UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DA COMPUTAÇÃO

THIAGO CABERLON SANTINI

Implementação de uma Rede em Chip com Suporte a *Clusters* Dinâmicos e *Multicast*

Trabalho de Conclusão apresentado como requisito parcial para a obtenção do grau de Engenheiro da Computação

Prof. Dr. Flávio Rech Wagner
Orientador

Mestre Gustavo Girão Barreto da Silva
Co-orientador

Porto Alegre, julho de 2012

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Santini, Thiago Caberlon

Implementação de uma Rede em Chip com Suporte a *Clusters* Dinâmicos e *Multicast* / Thiago Caberlon Santini. – Porto Alegre: IF da UFRGS, 2012.

50 f.: il.

Trabalho de Conclusão – Universidade Federal do Rio Grande do Sul. Curso de Engenharia da Computação, Porto Alegre, BR–RS, 2012. Orientador: Flávio Rech Wagner; Co-orientador: Gustavo Girão Barreto da Silva.

1. Redes em chip. 2. Systemc. 3. Microeletrônica. 4. Multicast. 5. Cluster. I. Wagner, Flávio Rech. II. da Silva, Gustavo Girão Barreto.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitora de Graduação: Prof.ª Valquiria Linck Bassani
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do curso: Prof. Sérgio Luís Cechin
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“An optimist will tell you the glass is half-full; the pessimist, half-empty; and the engineer will tell you the glass is twice the size it needs to be.“

— Oscar Wilde

AGRADECIMENTOS

Agradeço, primeiramente, aos meus pais, Gilberto Santini e Maria Inês Caberlon Santini, pelo apoio, carinho, oportunidades e lições durante todo esse tempo.

Ao meu irmão, Diego Caberlon Santini, agradeço pelo exemplo, incentivo e, claro, lutinhas!

À minha namorada, Nicole Fitarelli Gehrke, por seu carinho e compreensão.

Também agradeço ao meu orientador, Flávio Rech Wagner, pelos mais de três anos sob sua tutela, nos quais muito aprendi.

Ao meu co-orientador e amigo, Gustavo Girão, pelas valiosas discussões, idéias e conversas.

À minha família, colegas de curso, de laboratório, colegas da Datacom e aos professores da UFRGS pelas suas contribuições tanto ao desenvolvimento deste trabalho quando ao meu desenvolvimento pessoal.

Einen Dank auch an Patrick Heckeler für die Unterstützung in Deutschland.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

bop	begin of packet
eop	end of packet
E/S	Entrada e Saída
FCFS	First Come, First Served
FIFO	First In, First Out
HLP	Higher Level Protocol
HTV	High Threshold Voltage
IB	Input Buffer
IC	Input Controller
IFC	Input Flow Controller
ILP	Instruction Level Parallelism
IP	Intelectual Property
IRS	Input Read Switch
ISA	Instruction Set Architecture
LTV	Low Threshold Voltage
LUT	Look Up Table
ME	Multicast Enabler
NoC	Network on Chip
NTV	Normal Threshold Voltage
OC	Output Controller
ODS	Output Data Switch
ORS	Output Read Switch
RASoC	Router Architecture for System on Chip
RIB	Routing Information Bits
RTL	Register Transfer Level
SoCIN	System on Chip Interconnection Network

SoCMEIN	System on Chip Multicast Enabled Interconnection Network
SoC	System on Chip
TLP	Task Level Parallelism

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Sistemas em Chip de núcleo único utilizam o paralelismo de instruções para agilizar a computação e conforme o limite deste paralelismo é alcançado é necessário buscar alternativas. Graças ao aumento contínuo da taxa de integração de transistores tornou-se possível a multiplicação de núcleos dentro de um único chip, surgindo assim um novo paradigma cujo objetivo é o paralelismo de tarefas. Neste novo paradigma surge um aumento elevado nas taxas de comunicações entre os elementos do sistema. Isto exige um mecanismo de comunicação que possa atender um alto número de requisições simultaneamente e que seja facilmente replicável, eliminando assim a viabilidade de uso de barramentos tradicionais. Uma nova proposta de interconexão para estes sistemas são as Redes em Chip (NoCs). Mapeando tarefas em um conjunto de recursos, como núcleos e memórias, é possível criar *clusters* para tratar tarefas. Com a disponibilidade de múltiplos recursos é possível criar diversos *clusters*, e é desejável que eles não interfiram entre si. Além disso, uma parcela significativa da comunicação quando utiliza-se paralelismo a nível de tarefas tende a ser de um para muitos nodos. Atualmente, *multicasts* são suportados através de múltiplos *unicasts* ou *broadcast*. Infelizmente esses métodos não são eficientes. Este trabalho implementa e modifica uma Rede em Chip utilizando SystemC, adicionando suporte a *multicast* e *clusters* no nível de rede sem um grande aumento de área, comparando a primeira implementação com a implementação modificada. Também é feito um modelo de área e potência utilizando a biblioteca Orion 2.0.

Palavras-chave: Redes em chip, systemc, microeletrônica, multicast, cluster.

Implementation of a Network on a Chip Supporting Dynamic Clusters and Multicast

ABSTRACT

Single core System on Chip relays on ILP to make computing faster and, as this parallelism's limit is reached a need for alternatives arises. The continuous raise on the rate of transistor integration has enabled core multiplication into a single chip, giving birth to a new paradigm that benefits from task parallelism. In the light of this new paradigm SoCs tend to become communication-bound, requiring a communication bus that can serve a high number of requisitions simultaneously and that is easily replicable. These requirements prevent the use of traditional buses. A new proposal to solve this problem is based on Networks on Chip. By mapping tasks into a set of resources, like cores and memories, it is possible to create clusters to handle tasks. With multiple resources available it is possible to create several clusters, and it is desirable that they don't interfere with each other. Also, a significant part of the communication when using TLP tends to be from one to many nodes. Currently, multicasts are supported through multiple unicasts or a broadcast. Unfortunately these methods are not efficient. This work implements and modifies a Network on Chip using SystemC, adding multicast and dynamic cluster support at the network level without a big area overhead, comparing the first implementation with the modified one. Also an area and power model is created using Orion 2.0.

Keywords: network on chip, systemc, microeletronic, multicast, cluster.

1 INTRODUÇÃO

Um microprocessador é um circuito integrado que executa instruções definidas em sua arquitetura de conjunto de instruções. O primeiro microprocessador disponível comercialmente, o Intel 4004, foi introduzido em 1971 e possuia 2,300 transistores de 10 micrometros integrados, rodando a uma frequência máxima de 704KHz. Um processador moderno tende a ser complexo e pode ter na ordem de bilhões de transistores, cada um na ordem de dezenas de nanômetros rodando a frequências na ordem de gigahertz. A indústria de semicondutores tem se destacado nas últimas quatro décadas pela rapidez de seu crescimento e avanço tecnológico. Entre os responsáveis por esta evolução estão o nível de integração, redução do custo de fabricação, aumento de velocidade, redução de potência e novas funcionalidades (SEMATECH, 2010).

Entre os principais responsáveis pelo rápido progresso está a taxa de integração que, aproximadamente, dobra o número de componentes em um chip a cada vinte e quatro meses (MOORE, 1965). Este crescimento permitiu aumentar rapidamente a complexidade dos circuitos. No contexto de processadores, este fato culminou em processadores superescalares, que buscam aproveitar ao máximo o paralelismo a nível de instruções através da combinação de técnicas como *pipelines*, memórias *cache* e execução fora de ordem.

Devido à alta complexidade, estes sistemas possuem um alto tempo de projeto, inviável no ágil mercado atual. Além disso, obter maior desempenho através do aumento da frequência também se torna proibitivo devido ao fato de que a frequência influencia diretamente a potência dissipada. Como exposto na Equação 1.1, o aumento da frequência de operação acarreta em um aumento na potência dinâmica. Esta, por sua vez é a maior componente na potência total de um chip atualmente. A potência é altamente relevante atualmente devido à ampla adoção de sistemas embarcados como celulares, *tablets*. *netbooks* e *laptops* que possuem uma fonte de energia limitada. Este fato leva ao uso de múltiplos núcleos com uma frequência mais baixa ao invés de um núcleo mais complexo com frequência alta.

$$P = 0,5 f_c V_{dd}^2 \sum_{i=1}^{n} C_i \alpha(x_i) \tag{1.1}$$

Onde:

$f_c =$ frequência de operação (Hz)

$V_{dd} =$ tensão de alimentação do circuito [V]

$n =$ número de portas do circuito

$C_i =$ capacitância equivalente no nodo x_i $[F]$

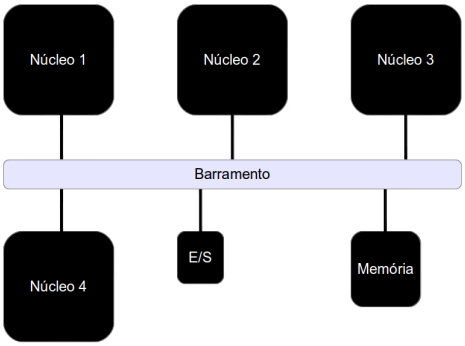


Figura 1.1: SoC interconectada por barramento simples.

$$\alpha(x_i) = \text{taxa de comutação de cada porta}$$

O aumento na densidade de transistores, frequências de operações mais altas, curto tempo de projeto para o mercado e o curto ciclo de vida de produtos caracterizam a indústria de semicondutores atual. Com estas condições impostas, projetistas estão integrando componentes complexos e funcionalmente heterogêneos em um único chip, chamado Sistema em Chip (SoC). Segundo (GUPTA; ZORIAN, 1997) et al. o projeto de SoCs se baseia no reuso de núcleos de propriedade intelectual (IP). Esses IPs consistem de blocos de *hardware* pré-verificados que podem ser usados para constituir um sistema maior e complexo. Dentre os tipos de núcleos podemos citar processadores, controladores de memória, periféricos, etc.

Porém, é necessário mais do que somente IPs para constituir um SoC. É necessário um mecanismo que torne possível a comunicação entres esses núcleos. Normalmente a interconexão dos elementos dos circuitos é feito através de fios ou barramentos compartilhados. Fios não apresentam uma boa flexibilidade nem reuso. Atualmente, a maioria dos SoCs usam uma arquitetura com um meio compartilhado (Figura 1.1), que possui estruturas de interconexão simples onde todos elementos compartilham o meio de comunicação e somente um elemento pode usar o barramento por vez, sendo necessária uma arbitragem quando múltiplos elementos tentam usar o barramento simultâneamente. Para utilizar o meio, o elemento precisa primeiro conseguir permissão do árbitro, isto gera uma transação de controle reduzindo a eficiência da comunicação, logo é desejável que estes eventos sejam rápidos e escassos. Apesar de bem compreendidos e amplamente utilizados, barramentos compartilhados possuem uma séria limitação de escalabilidade. Organizações baseadas em barramentos permanecem convenientes para SoCs que possuem menos do que cinco processadores (BENINI; MICHELI, 2002).

As arquiteturas do futuro necessitarão de comunicação em chip que sejam eficientes e que possuam uma boa escalabilidade pois conterão entre dezenas a centenas de núcleos (HELD; BAUTISTA; KOEHL, 2006). Uma alternativa de conexão é baseada em roteadores, assim, a comunicação entre dois elementos segue um caminho definido pelo algoritmo de roteamento. A cada roteador é ligado uma interface onde podem ser conectados IPs (Figura 1.2). Esta solução, conhecida como Rede em Chip (Noc), é apresentada como uma solução para o problema da complexidade, flexibilidade e escalabilidade de projeto.

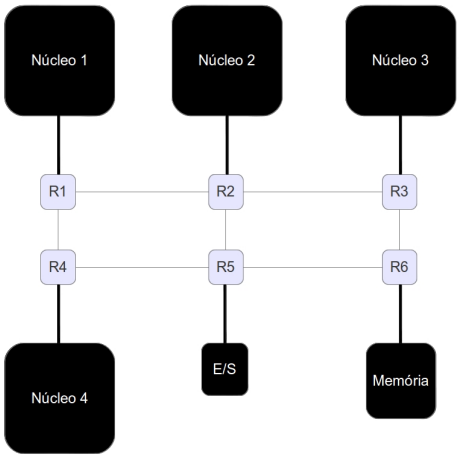


Figura 1.2: SoC interconectada por NoC.

A maioria das pesquisas em NoC até agora assumem que a maior parte do tráfego é de um-para-um (*unicast*), existindo assim a ausência de um método eficaz para realizar *multicasts* e *broadcasts*. Algumas propostas como (JERGER; PEH; LIPASTI, 2008) e (SAMMAN; HOLLSTEIN; GLESNER, 2008) trataram do assunto mas assumiram uma fase global de configuração e de liberação do *multicast*. (LIU; ZHENG; TENHUNEN, 2003) oferece multicast de um nodo para todos seus vizinhos.

Além disso, o compartilhamento de um barramento pode causar interferência na comunicação, levando assim a uma degradação na performance de uma das aplicações. Isto representa um problema grave caso alguma das aplicações possua requisitos temporais restritos. Uma funcionalidade interessante, seria um mecanismo que permita que a uma aplicação isolar-se do resto do sistema. Em NoCs, isto pode ser alcançado através de um mecanismo que separe a rede em *clusters* capazes de rejeitar tráfego oriundo de fora *cluster*.

1.1 Objetivos

Este trabalho propõe e implementa um método para a realização de *multicasts*, bem como um suporte simples para identificação de *clusters* dinâmicos na NoC SoCIN (ZEFERINO; SUSIN, 2003).

Como objetivos tem-se realizar uma implementação da SoCIN, criando também um modelo de área e potência, modificá-la para adicionar suporte a *multicast* e a *clusters* dinâmicos e realizar uma comparação entre a SoCIN original e a SoCIN com suporte a *multicast*.

1.2 Organização

O Capítulo 2 faz uma introdução a conceitos de redes e faz uma breve análise do estado da arte de suporte a *multicast* e suporte a *cluster* em NoCs. O Capítulo 3 entra em detalhes específicos ao projeto da SoCIN e identifica os módulos usados para construí-la. O Capítulo 4 detalha a implementação e modificações realizadas para adicionar suporte a *multicast* e *cluster*. O Capítulo 5 descreve os modelos de área e potência utilizados,

enquanto o Capítulo 6 apresenta e analisa as comparações feitas e resultados. O Capítulo 7 conclui o trabalho.

2 REDES EM CHIP E ESTADO DA ARTE

A Seção 2.1 introduz alguns conceitos básicos de redes necessários para a compreensão do trabalho, enquanto a Seção 2.2 faz uma breve análise de propostas de solução para implementação de suporte a *multicast* e *clusters* em NoCs.

2.1 Redes em Chip

Uma rede em chip consiste de um conjunto de roteadores e canais ponto a ponto interligando-os. Além disso, é necessário, em alguns roteadores, haver um canal de comunicação com o exterior da rede, por onde pacotes são inseridos e retirados da rede. NoCs baseiam-se no modelo de comunicação de troca de mensagens, no qual cada pacote é uma mensagem.

Normalmente o pacote divide-se em três partes distintas: cabeçalho, carga e cauda (Figura 2.1). A seguir são apresentados alguns conceitos de redes como *phit*, *flit*, *hop*, *unicast*, *multicast*, topologia, controle de fluxo, roteamento, arbitragem, chaveamento, memorização, *deadlock*, *livelock*, *starvation*, latência e vazão.

2.1.1 *Phit e Flit*

Um *phit* equivale à menor unidade física de informação que traféga na rede, ou seja, o tamanho físico dos canais de comunicação. Difere-se de *flit*, a menor unidade de informação na qual é realizado controle de fluxo. Apesar de serem conceitos diferentes, é

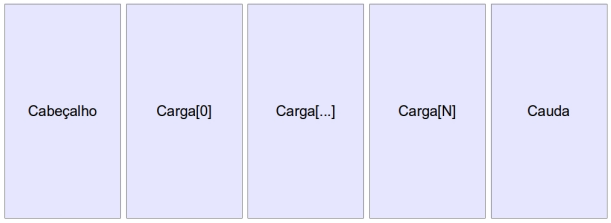


Figura 2.1: Exemplo de pacote.

Figura 2.2: Topologia grelha e toroidal.

possível que possuam o mesmo tamanho caso o controle de fluxo seja realizado a cada *Phit*.

2.1.2 *Hop*

Hop é uma unidade de medida utilizada para representar a passagem de um pacote ou *flit* de um roteador para um roteador adjacente na rota até seu destino. A quantidade de *hops* necessária para alcançar um roteador destino, a partir do roteador fonte, define a distância topológica entre eles.

2.1.3 *Unicast* e *Multicast*

Unicast é o termo usado para representar o envio de um pacote de um nodo a outro, sendo cada nodo identificado por um endereço único na rede. Analogamente, caso a mensagem seja transmitida a partir de um nodo para muitos nodos, utiliza-se o termo *multicast*. Por fim, caso todos os nodos da rede sejam alvos no *multicast* utiliza-se o termo *broadcast*.

2.1.4 Topologia

A topologia da rede consiste no modo como seus roteadores são interligados. O número de roteadores ao qual um roteador está conectado define o seu nível de conectividade. O nível de conectividade é proporcional ao custo e inversamente proporcional à escalabilidade da rede. Pode-se dividir as topologias em dois grupos, redes com conectividade direta e redes com conectividade indireta.

Redes com conectividade direta caracterizam-se por ter um recurso conectado a cada roteador, enquanto nas redes com conectividade indireta somente alguns roteadores possuem um recurso conectado. Alguns exemplos de topologias diretas são as topologias do tipo grelha e toroidal (Figura 2.2). As topologias indiretas variam muito e são representadas principalmente por redes multiestágio.

2.1.5 Controle de Fluxo

O controle de fluxo garante que os recursos necessários para uma mensagem avançar estão disponíveis, regulando o tráfego e evitando a perda de pacotes. Diversos tipos de

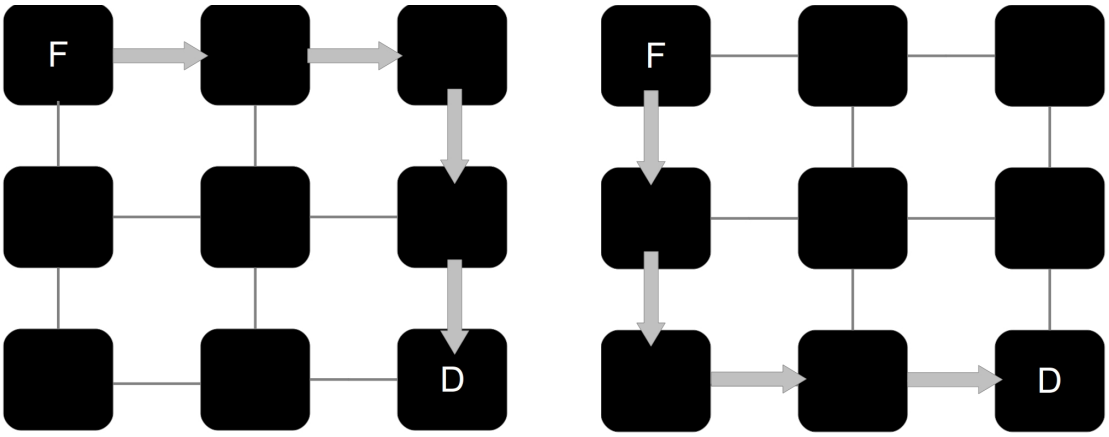


Figura 2.3: Roteamento XY vs YX.

controle de fluxo existem, entre eles *handshake*, canais virtuais e controle baseado em créditos.

O controle *handshake* consiste em um par válido-reconhecido. Assim que o nodo emissor possui informação a ser repassada, emite um sinal de válido para o receptor, que só irá emitir o sinal de reconhecido quando estiver apto a receber a informação. Após receber o sinal de reconhecido o emissor então transmite a informação.

No controle baseado em créditos, o receptor mantêm um sinal avisando ao emissor se possui créditos. Os créditos são representados por posições livres no buffer do receptor. Assim o emissor só envia informação se o receptor estiver apto a recebê-la.

O controle de fluxo com canais virtuais é uma técnica mais avançada e custosa, que apresenta um melhor aproveitamento do canal físico. Consiste em separar logicamente o canal físico e multiplexar os canais virtuais resultantes. Deste modo, em caso de bloqueio de um dos canais virtuais, pode-se utilizar outro.

2.1.6 Roteamento

Para um nodo fonte (F) se comunicar com um nodo destino (D), é necessário que o pacote entre na rede pelo roteador relacionado ao nodo fonte e seja propagado até o roteador relacionado ao nodo destino, onde deve então ser entregue (Figura 2.3). O mecanismo que decide qual será a rota tomada uma vez que o pacote entrou na rede é chamado de roteamento.

De acordo com (MOHAPATRA, 1998) podemos classificar o roteamento em duas categorias: roteamento de fonte ou roteamento distribuído. Quando o roteamento é de fonte, a rota é decidida na origem do pacote, antes de seu envio. Isto implica que o cabeçalho do pacote deve conter toda informação de roteamento. Quando o roteamento é distribuído, a decisão de rota é realizada localmente por cada roteador, decidindo assim qual será o próximo roteador a receber o pacote. Ainda existe uma terceira opção, raramente utilizada, que é o roteamento central. Neste caso todas as rotas são decididas por um elemento único.

Também é possível classificar o roteamento quanto a sua adaptabilidade. Um roteamento que sempre provê a mesma rota entre um par de nodos chama-se determinístico. Caso a rota não seja fixa, determina-se o algoritmo como adaptativo, subdividindo-se a categoria em parcialmente ou totalmente adaptativo, dependendo se o pacote pode utilizar qualquer caminho na rede (totalmente) ou apenas um subconjunto (parcialmente).

Algoritmos adaptativos levam em conta o tráfego da rede para decidir sua rota (DUATO; YALAMANCHILI; NI, 1997)

2.1.7 Arbitragem

Arbitragem define a política para a alocação dos recursos do roteador. Em outras palavras, quando e qual porta de entrada do roteador pode utilizar uma porta de saída.

A arbitragem pode ser implementada de maneira centralizada ou distribuída. Na versão centralizada, a decisão é tomada baseada no estado global do roteador. Na versão distribuída cada canal de saída possui um árbitro próprio, normalmente mais simples e rápidos do que uma versão centralizada. Existem diversos mecanismos diferentes para realizar a arbitração, incluíndo prioridades estáticas, dinâmicas, FCFS (*First Come, First Served*), randômicas e *Round Robin*.

