single_thread_proc]		
Microengine 0:0	18.51%	259.20 Mips
Microengine 0:1	10.32%	144.44 Mips
Microengine 0:2	0.00%	0.00 Mips
Microengine 0:3	0.00%	0.00 Mips
Microengine 0:4	0.00%	0.00 Mips
Microengine 0:5	0.00%	0.00 Mips
Microengine 0:6	0.00%	0.00 Mips
Microengine 0:7	0.00%	0.00 Mips
Microengine 1:0	0.00%	0.00 Mips
Microengine 1:1	0.00%	0.00 Mips
Microengine 1:2	0.00%	0.00 Mips
Microengine 1:3	0.00%	0.00 Mips
Microengine 1:4	0.00%	0.00 Mips
Microengine 1:5	0.00%	0.00 Mips
Microengine 1:6	0.00%	0.00 Mips
Microengine 1:7	5.33%	74.56 Mips
Total		478.20 Mips

Figura 4.10– Estadísticas del Rendimiento

Lista de eventos:

La figura 4.11 recoge los eventos que tienen lugar en distintos ciclos de ejecución de la aplicación.

single_thread_p | Filtering... | 97464 | 98463

Cycle	Thread	Type	SubType	Attributes
97902	(0:1) Thread8	Reference	DRAM read	PC=46, Start Address=0x0004C000, End Address=0x0004C007, Longwords=2
97946	(0:0) Thread0	Reference	MSF fast write	PC=81, Start Address=0x003A0044, End Address=0x003A0047, Longwords=1
97956	(0:0) Thread0	Reference	Scratch put	PC=98, Ring=0, Start Address=<unknown>, End Address=<unknown>, Longwords=4
98029	(0:0) Thread0	Reference	MSF fast write	PC=34, Start Address=0x40000030, End Address=0x40000033, Longwords=1
98151	(1:7) Thread120	Reference	MSF write	PC=100, Start Address=0x000018B8, End Address=0x000018BF, Longwords=2
98151	(1:7) Thread120	Packet	Transmitting	Packet ID=0x00800018, Device=1, Port=0
98151	(1:7) Thread120	Packet	Associate Memory	Packet ID=0x00800018, Region=MSF, Address=0x000005C0, Longwords=16
98151	(1:7) Thread120	Packet	M-Packet Transmitting	Packet ID=0x00800018, Device=1, Port=0
98201	(0:1) Thread8	Reference	SRAM read	PC=58, Start Address=0x00006000, End Address=0x00006007, Longwords=2
98216	(1:7) Thread120	Reference	SRAM get	PC=47, Ring=1, Start Address=<unknown>, End Address=<unknown>, Longwords=4
98244	(0:0) Thread0	Reference	DRAM rbuf read	PC=73, Start Address=0x000AB800, End Address=0x000AB81F, Longwords=8
98257	(1:7) Thread120	Packet	M-Packet Transmitted	Packet ID=0x00800018, Device=1, Port=0
98257	(1:7) Thread120	Packet	Transmitted	Packet ID=0x00800018, Device=1, Port=0
98302	(0:0) Thread0	Reference	MSF fast write	PC=81, Start Address=0x00460044, End Address=0x00460047, Longwords=1
98320	(0:0) Thread0	Reference	MSF fast write	PC=34, Start Address=0x40000030, End Address=0x40000033, Longwords=1
98351	(0:1) Thread8	Reference	SRAM increment	PC=81, Start Address=0x40300200, End Address=0x40300203, Longwords=1
98358	(0:1) Thread8	Reference	DRAM read	PC=88, Start Address=0x0004C008, End Address=0x0004C00F, Longwords=2
98365	(1:7) Thread120	Reference	SRAM get	PC=47, Ring=1, Start Address=<unknown>, End Address=<unknown>, Longwords=4
98404	(0:0) Thread0	Reference	DRAM rbuf read	PC=73, Start Address=0x000AB820, End Address=0x000AB83F, Longwords=8
98462	(0:0) Thread0	Reference	MSF fast write	PC=81, Start Address=0x00040044, End Address=0x00040047, Longwords=1

Figura 4.11– Lista de Eventos

Lista de paquetes:

La figura 4.12 muestra un ejemplo de un paquete transmitido y validado.

Packet List

single_thread_p

Packet ID	Tracking Type	Traffic Interface	Status	Disposition
00800018	Automatic Transmit	Device=1, Port=0	Transmitted/Validated	INVALID - No Stack Configured

Figura 4.12– Paquete validado y transmitido

Historia:

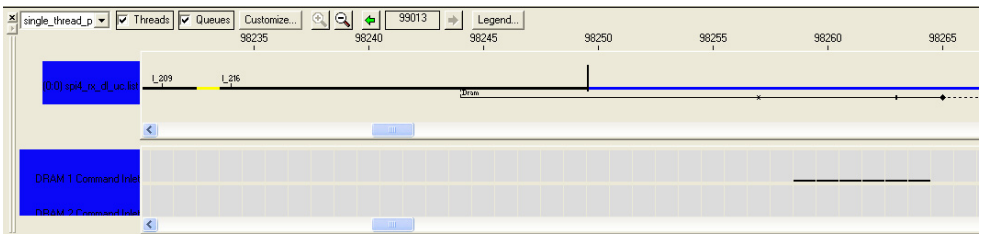


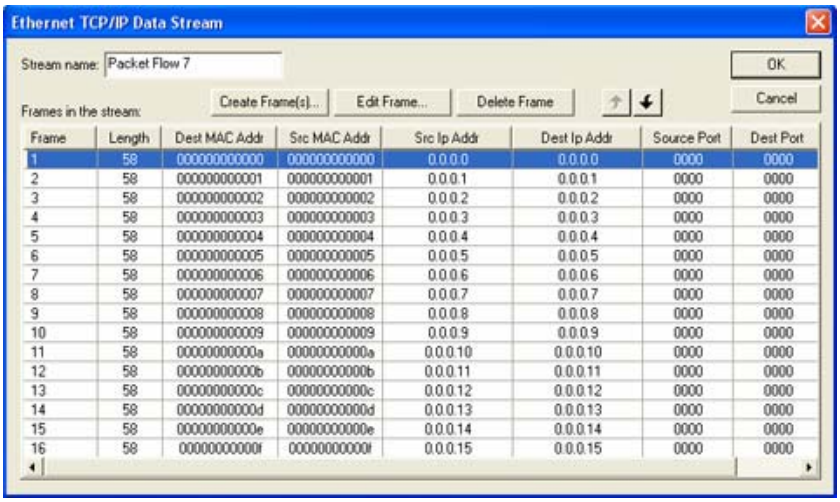
Figura 4.13– Historia

El tráfico de datos que se está generando es Ethernet TCP/IP, por lo que es lógico pensar que todos los paquetes Ethernet válidos y direccionados localmente son a su vez IP. Por tanto, cada vez que el contador incrementa el número de paquetes Ethernet, la siguiente vez que incrementa se está transmitiendo el anterior paquete, y así sucesivamente. Podemos apreciarlo en la figura 4.13.

4.3 CONTADOR DE PAQUETES INVÁLIDOS

Se entiende por paquetes inválidos aquellos paquetes que no cumplen con el tamaño de trama Ethernet, es decir, su tamaño no está comprendido entre 64 bytes y 1522 bytes. Para ello, habrá que seguir los siguientes pasos:

1. Crear flujo de tráfico de tipo Ethernet TCP/IP con un tamaño de trama de 58 bytes. La figura 4.14 recoge las tramas Ethernet TCP/IP que hemos creado.



Frame	Length	Dest MAC Addr	Src MAC Addr	Src Ip Addr	Dest Ip Addr	Source Port	Dest Port
1	58	000000000000	000000000000	0.0.0.0	0.0.0.0	0000	0000
2	58	000000000001	000000000001	0.0.0.1	0.0.0.1	0000	0000
3	58	000000000002	000000000002	0.0.0.2	0.0.0.2	0000	0000
4	58	000000000003	000000000003	0.0.0.3	0.0.0.3	0000	0000
5	58	000000000004	000000000004	0.0.0.4	0.0.0.4	0000	0000
6	58	000000000005	000000000005	0.0.0.5	0.0.0.5	0000	0000
7	58	000000000006	000000000006	0.0.0.6	0.0.0.6	0000	0000
8	58	000000000007	000000000007	0.0.0.7	0.0.0.7	0000	0000
9	58	000000000008	000000000008	0.0.0.8	0.0.0.8	0000	0000
10	58	000000000009	000000000009	0.0.0.9	0.0.0.9	0000	0000
11	58	00000000000a	00000000000a	0.0.0.10	0.0.0.10	0000	0000
12	58	00000000000b	00000000000b	0.0.0.11	0.0.0.11	0000	0000
13	58	00000000000c	00000000000c	0.0.0.12	0.0.0.12	0000	0000
14	58	00000000000d	00000000000d	0.0.0.13	0.0.0.13	0000	0000
15	58	00000000000e	00000000000e	0.0.0.14	0.0.0.14	0000	0000
16	58	00000000000f	00000000000f	0.0.0.15	0.0.0.15	0000	0000

Figura 4.14– Flujo de Datos para paquetes Ethernet TCP/IP

2. Asignar al WorkBench las tramas anteriormente creadas y las de tipo Ethernet válidas utilizadas en la primera simulación: **Simulation** -> **Packet Simulation Options…** -> **Traffic Assignment** -> **Assign Input…** y añadir Packet Flow 1.strm y Packet Flow 7.strm como único tráfico.

3. Modificar el código:

   a) **ethernet.c**:

```
void ethernet_validate()
{
…

    // Check the packet length
    if (dlMeta.bufferSize < MIN_ETHERNET_LENGTH ||
            dlMeta.bufferSize > MAX_ETHERNET_LENGTH)
    {
//dlNextBlock = ETHERNET_VALIDATE_INVALID;
            /* Contador */
            dlNextBlock = BID_COUNT;
            /* Contador */
            return;
    }
    …
```

```
    dlNextBlock = ETHERNET_VALIDATE_LOCAL;
    /* Contador */
            //dlNextBlock = BID_COUNT;
     /* Contador */
    return;
}
```

b) **counteth.c**:

```
extern __declspec(gp_reg) int        dlNextBlock;

INLINE void counteth ( void )

{
//comprobamos si este paquete es para nosotros
    if ( dlNextBlock != BID_COUNT )

 return ;

sram_incr( ( volatile void _declspec(sram) * )( COUNT_SRAM_ADDR ) ) ;

//enviamos el paquete al siguiente bloque
dlNextBlock = ETHERNET_VALIDATE_INVALID ;
return ;
}
```

4. El resto del código no hace falta modificarlo. Construiremos la aplicación para asegurarnos de que no produce ningún error.

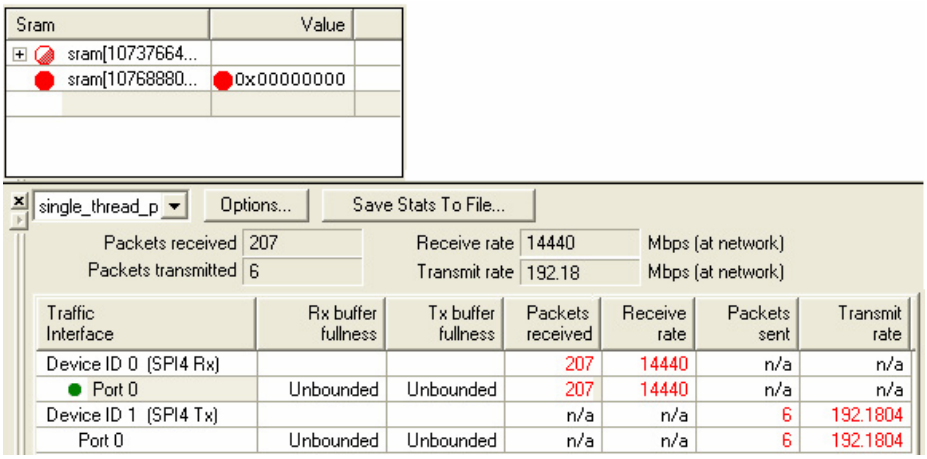
5. Iniciaremos el proceso de depurado. No hará falta añadir ningún watch point en la dirección SRAM 0x403002, ya que se añadió en la primera simulación y seguirá habilitada.

Estado de simulación de paquetes:

La tabla 4.2 recoge los valores tomados durante siete simulaciones. La figura 4.15 muestra un ejemplo del estado de simulación de paquetes recogida en la simulación 2.

Simulación	Micromotorores	Paquetes recibidos	Contador de paquetes inválidos	Paquetes transmitidos
1	16447	162	0x00000000	2
2	32467	207	0x00000000	6
3	44318	239	0x00000001	9
4	57282	270	0x00000011	12

Tabla 4.2– Estado de Simulación de Paquetes



Sram	Value
sram[10737664...	
sram[10768880...	0x00000000

single_thread_p | Options... | Save Stats To File...

Packets received 207 | Receive rate 14440 | Mbps (at network)
Packets transmitted 6 | Transmit rate 192.18 | Mbps (at network)

Traffic Interface	Rx buffer fullness	Tx buffer fullness	Packets received	Receive rate	Packets sent	Transmit rate
Device ID 0 (SPI4 Rx)			207	14440	n/a	n/a
Port 0	Unbounded	Unbounded	207	14440	n/a	n/a
Device ID 1 (SPI4 Tx)			n/a	n/a	6	192.1804
Port 0	Unbounded	Unbounded	n/a	n/a	6	192.1804

Figura 4.15– Simulación de paquetes de la simulación 2

Como se puede observar, al principio de ejecutar la simulación, se transmiten paquetes sin haber recogido todavía ninguno inválido. Será más adelante cuando le lleguen paquetes inválidos.

Estadísticas del rendimiento:

La figura 4.16 recoge las estadísticas del rendimiento de los tres micromotores que participan en la depuración de la aplicación.