2.1.8 Chaveamento

O chaveamento controla o método pelo qual a informação é transferida da porta de entrada do roteador para um canal de saída. As duas categorias mais utilizadas são chaveamento por circuito e chaveamento por pacotes (HWANG, 1992).

Quando o chaveamento por circuito é utilizado, inicialmente é estabelecida uma conexão entre o nodo fonte e o nodo destino e, somente após a conexão estar estabelecida, os dados são transmitidos. Isso implica que todos os recurso entre eles devem estar disponíveis e que a única memória necessária nos roteadores é para guardar o cabeçalho que define a conexão.

Se o chaveamento utilizado for por pacotes, os dados são divididos em unidades menores, chamadas de pacotes. Cada pacote é então enviado e roteado individualmente da fonte até o destino. A política de repasse destes pacotes define como estes pacotes são tratados pelo roteador. Existem três políticas usualmente utilizadas (MOHAPATRA, 1998). A primeira, *store-and-forward*, na qual um roteador sempre recebe todo o pacote antes de repassá-lo adiante. Na segunda, *virtual-cut-through*, pacotes são enviados para o próximo roteador na rota assim que esteja garantido que há capacidade para recebê-lo completamente. Ambas soluções necessitam de um buffer com, no mínimo, o tamanho de um pacote. A terceira variação é uma variação de *virtual-cut-through* visando reduzir o tamanho dos *buffers*, chamada *wormhole*. Nesta variação, os pacotes são dividídos em unidades menores, chamadas *flits* e somente o primeiro *flit* possui a informação de roteamento, todos os outros *flits* seguem o mesmo caminho do *flit* de cabeçalho. O canal utilizado só é liberado após a passagem do último *flit*, o qual contem a cauda do pacote.

2.1.9 Memorização

Nas redes que utilizam chaveamento baseado em pacotes, é necessário um mecanismo para armazenar os *flits* até que eles estejam aptos a serem consumidos pelo próximo roteador. Este mecanismo denomina-se memorização. A memorização pode ser classificada em centralizada, quando *flits* de todos canais são armazenados na mesma estrutura, ou distribuída, quando cada canal possui uma memória privada. A memorização não se limita aos canais de entrada, podendo ser utilizada nos canais de saída também. Normalmente são utilizadas FIFOs para implementar o armazenamento devido ao seu custo reduzido.

A capacidade de armazenamento interfere criticamente no desempenho do roteador e, idealmente, a memorização deve ter capacidade suficiente para que os pacotes nunca fiquem bloqueados.

2.1.10 *Deadlock*, *Livelock* e *Starvation*

Deadlock caracteriza-se por uma depedência cíclica quando um elemento espera a liberação de um recurso, que esta em uso por outro elemento que, por sua vez, esta aguardando a liberação de um recurso que esta sendo usado pelo primeiro elemento. Para existir *deadlock* em um sistema é necessário que todas as Condições de Coffman estejam presente simultaneamente no sistema (COFFMAN; ELPHICK; SHOSHANI, 1971). As quatro condições são:

- Exclusão Mútua: no máximo um elemento pode utilizar o recurso em um dado momento.
- Alocação de recurso enquanto aguarda pela alocação de recursos adicionais.
- Ausência de Preempção: uma vez que um recurso está alocado para um elemento ele só sera liberado quando o elemento liberá-lo voluntariamente.
- Depedência Cíclica.

Livelock caracteriza-se por a ação de um elemento causar uma outra ação que volta a causar a ação inicial do elemento. Desto modo, o estado do sistema muda constantemente sem fazer nenhum progresso. No contexto de NoCs, pode ser representado pelo caso em que um pacote trafega na rede sem nunca alcançar seu destino.

Starvation é caracterizado pela privação indefinida de acesso de um elemento a um recurso. Esta situação pode ser evitada através da arbitragem de acesso ao recurso compartilhado, de modo que, eventualmente, todos elementos terão o recurso alocado para si.

2.1.11 Vazão e Latência

A vazão e a latência são métricas utilizadas para avaliar a performance e caracterizar a rede. Idealmente, a latência deve ser minimizada e a vazão maximizada.

A vazão de uma rede pode ser definida como o tráfego máximo aceito pela rede (DUATO; YALAMANCHILI; NI, 1997). Este valor precisa ser normalizado para levar em conta tamanho da rede e dos pacotes. É utilizada para caracterizar a quantidade de informação que pode trafegar na rede em um dado momento.

Latência é o tempo necessário para uma mensagem se propagar do ponto fonte até seu destino. É utilizada para caracterizar a velocidade com que a informação trafega na rede.

2.2 Estado da Arte

2.2.1 *Multicast*

Atualmente, a maior parte dos trabalhos em NoCs focam em otimizar o tráfego de *unicasts*, provendo comunicação eficiente, com alta vazão e baixa latência. Porém, sem nenhum suporte a *multicast*, a ocorrência de tráfego de um para muitos pode degradar severamente a performance do sistema.

Em (JERGER; PEH; LIPASTI, 2008) são apresentados diversos protocolos de coerência que fazem uso de *multicast* a fim de justificar o suporte a este tipo de tráfego. Após, é introduzido um mecanismo baseado em um árvore de circuitos virtuais. Este método utiliza uma fase de preparação, na qual são enviados pacotes de *unicast* para montar uma árvore virtual, e após a árvore ter sido construída as cargas são enviadas.

Em (SAMMAN; HOLLSTEIN; GLESNER, 2008) é apresentada uma modificação em um roteador existente, utilizando uma combinação de lógica e LUTs alocadas a cada porta para oferecer suporte a roteamento paralelo, fazendo um controle de fluxo baseado em identificadores presentes em cada *flit*.

2.2.2 *Cluster*

Não foram encontradas referências explícitas ao suporte de *clusters* no nível de rede. Porém, foi encontrada uma área relacionada chamada *Cluster on Chip*. Em artigos dessa área, como em (LENG et al., 2005), foram encontradas breves referências a utilização de NoCs, porém nada mencionado sobre a adição de suporte na rede, somente de sua utilização como barramento de interconexão.

3 SOCIN

Este capítulo descreve a SoCIN (ZEFERINO; SUSIN, 2003), e relaciona seus componentes com os conceitos relacionados no Capítulo 2.

A SoCIN é uma rede em chip escalável, construída a partir de um roteador parametrizável chamado RASoC. Nesta rede o tamanho de *flit* equivale ao tamanho de *phit*. Ela poder ser parametrizável conforme as necessidades da aplicação de acordo com três critérios: grau de conectividade dos nodos, largura de canal (n) e profundidade de buffers (p). Seu grau de conectividade pode variar, com um limite superior de cinco. Tanto a largura do canal quanto a profundidade dos buffers não possui nenhuma limitação. Além disso, ela define que, para o cabeçalho, *n* pode ser divido duas partes: *High Level Protocol* (HLP) e *Routing Information Bits* (RIB). RIB são bits de informação de roteamento, utilizados por um dos módulos para decidir qual a porta destino de um *flit* de cabeçalho em uma porta de entrada. HLP é uma parte do cabeçalho dedicada a protocolos de níveis superiores, porém nenhum uso específico é definido. Por este motivo este trabalho considera *n* como sendo dedicado somente para RIB. Isso permite a construção de redes com topologias do tipo grelha e toroidal, com tamanho máximo dependente do tamanho da largura do canal, de acordo com a Equação 3.1. A Figura 3.1 mostra uma SoCIN com topologia tipo grelha e largura do canal *n*.

$$t_{max} = X * Y, onde : X = Y = 2^{(n-2)/2} \tag{3.1}$$

3.1 RASoC

O RASoC foi projetado para ser utilizado em NoCs para SoCs de sistemas embarcados, sem consumir muita área. Ele é construído de um modo distribuído, baseado em chaveamento do tipo *wormhole*. O algoritmo de roteamento utilizado é XY de fonte, que é determinístico e livre de *deadlocks*. Para controle de fluxo é utilizado um protocolo de *handshake*. Ele é formado por instâncias do módulo canal de entrada (Figura 3.2) e do módulo canal de saída (Figura 3.3), interligados por uma matriz de *cross* conexão. Originalmente, a diagonal principal da matriz não é implementada, porém neste trabalho optou-se por implementá-la. Nas subseções seguintes são descritos os módulos que compõem os módulos dos canais de entrada e saída.

3.1.1 IFC

O módulo *Input Flow Controller* implementa a lógica de controle de fluxo, realizando uma tradução dos sinais do protocolo de *handshake* utilizados pela porta do roteador para os sinais de controle da FIFO do *buffer* de entrada.

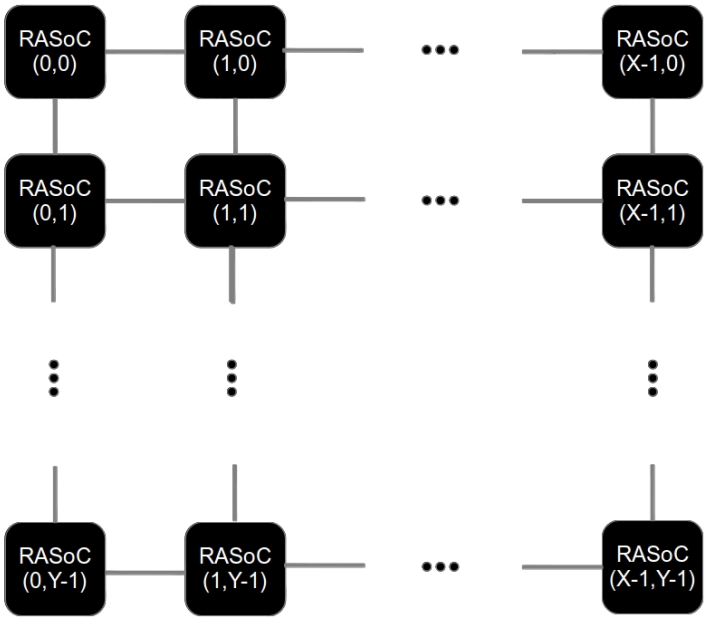


Figura 3.1: Uma instância da SoCIN de tamanho máximo, X e Y são definidos na Equação 3.1.

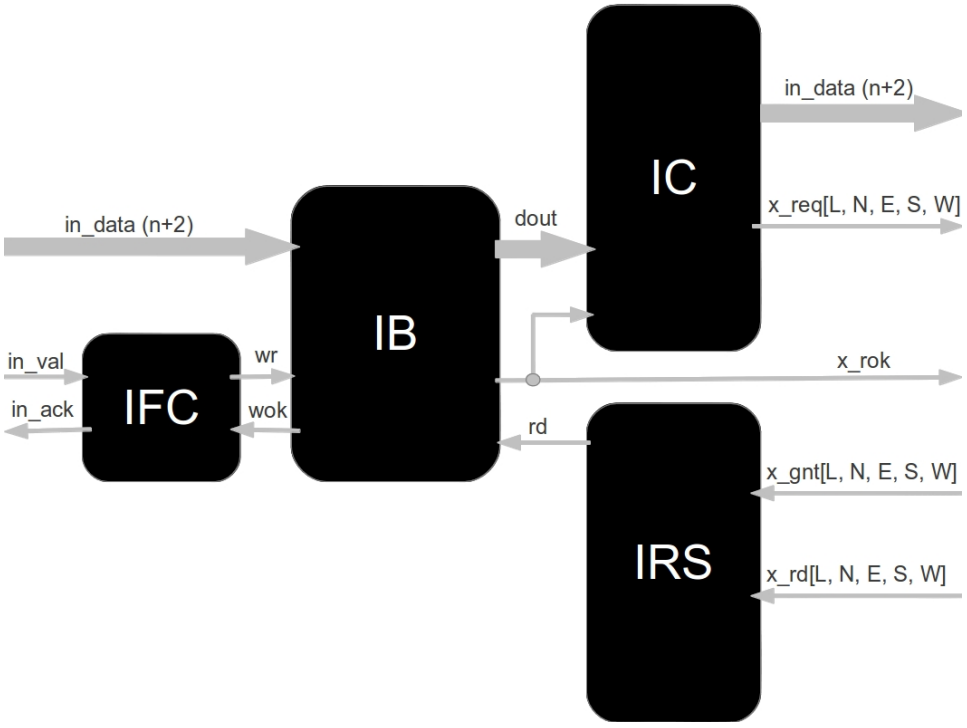


Figura 3.2: Módulo do canal de entrada.

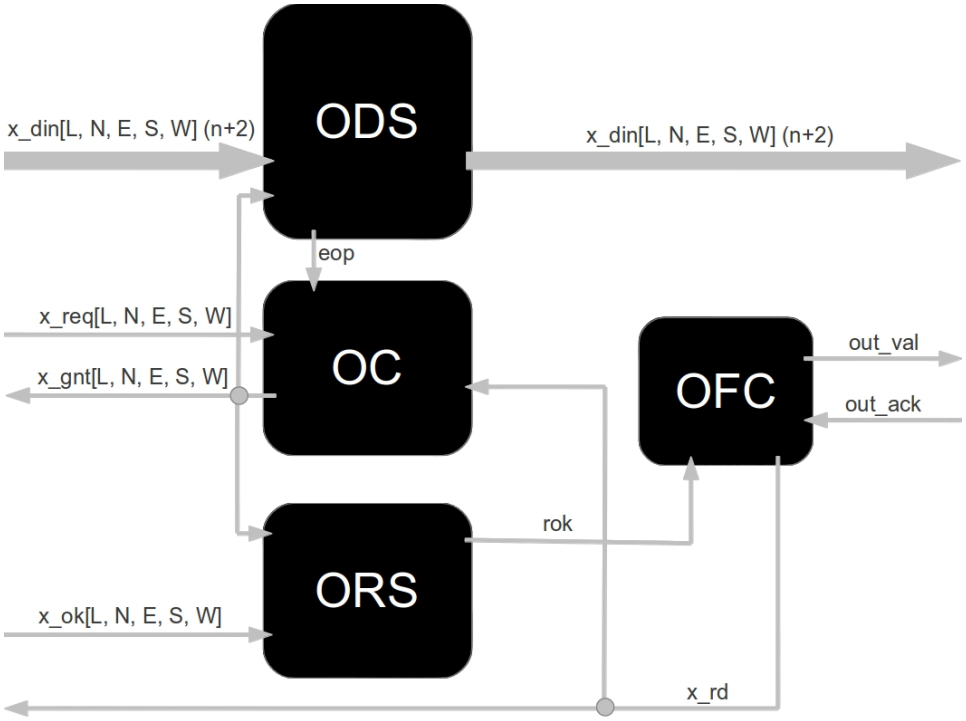


Figura 3.3: Módulo do canal de saída.

3.1.2 IB

O módulo *Input Buffer* é um *buffer* do tipo FIFO, e sua responsabilidade é armazenar os flits até que seja possível passá-los adiante na rede. Sua parametrização se dá pelo parâmetro *p* que define qual quantidade máxima de *flits* que podem ser armazenados neste módulo.

3.1.3 IC

O módulo *Input Controller* é responsável pelo roteamento dos pacotes. Ao detectar um *flit* de cabeçalho ele analisa os RIB, e, baseado no seu algoritmo de roteamento, emite um sinal de requisição para a porta de saída para onde o pacote deve ser roteado. As informações dos RIB (Figura 3.4) são interpretadas de acordo com a Tabela 3.1.

Após fazer a requisição, o módulo IC atualiza os campos Xmod ou Ymod, de acordo com a porta de saída escolhida. O algoritmo de roteamento XY consiste em sempre percorrer o eixo X primeiro, e após, percorrer o eixo Y. Seu funcionamento pode ser representado pelo fluxograma da Figura 3.5.

Este roteamento restringe a utilização total da banda disponível na NoC, porém é uma das alternativas mais baratas de se ôbter uma rede livre de *deadlocks*. O algoritmo de roteamento também limita a topologia da rede e o número máximo de canais, visto que um pacote só pode ser roteado no eixo X, que dá origem as portas Leste e Oeste, ou no eixo Y, que origina as portas Sul e Norte, ou não ser repassado a nenhum roteador implicando a existência da porta Local.

3.1.4 IRS

O módulo *Input Read Switch* é responsável por repassar ao bloco IB o sinal de que o *flit* atual foi lido pela porta de saída conectada. Para isso utiliza-se dos sinais de concessão

Campo	Significado
Xdir	Se 0 o pacote deve ser roteado na direção Leste Se 1 o pacote deve ser roteado na direção Oeste
Xmod	Número de *hops* restantes na direção X
Ydir	Se 0 o pacote deve ser roteado na direção Norte Se 1 o pacote deve ser roteado na direção Sul
Ymod	Número de *hops* restantes na direção Y

Tabela 3.1: Interpretação dos campos do *flit* de RIB.

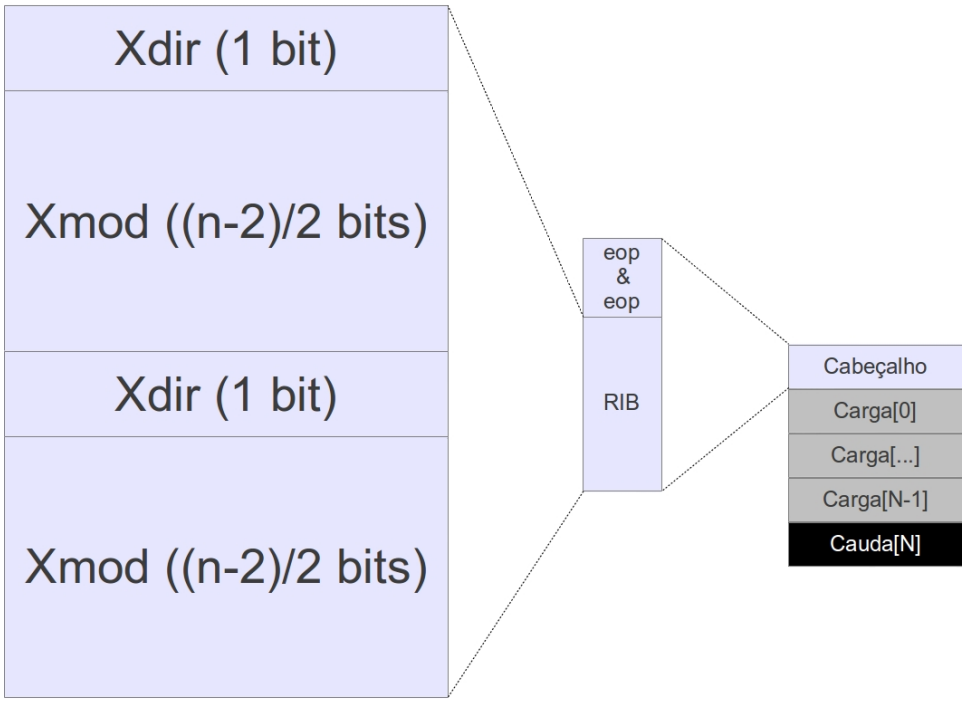


Figura 3.4: Flit de cabeçalho, contendo as informações de roteamento.

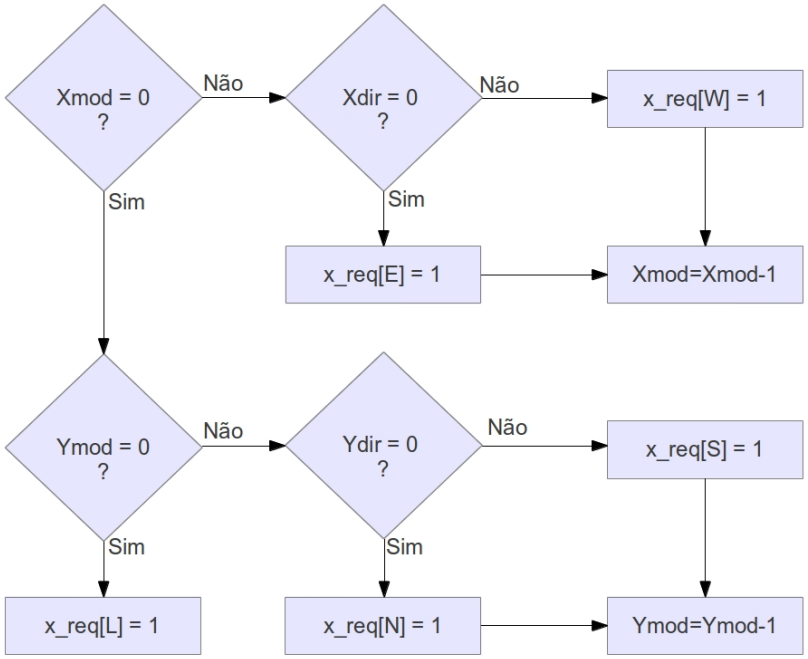


Figura 3.5: Fluxograma do roteamento XY de acordo com os RIB da SoCIN.

para determinar a qual porta de saída esta conectado.

3.1.5 OC

O módulo *Output Controller* utilizar um algoritmo *Round Robin* para decidir a qual porta de saída conceder as portas de entrada, garantindo que não ocorra *starvation*. Ele responde a ativações nos sinais de requisição vindo de cada porta de entrada, e ao determinar qual porta deve receber a concessão ativa o sinal de concessão correspondente àquela porta de entrada.

3.1.6 ORS

O módulo *Output Read Switch* é responsável por repassar ao bloco OFC o sinal de que existe um *flit* disponível no IB do canal de entrada que recebeu a concessão do OC. Utiliza-se dos sinais de concessão para determinar qual porta de entrada está conectada à sua porta de saída.

3.1.7 ODS

O módulo *Output Data Switch* é responsável por atualizar o *flit* atual que deve ser escrito na porta de saída com o *flit* vindo do módulo IC da porta de entrada conectada. Para isso utiliza-se dos sinais de concessão para determinar qual porta de entrada está conectada à sua porta de saída.

3.1.8 OFC

O módulo *Output Flow Control* é responsável pelo mecanismo de controle de fluxo na porta de saída. Por sua operação ser redundante com o sinal vindo do ORS pode ser implementado como dois fios.