	Active	Rate
Chip [single_thread_proc]		
Microengine 0:0	17.28%	241.87 Mips
Microengine 0:1	10.44%	146.15 Mips
Microengine 0:2	0.00%	0.00 Mips
Microengine 0:3	0.00%	0.00 Mips
Microengine 0:4	0.00%	0.00 Mips
Microengine 0:5	0.00%	0.00 Mips
Microengine 0:6	0.00%	0.00 Mips
Microengine 0:7	0.00%	0.00 Mips
Microengine 1:0	0.00%	0.00 Mips
Microengine 1:1	0.00%	0.00 Mips
Microengine 1:2	0.00%	0.00 Mips
Microengine 1:3	0.00%	0.00 Mips
Microengine 1:4	0.00%	0.00 Mips
Microengine 1:5	0.00%	0.00 Mips
Microengine 1:6	0.00%	0.00 Mips
Microengine 1:7	5.26%	73.70 Mips
Total		461.72 Mips

Figura 4.16– Estadísticas del Rendimiento

Lista de eventos:

La figura 4.17 muestra los diferentes valores del PC en cada ciclo de procesamiento, las direcciones de comienzo y fin, y el tipo de evento y su hilo donde ocurre.

single_thread_p | Filtering... | 56283 | 57282

Cycle	Thread	Type	SubType	Attributes
56785	(0:1) Thread8	Reference	Scratch read	PC=375, Start Address=0x00000010, End Address=0x00000013, Longwords=1
56818	(0:0) Thread0	Reference	DRAM rbuf read	PC=73, Start Address=0x00088000, End Address=0x0008803F, Longwords=16
56842	(0:1) Thread8	Reference	Scratch read	PC=379, Start Address=0x00000014, End Address=0x00000017, Longwords=1
56875	(1:7) Thread120	Reference	SRAM get	PC=47, Ring=1, Start Address=<unknown>, End Address=<unknown>, Longwords=4
56886	(0:0) Thread0	Reference	MSF fast write	PC=81, Start Address=0x00020044, End Address=0x00020047, Longwords=1
56906	(0:1) Thread8	Reference	Scratch increment	PC=389, Start Address=0x00000010, End Address=0x00000013, Longwords=1
56909	(0:0) Thread0	Reference	MSF fast write	PC=34, Start Address=0x40000030, End Address=0x40000033, Longwords=1
56911	(0:1) Thread8	Reference	SRAM put	PC=394, Ring=1, Start Address=<unknown>, End Address=<unknown>, Longwords=4
57023	(1:7) Thread120	Reference	SRAM get	PC=47, Ring=1, Start Address=<unknown>, End Address=<unknown>, Longwords=4
57086	(0:1) Thread8	Reference	Scratch get	PC=426, Ring=0, Start Address=<unknown>, End Address=<unknown>, Longwords=4
57126	(0:0) Thread0	Reference	DRAM rbuf read	PC=73, Start Address=0x00088800, End Address=0x0008883F, Longwords=16
57171	(0:1) Thread8	Reference	SRAM increment	PC=83, Start Address=0x40300200, End Address=0x40300203, Longwords=1
57184	(1:7) Thread120	Reference	Scratch decrement	PC=51, Start Address=0x00000010, End Address=0x00000013, Longwords=1
57185	(1:7) Thread120	Reference	Scratch read	PC=52, Start Address=0x00000010, End Address=0x00000013, Longwords=1
57186	(0:1) Thread8	Reference	Scratch get	PC=426, Ring=0, Start Address=<unknown>, End Address=<unknown>, Longwords=4
57194	(0:0) Thread0	Reference	MSF fast write	PC=81, Start Address=0x00320044, End Address=0x00320047, Longwords=1
57204	(0:0) Thread0	Reference	Scratch put	PC=98, Ring=0, Start Address=<unknown>, End Address=<unknown>, Longwords=4
57261	(1:7) Thread120	Reference	Scratch write	PC=57, Start Address=0x00000018, End Address=0x0000001B, Longwords=1
57267	(0:1) Thread8	Reference	SRAM increment	PC=83, Start Address=0x40300200, End Address=0x40300203, Longwords=1
57277	(0:0) Thread0	Reference	MSF fast write	PC=34, Start Address=0x40000030, End Address=0x40000033, Longwords=1

Figura 4.17– Lista de eventos

Lista de paquetes:

La transmisión de un paquete se ve reflejada en la figura 4.18. A continuación, la figura 4.19 muestra otro paquete que ha sido validado y transmitido.

Packet List

single_thread_p

Packet ID	Tracking Type	Traffic Interface	Status	Disposition
0080000F	Automatic Transmit	Device=1, Port=0	Transmitted	

Figura 4.18– Transmisión del paquete

Packet List

single_thread_p

Packet ID	Tracking Type	Traffic Interface	Status	Disposition
0080000F	Automatic Transmit	Device=1, Port=0	Transmitted/Validated	INVALID - No Stack Configured

Figura 4.19– Validación del paquete

Historia:

En la figura 4.20 se presenta la entrada de un paquete en el buffer de cola y su transmisión unos micromotores más tarde. En la figura 4.21 apreciaremos un hilo abortado debido al incremento del contador (línea amarilla).

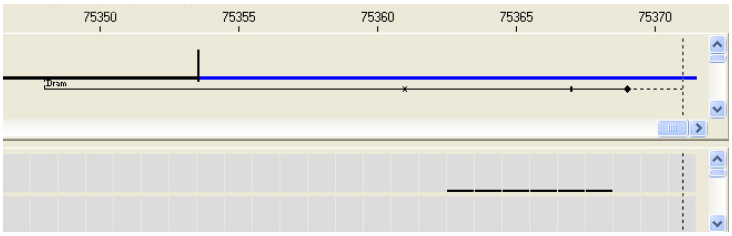


Figura 4.20– Lista de eventos: entrada de un paquete en cola y transmisión del paquete

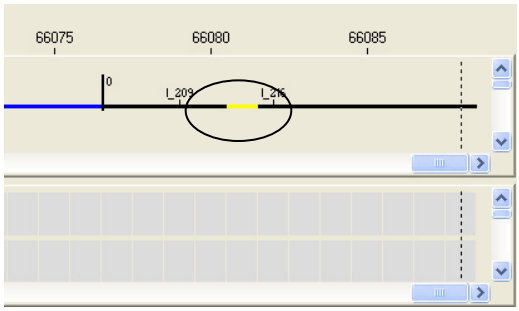


Figura 4.21– Lista de eventos: hilo abortado e incremento del contador

Estado de la cola:

En la figura 4.22 podemos observar como en la SRAM 1 A0 entra un valor. Éste valor es el incremento del contador.

Queue	# Entries	Command	Address	Thread	PC	Cycle	# Lwords	Sig Done
DRAM								
DRAM 0 Command Inlet	0							
DRAM 1 Command Inlet	0							
DRAM 2 Command Inlet	0							
SRAM								
SRAM 0 Command Inlet A0	0							
SRAM 0 Command Inlet A1	0							
SRAM 1 Command Inlet A0	1							
SRAM 1 Command Inlet A1	0							
SRAM 2 Command Inlet A0	0							
SRAM 2 Command Inlet A1	0							
SRAM 3 Command Inlet A0	0							
SRAM 3 Command Inlet A1	0							
Microengines								
Microengine 0:0 Command	0							
Microengine 0:1 Command	1							
Microengine 0:2 Command	0							
Microengine 0:3 Command	0							
Microengine 0:4 Command	0							
Microengine 0:5 Command	0							
Microengine 0:6 Command	0							

Figura 4.22– Estado de la cola: entrada del paquete en el bloque Contador

CAPÍTULO 5
CONCLUSIONES Y LÍNEAS FUTURAS

5.1 CONCLUSIONES Y LÍNEAS FUTURAS

El desarrollo de este proyecto surge como necesidad de aprender a utilizar los *network processors* para su implantación en futuras aplicaciones que nos solucionen los problemas vigentes que otros dispositivos no nos dan.

El primer paso fue investigar qué es realmente un *network processor*. Para ello se utilizaron diversas fuentes de información como libros, y búsquedas de documentos en internet. Para llegar a una mejor comprensión, se partió de estudiar lo que había anteriormente hasta llegar a los *network processors*.

Una vez entendido qué es un *network processor* y ver como estos dispositivos daban solución a problemas que cualquier otro dispositivo no era capaz de darlo, se decidió estudiar algunos tipos y elegir el que mejor se adecuaba a nuestras necesidades. Después de estudiarlos a fondo y con detenimiento, Intel era el que mejores prestaciones aportaba con vistas de futuro. Intel proporcionaba un SDK gratuito y aplicaciones ya desarrolladas que nos podían valer para entender el funcionamiento a través de un *network processor* y modificar estas aplicaciones según nuestros intereses.

Se instaló el Developer WorkBench sobre la plataforma Windows, aunque también soportaba la de Linux, y se estudió su funcionamiento ejecutando varias aplicaciones. Vimos que el lenguaje de programación que utilizaba el WorkBench soportaba ensamblador y C, decantándonos por C por ser un lenguaje más fácil de utilizar y del cual poseíamos más conocimiento. Este programa nos mostraba una serie de estadísticas que nos permitió entender qué es lo que realmente hacía la aplicación. Con todo esto, se adquirieron bastantes conocimientos en cuanto al funcionamiento interno del *network processor*, a parte de que su entendimiento era fácil por su modularidad, flexibilidad y sencillez.

Por último, se eligió una aplicación ya desarrollada consistiendo en el procesamiento de paquetes en un único hilo. Parecía la más sencilla para empezar a estudiar y para posteriormente modificarla. Estudiamos la aplicación sobre papel para ver las relaciones entre los distintos archivos por los que estaba formada la aplicación. Una vez asegurado su entendimiento y funcionamiento, simulamos esta aplicación y mediante las estadísticas que el propio programa nos proporcionaba vimos como devolvía los resultados esperados. Para ello, se crearon distintos flujos de tráfico y vimos el por qué con unos transmitía más, con otros transmitía menos, y con otros no transmitía nada.

Llegados a este punto, decidimos implementar un sencillo contador de paquetes Ethernet. Para ello creamos un fichero de nombre counteth.c que implementaba el mecanismo. Fue fácil modificar el código ya que sólo había que tener en cuenta el punto exacto donde había que implementar el contador para que el anterior microbloque llamara al nuevo bloque contador en vez del bloque IP que antes llamaba. Hubo que definir una ID para este contador y asignarle un espacio de memoria en la SRAM para que a la hora de depurar el código viéramos como iba incrementándose dicho contador. Insertamos una línea de código en el main() para llamar a la

función counteth(). Antes de compilar el código y cerciorarnos de que no produce ningún error, había que añadir el fichero counteth.c como parte de la aplicación, y añadir en el script de inicio una línea para inicializar la dirección de la SRAM para el contador. Para esta parte simulamos con tráfico de tipo Ethernet para comprobar que todos los paquetes Ethernet pasaban al contador, incrementándose éste cuando le llegaba, y transmitiéndose unos micromotores después. El siguiente paso fue modificar la aplicación para ver cuántos paquetes inválidos se descartaban. Sólo había que cambiar las líneas de código y ponerlas en la parte correspondiente del código. Efectivamente, vimos como el contador se incrementaba cada vez que un paquete había sido descartado por ser inválido, y que dicho paquete no se transmitía.

Como línea de futuro se plantea:

- Ampliar la investigación para desarrollar una aplicación, por ejemplo DiffServ, y que se lleve a cabo no sólo en el entorno de simulación, sino también en el entorno de la realidad.

- Plantear posibles soluciones a los problemas actuales y construir la aplicación necesaria con el software aportado.

APÉNDICE A
CÓDIGO DE LA APLICACIÓN

dispatch_loop.h

```
#ifndef __DISPATCH_LOOP__H__
#define __DISPATCH_LOOP__H__


#endif                  // __DISPATCH_LOOP__H__
```

dl_buf.c

```
#ifndef __DL_BUF_C__
#define __DL_BUF_C__
#include <dl_system.h>
#include <dl_buf.h>

INLINE void
Dl_BufInit()
{
        ixp_buf_freelist_create(AVLBL_PKT_BUFFERS, BUF_FREE_LIST0, BUF_POOL);
}


INLINE void
Dl_BufAlloc(
        __declspec(sram_read_reg)dl_buf_handle_t* bufHandle,
        pool_t          freeList,
        unsigned int    dBase,
        unsigned int    dSize,
        unsigned int    sBase,
        unsigned int    sSize,
        SIGNAL*         reqSig,
        unsigned int    sigAction,
        queue_t                 Q_OPTION)
{

        ixp_buf_alloc((void *) bufHandle, freeList, dBase, dSize, sBase, sSize, reqSig, \
                                        sigAction, ___);
}

INLINE void
Dl_BufFree(
   dl_buf_handle_t      bufHandle,
   pool_t               freeList,
   unsigned int         dBase,
```

```
    unsigned int        dSize,
    unsigned int        sBase,
    unsigned int        sSize)
{


ixp_buf_free(bufHandle, freeList, dBase, dSize, sBase, sSize);
}

INLINE unsigned int
Dl_BufGetDesc(dl_buf_handle_t bufHandle)
{
        //      Mask off ms 8 bits of buf_handle.
        //      The ls 24 bits of buf_handle represent the sram address in LW's.
        //      Convert ls 24 bits of buf_handle from LW address to byte address.
        //      Note: Only Channel 0 of SRAM is supported by this function.
return (bufHandle.lw_offset << 2);
}

INLINE unsigned int
Dl_BufGetData(dl_buf_handle_t bufHandle)
{
        unsigned int sdramOffset;

        //      Mask off ms 8 bits of buf_handle.
        //      The ls 24 bits of buf_handle represent the sram address in LW's.
        //      Convert ls 24 bits of buf_handle from LW address to byte address (shift by 2)
        //      Convert this sram byte address to the corres. dram address (shift by
        MULT_FACTOR)
        //      Note: Only SRAM channel 0 is supported by this function.

sdramOffset = (bufHandle.lw_offset) << (MULT_FACTOR + 2);

#if (DL_REL_BASE > 0)
        sdramOffset +=DL_REL_BASE;
#endif

        return sdramOffset;
}

INLINE unsigned int
Dl_BufGetDataFromMeta(unsigned int sramOffset)
{
        unsigned int sdramOffset;

        // Convert sram byte address to the corres. dram address (shift by MULT_FACTOR)
        sdramOffset = sramOffset << MULT_FACTOR;

#if (DL_REL_BASE > 0)
        sdramOffset +=DL_REL_BASE;
#endif

        return sdramOffset;
}
```

```
INLINE void
Dl_BufDrop(dl_buf_handle_t bufHandle)
{
        Dl_BufFree(bufHandle, BUF_FREE_LIST0, BUF_POOL);
}
INLINE void
Dl_BufDropChain(dl_buf_handle_t bufHandle, dl_buf_handle_t bufEopHandle)
{

#if     TO_BE_INCLUDED_LATER

        /*
         *      This function requires the followinf #defines.
         *      DL_DROP_RING_FULL and a value for dl_drop_ring.
         */

        __declspec(sram_write_reg) int sWrite[3];
        SIGNAL sigScratchWrite;
        unsigned int dropRing;

        //      Check if the ring this full.

        while (inp_state_test(DL_DROP_RING_FULL))
                ;

        sWrite[0] = bufHandle;
        sWrite[1] = bufEopHandle;
        sWrite[2] = QM_DROP_QUEUE | (1 << 31);
        dropRing  = DL_DROP_RING << 2;

        scratch_put_ring(sWrite, (volatile void __declspec(scratch) *)dropRing, 3, sig_done,
&sigScratchWrite);

#endif

}


#endif // __DL_BUF_C__
```

dl_buf.h

```
#ifndef __DL_BUF_H__
#define __DL_BUF_H__

#include <dl_system.h>
#include <ixp_buf.c>

/*
 *      These definitions are provided for backward compatibilty to enable smooth
```

```
 *      transition. They will be removed soon
 */
#ifdef  BACKWARD_COMPATIBILITY

#define dl_buf_init                     Dl_BufInit
#define dl_buf_alloc                    Dl_BufAlloc
#define dl_buf_free                     Dl_BufFree
#define dl_buf_get_desc                 Dl_BufGetDesc
#define dl_buf_get_data                 Dl_BufGetData
#define dl_buf_get_data_from_meta       Dl_BufGetDataFromMeta
#define dl_buf_drop                     Dl_BufDrop
#define dl_buf_drop_chain               Dl_BufDropChain

#endif

//      SRAM-QArray ID for enqueing/dequeing pkt buffers. (i.e maintinaing the free list)
//      Right now there is only one free list. But there may be multiple in the near future.

#define BUF_FREE_LIST0              BUF_QARRAY_BASE

//      Number of packet buffers

#define  NO_OF_PKT_BUFFERS      BUF_SRAM_SIZE/META_DATA_SIZE

//      Some sanity check. Check if the specified SDRAM memory is of suffcient size
//      to hold all buffers

#if BUFFER_SIZE == 0
#error "BUFER SIZE NOT DEFINED"
#endif

#define DRAM_COMPUTE            BUF_SDRAM_SIZE/BUFFER_SIZE

#if     (DRAM_COMPUTE != NO_OF_PKT_BUFFERS)
#error  "The amount of memory allocated in SRAM for packet buffers doesn't match with\
the SDRAM size. Check BUF_SRAM_SIZE and BUF_SDRAM_SIZE. Also check
META_DATA_SIZE and BUFFER_SIZE"
#endif

#undef  DRAM_COMPUTE

#define AVLBL_PKT_BUFFERS       NO_OF_PKT_BUFFERS


//      The combination of BUF_SDRAM_BASE, BUFFER_SIZE, BUF_SRAM_BASE,
META_DATA_SIZE
//      specify a particular buffer pool. The IXP buf functions need it this way and it needs
//      it in Long Words.
//      NOTE: Changed: The IXP buf functions needs it in bytes (not Long Words)!
//      BUt we still use *_LW for holding temp. values. But all values are in bytes.

#define BUF_SDRAM_BASE_LW       BUF_SDRAM_BASE
#define BUFFER_SIZE_LW          BUFFER_SIZE
#define BUF_SRAM_BASE_LW        BUF_SRAM_BASE
#define META_DATA_SIZE_LW       META_DATA_SIZE
```

```
// Specifying a base of address of 0 for SRAM (and DRAM) simplifies the arithmatic required
// to translate the handle to offset. (it saves one instruction. See dl_buf_get_desc)
// However, dl_buf_alloc will return 0 for a null buffer (i.e no buffer available) and it may
// not be possible to differentiate between a null buffer and a buffer at address 0.
// So we simply make that one buffer at address 0 unavailable for allocation/freeing, yet
// taking advantage of using a base address of 0. We end up wasting a little memory (1 buffer)
// but that's ok.

#if       (BUF_SRAM_BASE_LW == 0)

#undef  BUF_SRAM_BASE_LW
#define BUF_SRAM_BASE_LW        BUF_SRAM_BASE + META_DATA_SIZE_LW

#undef  BUF_SDRAM_BASE_LW
#define BUF_SDRAM_BASE_LW       BUF_SDRAM_BASE + BUFFER_SIZE_LW

#undef  AVLBL_PKT_BUFFERS
#define AVLBL_PKT_BUFFERS       NO_OF_PKT_BUFFERS - 1

#endif

#define BUF_POOL     BUF_SDRAM_BASE_LW, BUFFER_SIZE_LW,
BUF_SRAM_BASE_LW, META_DATA_SIZE_LW

//        DL_DS_RATIO = Sizeof (SDRAM entry) / Sizeof (SRAM Entry)

#define DL_DS_RATIO          BUFFER_SIZE/META_DATA_SIZE

// DL_REL_BASE = DRAM_BASE - (BUF_SRAM_BASE * DL_DS_RATIO)

#define DL_REL_BASE BUF_SDRAM_BASE - (BUF_SRAM_BASE * DL_DS_RATIO)

// ln(DL_DS_RATIO). i.e 2^MULT_FACTOR = DL_DS_RATIO. This value is required to
//  reduce multiplication in to simple shift operation. The following set of lines
//  computes MULT_FACTOR (at compile time) (XXX - This computation cannot be done with
// 'C' preprocessor. requires the assembler preprocessor)

#define MULT_FACTOR        6

typedef buf_handle_t     dl_buf_handle_t;

/*
 *        These are the Packet Buffer API functions
 */
void Dl_BufInit();
void Dl_BufAlloc(__declspec(sram_read_reg)dl_buf_handle_t* buf_handle,
                        pool_t          free_list,
                        unsigned int  d_base,
                        unsigned int  d_size,
                        unsigned int  s_base,
                        unsigned int  s_size,
                        SIGNAL*     req_sig,
                        unsigned int  sig_action,
                        queue_t        Q_OPTION);
```

```
void Dl_BufFree(dl_buf_handle_t buf_handle,
                pool_t          free_list,
                unsigned int  d_base,
                unsigned int  d_size,
                unsigned int  s_base,
                unsigned int  s_size);
unsigned int Dl_BufGetDesc(dl_buf_handle_t buf_handle);
unsigned int Dl_BufGetData(dl_buf_handle_t buf_handle);
unsigned int Dl_BufGetDataFromMeta(unsigned int sram_offset);

void Dl_BufDrop(dl_buf_handle_t buf_handle);
void Dl_BufDropChain(dl_buf_handle_t buf_handle, dl_buf_handle_t buf_eop_handle);

#endif  // __DL_BUF_H__
```

dl_meta.h

```
#ifndef __DL_META_H__
#define __DL_META_H__

#include <dl_buf.h>

/*
 *      These definitions are provided for backward compatibilty to enable smooth
 *      transition. They will be removed soon
 */

#define dl_meta_load_cache          Dl_MetaLoadCache
#define dl_meta_flush_cache         Dl_MetaFlushCache
#define dl_meta_get_buffer_next     Dl_MetaGetBufferNext
#define init_scratch_ring           InitScratchRing

typedef __declspec(packed) union
{
        struct {
                dl_buf_handle_t bufferNext;     /**< Next buffer in the chain */
                unsigned short  bufferSize;      /**< amount of data currently in buffer */
                unsigned short  offset;          /**< offset in DRAM where data begins */
                unsigned int    packetSize: 16;  /**< amount of data in the chain of buffers */
                unsigned int    freeListId: 4;   /**< Free List to which this buffer belongs to */
                unsigned int    rxStat: 4;       /**< Receive status */
                unsigned int    headerType: 8;   /**< HEader Type: IPv4, IPv6 etc */
                unsigned short  inputPort;       /**< Input port on which packet was received */
                unsigned short  outputPort;
                unsigned int    nextHopId: 16;   /**< Nexthop ID */
                unsigned int    fabricPort: 8;   /**< Blade:Port */
                unsigned int    reserved : 4;    /**< reserved */
                unsigned int    nhidType: 4;     /**< nexthop ID type */
                unsigned int    colorId:4;
                unsigned int    reserved1:4;
                unsigned int    flowId:24;       /**< FLow ID */
                unsigned short  classId;         /**< Class ID */
```

```
                unsigned short  reserved2;
                unsigned int    packetNext;     /**< Next packet in the chain */
                                                /**< (used only in Hierarchical Queuing) */
        };      // end of struct

        unsigned int value[8];                  /**< aggregate for the above fields */

} dl_meta_t;

/*
 *      These are the Meta Data API functions
 */

void Dl_MetaLoadCache(dl_buf_handle_t       bufHandle,
                      SIGNAL*               sig_requested,
                      unsigned int          numLongWords);

void Dl_MetaFlushCache(dl_buf_handle_t      buf_handle,
                       SIGNAL*              req_sig,
                       unsigned int         sig_action,
                       unsigned int         start_lw,
                       unsigned int         num_lw);

void Dl_MetaFlushCache2(dl_buf_handle_t     buf_handle,
                        __declspec(sram_write_reg) dl_meta_t *sram_wxfer,
                        SIGNAL*             req_sig,
                        unsigned int        sig_action,
                        unsigned int        start_lw,
                        unsigned int        num_lw);

void Dl_MetaWrite(dl_buf_handle_t   buf_handle,
                  __declspec(sram_write_reg) dl_meta_t *sram_wxfer,
                  unsigned int      buffer_next,
                  unsigned int      buffer_size,
                  unsigned int      offset,
                  unsigned int      packet_size,
                  unsigned int      free_list_id,
                  unsigned int      rx_stat,
                  unsigned int      header_type,
                  unsigned int      input_port,
                  unsigned int      output_port,
                  unsigned int      next_hop_id,
                  unsigned int      fabric_port,
                  unsigned int      flow_id,
                  unsigned int      calss_id,
                  unsigned int      packet_next,
                  SIGNAL*           req_sig,
                  unsigned int      sig_action,
                  unsigned int      start_lw,
                  unsigned int      num_lw);

void Dl_MetaFlushCacheSkip0(dl_buf_handle_t     buf_handle,
                  SIGNAL*       req_sig,
                  unsigned int  sig_action,
                  unsigned int  num_lw);
```

```
/*      XXX - Is this function really required */

int Dl_MetaGetBufferNext(void);

/*      XXX - Is there a better place where InitScratchRing can be put */

void InitScratchRing(int rbase, int rsize, int ring);


#endif  // __DL_META_H__
```

---

dl_source.c

```
#include "dl_source.h"
#include "dl_buf.c"
#include "dl_meta.h"
#include "scratch_rings.h"
#include "sram_rings.h"

extern dl_buf_handle_t          dlBufHandle;
extern __declspec(gp_reg) int   dlNextBlock;
extern dl_meta_t                dlMeta;

void dl_sink_init()
{
        if (ctx() == 0)
        {
// The sink block code for RX to processing
#ifdef RX_DL
                // Initialize the ring between rx and processing
                scratch_ring_init(
                        RX_TO_PROCESSING_RING,
                        RX_TO_PROCESSING_RING_BASE,
                        RX_TO_PROCESSING_RING_SIZE);

                // Send a singal to processing indicating the ring is ready
                cap_fast_write(
                   (RING_CONSUMER_CTX<<4) |
                   (__signal_number(&rx_ring_ready_sig,
                    RING_CONSUMER_ME)) |
                   (RING_CONSUMER_ME<<7),
                   csr_interthread_sig);
#endif

// The sink block code for processing to tx
#ifdef PROCESSING_DL
                // Fix the value of the incoming ring-ready signal so the
                // rx task can indicate when that ring is created.
                __assign_relative_register((void *)&rx_ring_ready_sig, 2);
```

```
                // Initialize the ring between tx and processing
                sram_ring_init(
                        PROCESSING_TO_TX_SRAM_RING,
                        PROCESSING_TO_TX_SRAM_RING_BASE,
                        PROCESSING_TO_TX_SRAM_RING_SIZE);

                // Send a singal to processing indicating the ring is ready
                cap_fast_write(
                        (RING_CONSUMER_CTX<<4) |
                        (__signal_number(&tx_ring_ready_sig,
                         RING_CONSUMER_ME)) |
                        (RING_CONSUMER_ME<<7),
                        csr_interthread_sig);
#endif

#ifdef TX_DL
                // Fix the value of the incoming ring-ready signal so the
                // rx task can indicate when that ring is created.
                __assign_relative_register((void *)&tx_ring_ready_sig, 2);
#endif
        }
}

void dl_sink()
{
        ring_data_t     r_data;

#ifdef RX_DL
        //      In the case of an exception packet, it needs to
        //  be sent to the core (Xscale)
        //      through a different ring.
        // For now just drop the packet.
        if (dlNextBlock == IX_EXCEPTION)
        {
                Dl_BufDrop(dlBufHandle);
                return;
        }

        // Enqueue the packet on the appropriate
        // scratch ring.  First check if the ring
        // is full, and if so drop the packet.
        if (scratch_ring_full(RX_TO_PROCESSING_RING))
        {
                Dl_BufDrop(dlBufHandle);
                return;
        }

        // Enqueue the handle, length and offset
        // of the packet
        r_data.handle = dlBufHandle;
        r_data.length = dlMeta.bufferSize;
        r_data.offset = dlMeta.offset;
        scratch_ring_put_buffer(
                RX_TO_PROCESSING_RING, r_data);
```

```
#endif

#ifdef PROCESSING_DL
#define __DL_SINK_RING                PROCESSING_TO_TX_RING
        //      In the case of an exception packet, it needs to
        //  be sent to the core (Xscale)
        //      through a different ring.
        // For now just drop the packet.
        if (dlNextBlock == IX_EXCEPTION)
        {
                Dl_BufDrop(dlBufHandle);
                return;
        }

        // Enqueue the handle, length and offset
        // of the packet
        r_data.handle = dlBufHandle;
        r_data.length = dlMeta.bufferSize;
        r_data.offset = dlMeta.offset;
        sram_ring_put_buffer(
        PROCESSING_TO_TX_SRAM_RING, r_data);
#endif

}

void dl_source_init()
{
        if (ctx() == 0)
        {
#ifdef PROCESSING_DL

                // Wait for a singal from RX indicating the
                // incoming ring is ready
                wait_for_all(&rx_ring_ready_sig);

#endif

#ifdef TX_DL

                // Wait for the queue to be set up
                wait_for_all(&tx_ring_ready_sig);

#endif
        }
}

void dl_source()
{
        ring_data_t       r_data;

#ifdef PROCESSING_DL
        // Dequeue a packet from the appropriate ring
        scratch_ring_get_buffer(
                RX_TO_PROCESSING_RING,
                &r_data);
```

```
        dlBufHandle      = r_data.handle;
        dlMeta.bufferSize = r_data.length;
        dlMeta.offset     = r_data.offset;
#endif

#ifdef TX_DL
        // Dequeue a packet from the appropriate ring
        sram_ring_get_buffer(
        PROCESSING_TO_TX_SRAM_RING,
        &r_data);
        dlBufHandle      = r_data.handle;
        dlMeta.bufferSize = r_data.length;
        dlMeta.offset     = r_data.offset;
#endif

}


unsigned int dl_get_sink_size()
{
        return sram_ring_get_size(dlMeta.flowId);
}

unsigned int dl_get_sink_empty_timestamp()
{
        return sram_ring_get_empty_timestamp(dlMeta.flowId);
}

void dl_set_exception(unsigned int core_id, unsigned int code)
{
// Since we don't implement the path to the core in our sample code, we don't implement this
// either.  If we did, this would set a global variable based on the input parameters.  The
// dl_sink function would then use the global variable to pack up the core ID and code and send
// the packet to the core.
}
```