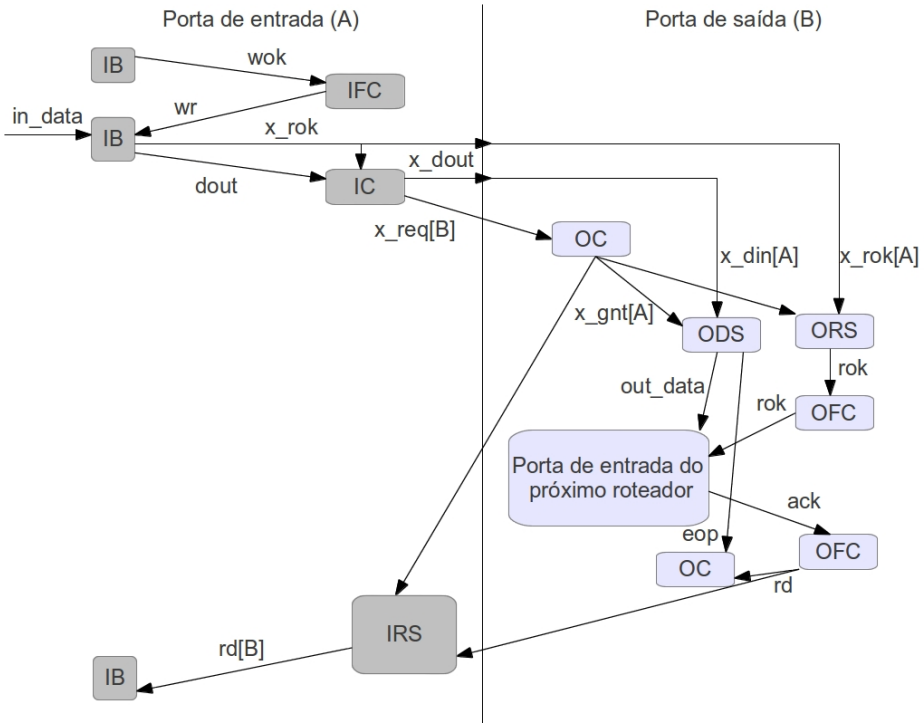


Figura 3.6: Interação entre os módulos do RASoC.

3.1.9 Fluxo interno

O diagrama da Figura 3.6 mostra como os blocos interagem entre si quando o *flit* é um cabeçalho e uma conexão é estabelecida entre uma porta de entrada (A) e uma porta de saída (B). O primeiro *flit* do pacote a chegar é o do cabeçalho, sendo primeiramente armazenado em IB. Quando ele alcança a frente do *buffer* o bloco IC detecta que se trata de um cabeçalho através do bit *bop*, e então executa o algoritmo de roteamento e modifica o *flit* de acordo. IC então emite uma requisição para o OC da porta desejada. O OC analisa todas as requisições presentes em sua entrada e, de acordo com um algoritmo *Round Robin*, emite uma concessão para a porta de entrada escolhida. As chaves ODS, ORS e IRS ao perceberem a presença de uma concessão conectam a entrada correspondente com sua saída. O bloco OFC então possui o *flit* e o sinal de pronto para ler referente à porta desejada e realiza o *handshake*. Quando o *handshake* é realizado com sucesso, OFC emite então um sinal de leitura realizada. Neste momento o bloco IRS já está com sua chave setada corretamente e o sinal se propaga até o bloco IB, que então remove o *flit* que encontra-se na ponta do *buffer* e um processo similar começa para os *flits* seguintes, com a excessão de que o bloco IC não roda o algoritmo de roteamento nem modifica o *flit*. Assim um circuito é fechado e os *flits* fluem em direção a porta desejada. A conexão é desfeita quando o bloco OC detecta, através do bit *eop*, que o *flit* escrito era cauda, fechando a conexão ao retirar o sinal de concessão. O diagrama da Figura 3.6 tenta dar uma visualização do fluxo de um *flit* pelo roteador e a interação de seus módulos quando todos os buffers estão vazios e existe um *flit* disponível numa porta de entrada.

3.2 *Link*

Os *links* que interligam os roteadores consistem de dois canais unidirecionais opostos, cada um composto por sinais de dados, dois bits para controle de quadro (*bop*, *eop*) e dois

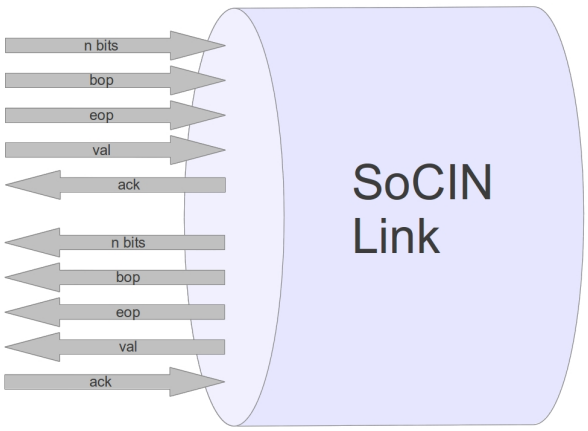


Figura 3.7: *Link* da SoCIN.

bits de controle de fluxo (*val*, *ack*) (Figura 3.7).

4 NOVOS RECURSOS E IMPLEMENTAÇÃO

Este capítulo é divido em quatro seções. A Seção 4.1 versa sobre a adição do suporte a *multicast* enquanto a Seção 4.2 discursa sobre a adição do suporte a *cluster*. A Seção 4.3 traz informações relativas à implementação e, por fim, a Seção 4.4 discute a validação da implementação.

4.1 Multicast

4.1.1 Motivação

Este trabalho propõe a modificação da SoCIN, visando diminuir o número de pacotes redundantes que trafegam nela quando a mesma mensagem deve ser transmitida de um nodo para vários.

Considere uma instância de SoCIN com tamanho de *flit* igual a *n* bits, contendo X*Y roteadores. A Equação 4.1 define a quantidade de informação (em *flits*) gerada no nodo origem para transmitir uma mensagem de S bytes entre dois nodos.

$$N_{flits/unicast} = N_{flits/cabecalho} + N_{flits/carga} = 1 + 8 * \lceil S/(n-2) \rceil \ (flits) \tag{4.1}$$

Considerando que todos os *flits* se propaguem pela NoC em condições ideais e sem intereferências numa taxa de um *flit* por ciclo, a quantidade de ciclos necessária para toda mensagem se propagar por um *hop* é dada pela Equação 4.2

$$N_{ciclos/hop} = N_{flits/unicast}/1 = N_{flits/unicast} \ (ciclos) \tag{4.2}$$

O número de *hops* necessários para alcançar um nodo pode ser calculado em função do endereço dos nodos atráves da Equação 4.3.

$$D(Xf, Yf, Xd, Yd) = |||Xf| - |Xd|| + ||Yf| - |Yd||| \tag{4.3}$$

Onde:

$Xf =$ é a posição no eixo X do nodo fonte

$Yf =$ é a posição no eixo Y do nodo fonte

$Xd =$ é a posição no eixo X do nodo destino

$Yd =$ é a posição no eixo Y do nodo destino

Logo, a quantidade de ciclos necessários para transmitir uma mensagem de um nodo (Xf,Yf) para outro nodo (Xd, Yd) é dada pela Equação 4.4.

$$N_{ciclos/unicast} = N_{ciclos/hop} * D(Xf, Yf, Xd, Yd) \tag{4.4}$$

Agora considere um *broadcast* da mesma mensagem baseado em *unicast*. A quantidade de ciclos necessários para alcançar todos os nodos é dado pela Equação 4.5.

$$N_{ciclos/broadcast_{unicast}} = N_{ciclos/hop} * \sum_{x=0}^{X-1} \sum_{y=0}^{Y-1} D(Xf, Yf, x, y) \quad (4.5)$$

É fácil de observar que alguns roteadores recebem a mensagem várias vezes, porém só a repassam para sua porta Local quando o *flit* de cabeçalho estiver com os campos Xmod e Ymod adequados (iguais a 0).

Com um suporte a *multicast* é possível reduzir o número de mensagens para uma única mensagem. Porém a mensagem deve ser maior pois deve conter um *flit* de cabeçalho para cada alvo. No caso de um *broadcast* temos o tamanho da mensagem de acordo com a Equação 4.6.

$$N_{flits/multicast} = N_{flits/cabecalhos} + N_{flits/carga} = X * Y + 8 * \lceil S/n \rceil \ (flits) \quad (4.6)$$

Considerando novamente que os *flits* fluem a uma taxa de um *flit* por segundo, o número de ciclos pode ser calculado para o pior caso. Este acontece quando o nodo origem encontra-se o mais distante possível de um dos alvos, e a Equação 4.7 representa o número de ciclos para alcançá-lo.

$$N_{ciclos/broadcast} = N_{flits/multicast} * D(Xf, Yf, Xd, Yd) \quad (4.7)$$

4.1.2 Metodologia

O suporte a *multicast* consiste, basicamente, em habilitar *multicast* dentro do RASoC. Uma vez que um roteador consiga transmitir um pacote a múltiplas portas o processo é replicado nos outros roteadores, tornando a NoC inteira passível de *multicast*. Verificando-se a conexão interna do RASoC, a qual é implementada através de um matriz de *cross* conexão (Figura 4.1) percebe-se que todos os caminhos de dados necessários para efetuar um *multicast* já estão presentes, só é necessário então inserir controles que permitam a conexão e sincronização de múltiplas portas de saída a uma única porta de entrada. A sincronização é importante para que um *flit* propague-se ao mesmo tempo para todas direções necessárias, evitando perda de *flits*.

Uma análise da SoCIN revela que a cada ciclo o bloco IC de uma porta de entrada faz, no máximo, uma requisição a uma porta de saída. Além disso, os árbitros implementados nos módulos OC também concedem a conexão a somente uma porta de entrada por ciclo. Isso gera a primeira demanda para a implementação do *multicast*, pois pode causar *deadlock*, visto que duas entidades distintas podem estar competindo por mais de um recurso ao mesmo tempo, e a concessão dos dois recursos não ocorre atomicamente. Isto implica que antes de começar a solicitar um *multicast* é necessário fechar um recurso para garantir a não ocorrência de *deadlocks*. Deste modo, antes de iniciar requisições aos blocos OC para fazer *multicast* o bloco IC deve garantir este recurso. Com isto em mente, é necessário analisar o comportamento esperado de cada tipo de *flit*. A SoCIN original implementa três tipos de *flits*, conforme a Tabela 4.1.

O comportamento projetado é que o cabeçalho define uma conexão, e a carga segue por essa conexão até que seja repassada a carga que contém a cauda, caso no qual a conexão é fechada. O que foi percebido é que é uma porta de entrada pode ser conectada a mais de uma porta de saída, desde que as portas de saída realizem a leitura de um modo síncrono. Com isto garantido, a mensagem será efetivamente replicada nas portas de

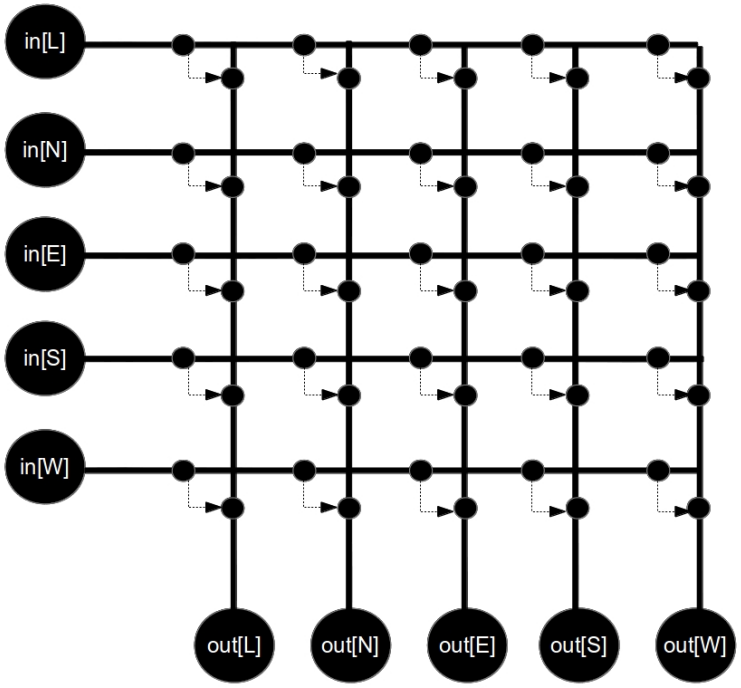


Figura 4.1: Matriz de *cross* conexão.

bop	*eop*	Tipo do *flit*
0	0	Carga
0	1	Cauda
1	0	Cabeçalho
1	1	Não definido

Tabela 4.1: Tipos de *flit* na SoCIN.

bop	*eop*	Tipo do *flit*
0	0	Carga
0	1	Cauda
1	0	Cabeçalho
1	1	Cabeçalho de *Multicast*

Tabela 4.2: Tipos de *flit* na SoCIN com suporte a *multicast*.

saída. Isso pode ser facilmente obtido enviando mais de um *flit* de cabeçalho. Contudo, este mecanismo afeta também os cabeçalhos que seguem o primeiro, sendo estes também replicados na porta de saída. Este efeito não é desejado pois provoca um comportamento inesperado e possivelmente leva a situações de *deadlock* e *livelock*. Para contornar este problema, criou-se uma nova classe de *flit* para cabeçalhos de *multicast* (Tabela 4.2), aproveitando os bits de *bop* e *eop*. Para esta ampliação foi necessário alterar as lógicas dos blocos para verificarem ambos os bits, ao invés de apenas um.

Com esta nova classe criada, foi definido que o sinal de requisição vindo do bloco IC só permanece setado quando um *flit* de um dos tipos de cabeçalho estiver presente. O bloco OC então detecta se o canal de entrada que tem a sua concessão está requisitando conexão novamente, e repassa isso para o bloco OFC. Quando o bloco OFC detecta um cabeçalho de multicast ele verifica a informação repassada pelo OC e só escreve o cabeçalho na porta de saída caso ambos, concessão e requerimento, estejam ativados.

Deste modo, os tipos de *flits* da SoCIN não têm seu comportamento alterado. Os *flits* de carga que seguem cabeçalhos de *multicast* serão replicados nas portas que estiverem conectadas na entrada somente quando todas estiverem aptas a receber o *flit*. Este controle é feito, começando pelo OFC enviando um sinal ao ME avisando se está apto a transmitir o *flit*. Este sinal só não é enviado caso não esteja apto e esteja fazendo parte do multicast, assim seu comportamento quando não está participando do multicast é transparente, como se sempre estivesse disponível. Já os cabeçalhos de *multicast* só serão repassados para as portas da rota que normalmente tomariam. A conexão é fechada normalmente quando a cauda é repassada adiante. Note que, no caso de *multicast*, independentemente de qual sinal de leitura feita o bloco IRS recebe, este sinal será redundante em relação aos outros já que a escrita ocorre em todos ao mesmo tempo. Após isso, é obrigação do bloco IC liberar o recurso de *multicast*.

4.1.3 Alterações nos módulos do RASoC e módulo adicional

Foi adicionado um bloco denominado de *Multicast Enabler* (Figura 4.2) por roteador, que realiza a seleção de qual requisição recebe a concessão, utilizando um algoritmo *Round Robin* para evitar *starvation*. Ele mantém a concessão até que o sinal de requisição não esteja mais presente. Além disso, visto que é o único elemento central nos roteadores, ele é responsável por unificar os sinais de disponibilidade das portas de saída em um único sinal, que é distribuído a todos OFCs.

No módulo do canal de entrada foi necessário alterar somente o bloco IC (Figura 4.3), adicionando uma lógica para requisitar o recurso ME antes de fazer requisições relativas a *multicast*. Para cabeçalhos normais as requisições são feitas normalmente. Ao detectar que um *flit* de cauda foi escrito, o bloco IC libera o recurso de *multicast* baixando o sinal de requisição para o ME.

No módulo canal de saída foram alterados os blocos OC, ODS e OFC (Figura 4.4). O

bloco OC agora repassa os sinais que representam a concessão feita a um canal de entrada e uma requisição do canal ao qual foi concedido a conexão. O bloco ODS também detecta se o *flit* presente possui ambos bits, *bop* e *eop*, setados. Caso estes bits estejam setados, ele seta o sinal de que o *flit* é um cabeçalho de *multicast* para o OFC.

O bloco OFC foi alterado para, quando o sinal que indica um *flit* de cabeçalho de *multicast* estiver setado, só repassar se os sinais de requisição e concessão vindos do OC estiverem habilitados. Os *flits* de carga que seguem um cabeçalho de *multicast* só serão repassados no momento em que o sinal de que todas saídas estão livres estiver setado. Cargas que seguem um cabeçalho regular são tratadas normalmente. Além disso, quando o módulo detecta *multicast* passa a ativar e desativar o sinal de livre conforme seu estado enquanto em outras condições sempre sinaliza como livre para não interferir com portas que estejam participando de um *multicast*.

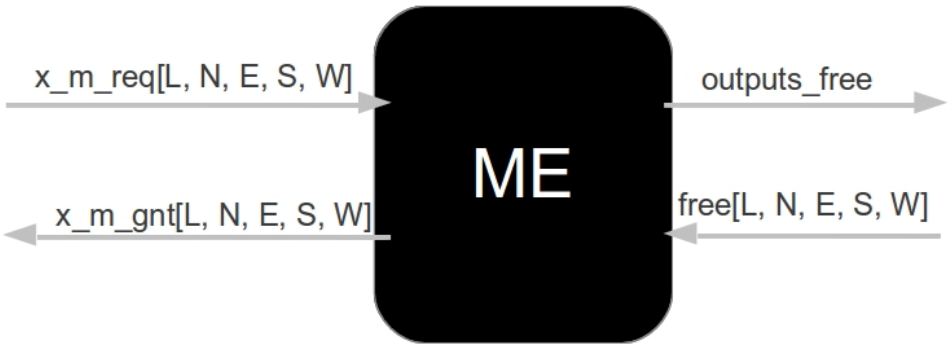


Figura 4.2: Módulo adicionado para dar suporte a *multicast*.

4.1.4 Limitações

A solução possui uma limitação de que o *multicast* seja único na rede. Caso não seja, poderão ocorrer *deadlocks*. Existem diversas soluções possíveis para garantir a unicidade do *multicast*, entre elas, criar um *mutex* central utilizado por todos roteadores ou utilizar um mecanismo de *tokens*.

4.2 *Clusters*

4.2.1 Motivação

O suporte a *clusters* visa possibilitar aos clientes da rede que possam isolar-se do resto do tráfego da NoC. Este isolamento permite que uma aplicação consiga rodar sem interferência de outras aplicações, facilitando sua análise, depuração e a não violação de requisitos temporais.

4.2.2 Metodologia

Para dar suporte a *clusters*, foi realizada uma modificação rudimentar, adicionando um registrador que guarda a identificação de *cluster* do roteador e um bit que representa o

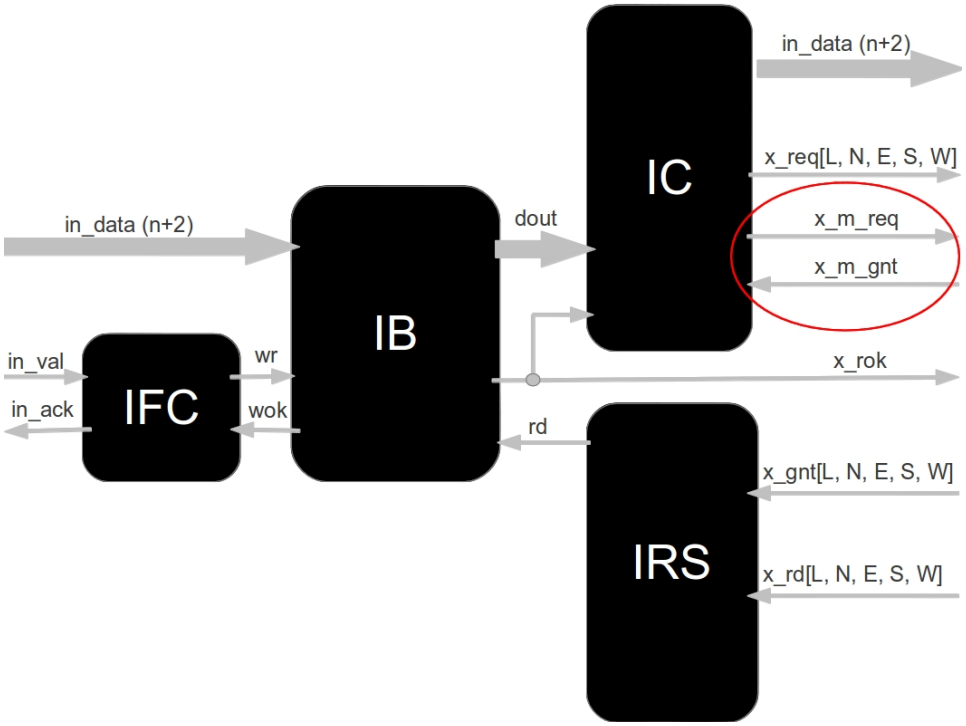


Figura 4.3: Módulo do canal de entrada modificado para dar suporte a *multicast*.