---

dl_source.h

```
#ifndef DL_SOURCE_H
#define DL_SOURCE_H

#include "ixp.h"

#ifdef RX_DL

// A signal in the processing task that indicates the processing
// task can begin to dequeue packets
__declspec(remote) SIGNAL rx_ring_ready_sig;

#endif
```

```
#ifdef PROCESSING_DL

//-------------------------------------------------------------------
// This signal is sent by the rx pipeline to indicate that
// the processing code can begin to dequeue packets
__declspec(visible) SIGNAL rx_ring_ready_sig;

//-------------------------------------------------------------------
// A signal in the tx task that indicates the tx
// task can begin to dequeue packets
__declspec(remote) SIGNAL tx_ring_ready_sig;

#endif

#ifdef TX_DL

//-------------------------------------------------------------------
// This signal is sent by the processing pipeline to indicate that
// the tx code can begin to dequeue packets
__declspec(visible) SIGNAL tx_ring_ready_sig;

#endif

extern void dl_sink_init();
extern void dl_sink();
extern void dl_source_init();
extern void dl_source();

unsigned int dl_get_sink_size();

unsigned int dl_get_sink_empty_timestamp();

void dl_set_exception(unsigned int core_id, unsigned int code);

#endif  // DL_SOURCE_H
```

dl_system.h

```
#ifndef SYSTEM__H
#define SYSTEM__H

//-------------------------------------------------------------------
//        System SRAM and SDRAM Memory Related Defines
//-------------------------------------------------------------------

/*      Amount of SRAM space available for (meta data of) packet buffers */
#ifndef MICRO_C
#define_eval    BUF_SRAM_SIZE      (1024 * 10)
#else
#define          BUF_SRAM_SIZE      (1024 * 10)
#endif
```

```
/*      Amount of SDRAM space available for Packet buffers. */
#ifndef MICRO_C
#define_eval    BUF_SDRAM_SIZE          (64 * 1024 * 10)
#else
#define         BUF_SDRAM_SIZE          (64 * 1024 * 10)
#endif

/*      Size of Meta Data (Buffer Descriptors) in SRAM. 32 entries per buffer.
 *      Should be a power of 2.
 */
#define  META_DATA_SIZE          32

/*      Size of packet buffer.  */
#define  BUFFER_SIZE             2048
#ifdef   USE_IMPORT_VAR
.import_var BUF_SRAM_BASE
.import_var DL_REL_BASE
.import_var BUF_FREE_LIST0

#else /* no import vars */
/* Base address in SRAM for the packet buffers. This is for the meta data (buffer descriptors). */
#define  BUF_SRAM_BASE           0x1000

/* Base Address in DRAM for the packet buffers. This is for the actual packet data. */
#define  BUF_SDRAM_BASE          0x6000

/* free list id that is stored in meta data */
#define  FREE_LIST_ID            0x0

/* BLADE ID */
#define  THIS_BLADE_ID                   1

// SRAM Q-Array entries allocated for buffer free list
#define  BUF_QARRAY_BASE         0
#define  BUF_QARRAY_SIZE         4

#endif /* USE_IMPORT_VAR */

//-------------------------------------------------------------------
// System Scratch Ring Related Defines
//-------------------------------------------------------------------

#define DL_DROP_RING                    11

/* Scratch for communicating between RX and Processing */
#define RX_TO_PROCESSING_RING           0
#define RX_TO_PROCESSING_RING_BASE      0x1000
#define RX_TO_PROCESSING_RING_SIZE      128

/* Scratch for communicating between Processing and Tx */
#define PROCESSING_TO_TX_RING           1
#define PROCESSING_TO_TX_RING_BASE      0x2000
#define PROCESSING_TO_TX_RING_SIZE      128
```

```
// These defines are used by the individual microcode to identify their scratch rings.
// The reason for these defines is to improve the portability of microblocks. User just
// has to make changes here and the microblocks will remain untouched.
// For the naming convention, the microblock name goes first, then the _RING_ specifier.
// Following this is the direction indicator. '_OUT' would imply the microblock is writing
// to this scratch ring and '_IN' would imply the microblock is reading from this scratch
// ring. If there is more than one 'IN' or 'OUT' ring the name will have '_<digit>' to
// differentiate the rings.

/* The Ring on which RX sends out msg to next block */

#define RX_RING_OUT             RX_TO_PROCESSING_RING
#define RX_RING_OUT_BASE        RX_TO_PROCESSING_RING_BASE
#define RX_RING_OUT_SIZE        RX_TO_PROCESSING_RING_SIZE

/*The input ring ring on which Processing receives msgs from prev block */
#define PROCESSING_RING_IN              RX_TO_PROCESSING_RING
#define PROCESSING_RING_IN_BASE         RX_TO_PROCESSING_RING_BASE
#define PROCESSING_RING_IN_SIZE         RX_TO_PROCESSING_RING_SIZE

/* The ring on which Processing sends out msg to next block  */
#define PROCESSING_RING_OUT             PROCESSING_TO_TX_RING
#define PROCESSING_RING_OUT_BASE        PROCESSING_TO_TX_RING_BASE
#define PROCESSING_RING_OUT_SIZE        PROCESSING_TO_TX_RING_SIZE

/*The input ring ring on which Tx receives msgs from prev block */
#define TX_RING_IN                      PROCESSING_TO_TX_RING
#define TX_RING_IN_BASE                 PROCESSING_TO_TX_RING_BASE
#define TX_RING_IN_SIZE                 PROCESSING_TO_TX_RING_SIZE

/*********************************************************************
 *      Some System Communication ID's and BlockID's Related Defines
 *********************************************************************
 */
/*      All the IX_* constants should be between 0 and 255 (i.e 1 byte) to enable using
 *      them as immed operands in instructions.
 *
 *      IX_NULL is used to denote a NULL (or empty) packet handle, It should not be zero,
 *      as it would interfere with ring empty (0) condition.
 *      Note: IX_NULL is too generic a name and may clash with core components code.
 *      So it's marked for deletion.  Start using UC_NULL
 */
#define IX_EXCEPTION            0x1
#define IX_DROP                 0x2
#define UC_NULL                 0xFF        /* NULL for microcode */
#define DL_BUF_NULL             UC_NULL     /* NULL Buffer representation */
#define BID_NULL                UC_NULL             /* NULL Block ID */
#define BID_EXCEPTION           0x1                 /* Exception Block ID */
#define BID_DROP                0x2                 /* Drop Block ID */

/*      Block IDs for various blocks. Block IDs Start from 0x20. (0 to 0x1f and 0xFF are
 *      reserved for IX_* )
 */

#define BID_SPI4_RX                         0x23
```

```
#define BID_DL_SINK                         0x24
#define BID_ETHERNET_VALIDATE               0x25
#define BID_ETHERNET_STRIP_HEADER           0x26
#define BID_IPV4_FIVE_TUPLE_CLASS           0x27
#define BID_ETHERNET_ADD_HEADER             0x28
#define BID_RED                             0x29


/*      Static Bindings (a.k.a targets) for various microblocks. */
#define SPI4_RX_NEXT_BLOCK                  BID_DL_SINK

/*************************************************************************
 *              Some System Signal Related Defines
 *************************************************************************/

/*      Signals being a precious resource, we use a single signal for
 *      inter-thread as well as any other inter-me signalling (init over signal)
 */

#define COMMON_INTER_THD_ME_SIGNAL      10

/*      We are not able to use a single signal before of limitations in assembler.
 *      until then, we need a separate signal for inter-thread signalling.
 */

#define INTER_THD_SIGNAL                    12

/*      ME_NIT_SIGNAL signal is used to let everyone know that initialisation is over
 *      and all blocks in all MEs can start running. This signal will be typically issued
 *      by the system initialisation code and all other MEs should wait for this signal
 *      before starting to run.
 */

#define ME_INIT_SIGNAL              COMMON_INTER_THD_ME_SIGNAL
// Set the meta data cache to be large enough so that
// the next hop ID and flow ID is at least temporarily
// saved
#define META_CACHE_SIZE         6
#endif          // SYSTEM__H

/* Contador */
#define BID_COUNT    0x2A
/* Contador */

/* Contador */
#define COUNT_SRAM_ADDR    0x40300200
/* Contador */

/* Contador */
//#define COUNT_NEXT_BLOCK    0x40300300
/* Contador */
```

errcode.h

```
#ifndef __ERRCODE_H_INCLUDED
#define __ERRCODE_H_INCLUDED

/*
 *  Common error codes for mutex/semaphores library
 */

typedef enum
{
   ERRCODE_EOK,
   ERRCODE_ENOLCK,
   ERRCODE_EBUSY,
   ERRCODE_EINVAL,
   ERRCODE_AGAIN

} ERRCODE;

#endif
```

---

ethernet.c

```
#include <dl_buf.c>
#include <dl_source.h>
#include "ethernet.h"
#include "ethernet_internal.h"

// Some Ethernet basics to be used for validating the packet
// and checking Ethernet fields
#define MIN_ETHERNET_LENGTH          64
#define MAX_ETHERNET_LENGTH          1518
#define ETHERNET_ADDR_SIZE           6
#define ETHERNET_HEADER_SIZE         14

// The format of the Ethernet packet header
typedef __declspec(packed) struct _ethernet_header
{
        unsigned int destination_addr_hi32;
        unsigned int destination_addr_lo16: 16;
        unsigned int source_addr_hi16: 16;
        unsigned int source_addr_lo32;
        short protocol;
} ethernet_header;

extern dl_buf_handle_t                  dlBufHandle;
```

```
extern dl_meta_t                    dlMeta;
extern __declspec(gp_reg) int       dlNextBlock;

void ethernet_init()
{
}

void ethernet_validate()
{
        ethernet_header header;
        int i = 0;
        __declspec(sram) ethernet_control_block *control =
                (__declspec(sram) ethernet_control_block*) (ETHERNET_DATA);

        // Check the packet length
        if (dlMeta.bufferSize < MIN_ETHERNET_LENGTH ||
                dlMeta.bufferSize > MAX_ETHERNET_LENGTH)
        {
                dlNextBlock = ETHERNET_VALIDATE_INVALID;
    /* Contador */
       //dlNextBlock = BID_COUNT;
    /* Contador */
                return;
        }

        // Now, get the Ethernet header
        header = *(__declspec(dram) ethernet_header*)
                (Dl_BufGetData(dlBufHandle) + dlMeta.offset);

        // Check for a broadcast or multicast source address
        if ((header.source_addr_hi16 >> 8) & 0x1)
        {
                dlNextBlock = ETHERNET_VALIDATE_INVALID;
                return;
        }

        // Check for broadcast or multicast destination address
        if ((header.destination_addr_hi32 >> 24) & 0x1)
        {
                if (header.destination_addr_hi32 != 0xffffffff &&
                        header.destination_addr_lo16 != 0xffff)
                {
                        dlNextBlock = ETHERNET_VALIDATE_MULTICAST;
    return;
                }
                dlNextBlock = ETHERNET_VALIDATE_BROADCAST;
    return;
        }

        // Check to see if the packet is locally addressed
        if (header.destination_addr_hi32 != control->device_addr_hi32 &&
                header.destination_addr_lo16 != control->device_addr_lo16)
        {
                dlNextBlock = ETHERNET_VALIDATE_OTHER;
    return;
```

```
        }

        dlNextBlock = ETHERNET_VALIDATE_LOCAL;
   /* Contador */
   //dlNextBlock = BID_COUNT;
   /* Contador */
        return;
}

void ethernet_strip_header()
{
        ethernet_header header;

        // Get the ethernet header so we can extract the
        // protocol
        header = *(__declspec(dram) ethernet_header*)
                (Dl_BufGetData(dlBufHandle) + dlMeta.offset);
        dlNextBlock = header.protocol;

        // Adjust the pointer and length
        dlMeta.bufferSize -= ETHERNET_HEADER_SIZE;
        dlMeta.offset += ETHERNET_HEADER_SIZE;
}

void ethernet_add_header(unsigned int eth_proto)
{
        ethernet_arp_table_entry  table_entry;
        __declspec(sram) ethernet_control_block *control =
                (__declspec(sram) ethernet_control_block*) (ETHERNET_DATA);
        unsigned int            eth_address_hi32;
        unsigned int            eth_address_lo16;
        bool                    found = 0;
        volatile __declspec(dram) ethernet_header*   header;

        // Look up the next hop IP address in the array
        // in memory, using the next hop ID as the index.
        table_entry =  control->arp_table_array[dlMeta.nextHopId];

        // If the entry is not valid, drop this packet.
        if (!table_entry.valid)
        {
                dl_set_exception(ETHERNET_EXCEPTION_ID ,0);
                dlNextBlock = IX_EXCEPTION;
                return;
        }

        eth_address_hi32 = table_entry.ethernet_address_hi32;
        eth_address_lo16 = table_entry.ethernet_address_lo16;

        // Since we found a MAC address, we can add the
        // header
        dlMeta.bufferSize += ETHERNET_HEADER_SIZE;
        dlMeta.offset -= ETHERNET_HEADER_SIZE;

        header = (__declspec(dram) ethernet_header*)
```

```
        (Dl_BufGetData(dlBufHandle) + dlMeta.offset);

        // Set the destination address in the header to be
        // the Ethernet address from the table.
        header->destination_addr_hi32 = eth_address_hi32;
        header->destination_addr_lo16 = eth_address_lo16;

        // Set the source address to be the device's MAC
        // address
        header->source_addr_hi16 = control->device_addr_hi32 >> 16;
        header->source_addr_lo32 = (control->device_addr_hi32 << 16) |
                                            (control->device_addr_lo16);

        // Set the protocol number to be the passed in protocol number
        header->protocol = eth_proto;
        dlNextBlock = ETHERNET_ADD_HEADER_PASS;
}
```

ethernet.h

```
#include <ixp.h>
#include <dl_meta.h>

// These defines are the outputs from
// ethernet_validate
// ---------------------------------------------------
// The packet is invalid:
#define ETHERNET_VALIDATE_INVALID       (IX_DROP + 1)
// The packet is valid and the packet's destination
// MAC address is the address of this device:
#define ETHERNET_VALIDATE_LOCAL         (IX_DROP + 2)
// The packet is valid and the packet's destination
// MAC address is a multicast address:
#define ETHERNET_VALIDATE_MULTICAST     (IX_DROP + 3)
// The packet is valid and the packet's destination
// MAC address is a broadcast address:
#define ETHERNET_VALIDATE_BROADCAST     (IX_DROP + 4)
// The packet is valid, but not local, multicast, or
// broadcast:
#define ETHERNET_VALIDATE_OTHER         (IX_DROP + 5)
// These defines are the outputs from
// ethernet_add_header
// ---------------------------------------------------
// The next hop MAC address was found and the Ethernet
// header was successfully added.
#define ETHERNET_ADD_HEADER_PASS        (IX_DROP + 1)

// The output from ethernet_strip_header is the
// Ethernet protocol number.  This particular one is
// for IP
#define ETHERNET_PROTO_IP               0x0800
```

```
void ethernet_init();
void ethernet_validate();
void ethernet_strip_header();
void ethernet_add_header(unsigned int eth_proto);
```

ethernet_internal.h

```
#ifndef ETHERNET_INTERNAL_H
#define ETHERNET_INTERNAL_H

// The number of entries in the top level hash array.  This
// number must be a power of 2.
#define ARP_TABLE_ARRAY_SIZE  256

// This is the ARP table entry format.  A hash table
// containing these entries is used to map destination
// IP addresses to Ethernet Destination MAC addresses.
typedef struct _ethernet_arp_table_entry
{
        unsigned int ethernet_address_hi32;
        unsigned int ethernet_address_lo16 : 16;
        unsigned int pad : 8;
        unsigned int valid,: 8;
} ethernet_arp_table_entry;

// This structure contains the single MAC address of the
// device, along with a pointer to the hash table used
// for ARP.
typedef struct
{
        unsigned int device_addr_hi32;
        unsigned int device_addr_lo16 : 16;
        unsigned int pad : 16;
        ethernet_arp_table_entry  arp_table_array[ARP_TABLE_ARRAY_SIZE];
} ethernet_control_block;

// If we are using hardware, the ETHERNET_DATA symbol
// is an imported variable.  In the simulator, it is
// a #define in the compile options
#ifdef HARDWARE
#define ETHERNET_SYMBOL_NAME          "ETHERNET_DATA"
#ifdef MICRO_C int ETHERNET_DATA =
                        LoadTimeConstant(ETHERNET_SYMBOL_NAME);
#endif
#endif /* HARDWARE */

// For now, the communication ID for packets going from
// the microblock to the core is set to 64.  When
// this path is actually implemented, it needs to be
// unique system-wide.  This is the exception ID used
// in the microblock and the communication ID in the
// core component
```

```
#define ETHERNET_EXCEPTION_ID64

#endif /*ETHERNET_INTERNAL_H*/
```

ipv4_five_tuple_class.c

```
#include <dl_buf.c>
#include "ipv4_five_tuple_class.h"
#include "ipv4_five_tuple_class_internal.h"

#define MIN_L4_SIZE             28
#define IP_PROTO_UDP            17
#define IP_PROTO_TCP            6

// Temporary, until compiler fixes problem
unsigned long long ua_get_64_dram(DRAM_I64 *p, unsigned int offset);

// This is the structure of the IP header
typedef __declspec(packed) struct
{
        unsigned int version       :4;
        unsigned int header_len    :4;
        unsigned int tos           :8;
        unsigned int total_len     :16;
        unsigned int id            :16;
        unsigned int flags         :4;
        unsigned int frag_offset   :12;
        unsigned int ttl           :8;
        unsigned int protocol      :8;
        unsigned int checksum      :16;
        unsigned int source_addr;
        unsigned int destination_addr;
        unsigned int source_port       :16;  // UDP and TCP only
        unsigned int destination_port  :16;  // UDP and TCP only
} ip_header;

// This structure exists to allow unaligned access to
// the IP header.
typedef __declspec(packed) struct
{
        unsigned long long header0;
        unsigned long long header1;
        unsigned long long header2;
} ip_header_mirror;

extern dl_buf_handle_t          dlBufHandle;
extern dl_meta_t                dlMeta;
extern __declspec(gp_reg) int   dlNextBlock;


void ipv4_five_tuple_class_init()
{
}
```

```
void ipv4_five_tuple_class()
{
        __declspec(dram) ip_header*  header_ptr;
        ip_header                header;
        ip_five_tuple            five_tuple = { 0 };
        unsigned int             array_index;
        hash_table_entry*        table_entry;

        dlNextBlock = IX_DROP;

        // Get the IP header
        header_ptr = (__declspec(dram) ip_header*)
                (Dl_BufGetData(dlBufHandle) + dlMeta.offset);
        read_unaligned_header(header_ptr, &header);

        // Do some basic validation of the packet.  Check the
        // length of the packet, make sure the TTL is > 0, and
        // make sure the protocol is either TCP or UDP.
        if (dlMeta.bufferSize < MIN_L4_SIZE ||header.ttl < 1 ||
                (header.protocol != IP_PROTO_UDP &&
                 header.protocol != IP_PROTO_TCP))
        {
                return;
        }

        // Now that we know the packet is acceptable, do the
        // hash lookup.
        five_tuple.tuple.source_addr =   header.source_addr;
        five_tuple.tuple.destination_addr = header.destination_addr;
        five_tuple.tuple.source_port = header.source_port;
        five_tuple.tuple.destination_port = header.destination_port;
        five_tuple.tuple.protocol = header.protocol;
        five_tuple.tuple.pad = 0;

        crc_write(0x42424242);
        crc_16_be(five_tuple.hash_io.value3, bytes_0_3);
        crc_16_be(five_tuple.hash_io.value2, bytes_0_3);
        crc_16_be(five_tuple.hash_io.value1, bytes_0_3);
        crc_16_be(five_tuple.hash_io.value0, bytes_0_3);
        array_index = crc_read();

        // Fold the hash result into a size that indexes into
        // the hash table initial array
        array_index = array_index &  (HASH_TABLE_ARRAY_SIZE - 1);

        // Get the array contents
        table_entry = *((hash_table_entry**)IP_FIVE_TUPLE_DATA + array_index);

        // In a loop, look for an entry that matches the key
        while (table_entry)
        {
                if (table_entry->key.tuple.source_addr == five_tuple.tuple.source_addr &&
                        table_entry->key.tuple.destination_addr ==
                                        five_tuple.tuple.destination_addr &&
```

```
                        table_entry->key.tuple.source_port ==
                                five_tuple.tuple.source_port &&
                        table_entry->key.tuple.destination_port ==
                                five_tuple.tuple.destination_port &&
                        table_entry->key.tuple.protocol ==
                                five_tuple.tuple.protocol)
                {
                        // If we find a match, set the out-params based on the
                        // contents of the table data.
                        if(!table_entry->drop)
                        {
                                dlNextBlock =  IPV4_FIVE_TUPLE_CLASS_PASS;
                        }
                        dlMeta.nextHopId = table_entry->next_hop_id;
                        dlMeta.flowId    = table_entry->flow_id;
                        break;
                }
                else
                {
                        // If this is not a match, check the next element
                        table_entry = table_entry->next;
                }
        }
}

__forceinline static void read_unaligned_header(
                                __declspec(dram) ip_header* header_ptr, ip_header* header)
{
        ip_header_mirror* mirror = (ip_header_mirror*)header;
        unsigned int offset = (unsigned int)header_ptr & 0x7;
        __declspec(dram) long long* base_ptr = (__declspec(dram) long long*)
                ((__declspec(dram) char*)header_ptr - offset);
        unsigned long long data;
        data = ua_get_64_dram(base_ptr, offset);
        mirror->header0 = data;
        data = ua_get_64_dram(base_ptr + 1, offset);
        mirror->header1 = data;
        data = ua_get_64_dram(base_ptr + 2, offset);
        mirror->header2 = data;
}
```

ipv4_five_tuple_class.h

```
#include <ixp.h>
#include <dl_meta.h>

void ipv4_five_tuple_class_init();
void ipv4_five_tuple_class();
ipv4_five_tuple_class_hash_unit.c

#include <dl_buf.c>
```

```
#include "ipv4_five_tuple_class.h"
#include "ipv4_five_tuple_class_internal.h"

#define MIN_L4_SIZE             28
#define IP_PROTO_UDP            17
#define IP_PROTO_TCP            6

// Temporary, until compiler fixes problem
unsigned long long ua_get_64_dram(DRAM_I64 *p, unsigned int offset);

// This is the structure of the IP header
typedef __declspec(packed) struct
{
        unsigned int version       :4;
        unsigned int header_len    :4;
        unsigned int tos           :8;
        unsigned int total_len     :16;
        unsigned int id            :16;
        unsigned int flags         :4;
        unsigned int frag_offset   :12;
        unsigned int ttl           :8;
        unsigned int protocol      :8;
        unsigned int checksum      :16;
        unsigned int source_addr;
        unsigned int destination_addr;
        unsigned int source_port       :16;   // UDP and TCP only
        unsigned int destination_port :16;  // UDP and TCP only
} ip_header;

// This structure exists to allow unaligned access to
// the IP header.
typedef __declspec(packed) struct
{
        unsigned long long header0;
        unsigned long long header1;
        unsigned long long header2;
} ip_header_mirror;

extern dl_buf_handle_t                  dlBufHandle;
extern dl_meta_t                        dlMeta;
extern __declspec(gp_reg) int           dlNextBlock;



__forceinline static void read_unaligned_header(
                                __declspec(dram) ip_header* header_ptr, ip_header* header);

void ipv4_five_tuple_class_init()
{
        unsigned int hash_mult = 0x42424242;
        SIGNAL                  csr_sig;
        cap_csr_write((void*)hash_mult,  csr_hash_multiplier_128_0, ctx_swap,  &csr_sig);
        cap_csr_write((void*)hash_mult,  csr_hash_multiplier_128_1, ctx_swap,  &csr_sig);
        cap_csr_write((void*)hash_mult,  csr_hash_multiplier_128_2, ctx_swap,  &csr_sig);
        cap_csr_write((void*)hash_mult,  csr_hash_multiplier_128_3, ctx_swap,  &csr_sig);
```

```
}

void ipv4_five_tuple_class()
{
        __declspec(dram) ip_header*                 header_ptr;
        ip_header                                   header;
        ip_five_tuple                               five_tuple;
        __declspec(sram_write_reg) ip_five_tuple    hash_input;
        __declspec(sram_read_reg) ip_five_tuple     hash_result;
        SIGNAL_PAIR                                 hash_signal;
        unsigned int                                array_index;
        hash_table_entry*                           table_entry;

        dlNextBlock = IX_DROP;

        // Get the IP header
        header_ptr = (__declspec(dram) ip_header*)
                (Dl_BufGetData(dlBufHandle) + dlMeta.offset);
        read_unaligned_header(header_ptr, &header);

        // Do some basic validation of the packet.  Check
        // the length of the packet, make sure the TTL is
        // > 0, and make sure the protocol is either TCP or
        // UDP.
        if (dlMeta.bufferSize < MIN_L4_SIZE || header.ttl < 1 ||
                (header.protocol != IP_PROTO_UDP &&
                 header.protocol != IP_PROTO_TCP))
        {
                return;
        }
        // Now that we know the packet is acceptable, do the
        // hash lookup.
        five_tuple.tuple.source_addr = header.source_addr;
        five_tuple.tuple.destination_addr = header.destination_addr;
        five_tuple.tuple.source_port = header.source_port;
        five_tuple.tuple.destination_port = header.destination_port;
        five_tuple.tuple.protocol = header.protocol;
        five_tuple.tuple.pad =  0;
        hash_input = five_tuple;
        hash_128(&hash_input, &hash_result, 1, sig_done, &hash_signal);
        wait_for_all(&hash_signal);

        // Fold the hash result into a size that indexes
        // into the hash table initial array
        array_index = (hash_result.hash_io.value0 ^ hash_result.hash_io.value1 ^
                                hash_result.hash_io.value2 ^ hash_result.hash_io.value3) &
                                                        (HASH_TABLE_ARRAY_SIZE - 1);

        // Get the array contents
        table_entry = *((hash_table_entry**)IP_FIVE_TUPLE_DATA + array_index);

        // In a loop, look for an entry that matches the key
        while (table_entry)
        {
                if (table_entry->key.tuple.source_addr == five_tuple.tuple.source_addr &&
```

```
                        table_entry->key.tuple.destination_addr ==
                                five_tuple.tuple.destination_addr &&
                        table_entry->key.tuple.source_port ==
                                five_tuple.tuple.source_port &&
                        table_entry->key.tuple.destination_port ==
                                five_tuple.tuple.destination_port &&
                        table_entry->key.tuple.protocol ==
                                five_tuple.tuple.protocol)
                {
                        // If we find a match, set the out-params
                        // based on the contents of the table data.
                        if(!table_entry->drop)
                        {
                                dlNextBlock = IPV4_FIVE_TUPLE_CLASS_PASS;
                        }
                        dlMeta.nextHopId = table_entry->next_hop_id;
                        dlMeta.flowId  = table_entry->flow_id;
                        break;
                }
                else
                {
                        // If this is not a match, check the next
                        // element
                        table_entry = table_entry->next;
                }
        }
}

__forceinline static void read_unaligned_header(
                                __declspec(dram) ip_header* header_ptr, ip_header* header)
{
        ip_header_mirror* mirror = (ip_header_mirror*)header;
        unsigned int offset = (unsigned int)header_ptr & 0x7;
        __declspec(dram) long long* base_ptr = (__declspec(dram) long long*)
                ((__declspec(dram) char*)header_ptr - offset);
        unsigned long long data;
        data = ua_get_64_dram(base_ptr, offset);
        mirror->header0 = data;
        data = ua_get_64_dram(base_ptr + 1, offset);
        mirror->header1 = data;
        data = ua_get_64_dram(base_ptr + 2, offset);
        mirror->header2 = data;
}
```

ipv4_five_tuple_class_internal.h

```
#ifndef IPV4_FIVE_TUPLE_CLASS_INTERNAL_H
#define IPV4_FIVE_TUPLE_CLASS_INTERNAL_H

#define HASH_TABLE_ARRAY_SIZE  256
```

```
// This is the hash key.  It is based on an IP 5-tuple
typedef union
{
	struct
	{
		unsigned int source_addr;
		unsigned int destination_addr;
		unsigned int source_port :16;
		unsigned int destination_port :16;
		unsigned int protocol :8;
		unsigned int pad :24;
	} tuple;
	struct
	{
		unsigned int value0;
		unsigned int value1;
		unsigned int value2;
		unsigned int value3;
	} hash_io;
} ip_five_tuple;

// This is the information in the hash table entries
typedef struct _hash_table_entry
{
	ip_five_tuple		key;
	unsigned int		flow_id;
	unsigned int		next_hop_id;
	unsigned int		drop;
	struct _hash_table_entry*  next;
} hash_table_entry;

// If we are using hardware, the IP_FIVE_TUPLE_DATA
// symbol is an imported variable.  In the simulator, it
// is a #define in the compile options
#ifdef HARDWARE
#define IP_FIVE_TUPLE_SYMBOL_NAME   "IP_FIVE_TUPLE_DATA"

#ifdef MICRO_C
int ETHERNET_DATA = LoadTimeConstant(IP_FIVE_TUPLE_SYMBOL_NAME);
#endif
#endif // HARDWARE


#endif // IPV4_FIVE_TUPLE_CLASS_INTERNAL_H
```

process_dl.c

```
#include	"dl_system.h"
#include	"system_init.h"
#include	"dispatch_loop.h"
#include	"dl_source.h"
#include	"dl_buf.c"
```

```
#include        "dl_meta.h"
#include        "ethernet.h"
#include        "ipv4_five_tuple_class.h"
#include        "red.h"

/* Contador */
#include "counteth.c"
/* Contador */

//-------------------------------------------------------------
// Global variables/Registers
//-------------------------------------------------------------
dl_buf_handle_t                 dlBufHandle;
__declspec(gp_reg) int          dlNextBlock;
extern dl_meta_t                dlMeta;

void main()
{
        unsigned int eth_proto = 0;
        //-------------------------------------------------
        // Initialize the blocks
        //-------------------------------------------------
        // Initialize the system
        system_init();

        // Initialize the outgoing ring
        dl_sink_init();

        // Initialize the incoming ring
        dl_source_init();

        // Call the initialization functions for those
        // modules that need it.
        ethernet_init();
        ipv4_five_tuple_class_init();

        while (1)
        {
                // Dequeue a packet from the rx task
                dl_source();
                if (dlBufHandle.value == 0)
                {
                        continue;
                }

                // Verify that this packet is an acceptable
                // Ethernet packet and that it is locally
                // addressed.
                ethernet_validate();

    /* contador */
    //counteth();
    /* contador */

                if (dlNextBlock != ETHERNET_VALIDATE_LOCAL)
```

```
                {
                        goto drop;
                }

        // At this point we know we have an Ethernet II
                // packet. Before we send it to the IPv4
                // classifier, we have to make sure that it is
                // an IP packet.  Then, since the classifier is
                // L2 agnostic, we have to move the packet data
                // pointer past the Ethernet header.
                ethernet_strip_header();
                if (dlNextBlock != ETHERNET_PROTO_IP)
                {
                        goto drop;
                }
                eth_proto = dlNextBlock;

                // Now that we have a packet, send it the IPv4
                // 5-tuple classifier.  The classifier will
                // assign an output ring, and a next hop IP
                // address.
                ipv4_five_tuple_class();
                if (dlNextBlock == IX_DROP)
                {
                        goto drop;
                }

                // Before we transmit the packet, we have to add
                // the Ethernet header back on.  We do this
                // based on the next hop IP address retrieved
                // from the classifier.
                ethernet_add_header(eth_proto);
                if (dlNextBlock == IX_DROP)
                {
                        goto drop;
                }

                // Now that the output ring is assigned, send it
                // to the RED buffer manager to either enqueue
                // or drop
                red();
                if (dlNextBlock == IX_DROP)
                {
                        goto drop;
                }

                // Once we get here, the packet is put on the
                // ring to go to transmit.
                dl_sink();
                continue;

        drop: Dl_BufDrop(dlBufHandle);
        }
}
```

```
void put(int x)
{
}
```