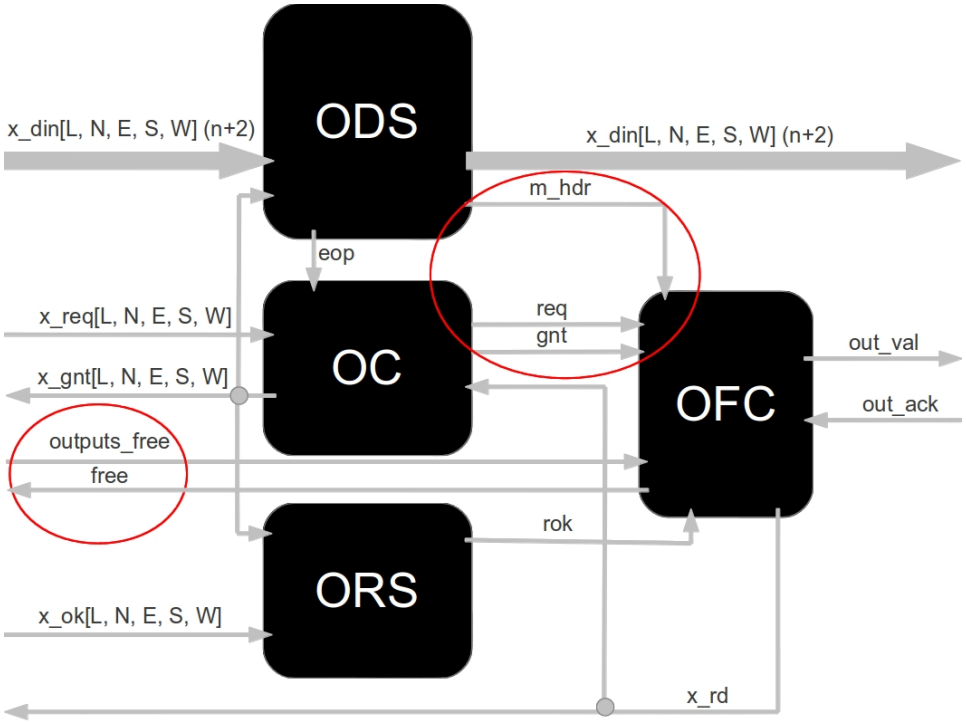


Figura 4.4: Módulo do canal de saída modificado para dar suporte a *multicast*.

modo de isolamento de tráfego (ligado/desligado). Estes registradores são disponibilizados para o exterior, deixando a responsabilidade de seu controle fora da NoC e garantindo que este mecanismo seja dinâmico.

4.2.3 Alterações

O valor do registrador de identificação sem o bit de controle é repassado aos roteadores vizinhos, através de portas adicionais criadas em cada roteador. O bloco IB local recebe todo o registrador de identificação. A lógica de controle de isolamento de tráfego é realizada no bloco IB. Ao receber um *flit* o IB verifica se o bit de isolamento de tráfego esta ativo, caso esteja, o valor do registrador de identificação do roteador local é comparado à identificação do roteador conectado a porta, e caso sejam diferentes o flit é simplesmente descartado ao invés de ser colocado no *buffer*. Para o roteador vizinho é como se o *flit* tivesse sido transferido normalmente. Isto garante a transparência do mecanismo visto de fora do roteador, e garante que tráfego só entrará por este roteador caso venha de um vizinho com o mesmo valor de identificação. A porta local tem seu identificador que viria do vizinho conectado ao próprio roteador local, garantindo que o tráfego da porta local nunca será descartado.

Este mecanismo permite que exista um multicast por *cluster*, pois separa o domínio da rede em subdomínios isolados.

4.2.4 Limitações

Deve ser tomado um cuidado em relação ao posicionamento e formato dos *clusters* para não isolar da rede alguns nodos. Isto se deve ao roteamento utilizado ser XY. Quanto ao posicionamento, caso a rota X ou Y seja bloqueada por um *cluster* o nodo pode ficar indisponível para outros nodos pois os *flits* serão descartados ao tentar atravessar o *cluster* (Figura 4.5). Em relação ao formato do *cluster* ele deve ser retangular, caso contrário alguns nodos também podem ficar inalcançáveis pois a rota pode envolver um roteador não pertencente ao *cluster*. Assim quando o *flit* tenta reentrar no *cluster* será descartado (Figura 4.6).

4.3 Implementação

A implementação foi feita em C++, utilizando a biblioteca SystemC (INITIATIVE, 2012). SystemC é uma coleção de classes e macros para C++ que provê um núcleo de simulação baseado em eventos, que pode ser utilizado para descrever *hardware* em vários níveis. Inicialmente, foi implementada do zero a SoCIN, tomando como base a descrição feita no Capítulo 3. Este modelo inicial foi utilizado como base para o resto do trabalho. A modelagem foi feita em RTL, sendo completamente parametrizável em termos de largura do canal e profundidade dos *buffers*. O número de portas é limitado pelo roteamento, mas é facilmente alterável para incluir mais portas. O tamanho da rede também é completamente parametrizável, suportando qualquer formato retangular. É importante ressaltar que, apesar de não houver qualquer limitação quanto ao tamanho, dependendo do tamanho escolhido para a NoC e para a largura do canal, não haverá conexão em nível de rede entre alguns nodos. Como exemplo, considere uma NoC de tamanho 3x3 e largura de canal de quatro bits. Assim será utilizado no máximo um bit para cada campo do RIB. Deste modo um *flit* poderá ser mandado no máximo um *hop* em cada eixo (limite imposto por *xMod* e *yMod*) e então será repassado a porta Local (Figura 4.7). Apesar disso, decidiu-se não limitar o tamanho da NoC pois pode ser utilizada uma rede virtual, na camada acima

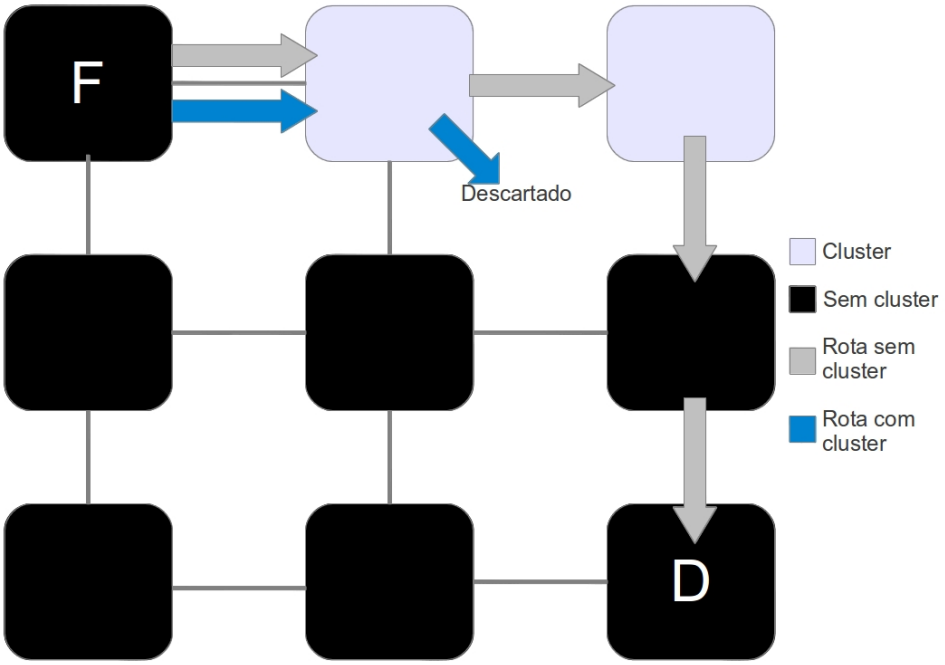


Figura 4.5: Exemplo de nodos inalcançáveis por bloqueio de rota por *cluster*.

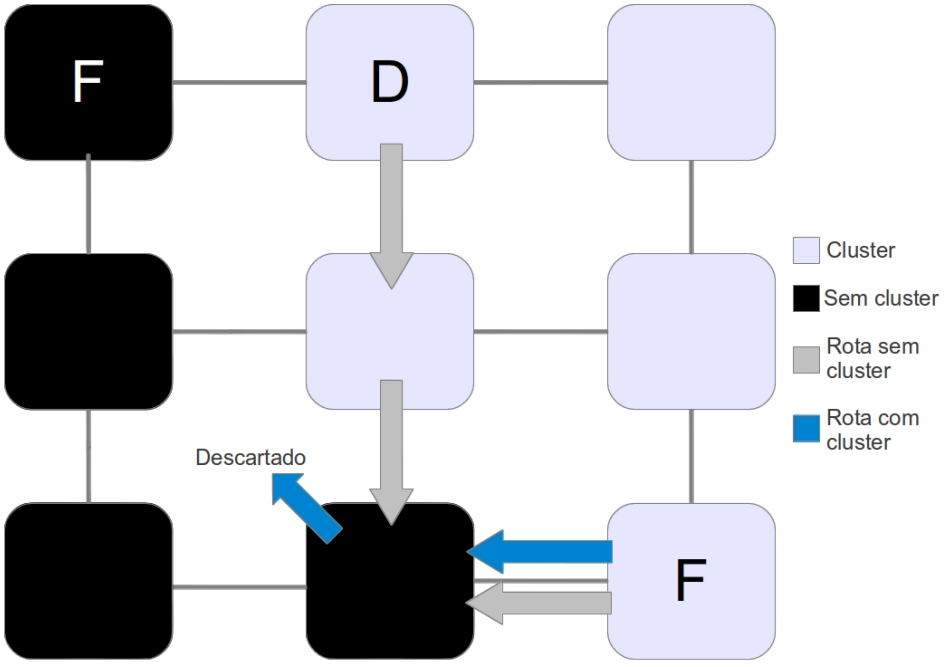


Figura 4.6: Exemplo de nodos inalcançáveis por formato de *cluster* não retangular.

da camada de rede, para conseguir interligar todos os nodos, um exemplo disso pode ser visto na Figura 4.8, na qual o *flit* é primeiro repassado ao vizinho alcançável e este repassa para o destino final.

Além disso, para fins de depuração, todos os blocos podem imprimir informações

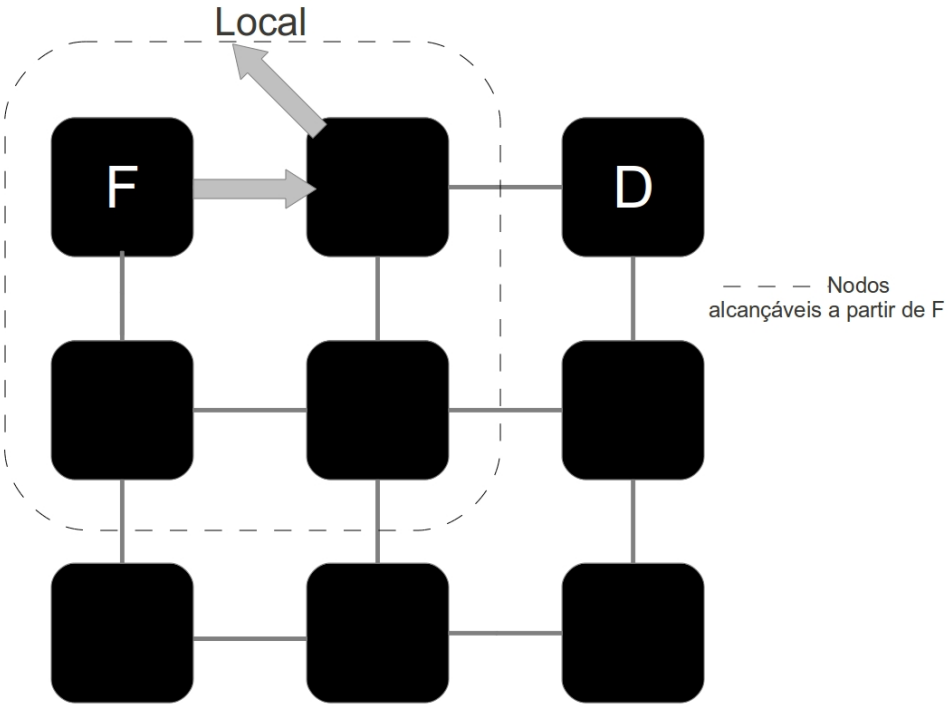


Figura 4.7: Exemplo de nodos inalcançáveis devido ao tamanho máximo do RIB.

de depuração que podem ser ligadas ou desligadas em tempo de compilação através de macros definidos em um cabeçalho único. Apesar destas informações, a grande quantidade de componentes no sistema torna a saída complexa de ser analisada. Para facilitar a verificação da implementação foram criadas algumas ferramentas utilizando *shellscript* para separar, organizar e visualizar as informações de saída. Uma das ferramentas separa um arquivo de saída em vários arquivos com informações relativas aos componentes. A ferramenta separa as informações relativas a cada roteador em arquivos nomeados a partir de seu endereço na rede com o formato *x_y*. Além disso, também separa por portas para cada roteador nos arquivos *x_y{local, north, east, west, south}* como pode ser visto na Figura 4.9. Também é possível marcar *flits* de interesse com um identificador único e após rastrear sua rota dentro da rede. A Figura 4.10 mostra o mecanismo em ação, detectando a rota percorrida por um cabeçalho e uma cauda marcados.

O código foi desenvolvido utilizando um controle de versão através da ferramenta Git (GIT, 2012). Todo o código foi documentando utilizando a ferramenta Doxygen (DOXYGEN, 2012).

4.4 Validação

Para validar a implementação foi criado um módulo gerador de tráfego, capaz de realizar *unicasts* e *multicasts* para ambas versões da NoC. Além de originar tráfego, este elemento também é capaz de receber dados. Foi instanciado um módulo para cada nodo da rede e inseridos diversos tipos de tráfego para estressar a rede. Deste modo, foram

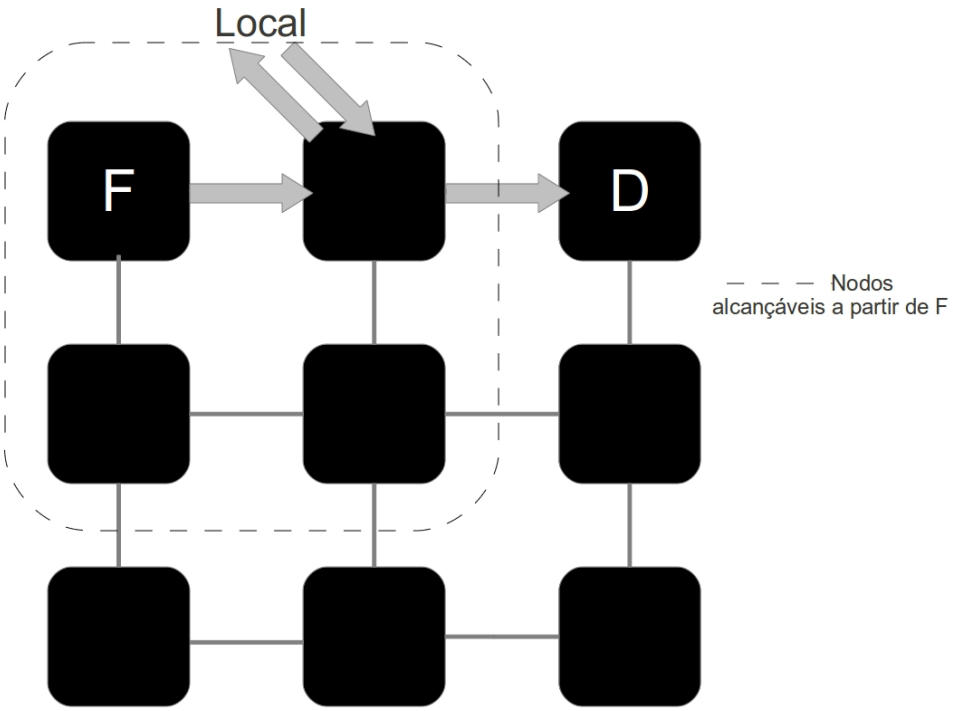


Figura 4.8: Exemplo de uma rede virtual acima da camada de rede para superar a limitação imposta pelo tamanho máximo de RIB.

```
File  Edit  View  Terminal  Help
santini@vannuys:~/bachelor_thesis/report$ ls
0_0       0_1west   1_0tf     1_2south  2_1north
0_0east   0_2       1_0west   1_2tf     2_1south
0_0local  0_2east   1_1       1_2west   2_1tf
0_0north  0_2local  1_1east   2_0       2_1west
0_0south  0_2north  1_1local  2_0east   2_2
0_0tf     0_2south  1_1north  2_0local  2_2east
0_0west   0_2tf     1_1south  2_0north  2_2local
0_1       0_2west   1_1tf     2_0south  2_2north
0_1east   1_0       1_1west   2_0tf     2_2south
0_1local  1_0east   1_2       2_0west   2_2tf
0_1north  1_0local  1_2east   2_1       2_2west
0_1south  1_0north  1_2local  2_1east   report
0_1tf     1_0south  1_2north  2_1local
santini@vannuys:~/bachelor_thesis/report$
```

Figura 4.9: Exemplo dos arquivos gerados após filtragem da informação de saída.

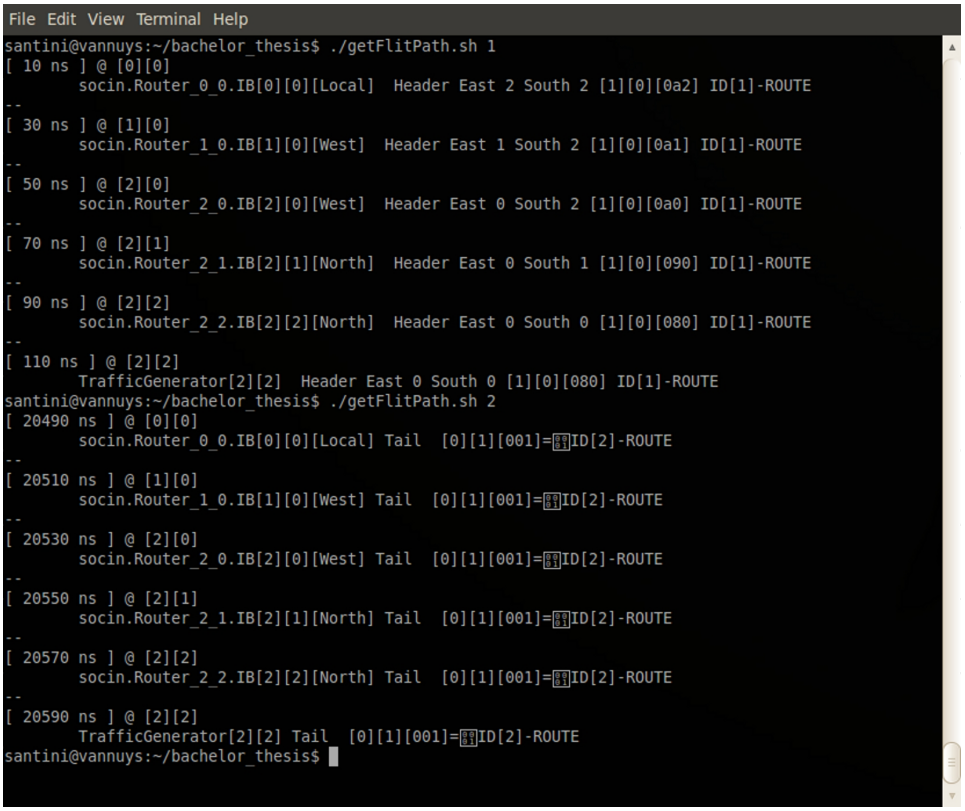


Figura 4.10: Exemplo de visualização de rota de *flits* de um pacote enviado de (0,0) para (2,2).

verificados rota dos *flits* de cabeçalho e cauda, valores dos *flits* de carga e quantidade de *flits* enviados/recebidos para cada gerador de tráfego. Entre os testes para ambas versões pode-se citar, comunicação de um para um, *multicast* sem suporte da rede, *broadcast* sem suporte da rede, comunicação de um para um com ambos nodos enviando pacotes simultaneamente e comunicações de um para um formando uma cruz na rede. Adicionalmente para a versão com suporte a *multicast* foram realizados *multicast* e *broadcast* fazendo uso do suporte.

O suporte a cluster foi testado a parte, instanciando uma NoC de dimensões 4x4 e manualmente setando os registradores de identificação dos nodos (0,0), (1,0), (0,1) e (1,1) de modo que estes formassem um *cluster*. A seguir foi verificado que não era possível enviar mensagens de fora para dentro do *cluster* quando o bit de isolamento estava setado e que mensagens trafegavam sem problemas dentro do *cluster*. Mensagens de dentro para fora do *cluster* funcionam normalmente.

5 MODELO DE ÁREA E POTÊNCIA

Os modelos de área e potência foram feitos utilizando a biblioteca Orion versão 2.0 (KAHNG et al., 2009). Devido à granularidade de modelagem da Orion foi feita uma modelagem única e utilizado este modelo para todas versões. Isto não reflete a realidade, visto que as versões com suporte a *multicast* e *cluster* diferem da original, principalmente em termos de área.

Este capítulo descreve e justifica os parâmetros que foram utilizados como entrada para a biblioteca, os resultados fornecidos pela biblioteca e como estes foram utilizados.

5.1 Parâmetros de Entrada da Orion

A Orion possui diversos parâmetros de entrada que podem ser alterados para modelar e especificar um roteador. As configurações são feitas no arquivo *SIM_port.h*, com exceção da carga. A seguir são descritos alguns parâmetros utilizados.

- **Tecnologia:** Este parâmetro define a tecnologia dos transistores do modelo. A Orion provê parâmetros precisos para tecnologias de 90nm e 65nm. Os parâmetros para 45nm e 32nm são feitos baseados em fatores de escala obtidos em (SEMATECH, 2007). Foi escolhido 65nm para o modelo, por ser o menor dentro das duas opções que representavam as melhores precisões.

- **Domínio da Aplicação:** A Orion modela três tipos distintos de domínios, roteadores de alta performance (LVT), performance normal (NVT) e baixa performance (HVT). Como a SoCIN foi inicialmente projetada para ser utilizada em sistemas embarcados, optou-se por modelar este parâmetro como baixa performance (HVT), pois também resulta em baixo consumo e área.

- **Tensão de Alimentação (Vdd):** Este valor foi retirada da documentação da Orion que sugere que um valor razoável para um modelo utilizando tecnologia de 65nm e HVT é 0.8V.

- **Frequência de Operação:** O valor utilizado para frêquencia de operação foi 100MHz, baseado em valores extraídos de (SILVA JR et al., 2008), o qual utilizava a SoCIN como barramento de interconexão. Este valor foi considerado razoável, visto que em (ZEFERINO; SUSIN, 2003) a SoCIN foi prototipada em FPGA obtendo frequências de operação da ordem de 60MHz.

- **Número de Portas:** Foram utilizadas cinco portas, tanto para saída quanto entrada, de acordo com as portas da SoCIN (Norte, Sul, Leste, Oeste e Local).

- **Modelo de Chave:** Foi escolhido um modelo baseado em matriz de *cross* conexão.
- **Modelo de Árbitro:** Foi escolhido um modelo que utiliza o algoritmo *Round Robin*.
- **Tamanho dos *Buffers* de Entrada:** Este valor é variado de acordo com a profundidade dos *buffers* de entrada escolhida pelo usuário para a simulação.
- **Largura dos Dados:** Este valor é variado de acordo com o tamanho de *flit* escolhido pelo usuário para a simulação.
- **Carga:** A Orion calcula a potência dos roteadores e *links* baseado na carga sobre eles. Este parâmetro é colhido durante a simulação e depende dos tipo de tráfego que passa na rede. Para cada roteador e para cada *link* é utilizada uma carga própria, logo a potência não é necessariamente a mesma entre roteadores e entre *links*.