---

red.c

```
#include <dl_meta.h>
#include "dl_source.h"
#include "red.h"

typedef struct {
        unsigned int min_threshold;   // The threshold below which no packets are dropped
        unsigned int max_threshold;  // The threshod above which all packets are dropped
        unsigned int average_length; // The average queue length
        unsigned int c1;             // The slope of the probability line
        unsigned int c2;             // The y-intersect of the probability line
} queue_info;

#define RATE_RIGHT_SHIFT                8
#define AVG_TO_ACTUAL_SHIFT_RIGHT  16
#define EXP_TABLE_SIZE                  256
#define EXP_TABLE { 65536, 63470, 61469, 59531, 57654, \
                55837, 54076, 52372, 50721, 49122, \ 47573, 46073, 44621, 43214, 41852, \
                40533, 39255, 38017, 36819, 35658, \ 34534, 33445, 32391, 31370, 30381, \
                29423, 28496, 27597, 26727, 25885, \ 25069, 24278, 23513, 22772, 22054, \
                21359, 20685, 20033, 19402, 18790, \ 18198, 17624, 17068, 16530, 16009, \
                15504, 15016, 14542, 14084, 13640, \ 13210, 12793, 12390, 11999, 11621, \
                11255, 10900, 10556, 10224, 9901, \ 9589, 9287, 8994, 8710, 8436, 8170, \
                7912, 7663, 7421, 7187, 6961, 6741, \ 6529, 6323, 6124, 5931, 5744, 5562, \
                5387, 5217, 5053, 4893, 4739, 4590, \ 4445, 4305, 4169, 4038, 3910, 3787, \
                3668, 3552, 3440, 3332, 3227, 3125, \ 3026, 2931, 2839, 2749, 2662, 2578, \
                2497, 2418, 2342, 2268, 2197, 2127, \ 2060, 1995, 1933, 1872, 1813, 1755, \
                1700, 1646, 1595, 1544, 1496, 1448, \ 1403, 1358, 1316, 1274, 1234, 1195, \
                1157, 1121, 1085, 1051, 1018, 986, \ 955, 925, 896, 867, 840, 814, 788, \
                763, 739, 716, 693, 671, 650, 629, \ 610, 590, 572, 554, 536, 519, 503, \
                487, 472, 457, 442, 428, 415, 402, \ 389, 377, 365, 353, 342, 331, 321, \
                311, 301, 292, 282, 273, 265, 256, \ 248, 240, 233, 226, 218, 212, 205, \
                198, 192, 186, 180, 174, 169, 164, \ 158, 153, 149, 144, 139, 135, 131, \
                126, 122, 119, 115, 111, 108, 104, \ 101, 98, 95, 92, 89, 86, 83, 81, 78, \
                76, 73, 71, 69, 66, 64, 62, 60, 58, \ 57, 55, 53, 51, 50, 48, 47, 45, 44, \
                42, 41, 40, 38, 37, 36, 35, 34, 33, \ 32, 31, 30, 29, 28, 27, 26, 25, 24, \
                24, 23, 22, 21, 21, 20, 19, 19, 18 }

#define NON_EMPTY_AVERAGE_LEFT_SHIFT      7

// This is the table of exponents defined above
static __declspec(shared local_mem)  unsigned int g_exp_table[] = EXP_TABLE;

// This is the per-queue data structure
static __declspec(shared local_mem)  queue_info g_queue_data[16];

extern dl_meta_t                dlMeta;
```

```
extern __declspec(gp_reg) int    dlNextBlock;

void red()
{
        __declspec(local_mem shared) queue_info* queue;
        unsigned int queue_length;
        unsigned int real_average;

        // Use the ring number to get an index into local
        // memory to get the queue_info structure
        queue = g_queue_data + dlMeta.flowId;

        // Get the number of packets in the ring.
        queue_length = dl_get_sink_size();

        // Calculating the average queue length is different
        // if the queue is empty or not
        if (queue_length)
        {
                unsigned int q_minus_avg;
                q_minus_avg = queue_length - (queue->average_length >>
                                        AVG_TO_ACTUAL_SHIFT_RIGHT);
                queue->average_length = queue->average_length +
                        (q_minus_avg << NON_EMPTY_AVERAGE_LEFT_SHIFT);
        }
        else
        {
                unsigned int idle_time;
                unsigned int m;

                idle_time = local_csr_read(local_csr_timestamp_low) -
                        dl_get_sink_empty_timestamp();
                // Shift the idle time by a scaling factor that
                // approximates division by the number of packets
                // that could have arrived in this amount of time
                m = idle_time >> RATE_RIGHT_SHIFT;

                // Now, perform a lookup to get the new average
                if (m > 255)
                {
                        queue->average_length = 0;
                }
                else
                {
                        queue->average_length = (queue->average_length >>
                                AVG_TO_ACTUAL_SHIFT_RIGHT) * g_exp_table[m];
                }
        }

        // We don't need scaled averages at this point, so
        // switch back to real averages
        real_average = queue->average_length >> AVG_TO_ACTUAL_SHIFT_RIGHT;

        // Now that the average queue length is computed,
        // decide whether or not to drop the packet
```

```
        if (real_average >= queue->min_threshold && real_average < queue->max_threshold)
        {
                unsigned int random;
                unsigned int probability;

                // Compute the probability of dropping the packet
                probability = queue->c1 * real_average -  queue->c2;

                // Get a pseudo-random number
                random = local_csr_read(local_csr_pseudo_random_number);

                if (random > probability)
                {
                        dlNextBlock = RED_PASS;
                }
                else
                {
                        dlNextBlock = IX_DROP;
                }
        }
        else if (real_average >= queue->max_threshold)
        {
                dlNextBlock = IX_DROP;
        }
        else
        {
                dlNextBlock = RED_PASS;
        }
}
```

---

red.h

```
#include <ixp.h>

// These are the outputs to red
#define RED_PASS      (IX_DROP + 1)

void red();
```

---

rx.h

```
#ifndef RX_H
#define RX_H

//------------------------------------------------------------------
// Bit positions within the MSF receive control CSR
#define RX_EN_SPHY_BITPOS                                30
#define RBUF_ELE_SIZE_0_BITPOS                           2
#define RBUF_PARTITION_BITPOS                            0
```

```
//-----------------------------------------------------------------
// Bit positions within the SPHY4 receive status word
#define RSW_SPHY4_ADDR_BITPOS                   0
#define RSW_SPHY4_TYPE_BITPOS                   7
#define RSW_SPHY4_NULL_BITPOS                   9
#define RSW_SPHY4_ERRORS_BITPOS                 13
#define RSW_SPHY4_EOP_BITPOS                    14
#define RSW_SPHY4_SOP_BITPOS                    15
#define RSW_SPHY4_BYTECOUNT_BITPOS              16
#define RSW_SPHY4_BYTECOUNT_MASK                0xff
#define RSW_SPHY4_ELEMENT_BITPOS                24
#define RSW_SPHY4_ELEMENT_MASK                  0x7f

//-----------------------------------------------------------------
// MSF memory map
#define MSF_RX_CONTROL_ADDR                     0x0
#define MSF_HWM_CONTROL_ADDR                    0x24
#define MST_HWM_CONTROL__RBUF_S_HWM__MASK       0xFFFFFFCF
#define MST_HWM_CONTROL__RBUF_S_HWM__00         0x00000000
#define MST_HWM_CONTROL__RBUF_S_HWM__01         0x00000010
#define MST_HWM_CONTROL__RBUF_S_HWM__10         0x00000020
#define MST_HWM_CONTROL__RBUF_S_HWM__11         0x00000030
#define MSF_RX_THREAD_FREELIST_0_ADDR           0x30
#define MSF_RX_PORT_MAP_ADDR                    0x40
#define MSF_RBUF_ELEMENT_DONE_ADDR              0x44
#define MSF_RX_CALENDAR_LENGTH_ADDR             0x48
#define MSF_TX_CALENDAR_LENGTH_ADDR             0x70
#define MSF_TRAIN_DATA_ADDR                     0xA0
#define MSF_TRAIN_DATA__RSTAT_EN                0x00000002
#define MSF_RBUF_BASE_ADDR                      0x2000


//-----------------------------------------------------------------
// MicroC data structures
#ifdef MICRO_C

//-----------------------------------------------------------------
// The SPI4 receive status words
typedef struct s_spi4_rsw
{
        // The first status word
        union
        {
                struct
                {
                        unsigned int    rprot           : 1,
                                        element         : 7,
                                        byte_count      : 8,
                                        sop             : 1,
                                        eop             : 1,
                                        err             : 1,
                                        len_err         : 1,
                                        par_err         : 1,
                                        abort_err       : 1,
```

```
                                        null            : 1,
                                        type            : 1,
                                        addr            : 8;
                } parts;
                unsigned int whole;
        } w1;

        // The second status word union
        {
                unsigned int whole;
        } w2;
} spi4_rsw_t;

#endif // MICRO_C

#endif // RX_H
```

---

scratch_rings.c

```
#include "ixp.h"
#include "scratch_rings.h"


void scratch_ring_init(unsigned int ring_number, unsigned int ring_base,unsigned int ring_size)
{
        __declspec(sram_write_reg) unsigned int ring_init;
        __declspec(sram_write_reg) unsigned int ring_head;
        __declspec(sram_write_reg) unsigned int ring_tail;
        SIGNAL ring_init_sig,
                ring_head_sig,
                ring_tail_sig;

        ring_init = ring_base |  (((ring_size/128)-1) << RING_BASE_SIZE_BITPOS);

        cap_write(&ring_init, csr_scratch_ring_base_0 + (ring_number * 16), 1,
                        sig_initiator,  sig_done, &ring_init_sig);

        ring_head = 0;
        cap_write(&ring_head, csr_scratch_ring_head_0 + (ring_number * 16), 1,
                        sig_initiator, sig_done, &ring_head_sig);

        ring_tail = 0;
        cap_write(&ring_tail, csr_scratch_ring_tail_0 + (ring_number * 16), 1,
                        sig_initiator, sig_done, &ring_tail_sig);

        wait_for_all(&ring_init_sig,  &ring_head_sig,  &ring_tail_sig);
        free_write_buffer(&ring_init);
        free_write_buffer(&ring_head);
        free_write_buffer(&ring_tail);
}
```

```
bool scratch_ring_empty(unsigned int ring_number)
{
        __declspec(sram_read_reg) unsigned int ring_head;
        __declspec(sram_read_reg) unsigned int ring_tail;
        SIGNAL ring_head_sig;
        SIGNAL ring_tail_sig;

        cap_read(&ring_head, csr_scratch_ring_head_0 + (ring_number * 16), 1,
                        sig_initiator, sig_done, &ring_head_sig);

        cap_read(&ring_tail, csr_scratch_ring_tail_0 + (ring_number * 16), 1,
                        sig_initiator, sig_done, &ring_tail_sig);

        wait_for_all(&ring_head_sig, &ring_tail_sig);
        return ring_head == ring_tail ? 1 : 0;
}


void scratch_ring_get_buffer(unsigned int ring_number, ring_data_t*data)
{
        __declspec(scratch) void* ring_addr = (void*)(ring_number << 2);
        SIGNAL ring_signal;
        __declspec(sram_read_reg) ring_data_t packet;

        scratch_get_ring(&packet, ring_addr, sizeof(packet) / sizeof(unsigned int),
                                                                ctx_swap, &ring_signal);
        *data = packet;
}



void scratch_ring_put_buffer(unsigned int    ring_number,ring_data_t data)
{
        __declspec(scratch) void* ring_addr = (void*)(ring_number << 2);
        SIGNAL ring_signal;
        __declspec(sram_write_reg) ring_data_t packet;

        packet = data;
        scratch_put_ring(&packet, ring_addr, sizeof(packet) / sizeof(unsigned int),
                                                                ctx_swap, &ring_signal);
}
```

scratch_rings.h

```
#ifndef SCRATCH_RINGS_H
#define SCRATCH_RINGS_H


#define RING_BASE_SIZE_BITPOS                   30
```

```
#define RING_DATA_SIZE_BYTES                    16

//--------------------------------------------------------------------
// MicroC data structures
#ifdef MICRO_C

#include "dl_buf.h"

// This structure represents the data placed on the rx->processing
// and processing->tx rings.
typedef struct s_ring_data
{
        dl_buf_handle_t         handle;
        unsigned int            length;
        unsigned int            offset;
        unsigned int            sequence;
} ring_data_t;

// Function prototypes

void scratch_ring_init(unsigned int ring_number,unsigned int ring_base,
                                                                unsigned int ring_size);
bool scratch_ring_empty(unsigned int ring_number);

void scratch_ring_get_buffer( unsigned int ring_number, ring_data_t*data);

void scratch_ring_put_buffer(unsigned int    ring_number, ring_data_t data);

__forceinline bool scratch_ring_full(ring_number)
{
        return inp_state_test(inp_state_scr_ring0_full + ring_number);
}

#endif // MICRO_C

#endif // SCRATCH_RINGS_H
```

---

single_thread_proc_uc.h

```
/* Intel Corporation -- UcLd generated file. DO NOT EDIT THIS FILE!! */
/*
UcLd version: 4.3,  UOF: C:\IXP2400_2800_book\Chapter06\single_thread_proc_uc.uof, Date:
Sun Jun 24 20:53:53 2007
*/
#ifndef __SINGLE_THREAD_PROC_UC_H__
#define __SINGLE_THREAD_PROC_UC_H__

#define SRAM0_DATASEG_BASE 0xc000               /* SRAM0 data seg byteAddr */
#define SRAM0_DATASEG_SIZE 0x0                  /* SRAM0 data seg byteSize */
#define SRAM1_DATASEG_BASE 0x4                  /* SRAM1 data seg byteAddr */
#define SRAM1_DATASEG_SIZE 0x0                  /* SRAM1 data seg byteSize */
```

```
#define SRAM2_DATASEG_BASE 0x4              /* SRAM2 data seg byteAddr */
#define SRAM2_DATASEG_SIZE 0x0              /* SRAM2 data seg byteSize */
#define SRAM3_DATASEG_BASE 0x4              /* SRAM3 data seg byteAddr */
#define SRAM3_DATASEG_SIZE 0x0              /* SRAM3 data seg byteSize */

#define DRAM_DATASEG_BASE  0x10             /* DRAM data seg byteAddr */
#define DRAM_DATASEG_SIZE  0x0              /* DRAM data seg byteSize */
#define DRAM1_DATASEG_BASE 0x0              /* DRAM1 data seg byteAddr */
#define DRAM1_DATASEG_SIZE 0x0              /* DRAM1 data seg byteSize */

#define SCRATCH_DATASEG_BASE   0x4          /* SCRATCH data seg byteAddr */
#define SCRATCH_DATASEG_SIZE   0x300        /* SCRATCH data seg byteSize */

#define ME0_USTORE_USED   0x82          /* Num USTORE being used in ME0*/
#define ME1_USTORE_USED   0x1d3         /* Num USTORE being used in ME1*/
#define ME2_USTORE_USED   0x77          /* Num USTORE being used in ME2*/
#define ME3_USTORE_USED   0x0           /* Num USTORE being used in ME3*/
#define ME4_USTORE_USED   0x0           /* Num USTORE being used in ME4*/
#define ME5_USTORE_USED   0x0           /* Num USTORE being used in ME5*/
#define ME6_USTORE_USED   0x0           /* Num USTORE being used in ME6*/
#define ME7_USTORE_USED   0x0           /* Num USTORE being used in ME7*/
#define ME16_USTORE_USED   0x0          /* Num USTORE being used in ME16*/
#define ME17_USTORE_USED   0x0          /* Num USTORE being used in ME17*/
#define ME18_USTORE_USED   0x0          /* Num USTORE being used in ME18*/
#define ME19_USTORE_USED   0x0          /* Num USTORE being used in ME19*/
#define ME20_USTORE_USED   0x0          /* Num USTORE being used in ME20*/
#define ME21_USTORE_USED   0x0          /* Num USTORE being used in ME21*/
#define ME22_USTORE_USED   0x0          /* Num USTORE being used in ME22*/
#define ME23_USTORE_USED   0x77         /* Num USTORE being used in ME23*/

#endif      /* __SINGLE_THREAD_PROC_UC_H__   */
```