5.2 Parâmetros de Saída e Utilização

A partir dos parâmetros de entrada, a Orion fornece resultados de área e potência para um *link* e para um roteador.

Após uma simulação, a carga é calculada para cada *link* e para cada roteador da simulação, baseado nos ciclos que eles foram utilizados e no número total de ciclos da simulação (Equação 5.1). Após, este valor é utilizado como entrada para a Orion para cada elemento, resultando na área e potência do elemento (Figura 5.1). A potência é então multiplicada pelo tempo total da simulação chegando assim na energia gasta pelo elemento (Equação 5.2). A energia total da rede é dada então pelo somatório da energia de todos elementos (Equação 5.3). Os resultados de área são obtidos diretamente da Orion e são somados para cada elemento chegando à área total da rede (Equação 5.4).

$$Carga_{elemento} = N_{ciclos\ com\ atividade}/N_{ciclos\ totais} \tag{5.1}$$

$$W_{elemento} = P_{elemento} * T_{tempo\ total}\ \ (J) \tag{5.2}$$

$$W_{NoC} = (\sum_{roteadores} W_{elemento}) + (\sum_{links} W_{elemento})\ \ (J) \tag{5.3}$$

$$A_{NoC} = (\sum_{roteadores} A_{elemento}) + (\sum_{links} A_{elemento})\ \ (m) \tag{5.4}$$

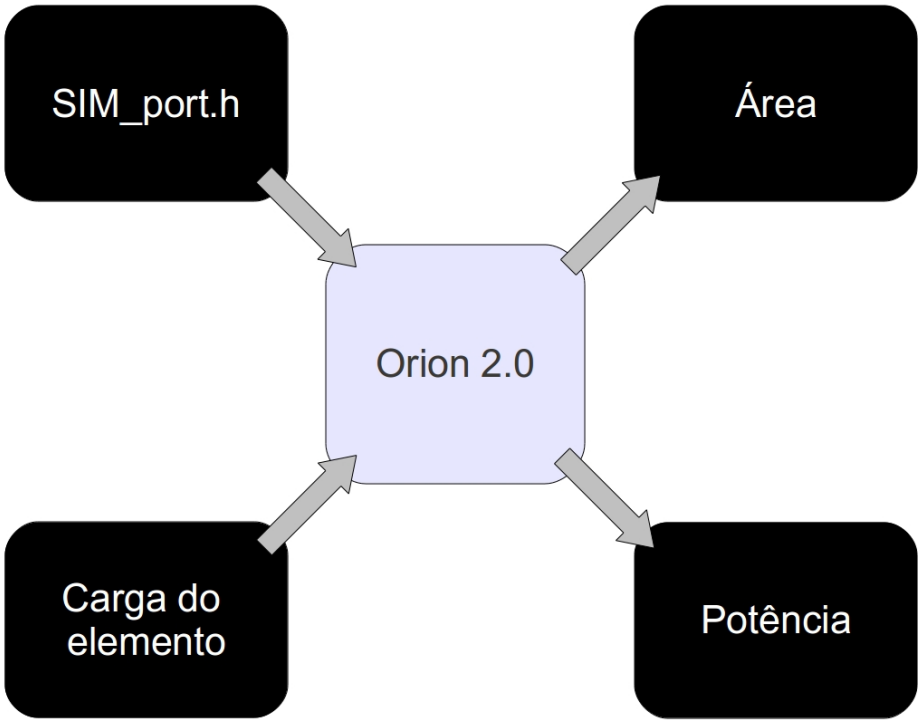


Figura 5.1: Entrada e saída da Orion.

6 RESULTADOS

Foram feitas comparações entre a primeira versão da SoCIN e a versão com *multicast* para demonstrar o ganho que pode ser obtido com o suporte adicionado. É importante ressaltar que os resultados aqui exibidos são resultado do suporte a *multicast*. Ganhos para o suporte a *clusters* se dão em forma de uma melhor infraestrutura na NoC e são mais complexos de serem analisados.

Para obter resultados, foi fixado o tamanho da rede como sendo 3x3, e seu tamanho de *flit* em dez bits e numa segunda rodada de experimentos trinta e quatro bits.

A comparação é feita entre *unicasts* para mensagens de oito e 1024 bytes para demonstrar que a rede não é afetada nesse aspecto, e após é comparado o *multicast* baseado em *unicast* com o *multicast* utilizando o suporte da NoC. O *unicast* foi feito do nodo (0,0) para o nodo (2,2). Como *multicast*, optou-se por utilizar um *broadcast* do nodo (0,0) para todos nodos, incluindo ele mesmo. Neste Capítulo, refere-se à versão modificada como SoCMEIN (System on Chip Multicast Enabled Interconnection Network). Quando não especificado, as referências para as observações são as figuras 6.1 e 6.2.

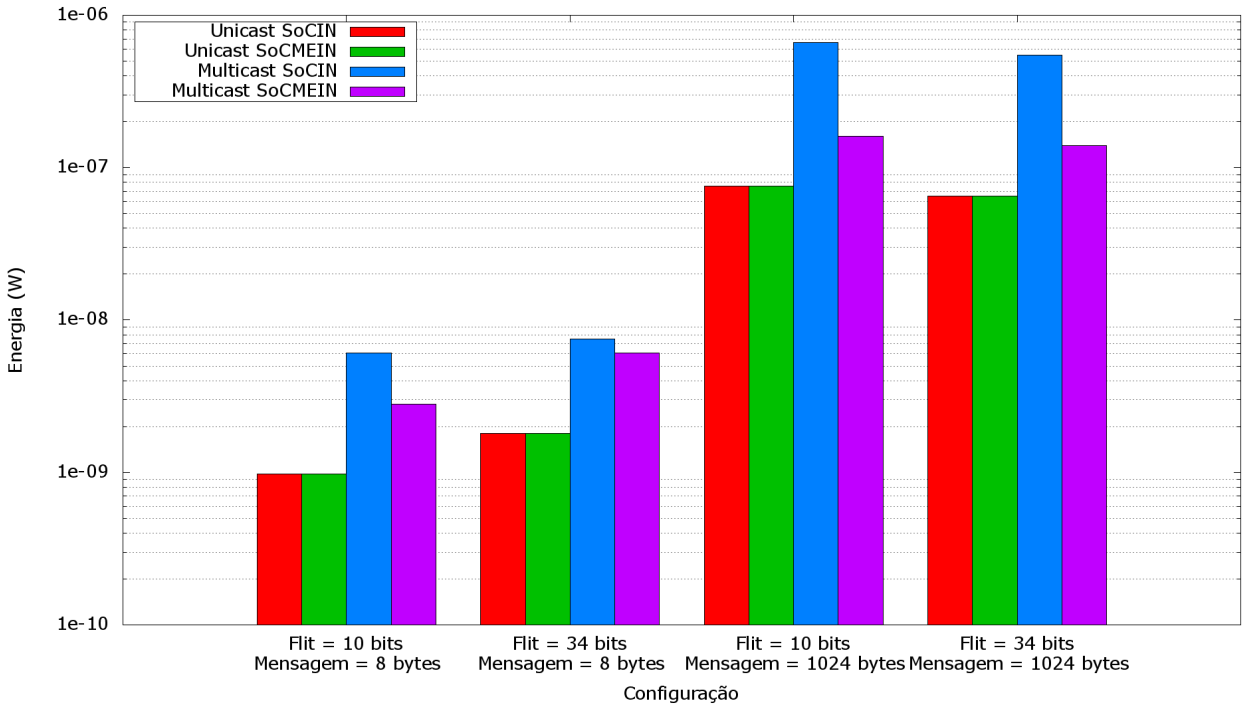


Figura 6.1: Resultados de energia para diversas configurações.

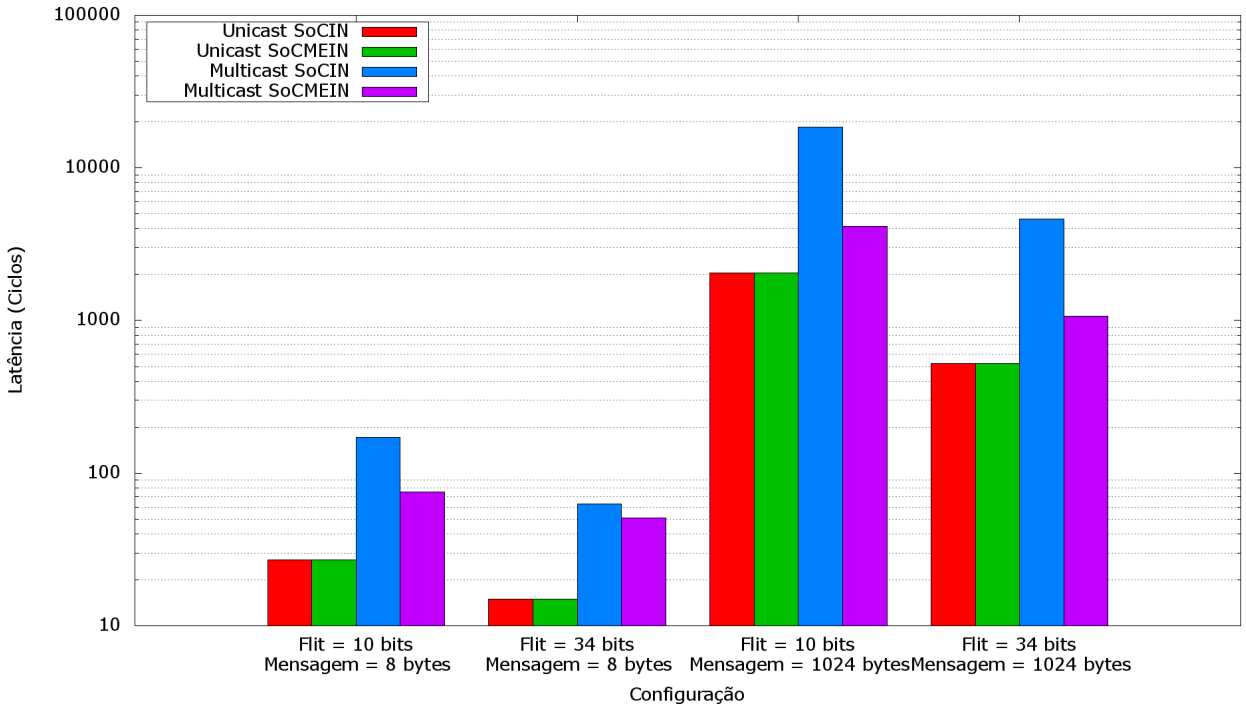


Figura 6.2: Resultados de latência para diversas configurações.

Tamanho de *flit* (bits)	Tamanho da carga (bytes)	Cabeçalho em relação à mensagem
10	8	11,11%
34	8	33,33%
10	1024	0,10%
34	1024	0.38%

Tabela 6.1: Tamanho do Cabeçalho em relação à mensagem para *unicast*.

Pode-se observar que para mensagens de *unicast* não há alteração, tanto na latência quanto na energia consumida. Isto condiz com o esperado, visto que para esta classe de mensagens não há nenhuma diferença entre as duas versões de NoC. Ainda, é possível ver uma redução, que é esperada, na latência ao aumentar o tamanho de *flit*. Como a mensagem será dividida em menos pedaços é de se esperar que, sob as mesmas condições, pedaços menores chegarão na mesma velocidade que pedaços maiores, logo a mensagem com menos pedaços chegará em menos ciclos. É possível também observar o impacto de *flits* mal dimensionados ao transmitirmos a mensagem de 8 bytes com diferentes tamanhos de *flit*. Este erro no dimensionamento reflete-se em um aumento da energia gasta. Isto se deve ao tamanho do cabeçalho aumentar em conjunto com o tamanho de *flit* tornando-o, para mensagens pequenas, uma grande parte do pacote (Tabela 6.1). Deste modo, o custo de transmissão do cabeçalho não é amortizado ao longo do resto do pacote e observa-se um consumo maior de energia.

Como mencionado anteriormente, mensagens de *multicast* que utilizam o suporte da NoC possuem mensagens mais longas, devido aos múltiplos cabeçalhos. Porém, este custo é amortizado devido ao envio de um pacote único (Tabela 6.2 e Tabela 6.3).

Pode-se fazer algumas observações quanto aos resultados relativos aos *multicasts*. Espera-se que, com a redução do número de *flits* (Tabela 6.2 e Tabela 6.3), tenha-se uma redução tanto na latência quanto na energia gasta. De fato, os resultados demonstram

Configuração	Tamanho do Pacote (*flits*)	Número de Pacotes	Total de *Flits*	Redução
SoCIN (Flit = 10 bits)	9	9	81	Base
SoCMEIN (Flit = 10 bits)	17	1	17	79,01%
SoCIN (Flit = 34 bits)	3	9	27	66,67%
SoCMEIN (Flit = 34 bits)	12	1	12	85,19%

Tabela 6.2: Diferença de *flits* para *multicast* de uma mensagem de 8 Bytes.

Configuração	Tamanho do Pacote (*flits*)	Número de Pacotes	Total de *Flits*	Redução
SoCIN (Flit = 10 bits)	1025	9	9225	Base
SoCMEIN (Flit = 10 bits)	1033	1	1033	88,80%
SoCIN (Flit = 34 bits)	257	9	2313	74,93%
SoCMEIN (Flit = 34 bits)	265	1	265	97,13%

Tabela 6.3: Diferença de *flits* para *multicast* de uma mensagem de 1024 Bytes.

reduções expressivas, chegando a reduções maiores do que 75% (Figura 6.3).

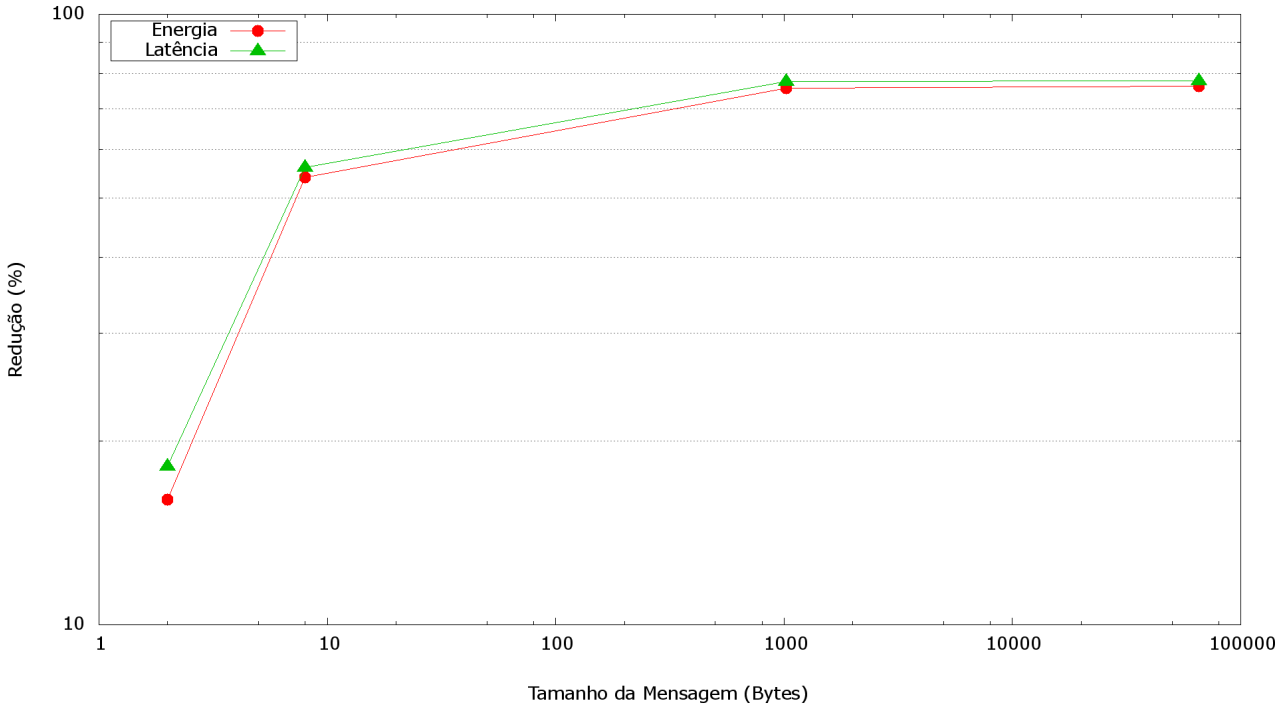


Figura 6.3: Redução de energia e latência para *flits* de dez bits e mensagem variável.

Uma análise mais profunda da Figura 6.3 mostra que as reduções tendem a crescer até um limite próximo a 80%, de acordo com o tamanho da mensagem. Isto pode ser explicado generalizando uma fórmula para o número de *flits* enviados pela fonte sem suporte (Equação 6.1) e com suporte (Equação 6.2), analogamento ao feito no Capítulo 4.

$$N_{flits\ sem\ suporte} = N_{nodos} * ((T_{mensagem}/(T_{flit} - 2)) + 1) \quad (6.1)$$

$$N_{flits\ com\ suporte} = N_{nodos} + T_{mensagem}/(T_{flit} - 2) \tag{6.2}$$

Generalizando uma equação para a redução, supondo que com suporte sempre terá menos *flits*:

$$\textit{Redução} = (N_{flits\ sem\ suporte} - N_{flits\ com\ suporte})/N_{flits\ sem\ suporte} \tag{6.3}$$

$$\textit{Redução} = (N_{nodos} - 1)/(N_{nodos} + N_{nodos}/(T_{mensagem}/(T_{flit} - 2))) \tag{6.4}$$

Se aumentarmos o tamanho da mensagem indefinidamente então tem-se:

$$\lim_{T_{mensagem} \to \infty} \textit{Redução} = (N_{nodos} - 1)/N_{nodos} \tag{6.5}$$

Assim é possível observar que, com o número de nodos não fixado, existe um ponto em que o número de *flits* gerado com suporte passa a ser muito pequeno comparado com o número gerado pela versão sem suporte e uma redução máxima é alcançada. Para um número de nodos fixo não existe redução maior do que a Equação 6.5. Logo, a partir deste ponto, não observa-se mais nenhuma redução na latência nem em energia, visto que o ganho está na redução de *flits*.

Este resultado implica que, quanto maior a rede, mais efetivo será o suporte. Como exemplo, a redução máxima para uma NoC com nove elementos é 88,89%, enquanto para uma NoC com trinta e dois elementos tem-se uma redução máxima de 96,88%. Mesmo no caso em que a NoC possua o menor número de elementos possível, tem-se uma redução de até 50%, mostrando que a solução proposta é efetiva para qualquer dimensão de rede.

Em termos de área, com *flits* de 10 bits, cada *link* resultou em uma área de aproximadamente $6.3\mu m^2$, enquanto cada roteador resultou em uma área de $17502.3\mu m^2$.

7 CONCLUSÃO

Este trabalho apresentou e implementou uma modificação não intrusiva para a rede SoCIN. É importante o fato de qualquer aplicação que utiliza a SoCIN é capaz de utilizar a versão modificada sem necessidade de alterações.

Com estas modificações, aplicações que se utilizam de *multicast* serão amplamente beneficiadas ao serem migradas. A migração é simples, sendo necessário somente garantir a unicidade do *multicast* e substituir o envio de múltiplos pacotes pelo envio de um pacote com vários cabeçalhos. Além da redução na latência e energia da NoC, também serão reduzidos a energia e o tempo gasto pelo elemento conectado ao roteador para inserir a mensagem na rede. Resultados mostraram que, utilizando o suporte, pode haver uma redução em energia e latência para *multicasts* de até 80%, com espaço para ganhos maiores, dependente do tamanho de mensagens e dimensões da NoC.

Graças à adição de *clusters*, tarefas agora podem optar por isolar-se do resto da NoC, sendo um mecanismo muito prático para tarefas de tempo real, as quais podem ser verificadas e validadas isoladamente da complexidade do resto da rede.

Não foram feitas tentativas de sintetizar o modelo, mas foi feito um esforço no sentido de tentar escrever o máximo possível de código sintetizável dentro dos módulos que compõem a rede.

Este trabalho cumpriu com seu objetivo, adicionando uma modelagem em RTL, parametrizável, com suporte a *multicast* e *clusters*, com resultados de área e potência aproximados, documentada e com extensões para depuração que pode ser utilizada em trabalhos futuros.

REFERÊNCIAS

BENINI, L.; MICHELI, G. D. Networks on Chip: a new paradigm for systems on chip design. In: IN PROCEEDINGS OF CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE. **Anais...** IEEE Computer Society, 2002. p.418–419.

COFFMAN, E. G.; ELPHICK, M.; SHOSHANI, A. System Deadlocks. **ACM Comput. Surv.**, New York, NY, USA, v.3, n.2, p.67–78, June 1971.

DOXYGEN. [S.l.]: http://www.doxygen.org/, 2012.

DUATO, J.; YALAMANCHILI, S.; NI, L. **Interconnection Networks**: an engineering approach. 1st.ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 1997.

GIT. [S.l.]: http://git-scm.com/, 2012.

GUPTA, R. K.; ZORIAN, Y. Introducing Core-Based System Design. **IEEE Des. Test**, Los Alamitos, CA, USA, v.14, p.15–25, October 1997.

HELD, J.; BAUTISTA, J.; KOEHL, S. **White Paper From a Few Cores to Many**: a tera-scale computing research review. 2006.

HWANG, K. **Advanced Computer Architecture**: parallelism,scalability,programmability. 1st.ed. [S.l.]: McGraw-Hill Higher Education, 1992.

INITIATIVE, O. S. [S.l.]: http://www.systemc.org/home/, 2012.

JERGER, N. E.; PEH, L. shiuan; LIPASTI, M. Virtual circuit tree multicasting: a case for on-chip hardware multicast support. In: PROCEEDINGS OF THE INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE (ISCA-35). **Anais...** [S.l.: s.n.], 2008.

KAHNG, A. B. et al. ORION 2.0: a fast and accurate noc power and area model for early-stage design space exploration. In: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, 3001 Leuven, Belgium, Belgium. **Proceedings...** European Design and Automation Association, 2009. p.423–428. (DATE '09).

LENG, X. et al. Implementation and simulation of a cluster-based hierarchical NoC architecture for multi-processor SoC. In: COMMUNICATIONS AND INFORMATION TECHNOLOGY, 2005. ISCIT 2005. IEEE INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2005. v.2, p.1203 – 1206.

LIU, J.; ZHENG, L.-R.; TENHUNEN, H. A guaranteed-throughput switch for network-on-chip. In: SYSTEM-ON-CHIP, 2003. PROCEEDINGS. INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2003. p.31 – 34.

MOHAPATRA, P. Wormhole routing techniques for directly connected multicomputer systems. **ACM Comput. Surv.**, New York, NY, USA, v.30, n.3, p.374–410, Sept. 1998.

MOORE, G. E. Cramming more components onto integrated circuits. **Electronics**, [S.l.], v.38, n.8, April 1965.

SAMMAN, F.; HOLLSTEIN, T.; GLESNER, M. Multicast Parallel Pipeline Router Architecture for Network-on-Chip. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 2008. DATE '08. **Anais...** [S.l.: s.n.], 2008. p.1396 –1401.

SEMATECH, I. **International Technology Roadmap for Semiconductors - 2007**. [S.l.]: http://www.itrs.net/Links/2007ITRS/, 2007.

SEMATECH, I. **International Technology Roadmap for Semiconductors - 2010 Update**. [S.l.]: http://www.itrs.net/Links/2010ITRS/, 2010.

SILVA JR, E. T. et al. A virtual platform for multiprocessor real-time embedded systems. In: JAVA TECHNOLOGIES FOR REAL-TIME AND EMBEDDED SYSTEMS, 6., New York, NY, USA. **Proceedings...** ACM, 2008. p.31–37. (JTRES '08).

ZEFERINO, C. A.; SUSIN, A. A. SoCIN: a parametric and scalable network-on-chip. In: SBCCI. **Anais...** [S.l.: s.n.], 2003. p.169–.

APÊNDICE A MANUAL

Implementação de uma Rede em Chip com Suporte a Clusters Dinâmicos e Multicast

Generated by Doxygen 1.7.6.1

Thu Jul 12 2012 22:32:34

Contents

Chapter 1

Directory Hierarchy

1.1 Directories

This directory hierarchy is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Directory Documentation

4.1 common/ Directory Reference

Directory dependency graph for common/:

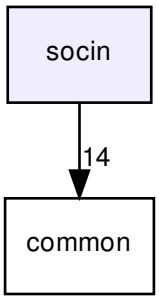
Files

- file config_2d.cpp
- file config_2d.h
- file Flit.cpp
- file Flit.h
- file IRS.cpp
- file IRS.h
- file Link.cpp
- file Link.h
- file ODS.cpp
- file ODS.h
- file ORS.cpp
- file ORS.h
- file Port.cpp

- file Port.h
- file Position.cpp
- file Position.h
- file RIB.cpp
- file RIB.h
- file RouterMatrix.cpp
- file RouterMatrix.h
- file Socin.cpp
- file Socin.h
- file TrafficGenerator.cpp
- file TrafficGenerator.h

4.2 socin/ Directory Reference

Directory dependency graph for socin/:

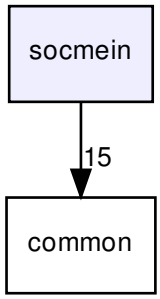


Files

- file IB.cpp
- file IB.h
- file IC.cpp
- file IC.h
- file main.cpp
- file main.h
- file OC.cpp
- file OC.h
- file OFC.cpp
- file OFC.h
- file Router.cpp
- file Router.h

4.3 socmein/ Directory Reference

Directory dependency graph for socmein/:

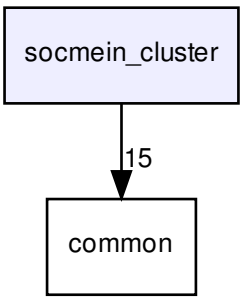


Files

- file IB.cpp
- file IB.h
- file IC.cpp
- file IC.h
- file main.cpp
- file main.h
- file ME.cpp
- file ME.h
- file OC.cpp
- file OC.h
- file OFC.cpp
- file OFC.h
- file Router.cpp
- file Router.h

4.4 socmein_cluster/ Directory Reference

Directory dependency graph for socmein_cluster/:



Files

- file IB.cpp
- file IB.h
- file IC.cpp
- file IC.h
- file main.cpp
- file main.h
- file ME.cpp
- file ME.h
- file OC.cpp
- file OC.h
- file OFC.cpp
- file OFC.h
- file Router.cpp
- file Router.h

Chapter 5

Class Documentation

5.1 Flit Class Reference

`#include <Flit.h>`

Public Member Functions

- Flit (bool bop=false, bool eop=false, sc_bv< N_SIZE > data=0)
- void print ()
- string toString (void)
- bool operator== (const Flit &flit) const
- Flit & operator= (const Flit &flit)

Public Attributes

- bool bop
- bool eop
- sc_bv< N_SIZE > data
- unsigned int ID

Friends

- ostream & operator<< (ostream &os, const Flit &flit)
- void sc_trace (sc_trace_file *tf, const Flit &flit, const std::string &NAME)

5.1.1 Detailed Description

Class representing a flit

Definition at line 8 of file Flit.h.

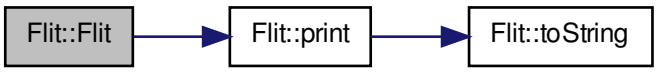
5.1.2 Constructor & Destructor Documentation

5.1.2.1 Flit::Flit (bool *bop* = `false`, bool *eop* = `false`, sc_bv< N_SIZE > *data* = `0`)

Definition at line 3 of file Flit.cpp.

References ID, and print().

Here is the call graph for this function:



5.1.3 Member Function Documentation

5.1.3.1 Flit& Flit::operator= (const Flit & *flit*) `[inline]`

Definition at line 25 of file Flit.h.

References bop, data, eop, and ID.

5.1.3.2 bool Flit::operator== (const Flit & *flit*) const `[inline]`

Definition at line 21 of file Flit.h.

References bop, data, and eop.

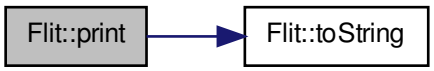
5.1.3.3 void Flit::print ( )

Definition at line 59 of file Flit.cpp.

References toString().

Referenced by Flit().

Here is the call graph for this function:



Here is the caller graph for this function:

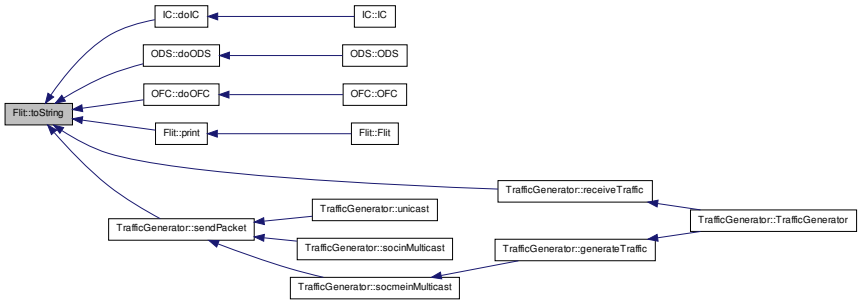
5.1.3.4 string Flit::toString (void)

Definition at line 13 of file Flit.cpp.

References bop, data, EAST, eop, ID, and NORTH.

Referenced by IC::doIC(), ODS::doODS(), OFC::doOFC(), print(), TrafficGenerator-::receiveTraffic(), and TrafficGenerator::sendPacket().

Here is the caller graph for this function:



5.1.4 Friends And Related Function Documentation

5.1.4.1 ostream& operator<< (ostream & *os,* const Flit & *flit*) [friend]

Definition at line 33 of file Flit.h.

5.1.4.2 void sc_trace (sc_trace_file * *tf,* const Flit & *flit,* const std::string & *NAME*) [friend]

Definition at line 37 of file Flit.h.

5.1.5 Member Data Documentation

5.1.5.1 bool Flit::bop

Definition at line 10 of file Flit.h.

Referenced by operator=(), operator==(), RIB::RIB(), TrafficGenerator::sendPacket(), and toString().

5.1.5.2 sc_bv<N_SIZE> Flit::data

Definition at line 12 of file Flit.h.

Referenced by operator=(), operator==(), RIB::RIB(), TrafficGenerator::sendPacket(), RIB::toFlit(), and toString().

5.1.5.3 bool Flit::eop

Definition at line 11 of file Flit.h.

Referenced by operator=(), operator==(), RIB::RIB(), TrafficGenerator::sendPacket(), and toString().

5.1.5.4 unsigned int Flit::ID

Definition at line 13 of file Flit.h.

Referenced by Flit(), operator=(), TrafficGenerator::receiveTraffic(), TrafficGenerator::sendPacket(), TrafficGenerator::socmeinMulticast(), and toString().

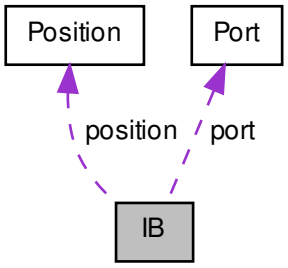
The documentation for this class was generated from the following files:

- common/Flit.h
- common/Flit.cpp

5.2 IB Struct Reference

`#include <IB.h>`

Collaboration diagram for IB:



Public Types

- typedef IB SC_CURRENT_USER_MODULE
- typedef IB SC_CURRENT_USER_MODULE
- typedef IB SC_CURRENT_USER_MODULE

Public Member Functions

- IB (sc_module_name name, Position *position, Port *port, unsigned int maxSize)
- void doIB ()
- void updateBuffer ()
- IB (sc_module_name name, Position *position, Port *port, unsigned int maxSize)
- void doIB ()
- void updateBuffer ()
- IB (sc_module_name name, Position *position, Port *port, unsigned int maxSize)
- void doIB ()
- void updateBuffer ()

Public Attributes

- sc_in< bool > clk
- sc_fifo_in< Flit > din
- sc_in< bool > rd
- sc_out< bool > rok

- sc_out< Flit > dout
- list< Flit > buffer
- unsigned int maxSize
- Position * position
- Port * port
- sc_in< N_SIZE > nID
- sc_bv< N_SIZE > * ID

5.2.1 Detailed Description

Definition at line 10 of file IB.h.

5.2.2 Member Typedef Documentation

5.2.2.1 typedef IB IB::SC_CURRENT_USER_MODULE

Definition at line 27 of file IB.h.

5.2.2.2 typedef IB IB::SC_CURRENT_USER_MODULE

Definition at line 27 of file IB.h.

5.2.2.3 typedef IB IB::SC_CURRENT_USER_MODULE

Definition at line 30 of file IB.h.

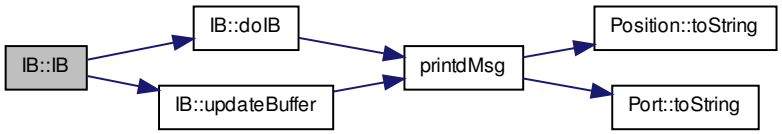
5.2.3 Constructor & Destructor Documentation

5.2.3.1 IB::IB (sc_module_name *name*, Position * *position*, Port * *port*, unsigned int *maxSize*)

Definition at line 3 of file IB.cpp.

References clk, doIB(), port, position, rd, and updateBuffer().

Here is the call graph for this function:



5.2.3.2 IB::IB (sc_module_name *name*, Position * *position*, Port * *port*, unsigned int *maxSize*)

5.2.3.3 IB::IB (sc_module_name *name*, Position * *position*, Port * *port*, unsigned int *maxSize*)

5.2.4 Member Function Documentation

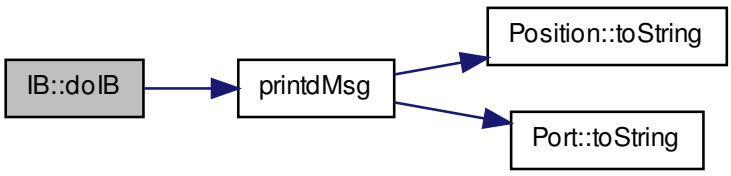
5.2.4.1 void IB::doIB ( )

Definition at line 25 of file IB.cpp.

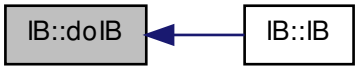
References buffer, din, dMsg, dout, maxSize, position, printdMsg(), and rok.

Referenced by IB().

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.4.2 void IB::doIB ( )

5.2.4.3 void IB::doIB ( )

5.2.4.4 void IB::updateBuffer ( )

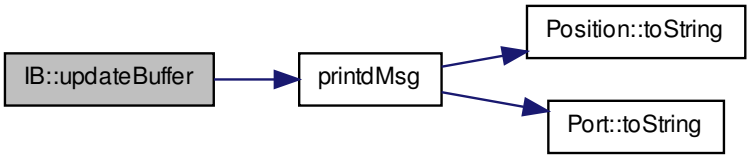
5.2.4.5 void IB::updateBuffer ( )

Definition at line 13 of file IB.cpp.

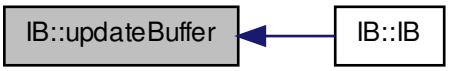
References buffer, dMsg, position, printdMsg(), and rd.

Referenced by IB().

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.4.6 void IB::updateBuffer ( )

5.2.5 Member Data Documentation

5.2.5.1 list< Flit > IB::buffer

Definition at line 20 of file IB.h.

Referenced by doIB(), and updateBuffer().

5.2.5.2 sc_in< bool > IB::clk

Definition at line 12 of file IB.h.

Referenced by IB(), and Router::Router().

5.2.5.3 sc_fifo_in< Flit > IB::din

Definition at line 14 of file IB.h.

Referenced by doIB(), and Router::Router().

5.2.5.4 sc_out< Flit > IB::dout

Definition at line 18 of file IB.h.

Referenced by doIB().

5.2.5.5 sc_bv<N_SIZE>* IB::ID

Definition at line 28 of file IB.h.

5.2.5.6 unsigned int IB::maxSize

Definition at line 21 of file IB.h.

Referenced by doIB().

5.2.5.7 sc_in<N_SIZE> IB::nID

Definition at line 16 of file IB.h.

5.2.5.8 Port * IB::port

Definition at line 25 of file IB.h.

Referenced by IB().

5.2.5.9 Position * IB::position

Definition at line 24 of file IB.h.

Referenced by doIB(), IB(), and updateBuffer().

5.2.5.10 sc_in< bool > IB::rd

Definition at line 15 of file IB.h.

Referenced by IB(), and updateBuffer().

5.2.5.11 sc_out< bool > IB::rok

Definition at line 17 of file IB.h.

Referenced by doIB(), and Router::Router().

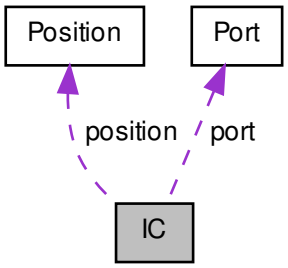
The documentation for this struct was generated from the following files:

- socin/IB.h
- socmein/IB.h
- socmein_cluster/IB.h
- socin/IB.cpp
- socmein/IB.cpp
- socmein_cluster/IB.cpp

5.3 IC Struct Reference

`#include <IC.h>`

Collaboration diagram for IC:



Public Types

- typedef IC SC_CURRENT_USER_MODULE
- typedef IC SC_CURRENT_USER_MODULE
- typedef IC SC_CURRENT_USER_MODULE

Public Member Functions

- IC (sc_module_name name, Position ∗position, Port ∗port)
- void doIC ()
- IC (sc_module_name name, Position ∗position, Port ∗port)
- void doIC ()
- IC (sc_module_name name, Position ∗position, Port ∗port)
- void doIC ()

Public Attributes

- sc_in< Flit > dout
- sc_in< bool > x_rok
- sc_out< bool > x_req [NUMBER_OF_DIRECTIONS]
- sc_out< Flit > x_dout
- Position ∗ position
- Port ∗ port
- bool active
- sc_in< bool > x_m_gnt
- sc_out< bool > x_m_req

5.3.1 Detailed Description

Definition at line 9 of file IC.h.

5.3.2 Member Typedef Documentation

5.3.2.1 typedef IC IC::SC_CURRENT_USER_MODULE

Definition at line 23 of file IC.h.

5.3.2.2 typedef IC IC::SC_CURRENT_USER_MODULE

Definition at line 25 of file IC.h.

5.3.2.3 typedef IC IC::SC_CURRENT_USER_MODULE

Definition at line 25 of file IC.h.

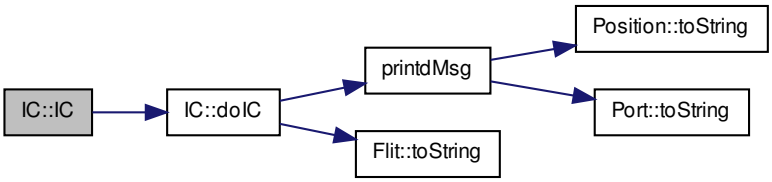
5.3.3 Constructor & Destructor Documentation

5.3.3.1 IC::IC (sc_module_name *name,* Position * *position,* Port * *port*)

Definition at line 3 of file IC.cpp.

References active, doIC(), dout, port, position, and x_rok.

Here is the call graph for this function:



5.3.3.2 IC::IC (sc_module_name *name,* Position * *position,* Port * *port*)

5.3.3.3 IC::IC (sc_module_name *name,* Position * *position,* Port * *port*)

5.3.4 Member Function Documentation

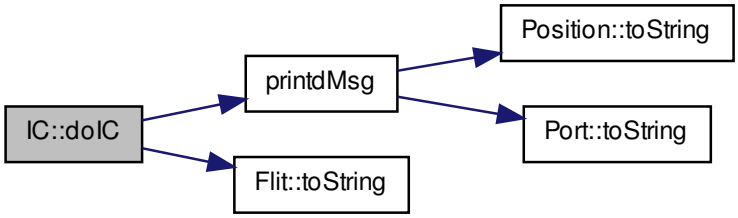
5.3.4.1 void IC::doIC ( )

Definition at line 14 of file IC.cpp.

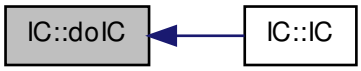
References active, dMsg, dout, E, EAST, L, N, NUMBER_OF_DIRECTIONS, position, printdMsg(), S, SOUTH, Flit::toString(), W, x_dout, and x_req.

Referenced by IC().

Here is the call graph for this function:



Here is the caller graph for this function:



5.3.4.2 void IC::doIC ( )

5.3.4.3 void IC::doIC ( )

5.3.5 Member Data Documentation

5.3.5.1 bool IC::active

Definition at line 21 of file IC.h.

Referenced by Router::checkForActivity(), doIC(), and IC().

5.3.5.2 sc_in< Flit > IC::dout

Definition at line 11 of file IC.h.

Referenced by doIC(), and IC().

5.3.5.3 Port * IC::port

Definition at line 19 of file IC.h.

Referenced by IC().

5.3.5.4 Position * IC::position

Definition at line 18 of file IC.h.

Referenced by doIC(), and IC().

5.3.5.5 sc_out< Flit > IC::x_dout

Definition at line 15 of file IC.h.

Referenced by doIC(), and Router::Router().

5.3.5.6 sc_in< bool > IC::x_m_gnt

Definition at line 13 of file IC.h.

5.3.5.7 sc_out< bool > IC::x_m_req

Definition at line 17 of file IC.h.

5.3.5.8 sc_out< bool > IC::x_req

Definition at line 14 of file IC.h.

Referenced by doIC().

5.3.5.9 sc_in< bool > IC::x_rok

Definition at line 12 of file IC.h.

Referenced by IC(), and Router::Router().

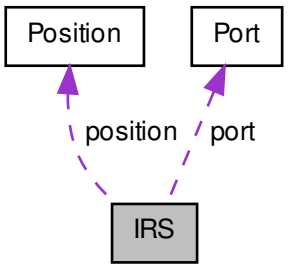
The documentation for this struct was generated from the following files:

- socin/IC.h
- socmein/IC.h
- socmein_cluster/IC.h
- socin/IC.cpp
- socmein/IC.cpp
- socmein_cluster/IC.cpp

5.4 IRS Struct Reference

`#include <IRS.h>`

Collaboration diagram for IRS:



Public Types

- typedef IRS SC_CURRENT_USER_MODULE

Public Member Functions

- IRS (sc_module_name name, Position *position, Port *port)
- void doIRS ()

Public Attributes

- sc_in< bool > x_gnt [NUMBER_OF_DIRECTIONS]
- sc_in< bool > x_rd [NUMBER_OF_DIRECTIONS]
- sc_out< bool > rd
- Position * position
- Port * port

5.4.1 Detailed Description

Definition at line 8 of file IRS.h.

5.4.2 Member Typedef Documentation

5.4.2.1 typedef IRS IRS::SC_CURRENT_USER_MODULE

Definition at line 19 of file IRS.h.

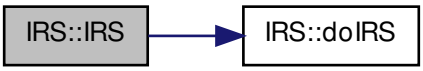
5.4.3 Constructor & Destructor Documentation

5.4.3.1 IRS::IRS (sc_module_name *name*, Position * *position*, Port * *port*)

Definition at line 3 of file IRS.cpp.

References doIRS(), NUMBER_OF_DIRECTIONS, port, position, x_gnt, and x_rd.

Here is the call graph for this function:



5.4.4 Member Function Documentation

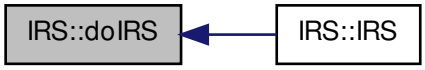
5.4.4.1 void IRS::doIRS ( )

Definition at line 14 of file IRS.cpp.

References NUMBER_OF_DIRECTIONS, rd, x_gnt, and x_rd.

Referenced by IRS().

Here is the caller graph for this function:



5.4.5 Member Data Documentation

5.4.5.1 Port* IRS::port

Definition at line 17 of file IRS.h.

Referenced by IRS().

5.4.5.2 Position* IRS::position

Definition at line 16 of file IRS.h.

Referenced by IRS().

5.4.5.3 sc_out<bool> IRS::rd

Definition at line 13 of file IRS.h.