spi4_rx.c

```
// Include files
#include "ixp.h"
#include "spi4_rx.h"
#include "dl_buf.c"
#include "dl_meta.h"

extern dl_buf_handle_t          dlBufHandle;
extern __declspec(gp_reg) int   dlNextBlock;
extern dl_meta_t                dlMeta;


#define RBUF_ELEM_SIZE          128
#define RX_CONTROL_VAL          ((1 << RX_EN_SPHY_BITPOS) | \
                                        (1 << RBUF_ELE_SIZE_0_BITPOS))


__forceinline static void _spi4_rx_free_rbuf(unsigned int in_elem)
```

```
{
        void* val_and_addr;

        // The element number is placed into the upper 16
        // bits as required by the MSF fast_wr instruction
        val_and_addr = (void*)(MSF_RBUF_ELEMENT_DONE_ADDR | (in_elem << 16));
        msf_fast_write(val_and_addr);
}

void spi4_rx_init()
{
        void* addr; // Holds the address of the CSR being set
        __declspec(sram_write_reg) unsigned int rx_ctl;
        SIGNAL msf_rx_ctl_sig;

        //----------- Turn off packet reception while configuring the MSF
        rx_ctl = (1 << RBUF_ELE_SIZE_0_BITPOS);
        addr   = (void *)MSF_RX_CONTROL_ADDR;
        msf_write(&rx_ctl, addr, 1, ctx_swap, &msf_rx_ctl_sig);

        //----------- Specify the RBUF SPI4 High Water Mark
        //            (should be read & mask & rewrite)
        rx_ctl = MST_HWM_CONTROL__RBUF_S_HWM__01;
        addr   = (void *)MSF_HWM_CONTROL_ADDR;
        msf_write(&rx_ctl, addr, 1, ctx_swap, &msf_rx_ctl_sig);

        //----------- Enable the MSF Rx flow control (should be read & mask)
        rx_ctl = MSF_TRAIN_DATA__RSTAT_EN;
        addr   = (void *)MSF_TRAIN_DATA_ADDR;
        msf_write(&rx_ctl, addr, 1, ctx_swap, &msf_rx_ctl_sig);

        //----------- Turn on the receive control CSR in the MSF.
        rx_ctl = RX_CONTROL_VAL;
        addr   = (void *)MSF_RX_CONTROL_ADDR;
        msf_write(&rx_ctl, addr, 1, ctx_swap, &msf_rx_ctl_sig);
}
__forceinline
static spi4_rsw_t _spi4_rx_get_mpacket()
{
        SIGNAL rx_complete_sig;
        __declspec(sram_read_reg) spi4_rsw_t rsw;
        // This union represents the fields written into
        // the thread freelist CSR
        union
        {
                struct
                {
                        unsigned int    res             : 16,
                                        sig_no          : 4,
                                        me_no           : 5,
                                        thd             : 3,
                                        xfer_reg        : 4;
                } parts;
                unsigned int whole;
        } rx_tfl;
```

```
        void* rx_tfl_addr_and_val;

        rx_tfl.whole = 0;

        // Add the wakeup signal when an mpacket arrives
        rx_tfl.parts.sig_no = __signal_number(&rx_complete_sig);

        // Add the microengine number to singal
        rx_tfl.parts.me_no = __ME();

        // Add the context to signal
        rx_tfl.parts.thd = ctx();

        // Add the transfer register address
        // where the RSW words should be placed
        rx_tfl.parts.xfer_reg = __xfer_reg_number(&rsw);

        // Place the data into the upper 16 bits for
        // the fast_wr operation
        rx_tfl_addr_and_val = (void*) (MSF_RX_THREAD_FREELIST_0_ADDR |
                                                        (rx_tfl.whole << 16));
        msf_fast_write(rx_tfl_addr_and_val);
        wait_for_all(&rx_complete_sig); // wait for an mpacket
        return (rsw);
}


__forceinline static void _spi4_rx_move_rbuf_to_dram( unsigned int in_rbuf_elem,
                                void __declspec(dram) *in_dram_addr, unsigned int in_size,
                                                        SIGNAL_PAIR *in_dram_sig)
{
        dram_rbuf_tbuf_ind_t indir;
        unsigned int rbuf_addr;

        // Compute the RBUF address.  This is the base RBUF
        // address in the MSF plus the element number times
        // 64.  The multiplication by 64 comes from the fact
        // that the element number given in the RSW is divided
        // by 64
        rbuf_addr = MSF_RBUF_BASE_ADDR + (in_rbuf_elem << 6);

        // Override the rbuf addr
        indir.value = 0;
        indir.ov_buf_addr = 1;
        indir.buf_addr = rbuf_addr;

        // Override the transfer size
        indir.ov_ref_count = 1;
        indir.ref_count = ((in_size + 7) >> 3) - 1;

        dram_rbuf_read_ind(in_dram_addr, 16, indir, sig_done, in_dram_sig);
}


void spi4_rx()
```

```
{
        // A pointer into dram where
        // the next mpacket should be placed
        __declspec(dram) unsigned char *cur_mpacket_addr;
        __declspec(sram_read_reg) dl_buf_handle_t alloc_handle;
        spi4_rsw_t rsw;             // The receive status words
        SIGNAL_PAIR          rbuf_to_dram_sig;
        SIGNAL               buf_alloc_sig;
        dlBufHandle.value = 0;
        dlMeta.bufferSize  = 0;

        while(1)
        {
                // Get the next mpacket
                rsw = _spi4_rx_get_mpacket();

                // Check for errors in the packet
                // These indicate that the current buffer, if any,
                // should be discarded
                if (rsw.w1.parts.err)
                {
                        if (dlBufHandle.value != 0)
                        {
                                // Drop the packet
                                Dl_BufDrop(dlBufHandle);
                        }

                        _spi4_rx_free_rbuf(rsw.w1.parts.element);
                        dlBufHandle.value = 0;
                        dlMeta.bufferSize = 0;
                        continue;
                }

                // If this is the SOP, allocate a new buffer
                if (rsw.w1.parts.sop)
                {
                        if (dlBufHandle.value == 0)
                        {
                        Dl_BufAlloc(&alloc_handle,BUF_FREE_LIST0,BUF_POOL,
                        &buf_alloc_sig,          SIG_NONE,___);
                                wait_for_all(&buf_alloc_sig);
                                dlBufHandle = alloc_handle;

                                if (dlBufHandle.value == 0)
                                {
                                        // No more buffers
                                        _spi4_rx_free_rbuf(rsw.w1.parts.element);
                                        continue;
                                }
                        }
                        cur_mpacket_addr = (__declspec(dram) unsigned char *)
                                Dl_BufGetData(dlBufHandle);
                }
                else if (dlBufHandle.value == 0)
                {
```

```
                        // An MOP or EOP mpacket was received
                        // without an SOP mpacket first
                        _spi4_rx_free_rbuf(rsw.w1.parts.element);
                        continue;
                }

                // Move the mpacket into DRAM
                _spi4_rx_move_rbuf_to_dram(
                                        rsw.w1.parts.element, cur_mpacket_addr,
                                        rsw.w1.parts.byte_count, &rbuf_to_dram_sig);

                // Update the buffer length
                dlMeta.bufferSize += rsw.w1.parts.byte_count;

                // Wait for the mpacket to move into DRAM
                __wait_for_all(&rbuf_to_dram_sig);
                _spi4_rx_free_rbuf(rsw.w1.parts.element);

                // If this is the EOP mpacket then return
                if (rsw.w1.parts.eop)
                {
                        break;
                }

                // Update the reassembly pointer
                cur_mpacket_addr += rsw.w1.parts.byte_count;
        }
        dlNextBlock = SPI4_RX_NEXT_BLOCK;
}
```

---

spi4_rx.h

```
#ifndef SPI4_RX_H
#define SPI4_RX_H

// Include files
#include "rx.h"

extern void spi4_rx_init();
extern void spi4_rx();

#endif // SPI4_RX_H
```

---

spi4_rx_dl.c

```
//      Include Files
#include        "dl_system.h"
#include        "dl_source.h"
```

```
#include        "system_init.h"
#include        "dispatch_loop.h"
#include        "spi4_rx.h"

#include        "dl_buf.c"
#include        "dl_meta.h"

//-------------------------------------------------------------
// Global variables/Registers
//-------------------------------------------------------------
dl_buf_handle_t                 dlBufHandle;
__declspec(gp_reg) int          dlNextBlock;
extern dl_meta_t                dlMeta;

void main()
{
        //-------------------------------------------------------
        // Initialize the blocks
        //-------------------------------------------------------
        // Initialize the system
        system_init();

        // Initialize the outgoing ring
        dl_sink_init();

        // spi4 rx initialization
        spi4_rx_init();

        //-------------------------------------------------------
        // dispatch loop
        //-------------------------------------------------------
        while(1)
        {
                // Reassembly a packet
                spi4_rx();

                // Enqueue the packet on the rx to processing
                dl_sink();
        }
}
```

spi4_tx.c

```
#include "ixp.h"
#include "spi4_tx.h"
#include "dl_buf.c"
#include "dl_meta.h"
#include "dl_source.h"

extern dl_buf_handle_t          dlBufHandle;
extern __declspec(gp_reg) int   dlNextBlock;
```

```
extern dl_meta_t                dlMeta;


#define TBUF_ELEM_SIZE          64
#define TBUF_ELEM_SIZE_SHIFT    6
#define NUM_TBUFS               (128 * 64 / TBUF_ELEM_SIZE)
#define TX_CONTROL_VAL          ((1 << TX_EN_SPHY_BITPOS) | \
                (1 << TX_ENABLE_BITPOS) | \ (0 << TBUF_ELE_SIZE_0_BITPOS))

//--------------------------------------------------------------------
// This state is used to store the current packet's segmentation
// state
typedef struct s_tx_state
{
        // 1 if the current mpacket is SOP/EOP respectively
        unsigned int sop, eop;
        // A pointer to the current mpacket
        __declspec(dram) unsigned char *cur_mpacket_addr;
        // Length remaining, in bytes, of the current mpacket
        unsigned int remaining_length;
        // The handle of the current buffer
        dl_buf_handle_t cur_buf_handle;
} tx_state_t;
static tx_state_t tx_state;

void spi4_tx_init()
{
        void* addr; // Holds the address of the CSR being set
        __declspec(sram_write_reg) unsigned int tx_ctl;
        SIGNAL msf_tx_ctl_sig;

        tx_ctl = TX_CONTROL_VAL;
        addr   = (void *)MSF_TX_CONTROL_ADDR;
        msf_write(&tx_ctl, addr, 1, ctx_swap, &msf_tx_ctl_sig);
}
__forceinline
static void _spi4_tx_move_dram_to_tbuf(
        unsigned int in_tbuf_elem, void __declspec(dram) *in_dram_addr,
        unsigned int in_size, SIGNAL_PAIR *in_dram_sig)
{
        dram_rbuf_tbuf_ind_t indir;
        unsigned int tbuf_addr;

        // Compute the TBUF address.  This is the base TBUF
        // address in the MSF plus the element number times 64.
        tbuf_addr = MSF_TBUF_BASE_ADDR + (in_tbuf_elem << 6);
        // Override the tbuf address
        indir.value = 0;
        indir.ov_buf_addr = 1;
        indir.buf_addr = tbuf_addr;

        // Override the transfer size
        indir.ov_ref_count = 1;
        indir.ref_count = ((in_size + 7) >> 3) - 1;
        dram_tbuf_write_ind(in_dram_addr, 8, indir, sig_done, in_dram_sig);
```

```
}


__forceinline
static void _spi4_tx_validate_tbuf(unsigned int in_tbuf_elem, unsigned int in_sop,
                                                    unsigned int in_eop, unsigned int in_size)
{
        spi4_tcw_t tbuf_control;
        __declspec(sram_write_reg) spi4_tcw_t tbuf_control_wr;
        SIGNAL msf_sig;

        tbuf_control.control_word.whole = 0;
        tbuf_control.reserved = 0;

        // Set the mpacket length
        tbuf_control.control_word.parts.payload_length = in_size;

        // Set SOP and EOP
        tbuf_control.control_word.parts.sop = in_sop;
        tbuf_control.control_word.parts.eop = in_eop;
        tbuf_control_wr = tbuf_control;

        msf_write(&tbuf_control_wr,
                (void *)(MSF_TBUF_CONTROL_BASE_ADDR + (in_tbuf_elem<<3)),
                sizeof(tbuf_control_wr) / sizeof(unsigned int), ctx_swap, &msf_sig);
}

__forceinline
static void _spi4_tx_update_tbufs_in_flight( unsigned int *io_tbufs_in_flight,
                                                        unsigned int *io_last_tx_seq)
{
        unsigned int cur_tx_seq;
        __declspec(sram_read_reg) unsigned int cur_tx_seq_rd;
        unsigned int tbufs_used;
        SIGNAL msf_sig;

        // First read the current sequence number
        msf_read(&cur_tx_seq_rd,
                        (void *)MSF_TX_SEQUENCE_0_ADDR,sizeof(cur_tx_seq_rd) /
                        sizeof(unsigned int), ctx_swap, &msf_sig);
        cur_tx_seq = cur_tx_seq_rd & 0xff;

        // Compute how many TBUFs have been consumed since
        // the last read.  Account for wrap around
        if (*io_last_tx_seq <= cur_tx_seq)
        {
                tbufs_used = cur_tx_seq - *io_last_tx_seq;
        }
        else
        {
                tbufs_used = (256 + cur_tx_seq) - *io_last_tx_seq;
        }

        // Subtract the tbufs_used from the current number of
        // tbufs in flight
```

```
        *io_tbufs_in_flight -= tbufs_used;

        // Save the sequence number
        *io_last_tx_seq = cur_tx_seq;
}


__forceinline
static tx_state_t _spi4_tx_get_and_update_state(
        unsigned int in_next_tbuf_elem)
{
        tx_state_t ret_state;

        // If EOP is true, get a new packet
        if (tx_state.eop)
        {
                while (1)
                {
                        // Dequeue a packet from the processing task
                        dl_source();

                        // Check for an empty queue
                        if (dlBufHandle.value != 0)
                        {
                                // The queue was not empty
                                tx_state.sop = 1;
                                tx_state.cur_buf_handle
                                        = dlBufHandle;
                                tx_state.cur_mpacket_addr =
                                        (__declspec(dram) unsigned char *)
                                        Dl_BufGetData(dlBufHandle);
                                tx_state.remaining_length
                                        = dlMeta.bufferSize;
                                break;
                        }
                }
        }

        ret_state.cur_mpacket_addr =  tx_state.cur_mpacket_addr;
        ret_state.cur_buf_handle   =  tx_state.cur_buf_handle;
        ret_state.sop = tx_state.sop;

        // Update the global state for the next call to
        // this macro.  Check for EOP
        if (tx_state.remaining_length <= TBUF_ELEM_SIZE)
        {
                tx_state.eop = 1;
                ret_state.remaining_length = tx_state.remaining_length;
        }
        else
        {
                tx_state.eop = 0;
                ret_state.remaining_length = TBUF_ELEM_SIZE;
        }
```

```
        tx_state.cur_mpacket_addr += TBUF_ELEM_SIZE;
        tx_state.remaining_length -= TBUF_ELEM_SIZE;
        tx_state.sop = 0;
        ret_state.eop = tx_state.eop;
        return ret_state;
}

void spi4_tx()
{
        // This state is used during the transmission process to ensure
        // that the TBUFs are used in order and without overrunning the
        // hardware

        // The index of the next tbuf element to be
        // used
        unsigned int next_tbuf_elem;
        // The value of the last read to the
        // tx_sequence number
        unsigned int last_tx_seq;
        // The number of TBUFs currently being
        // transmitted.  Based on the the last
        // time the tx sequence number was read
        unsigned int tbufs_in_flight;

        SIGNAL_PAIR dram_to_tbuf_sig;
        // State associated with the current mpacket
        tx_state_t cur_state;

        // Initialize the transmit state
        next_tbuf_elem  = 0;
        tbufs_in_flight = 0;
        last_tx_seq     = 0;

        // Setting the global EOP = 1 will force a dequeue
        tx_state.eop    = 1;

        while(1)
        {
                // Check that the TBUF is available for use
                while (tbufs_in_flight == NUM_TBUFS)
                {
                        // We are out of TBUFs, wait for the
                        // sequence number to increase
                        _spi4_tx_update_tbufs_in_flight(&tbufs_in_flight,&last_tx_seq);
                }

                // Get the state (next mpacket) for the current
                // TBUF element.
                cur_state = _spi4_tx_get_and_update_state(next_tbuf_elem);

                // Move the next portion of the packet into
                // the next tbuf
                _spi4_tx_move_dram_to_tbuf(next_tbuf_elem, cur_state.cur_mpacket_addr,
                                           cur_state.remaining_length,  &dram_to_tbuf_sig);
```

```
                // As an optimization, if we have only one
                // more TBUF available, then
                // read and update the tbufs in fight during
                // the transfer from DRAM to TBUF
                if (tbufs_in_flight == (NUM_TBUFS - 1))
                {
                        _spi4_tx_update_tbufs_in_flight(&tbufs_in_flight, &last_tx_seq);
                }

                // Wait for the TBUF to be filled
                wait_for_all(&dram_to_tbuf_sig);

                // Write the TBUF control word to validate
                // the entry
                _spi4_tx_validate_tbuf(next_tbuf_elem, cur_state.sop, cur_state.eop,
                                                                cur_state.remaining_length);

                // Update the global transmit state
                next_tbuf_elem += TBUF_ELEM_SIZE / 64;
                next_tbuf_elem &= 0x7f;
                tbufs_in_flight++;

                if (cur_state.eop)
                {
                        // Free the buffer
                        Dl_BufDrop(cur_state.cur_buf_handle);
                }
        }
}

void exit(unsigned int code)
{
        /* EMPTY */
}
```