Referenced by doIRS().

5.4.5.4 sc_in<bool> IRS::x_gnt[NUMBER_OF_DIRECTIONS]

Definition at line 10 of file IRS.h.

Referenced by doIRS(), IRS(), and Router::Router().

5.4.5.5 sc_in<bool> IRS::x_rd[NUMBER_OF_DIRECTIONS]

Definition at line 11 of file IRS.h.

Referenced by doIRS(), and IRS().

The documentation for this struct was generated from the following files:

- common/IRS.h
- common/IRS.cpp

5.5 Link Class Reference

`#include <Link.h>`

Public Member Functions

- Link ()
- ∼Link ()
- void write (const Flit &flit)
- bool nb_write (const Flit &flit)
- void read (Flit &flit)
- Flit read ()
- void orion_link (unsigned int cycleCount, LinkReport ∗linkReport)

Public Attributes

- unsigned int activityCount
- u_int data_width
- double link_len
- double load
- double freq
- double dynamicPower
- double leakagePower
- double totalPower
- double link_area
- unsigned int writeCount
- unsigned int readCount

5.5.1 Detailed Description

Definition at line 22 of file Link.h.

5.5.2 Constructor & Destructor Documentation

5.5.2.1 Link::Link ( )

Definition at line 3 of file Link.cpp.

References activityCount, data_width, dynamicPower, freq, leakagePower, link_area, link_len, load, and totalPower.

5.5.2.2 Link::∼Link ( )

Definition at line 18 of file Link.cpp.

5.5.3 Member Function Documentation

5.5.3.1 bool Link::nb_write (const Flit & *flit*)

Definition at line 28 of file Link.cpp.

References activityCount.

5.5.3.2 void Link::orion_link (unsigned int *cycleCount,* LinkReport * *linkReport*)

Definition at line 44 of file Link.cpp.

References activityCount, LinkReport::area, data_width, dynamicPower, freq, leakagePower, link_area, link_len, load, LinkReport::power, and totalPower.

5.5.3.3 void Link::read (Flit & *flit*)

Definition at line 39 of file Link.cpp.

References read().

Here is the call graph for this function:

5.5.3.4 Flit Link::read ( )

Definition at line 34 of file Link.cpp.

Referenced by read().

Here is the caller graph for this function:

5.5.3.5 void Link::write (const Flit & *flit*)

Definition at line 22 of file Link.cpp.

References activityCount.

5.5.4 Member Data Documentation

5.5.4.1 unsigned int Link::activityCount

Definition at line 24 of file Link.h.

Referenced by Link(), nb_write(), orion_link(), and write().

5.5.4.2 u_int Link::data_width

Definition at line 27 of file Link.h.

Referenced by Link(), and orion_link().

5.5.4.3 double Link::dynamicPower

Definition at line 33 of file Link.h.

Referenced by Link(), and orion_link().

5.5.4.4 double Link::freq

Definition at line 30 of file Link.h.

Referenced by Link(), and orion_link().

5.5.4.5 double Link::leakagePower

Definition at line 34 of file Link.h.

Referenced by Link(), and orion_link().

5.5.4.6 double Link::link_area

Definition at line 36 of file Link.h.

Referenced by Link(), and orion_link().

5.5.4.7 double Link::link_len

Definition at line 28 of file Link.h.

Referenced by Link(), and orion_link().

5.5.4.8 double Link::load

Definition at line 29 of file Link.h.

Referenced by Link(), and orion_link().

5.5.4.9 unsigned int Link::readCount

Definition at line 39 of file Link.h.

5.5.4.10 double Link::totalPower

Definition at line 35 of file Link.h.

Referenced by Link(), and orion_link().

5.5.4.11 unsigned int Link::writeCount

Definition at line 38 of file Link.h.

The documentation for this class was generated from the following files:

- common/Link.h
- common/Link.cpp

5.6 LinkReport Class Reference

```
#include <Link.h>
```

Public Member Functions

- LinkReport ()

Public Attributes

- double power
- double area

5.6.1 Detailed Description

Definition at line 11 of file Link.h.

5.6.2 Constructor & Destructor Documentation

5.6.2.1 LinkReport::LinkReport ( ) [inline]

Definition at line 16 of file Link.h.

References area, and power.

5.6.3 Member Data Documentation

5.6.3.1 double LinkReport::area

Definition at line 15 of file Link.h.

Referenced by LinkReport(), Link::orion_link(), and RouterMatrix::∼RouterMatrix().

5.6.3.2 double LinkReport::power

Definition at line 14 of file Link.h.

Referenced by LinkReport(), Link::orion_link(), and RouterMatrix::∼RouterMatrix().

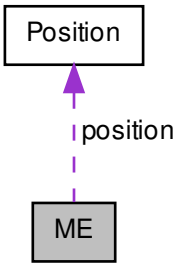
The documentation for this class was generated from the following file:

- common/Link.h

5.7 ME Struct Reference

```
#include <ME.h>
```

Collaboration diagram for ME:



Public Types

- typedef ME SC_CURRENT_USER_MODULE
- typedef ME SC_CURRENT_USER_MODULE

Public Member Functions

- ME (sc_module_name name, Position ∗position)
- void doME ()
- void outputSync ()
- ME (sc_module_name name, Position ∗position)
- void doME ()
- void outputSync ()

Public Attributes

- sc_in< bool > clk
- sc_in< bool > x_m_req [NUMBER_OF_DIRECTIONS]
- sc_in< bool > free [NUMBER_OF_DIRECTIONS]
- sc_out< bool > x_m_gnt [NUMBER_OF_DIRECTIONS]
- sc_out< bool > outputs_free
- list< en_directions > priorityList
- bool connected
- en_directions connection
- Position ∗ position

5.7.1 Detailed Description

Definition at line 10 of file ME.h.

5.7.2 Member Typedef Documentation

5.7.2.1 typedef ME ME::SC_CURRENT_USER_MODULE

Definition at line 27 of file ME.h.

5.7.2.2 typedef ME ME::SC_CURRENT_USER_MODULE

Definition at line 27 of file ME.h.

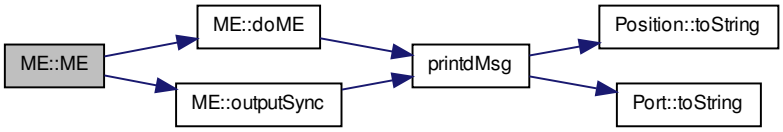
5.7.3 Constructor & Destructor Documentation

5.7.3.1 ME::ME (sc_module_name *name,* Position * *position*)

Definition at line 3 of file ME.cpp.

References clk, connected, doME(), NUMBER_OF_DIRECTIONS, outputSync(), position, priorityList, and x_m_req.

Here is the call graph for this function:



5.7.3.2 ME::ME (sc_module_name *name,* Position * *position*)

5.7.4 Member Function Documentation

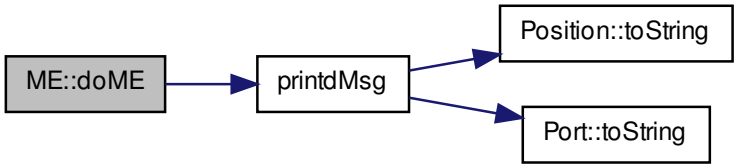
5.7.4.1 void ME::doME ( )

Definition at line 18 of file ME.cpp.

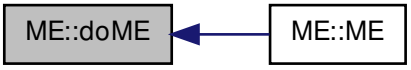
References connected, connection, directionString, dMsg, position, printdMsg(), priorityList, x_m_gnt, and x_m_req.

Referenced by ME().

Here is the call graph for this function:



Here is the caller graph for this function:



5.7.4.2 void ME::doME ( )

5.7.4.3 void ME::outputSync ( )

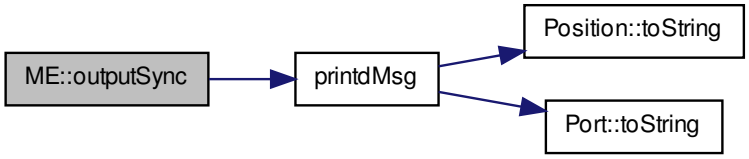
5.7.4.4 void ME::outputSync ( )

Definition at line 50 of file ME.cpp.

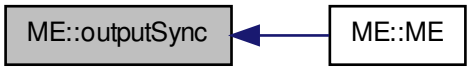
References dMsg, free, NUMBER_OF_DIRECTIONS, outputs_free, position, and printdMsg().

Referenced by ME().

Here is the call graph for this function:



Here is the caller graph for this function:



5.7.5 Member Data Documentation

5.7.5.1 sc_in< bool > ME::clk

Definition at line 12 of file ME.h.

Referenced by ME().

5.7.5.2 bool ME::connected

Definition at line 21 of file ME.h.

Referenced by doME(), and ME().

5.7.5.3 en_directions ME::connection

Definition at line 22 of file ME.h.

Referenced by doME().

5.7.5.4 sc_in< bool > ME::free

Definition at line 14 of file ME.h.

Referenced by outputSync().

5.7.5.5 sc_out< bool > ME::outputs_free

Definition at line 17 of file ME.h.

Referenced by outputSync().

5.7.5.6 Position * ME::position

Definition at line 25 of file ME.h.

Referenced by doME(), ME(), and outputSync().

5.7.5.7 list< en_directions > ME::priorityList

Definition at line 19 of file ME.h.

Referenced by doME(), and ME().

5.7.5.8 sc_out< bool > ME::x_m_gnt

Definition at line 16 of file ME.h.

Referenced by doME().

5.7.5.9 sc_in< bool > ME::x_m_req

Definition at line 13 of file ME.h.

Referenced by doME(), and ME().

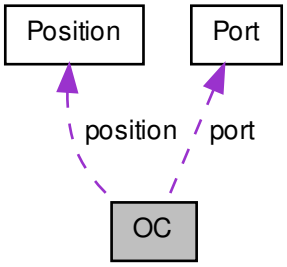
The documentation for this struct was generated from the following files:

- socmein/ME.h
- socmein_cluster/ME.h
- socmein/ME.cpp
- socmein_cluster/ME.cpp

5.8 OC Struct Reference

```
#include <OC.h>
```

Collaboration diagram for OC:



Public Types

- typedef OC SC_CURRENT_USER_MODULE
- typedef OC SC_CURRENT_USER_MODULE
- typedef OC SC_CURRENT_USER_MODULE

Public Member Functions

- OC (sc_module_name name, Position ∗position, Port ∗port)
- void doOC ()
- OC (sc_module_name name, Position ∗position, Port ∗port)
- void doOC ()
- OC (sc_module_name name, Position ∗position, Port ∗port)
- void doOC ()

Public Attributes

- sc_in< bool > eop
- sc_in< bool > x_req [NUMBER_OF_DIRECTIONS]
- sc_in< bool > x_rd
- sc_out< bool > x_gnt [NUMBER_OF_DIRECTIONS]
- list< en_directions > priorityList
- bool connected
- Position ∗ position
- Port ∗ port
- sc_out< bool > gnt_and_req
- en_directions connection

5.8.1 Detailed Description

Definition at line 10 of file OC.h.

5.8.2 Member Typedef Documentation

5.8.2.1 typedef OC OC::SC_CURRENT_USER_MODULE

Definition at line 26 of file OC.h.

5.8.2.2 typedef OC OC::SC_CURRENT_USER_MODULE

Definition at line 28 of file OC.h.

5.8.2.3 typedef OC OC::SC_CURRENT_USER_MODULE

Definition at line 28 of file OC.h.

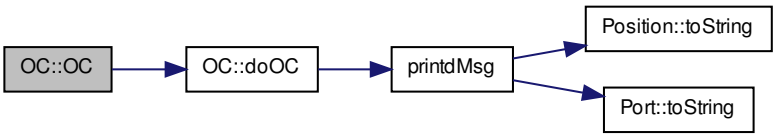
5.8.3 Constructor & Destructor Documentation

5.8.3.1 OC::OC (sc_module_name *name,* Position * *position,* Port * *port*)

Definition at line 3 of file OC.cpp.

References connected, doOC(), eop, NUMBER_OF_DIRECTIONS, port, position, priorityList, x_rd, and x_req.

Here is the call graph for this function:



5.8.3.2 OC::OC (sc_module_name *name,* Position * *position,* Port * *port*)

5.8.3.3 OC::OC (sc_module_name *name,* Position * *position,* Port * *port*)

5.8.4 Member Function Documentation

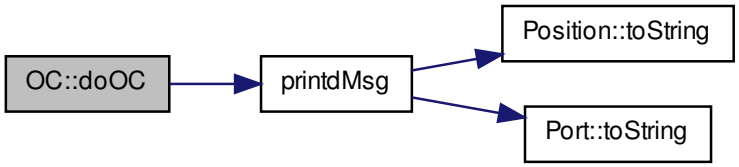
5.8.4.1 void OC::doOC ( )

Definition at line 18 of file OC.cpp.

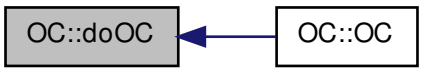
References connected, directionString, dMsg, eop, NUMBER_OF_DIRECTIONS, position, printdMsg(), priorityList, x_gnt, x_rd, and x_req.

Referenced by OC().

Here is the call graph for this function:



Here is the caller graph for this function:



5.8.4.2 void OC::doOC ( )

5.8.4.3 void OC::doOC ( )

5.8.5 Member Data Documentation

5.8.5.1 bool OC::connected

Definition at line 20 of file OC.h.

Referenced by doOC(), and OC().

5.8.5.2 en_directions OC::connection

Definition at line 22 of file OC.h.

5.8.5.3 sc_in< bool > OC::eop

Definition at line 12 of file OC.h.

Referenced by doOC(), and OC().

5.8.5.4 sc_out< bool > OC::gnt_and_req

Definition at line 17 of file OC.h.

5.8.5.5 Port * OC::port

Definition at line 24 of file OC.h.

Referenced by OC().

5.8.5.6 Position * OC::position

Definition at line 23 of file OC.h.

Referenced by doOC(), and OC().

5.8.5.7 list< en_directions > OC::priorityList

Definition at line 18 of file OC.h.

Referenced by doOC(), and OC().

5.8.5.8 sc_out< bool > OC::x_gnt

Definition at line 16 of file OC.h.

Referenced by doOC(), and Router::Router().

5.8.5.9 sc_in< bool > OC::x_rd

Definition at line 14 of file OC.h.

Referenced by doOC(), OC(), and Router::Router().

5.8.5.10 sc_in< bool > OC::x_req

Definition at line 13 of file OC.h.

Referenced by doOC(), and OC().

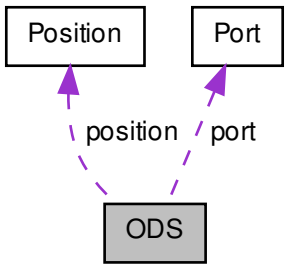
The documentation for this struct was generated from the following files:

- socin/OC.h
- socmein/OC.h
- socmein_cluster/OC.h
- socin/OC.cpp
- socmein/OC.cpp
- socmein_cluster/OC.cpp

5.9 ODS Struct Reference

`#include <ODS.h>`

Collaboration diagram for ODS:



Public Types

- typedef ODS SC_CURRENT_USER_MODULE

Public Member Functions

- ODS (sc_module_name name, Position *position, Port *port)
- void doODS ()

Public Attributes

- sc_in< bool > x_gnt [NUMBER_OF_DIRECTIONS]
- sc_in< Flit > x_din [NUMBER_OF_DIRECTIONS]
- sc_out< Flit > out_data
- sc_out< bool > eop
- Position * position
- Port * port

5.9.1 Detailed Description

Definition at line 8 of file ODS.h.

5.9.2 Member Typedef Documentation

5.9.2.1 typedef ODS ODS::SC_CURRENT_USER_MODULE

Definition at line 20 of file ODS.h.

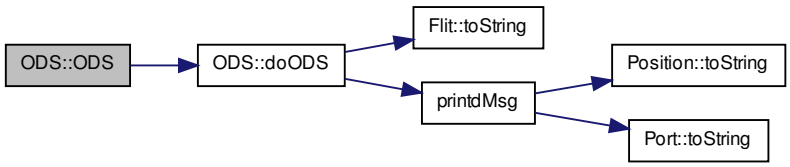
5.9.3 Constructor & Destructor Documentation

5.9.3.1 ODS::ODS (sc_module_name *name*, Position * *position*, Port * *port*)

Definition at line 3 of file ODS.cpp.

References doODS(), NUMBER_OF_DIRECTIONS, port, position, x_din, and x_gnt.

Here is the call graph for this function:



5.9.4 Member Function Documentation

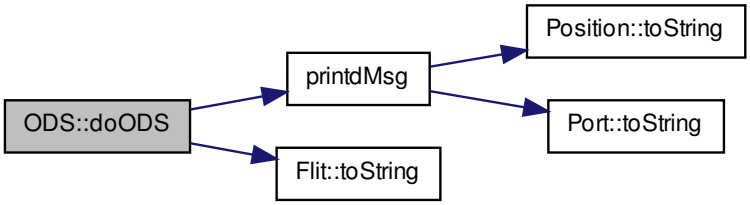
5.9.4.1 void ODS::doODS ( )

Definition at line 14 of file ODS.cpp.

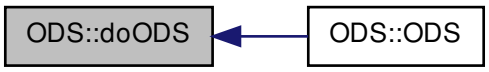
References directionString, dMsg, eop, NUMBER_OF_DIRECTIONS, out_data, position, printdMsg(), Flit::toString(), x_din, and x_gnt.

Referenced by ODS().

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.5 Member Data Documentation

5.9.5.1 sc_out<bool> ODS::eop

Definition at line 14 of file ODS.h.

Referenced by doODS().

5.9.5.2 sc_out<Flit> ODS::out_data

Definition at line 13 of file ODS.h.

Referenced by doODS().

5.9.5.3 Port* ODS::port

Definition at line 18 of file ODS.h.

Referenced by ODS().

5.9.5.4 Position* ODS::position

Definition at line 17 of file ODS.h.

Referenced by doODS(), and ODS().

5.9.5.5 sc_in<Flit> ODS::x_din[NUMBER_OF_DIRECTIONS]

Definition at line 11 of file ODS.h.

Referenced by doODS(), and ODS().

5.9.5.6 sc_in<bool> ODS::x_gnt[NUMBER_OF_DIRECTIONS]

Definition at line 10 of file ODS.h.

Referenced by doODS(), ODS(), and Router::Router().

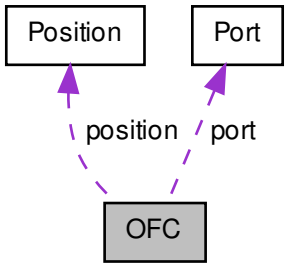
The documentation for this struct was generated from the following files:

- common/ODS.h
- common/ODS.cpp

5.10 OFC Struct Reference

```
#include <OFC.h>
```

Collaboration diagram for OFC:



Public Types

- typedef OFC SC_CURRENT_USER_MODULE
- typedef OFC SC_CURRENT_USER_MODULE
- typedef OFC SC_CURRENT_USER_MODULE

Public Member Functions

- OFC (sc_module_name name, Position *position, Port *port)
- void doOFC ()
- OFC (sc_module_name name, Position *position, Port *port)
- void doOFC ()
- OFC (sc_module_name name, Position *position, Port *port)
- void doOFC ()

Public Attributes

- sc_in< bool > clk
- sc_in< Flit > out_data
- sc_in< bool > rok
- sc_fifo_out< Flit > dout
- sc_out< bool > x_rd
- Position * position
- Port * port
- sc_in< bool > gnt_and_req
- sc_in< bool > outputs_free
- sc_out< bool > free
- bool onMulticast

5.10.1 Detailed Description

Definition at line 8 of file OFC.h.

5.10.2 Member Typedef Documentation

5.10.2.1 typedef OFC OFC::SC_CURRENT_USER_MODULE

Definition at line 22 of file OFC.h.

5.10.2.2 typedef OFC OFC::SC_CURRENT_USER_MODULE

Definition at line 27 of file OFC.h.

5.10.2.3 typedef OFC OFC::SC_CURRENT_USER_MODULE

Definition at line 27 of file OFC.h.

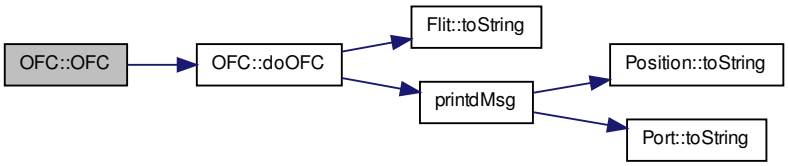
5.10.3 Constructor & Destructor Documentation

5.10.3.1 OFC::OFC (sc_module_name *name*, Position * *position*, Port * *port*)

Definition at line 3 of file OFC.cpp.

References clk, doOFC(), port, and position.

Here is the call graph for this function:



5.10.3.2 OFC::OFC (sc_module_name *name*, Position * *position*, Port * *port*)

5.10.3.3 OFC::OFC (sc_module_name *name*, Position * *position*, Port * *port*)

5.10.4 Member Function Documentation

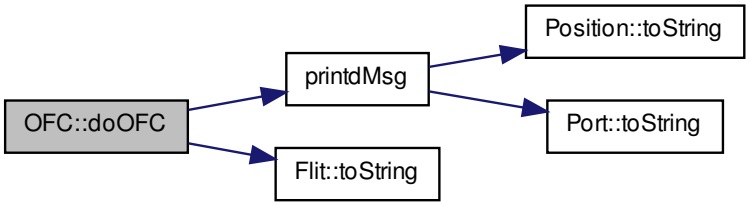
5.10.4.1 void OFC::doOFC ( )

Definition at line 11 of file OFC.cpp.

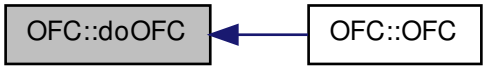
References dMsg, dout, out_data, position, printdMsg(), rok, Flit::toString(), and x_rd.

Referenced by OFC().

Here is the call graph for this function:



Here is the caller graph for this function:



5.10.4.2 void OFC::doOFC ( )

5.10.4.3 void OFC::doOFC ( )

5.10.5 Member Data Documentation

5.10.5.1 sc_in< bool > OFC::clk

Definition at line 10 of file OFC.h.