---

spi4_tx.h

```
#include "tx.h"

void spi4_tx_init();
void spi4_tx();
```

---

spi4_tx_dl.c

```
// Include Files
#include        "dl_system.h"
#include        "dl_source.h"
#include        "system_init.h"
```

```
#include        "dispatch_loop.h"
#include        "spi4_tx.h"
#include        "dl_buf.c"
#include        "dl_meta.h"

//-------------------------------------------------------------
// Global variables/Registers
//-------------------------------------------------------------
dl_buf_handle_t                 dlBufHandle;
__declspec(gp_reg) int          dlNextBlock;
extern dl_meta_t                dlMeta;


void main()
{
        //-------------------------------------------------------
        // Initialize the blocks
        //-------------------------------------------------------
        // Initialize the system
        system_init();

        // spi4 tx initialization
        spi4_tx_init();

        // Initialize the incoming ring
        dl_source_init();

        //-------------------------------------------------------
        // dispatch loop
        //-------------------------------------------------------
        while(1)
        {
                // Get, segment and transmit the
                // packet
                spi4_tx();
        }
}
```

sram_rings.c

```
#include "ixp.h"
#include "dl_buf.h" // Include the buf_form_q_descriptor prototype
#include "scratch_rings.h" // Include the ring data definition
#include "sram_rings.h"

__declspec(sram) unsigned char *buf_form_q_desc_addr(
                        unsigned int in_channel, unsigned int in_entry,  unsigned int in_offset)
{
        return (__declspec(sram) unsigned char *)
                ((in_entry << QDESC_ENTRY_BITPOS) |
                (in_channel << QDESC_CHANNEL_BITPOS) |
```

```
                (in_offset >> 2)); // Convert to SRAM words
}

void sram_ring_init(unsigned int ring_number,  unsigned int ring_base, unsigned int ring_size)
{
        typedef struct
        {
                unsigned int head;
                unsigned int tail;
                unsigned int count;
                unsigned int other;
        } q_desc_t;
        __declspec(sram_write_reg) q_desc_t wr_q_desc;
        __declspec(sram_read_reg) q_desc_t rd_q_desc;

        unsigned int addr, ring_base_no_channel;
        __declspec(sram) unsigned char *q_desc_addr;
        SIGNAL init_sig;
        unsigned int count;
        dl_buf_handle_t handle;

        // Bits 31 - 29 of the head pointer contain an
        // encoding of the size. 000 = 512 words,
        // 001 = 1024, etc., followed by a long word
        // pointer to the base address (without the channel)
        // Write these words into the begining of the
        // ring data
        ring_base_no_channel = ring_base & (~(0x3<<QDESC_CHANNEL_BITPOS));
        wr_q_desc.head  = ((ring_base_no_channel>>2) |
                (((ring_size/512)-1)<<SRAM_RING_BASE_SIZE_BITPOS));
        wr_q_desc.tail  = ((ring_base_no_channel>>2) |
                (((ring_size/512)-1)<<SRAM_RING_BASE_SIZE_BITPOS));
        wr_q_desc.count = 0;
        wr_q_desc.other = 0;
        addr = ring_base;
        sram_write(&wr_q_desc, (__declspec(sram) unsigned char *)addr,
                        sizeof(wr_q_desc) / sizeof(unsigned int), ctx_swap, &init_sig);

        // Read the zero's back into the queue array to
        // initialize the freelist
        q_desc_addr = buf_form_q_desc_addr(SRAM_RING_CHANNEL, ring_number,
                                                ring_base_no_channel);
        sram_read_qdesc_head((__declspec(sram_read_reg) unsigned int *)
                                        &rd_q_desc, q_desc_addr, 2,ctx_swap, &init_sig);
        sram_read_qdesc_other(q_desc_addr);

        // Initialize the global ring data
        g_rings[ring_number].num_packets = 0;
        g_rings[ring_number].max_packets = (ring_size/512 * 2048) / sizeof(ring_data_t);
        g_rings[ring_number].last_empty_timestamp = 0;
}



void sram_ring_put_buffer(unsigned int     ring_number, ring_data_t data)
```

```
{
        __declspec(sram) void* ring_addr =(__declspec(sram) void *)
                        ((SRAM_RING_CHANNEL<<QDESC_CHANNEL_BITPOS) |
                          (ring_number<<2));
        SIGNAL_PAIR ring_signal;
        volatile __declspec(sram_write_reg) ring_data_t packet;

        // !The compiler is associating the size of the put data with the size of the returned status
        __declspec(sram_read_reg) unsigned int status[4];

        // Wait for the ring to be not full
        while(g_rings[ring_number].num_packets >= g_rings[ring_number].max_packets)
        {
        }

        // Increment length of ring
        scratch_incr((__declspec(scratch) void*) & (g_rings[ring_number].num_packets));
        packet = data;
        sram_put_ring(&status[0], (void *)&packet, ring_addr,
                                sizeof(packet) / sizeof(unsigned int), sig_done, &ring_signal);
        wait_for_all(&ring_signal);
        // The status should never indicate the ring is full because of the check above
}

unsigned int sram_ring_get_size(unsigned int ring_number)
{
        return g_rings[ring_number].num_packets;
}

unsigned int sram_ring_get_empty_timestamp(unsigned int ring_number)
{
        return g_rings[ring_number].last_empty_timestamp;
}
```

---

sram_rings.h

```
#ifndef SRAM_RINGS_H
#define SRAM_RINGS_H

// All SRAM rings accessed with this code come from the same controller (i.e., channel), as
// defined in the following. This has the benefit of spreading the SRAM accesses across
// different SRAM channels as well as avoiding collisions between the SRAM ring memory
// and other data structures
#define SRAM_RING_CHANNEL                               1

// Override the processing to tx ring definition
#define PROCESSING_TO_TX_SRAM_RING                      1
#define PROCESSING_TO_TX_SRAM_RING_BASE                 0x40006000
#define PROCESSING_TO_TX_SRAM_RING_SIZE                 512
#define PROCESSING_TO_TX_SRAM_RING_SIZE_SHIFT  9
#define SRAM_RING_BASE_SIZE_BITPOS                      29
```

```
//--------------------------------------------------------------------
// Bit positions for forming a queue descriptor address
#define QDESC_CHANNEL_BITPOS                    30
#define QDESC_ENTRY_BITPOS                      24

//--------------------------------------------------------------------
// MicroC data structures
#ifdef MICRO_C
#include "dl_buf.h"
#include "scratch_rings.h"

// For each ring, the following information is maintained.
// This information is used by RED to maintain statistics about
// the state of the ring
typedef struct
{
        unsigned int            num_packets;
        unsigned int            max_packets;
        unsigned int            last_empty_timestamp;
} ring_t;

extern __declspec(export shared scratch) ring_t g_rings[64];

// Function prototypes

void sram_ring_init(unsigned int ring_number, unsigned int ring_base, unsigned int ring_size);


void sram_ring_put_buffer(unsigned int ring_number, ring_data_t data);

unsigned int sram_ring_get_size(unsigned int ring_number);

unsigned int sram_ring_get_empty_timestamp(unsigned int ring__number);

static __forceinline
void sram_ring_get_buffer(unsigned int     ring_number, ring_data_t *data)
{
        __declspec(sram) void* ring_addr =(__declspec(sram) void *)
                        ((SRAM_RING_CHANNEL<<QDESC_CHANNEL_BITPOS) |
                          (ring_number<<2));
        SIGNAL ring_signal;
        __declspec(sram_read_reg) ring_data_t packet;

        sram_get_ring(&packet, ring_addr, sizeof(packet) / sizeof(unsigned int),
                                                        ctx_swap, &ring_signal);

        *data = packet;

        // Decrement length of ring
        if (packet.handle.value)
        {
                scratch_decr((__declspec(scratch) void*)
                &(g_rings[ring_number].num_packets));
```

```
                // If the ring is now empty, update the timestamp
                if (!g_rings[ring_number].num_packets)
                {
                        g_rings[ring_number].last_empty_timestamp =
                                local_csr_read(local_csr_timestamp_low);
                }
        }
}

#endif // MICRO_C
#endif // RINGS_H
```

---

system_init.h

```
#ifndef SYSTEM_INIT_H
#define SYSTEM_INIT_H

#include "dl_buf.h"

__forceinline
static void system_init()
{
        if (ctx() == 0 && __ME() == 0)
        {
                // Initialise the free buffer list
                Dl_BufInit();
        }
}

#endif  //SYSTEM_INIT_H
```

---

tx.h

```
#ifndef TX_H
#define TX_H

//---------------------------------------------------------------------
// Bit positions within the MSF transmit control CSR
#define TX_EN_SPHY_BITPOS                       29
#define TX_ENABLE_BITPOS                        10
#define TBUF_ELE_SIZE_0_BITPOS                  2
#define TBUF_PARTITION_BITPOS                   0

//---------------------------------------------------------------------
// MSF memory map
#define MSF_TX_CONTROL_ADDR                     0x4
#define MSF_TX_SEQUENCE_0_ADDR                  0x60
#define MSF_TBUF_CONTROL_BASE_ADDR              0x1800
#define MSF_TBUF_BASE_ADDR                      0x2000
```

```
//-----------------------------------------------------------------
// MicroC data structures
#ifdef MICRO_C

//-----------------------------------------------------------------
// The SPI4 tbuf control words
typedef struct s_spi4_tcw
{
        union
        {
                struct
                {
                        unsigned int payload_length : 8,
                                     prepend_offset : 3,
                                     prepend_length : 5,
                                     payload_offset : 3,
                                     res1 : 1,
                                     skip : 1,
                                     res2 : 1,
                                     sop : 1,
                                     eop : 1,
                                     addr : 8;
                } parts;
                unsigned int whole;
        } control_word;
        unsigned int reserved;
} spi4_tcw_t;

#endif // MICRO_C
#endif //TX_H
```

counteth.c

```
extern __declspec(gp_reg) int    dlNextBlock;

INLINE void counteth ( void )

{
   //comprobamos si este paquete es para nosotros
        if ( dlNextBlock != BID_COUNT )

      return ;

    sram_incr( ( volatile void _declspec(sram) * )( COUNT_SRAM_ADDR ) ) ;

    //enviamos el paquete al siguiente bloque
         dlNextBlock = ETHERNET_VALIDATE_LOCAL ;
    return ;
}
```

BIBLIOGRAFÍA

[1] Patrick Crowley, Mark A. Franklin, Haldun Hadimioglu, Peter Z. Onufry. Network Processor Design, Issues and Practices. Volume 1. Morgan Kaufmann Publishers, 2003.

[2] Panos C. Lekkas. Network Processors, Architectures, Protocols, and Platforms. McGraw-Hill, 2003.

[3] Intel® IXP2XXX Product Line of Network Processors, Development Tools User's Guide. 2006.

[4] Intel® Internet Exchange Architecture Software Development Kit, Installation Guide. 2006.

[5] Niraj Shah. Understanding Network Processors.2001

[6] www.Embedded.com - Network Processor Programming

[7] SISTEMAS AGERE:
http://www.lsi.com/networking_home/networking_products/network_processors/index.html

[8] PMC-SIERRA
http://www.pmc-sierra.com/
SDK ClassiPI:
https://www.pmc-sierra.com/products/details/pm2329/

[9] TRANSWITCH-ASPEN
http://server-die.alc.upv.es/electroweb/fabricantes/empresas/transwitch.htm

[10] INTEL
http://www.intel.com/design/network/products/npfamily/ixp2400.htm
Para descargar el SDK:
http://www.intel.com/design/network/products/npfamily/sdk_download.htm
Premier Members en Madrid:
http://premierlocator.intel.com/select_profile.asp
Contacto:
http://www.intel.com/cd/corporate/contact/emea/spa/254122.htm
Distribuidores autorizados:
http://indigo.cps.intel.com/dlt/distributorList.aspx?country=Spain
Dónde comprar:
http://www.intel.com/buy/networking/design.htm?matrix=npfamily&prod=IXP2400

[11] IBM POWERNP
http://www.ibm.com/es/

[12] MOTOROLA C-5E
http://www.motorola.com/mediacenter/news/detail.jsp?globalObjectId=3332_2752_23

[13] CISCO
http://www.cisco.com/global/ES/index.shtml
Para poder bajar software, es necesario registrarse:
http://tools.cisco.com/RPF/register/register.do
El software que se puede adquirir es el siguiente:
http://www.cisco.com/public/swcenter/index.shtml