Referenced by OFC(), and Router::Router().

5.10.5.2 sc_fifo_out< Flit > OFC::dout

Definition at line 15 of file OFC.h.

Referenced by doOFC(), and Router::Router().

5.10.5.3 sc_out< bool > OFC::free

Definition at line 19 of file OFC.h.

5.10.5.4 sc_in< bool > OFC::gnt_and_req

Definition at line 14 of file OFC.h.

5.10.5.5 bool OFC::onMulticast

Definition at line 21 of file OFC.h.

5.10.5.6 sc_in< Flit > OFC::out_data

Definition at line 12 of file OFC.h.

Referenced by doOFC().

5.10.5.7 sc_in< bool > OFC::outputs_free

Definition at line 15 of file OFC.h.

5.10.5.8 Port * OFC::port

Definition at line 20 of file OFC.h.

Referenced by OFC().

5.10.5.9 Position * OFC::position

Definition at line 19 of file OFC.h.

Referenced by doOFC(), and OFC().

5.10.5.10 sc_in< bool > OFC::rok

Definition at line 13 of file OFC.h.

Referenced by doOFC().

5.10.5.11 sc_out< bool > OFC::x_rd

Definition at line 16 of file OFC.h.

Referenced by doOFC(), and Router::Router().

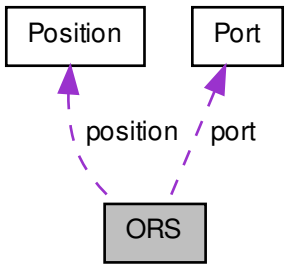
The documentation for this struct was generated from the following files:

- socin/OFC.h
- socmein/OFC.h
- socmein_cluster/OFC.h
- socin/OFC.cpp
- socmein/OFC.cpp
- socmein_cluster/OFC.cpp

5.11 ORS Struct Reference

`#include <ORS.h>`

Collaboration diagram for ORS:



Public Types

- typedef ORS SC_CURRENT_USER_MODULE

Public Member Functions

- ORS (sc_module_name name, Position *position, Port *port)
- void doORS ()

Public Attributes

- sc_in< bool > x_gnt [NUMBER_OF_DIRECTIONS]
- sc_in< bool > x_rok [NUMBER_OF_DIRECTIONS]
- sc_out< bool > rok
- Position * position
- Port * port

5.11.1 Detailed Description

Definition at line 8 of file ORS.h.

5.11.2 Member Typedef Documentation

5.11.2.1 typedef ORS ORS::SC_CURRENT_USER_MODULE

Definition at line 19 of file ORS.h.

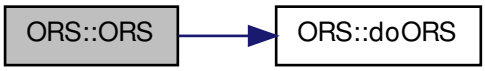
5.11.3 Constructor & Destructor Documentation

5.11.3.1 ORS::ORS (sc_module_name *name*, Position * *position*, Port * *port*)

Definition at line 3 of file ORS.cpp.

References doORS(), NUMBER_OF_DIRECTIONS, port, position, x_gnt, and x_rok.

Here is the call graph for this function:



5.11.4 Member Function Documentation

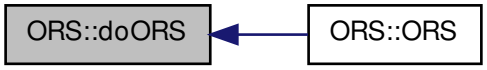
5.11.4.1 void ORS::doORS ( )

Definition at line 14 of file ORS.cpp.

References NUMBER_OF_DIRECTIONS, rok, x_gnt, and x_rok.

Referenced by ORS().

Here is the caller graph for this function:



5.11.5 Member Data Documentation

5.11.5.1 Port* ORS::port

Definition at line 17 of file ORS.h.

Referenced by ORS().

5.11.5.2 Position* ORS::position

Definition at line 16 of file ORS.h.

Referenced by ORS().

5.11.5.3 sc_out<bool> ORS::rok

Definition at line 13 of file ORS.h.

Referenced by doORS().

5.11.5.4 sc_in<bool> ORS::x_gnt[NUMBER_OF_DIRECTIONS]

Definition at line 10 of file ORS.h.

Referenced by doORS(), ORS(), and Router::Router().

5.11.5.5 sc_in<bool> ORS::x_rok[NUMBER_OF_DIRECTIONS]

Definition at line 11 of file ORS.h.

Referenced by doORS(), and ORS().

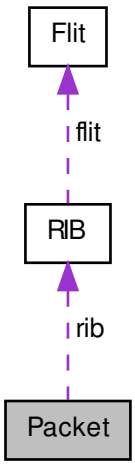
The documentation for this struct was generated from the following files:

- common/ORS.h
- common/ORS.cpp

5.12 Packet Class Reference

`#include <TrafficGenerator.h>`

Collaboration diagram for Packet:



Public Member Functions

- Packet (unsigned int sX, unsigned int sY, unsigned int dX, unsigned int dY, unsigned int payloadSize, bool multicast)
- Packet (unsigned int sX, unsigned int sY, unsigned int dX, unsigned int dY, string payload, bool multicast)

Public Attributes

- unsigned int sX
- unsigned int sY
- unsigned int dX
- unsigned int dY
- unsigned int payloadSize
- string payload
- RIB * rib
- bool meaningfulPayload

5.12.1 Detailed Description

Definition at line 12 of file TrafficGenerator.h.

5.12.2 Constructor & Destructor Documentation

5.12.2.1 Packet::Packet (unsigned int *sX,* unsigned int *sY,* unsigned int *dX,* unsigned int *dY,* unsigned int *payloadSize,* bool *multicast*)

Definition at line 122 of file TrafficGenerator.cpp.

References meaningfulPayload, and rib.

5.12.2.2 Packet::Packet (unsigned int *sX,* unsigned int *sY,* unsigned int *dX,* unsigned int *dY,* string *payload,* bool *multicast*)

Definition at line 129 of file TrafficGenerator.cpp.

References meaningfulPayload, payloadSize, and rib.

5.12.3 Member Data Documentation

5.12.3.1 unsigned int Packet::dX

Definition at line 16 of file TrafficGenerator.h.

5.12.3.2 unsigned int Packet::dY

Definition at line 17 of file TrafficGenerator.h.

5.12.3.3 bool Packet::meaningfulPayload

Definition at line 21 of file TrafficGenerator.h.

Referenced by Packet(), and TrafficGenerator::sendPacket().

5.12.3.4 string Packet::payload

Definition at line 19 of file TrafficGenerator.h.

Referenced by TrafficGenerator::sendPacket().

5.12.3.5 unsigned int Packet::payloadSize

Definition at line 18 of file TrafficGenerator.h.

Referenced by Packet(), and TrafficGenerator::sendPacket().

5.12.3.6 RIB* Packet::rib

Definition at line 20 of file TrafficGenerator.h.

Referenced by Packet(), and TrafficGenerator::sendPacket().

5.12.3.7 unsigned int Packet::sX

Definition at line 14 of file TrafficGenerator.h.

Referenced by TrafficGenerator::sendPacket().

5.12.3.8 unsigned int Packet::sY

Definition at line 15 of file TrafficGenerator.h.

Referenced by TrafficGenerator::sendPacket().

The documentation for this class was generated from the following files:

- common/TrafficGenerator.h
- common/TrafficGenerator.cpp

5.13 Port Class Reference

`#include <Port.h>`

Public Member Functions

- Port (en_directions port)
- Port ()
- string toString ()

Public Attributes

- en_directions port
- string portString

5.13.1 Detailed Description

Definition at line 6 of file Port.h.

5.13.2 Constructor & Destructor Documentation

5.13.2.1 Port::Port (en_directions *port*)

Definition at line 3 of file Port.cpp.

References directionString, port, and portString.

5.13.2.2 Port::Port ( )

Definition at line 9 of file Port.cpp.

5.13.3 Member Function Documentation

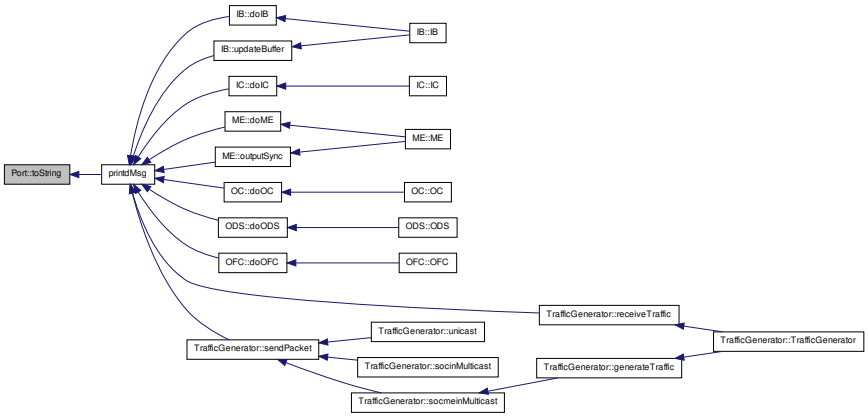
5.13.3.1 string Port::toString (void)

Definition at line 13 of file Port.cpp.

References portString.

Referenced by printdMsg().

Here is the caller graph for this function:



5.13.4 Member Data Documentation

5.13.4.1 en_directions Port::port

Definition at line 8 of file Port.h.

Referenced by Port(), and portInit().

5.13.4.2 string Port::portString

Definition at line 9 of file Port.h.

Referenced by Port(), portInit(), and toString().

The documentation for this class was generated from the following files:

- common/Port.h
- common/Port.cpp

5.14 Position Class Reference

`#include <Position.h>`

Public Member Functions

- Position (unsigned int x, unsigned int y)
- string toString ()
- string toPrefixedString (string prefix)

Public Attributes

- unsigned int x
- unsigned int y

5.14.1 Detailed Description

Definition at line 6 of file Position.h.

5.14.2 Constructor & Destructor Documentation

5.14.2.1 Position::Position (unsigned int *x*, unsigned int *y*)

Definition at line 3 of file Position.cpp.

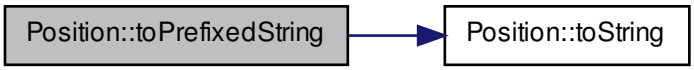
5.14.3 Member Function Documentation

5.14.3.1 string Position::toPrefixedString (string *prefix*)

Definition at line 15 of file Position.cpp.

References toString().

Here is the call graph for this function:



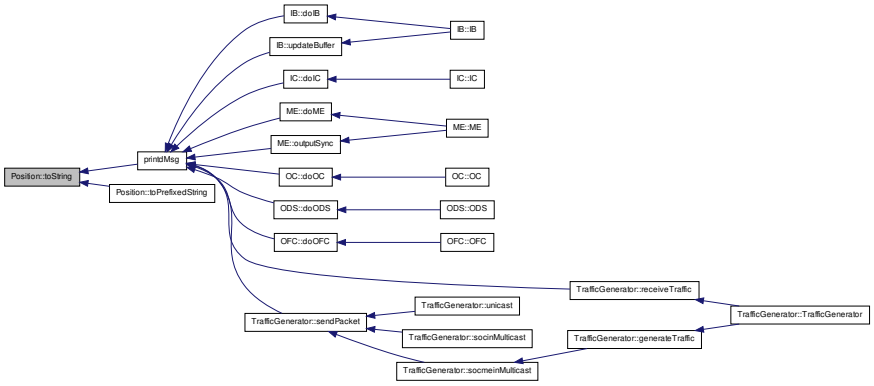
5.14.3.2 string Position::toString (void)

Definition at line 8 of file Position.cpp.

References x, and y.

Referenced by printdMsg(), and toPrefixedString().

Here is the caller graph for this function:



5.14.4 Member Data Documentation

5.14.4.1 unsigned int Position::x

Definition at line 8 of file Position.h.

Referenced by TrafficGenerator::sendPacket(), TrafficGenerator::socmeinMulticast(), and toString().

5.14.4.2 unsigned int Position::y

Definition at line 9 of file Position.h.

Referenced by TrafficGenerator::sendPacket(), TrafficGenerator::socmeinMulticast(), and toString().

The documentation for this class was generated from the following files:

- common/Position.h
- common/Position.cpp

5.15 RIB Class Reference

`#include <RIB.h>`

Collaboration diagram for RIB:

Public Member Functions

- RIB (Flit flit)
- RIB (unsigned int sX, unsigned int sY, unsigned int dX, unsigned int dY, bool multicast)
- Flit toFlit ()
- string toString ()

Public Attributes

- Flit flit
- bool xDir

- bool yDir
- sc_bv<(N_SIZE-2)/2 > xMod
- sc_bv<(N_SIZE-2)/2 > yMod

5.15.1 Detailed Description

Definition at line 8 of file RIB.h.

5.15.2 Constructor & Destructor Documentation

5.15.2.1 RIB::RIB (Flit *flit*)

Definition at line 3 of file RIB.cpp.

References Flit::data, xDir, xMod, yDir, and yMod.

5.15.2.2 RIB::RIB (unsigned int *sX,* unsigned int *sY,* unsigned int *dX,* unsigned int *dY,* bool *multicast*)

Definition at line 13 of file RIB.cpp.

References Flit::bop, EAST, Flit::eop, flit, NORTH, SOUTH, WEST, xDir, xMod, yDir, and yMod.

5.15.3 Member Function Documentation

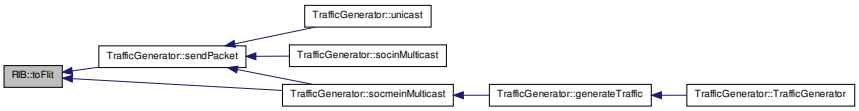
5.15.3.1 Flit RIB::toFlit ( )

Definition at line 63 of file RIB.cpp.

References Flit::data, flit, xDir, xMod, yDir, and yMod.

Referenced by TrafficGenerator::sendPacket(), and TrafficGenerator::socmeinMulticast().

Here is the caller graph for this function:



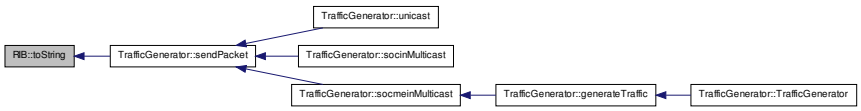
5.15.3.2 string RIB::toString (void)

Definition at line 46 of file RIB.cpp.

References EAST, NORTH, xDir, xMod, yDir, and yMod.

Referenced by TrafficGenerator::sendPacket().

Here is the caller graph for this function:



5.15.4 Member Data Documentation

5.15.4.1 Flit RIB::flit

Definition at line 10 of file RIB.h.

Referenced by RIB(), TrafficGenerator::sendPacket(), TrafficGenerator::socmeinMulticast(), and toFlit().

5.15.4.2 bool RIB::xDir

Definition at line 11 of file RIB.h.

Referenced by RIB(), toFlit(), and toString().

5.15.4.3 sc_bv<(N_SIZE-2)/2> RIB::xMod

Definition at line 13 of file RIB.h.

Referenced by RIB(), toFlit(), and toString().

5.15.4.4 bool RIB::yDir

Definition at line 12 of file RIB.h.

Referenced by RIB(), toFlit(), and toString().

5.15.4.5 sc_bv<(N_SIZE-2)/2> RIB::yMod

Definition at line 14 of file RIB.h.

Referenced by RIB(), toFlit(), and toString().

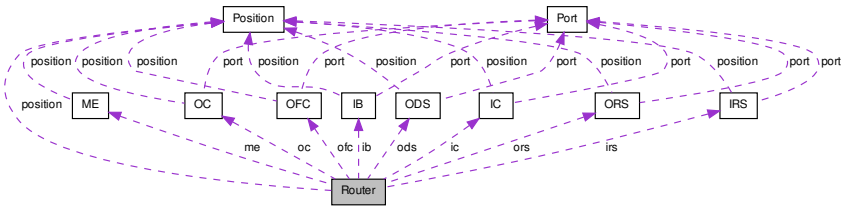
The documentation for this class was generated from the following files:

- common/RIB.h
- common/RIB.cpp

5.16 Router Struct Reference

`#include <Router.h>`

Collaboration diagram for Router:



Public Types

- typedef Router SC_CURRENT_USER_MODULE
- typedef Router SC_CURRENT_USER_MODULE
- typedef Router SC_CURRENT_USER_MODULE

Public Member Functions

- Router (sc_module_name name, unsigned int x, unsigned int y)
- void checkForActivity ()
- void connect (sc_in< Flit > &a, sc_out< Flit > &b)
- void connect (sc_in< bool > &a, sc_out< bool > &b)
- void orion_router_area (RouterReport *routerReport)
- void orion_router_power (RouterReport *routerReport, unsigned int cycleCount)
- Router (sc_module_name name, unsigned int x, unsigned int y)
- void checkForActivity ()
- void connect (sc_in< Flit > &a, sc_out< Flit > &b)
- void connect (sc_in< bool > &a, sc_out< bool > &b)
- void orion_router_area (RouterReport *routerReport)
- void orion_router_power (RouterReport *routerReport, unsigned int cycleCount)
- Router (sc_module_name name, unsigned int x, unsigned int y)
- void checkForActivity ()
- void connect (sc_in< Flit > &a, sc_out< Flit > &b)

- void connect (sc_in< bool > &a, sc_out< bool > &b)
- void orion_router_area (RouterReport *routerReport)
- void orion_router_power (RouterReport *routerReport, unsigned int cycleCount)

Public Attributes

- sc_in< bool > clk
- sc_fifo_in< Flit > in [NUMBER_OF_DIRECTIONS]
- sc_fifo_out< Flit > out [NUMBER_OF_DIRECTIONS]
- IB * ib [NUMBER_OF_DIRECTIONS]
- IC * ic [NUMBER_OF_DIRECTIONS]
- ODS * ods [NUMBER_OF_DIRECTIONS]
- ORS * ors [NUMBER_OF_DIRECTIONS]
- OC * oc [NUMBER_OF_DIRECTIONS]
- IRS * irs [NUMBER_OF_DIRECTIONS]
- OFC * ofc [NUMBER_OF_DIRECTIONS]
- Position * position
- list< sc_fifo< Flit > * > fifoSignal
- list< sc_signal< bool > * > boolSignal
- list< sc_signal< Flit > * > flitSignal
- double totalPower
- unsigned int activeCount
- ME * me
- sc_in< N_SIZE > nID [NUMBER_OF_DIRECTIONS]
- sc_bv< N_SIZE > ID

5.16.1 Detailed Description

Definition at line 37 of file Router.h.

5.16.2 Member Typedef Documentation

5.16.2.1 typedef Router Router::SC_CURRENT_USER_MODULE

Definition at line 63 of file Router.h.

5.16.2.2 typedef Router Router::SC_CURRENT_USER_MODULE

Definition at line 65 of file Router.h.

5.16.2.3 typedef Router Router::SC_CURRENT_USER_MODULE

Definition at line 68 of file Router.h.

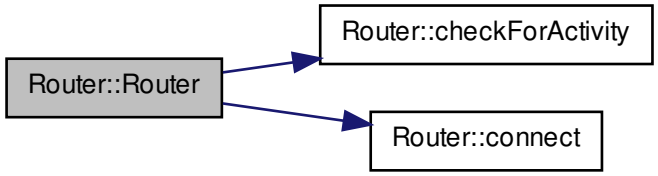
5.16.3 Constructor & Destructor Documentation

5.16.3.1 Router::Router (sc_module_name *name,* unsigned int *x,* unsigned int *y*)

Definition at line 3 of file Router.cpp.

References activeCount, boolSignal, checkForActivity(), OFC::clk, IB::clk, clk, connect(), IB::din, OFC::dout, flitSignal, ib, ic, in, irs, NUMBER_OF_DIRECTIONS, oc, ods, ofc, ors, out, portArray, position, IB::rok, totalPower, IC::x_dout, ORS::x_gnt, IRS::x_gnt, ODS::x_gnt, OC::x_gnt, OC::x_rd, OFC::x_rd, and IC::x_rok.

Here is the call graph for this function:



5.16.3.2 Router::Router (sc_module_name *name,* unsigned int *x,* unsigned int *y*)

5.16.3.3 Router::Router (sc_module_name *name,* unsigned int *x,* unsigned int *y*)

5.16.4 Member Function Documentation

5.16.4.1 void Router::checkForActivity ( )

Definition at line 115 of file Router.cpp.

References IC::active, activeCount, ic, and NUMBER_OF_DIRECTIONS.

Referenced by Router().

Here is the caller graph for this function:

5.16.4.2 void Router::checkForActivity ( )

5.16.4.3 void Router::checkForActivity ( )

5.16.4.4 void Router::connect (sc_in< Flit > & *a*, sc_out< Flit > & *b*)

Definition at line 80 of file Router.cpp.

References flitSignal.

Referenced by Router().

Here is the caller graph for this function:

5.16.4.5 void Router::connect (sc_in< bool > & *a*, sc_out< bool > & *b*)

Definition at line 88 of file Router.cpp.

References boolSignal.

5.16.4.6 void Router::connect (sc_in< Flit > & *a*, sc_out< Flit > & *b*)

5.16.4.7 void Router::connect (sc_in< bool > & *a*, sc_out< bool > & *b*)

5.16.4.8 void Router::connect (sc_in< Flit > & *a,* sc_out< Flit > & *b*)

5.16.4.9 void Router::connect (sc_in< bool > & *a,* sc_out< bool > & *b*)

5.16.4.10 void Router::orion_router_area (RouterReport * *routerReport*)

Definition at line 96 of file Router.cpp.

References RouterReport::area.

5.16.4.11 void Router::orion_router_area (RouterReport * *routerReport*)

5.16.4.12 void Router::orion_router_area (RouterReport * *routerReport*)

5.16.4.13 void Router::orion_router_power (RouterReport * *routerReport,* unsigned int *cycleCount*)

Definition at line 104 of file Router.cpp.

References activeCount, RouterReport::power, and totalPower.

5.16.4.14 void Router::orion_router_power (RouterReport * *routerReport,* unsigned int *cycleCount*)

5.16.4.15 void Router::orion_router_power (RouterReport * *routerReport,* unsigned int *cycleCount*)

5.16.5 Member Data Documentation

5.16.5.1 unsigned int Router::activeCount

Definition at line 61 of file Router.h.

Referenced by checkForActivity(), orion_router_power(), and Router().

5.16.5.2 list< sc_signal< bool > * > Router::boolSignal

Definition at line 55 of file Router.h.

Referenced by connect(), and Router().

5.16.5.3 sc_in< bool > Router::clk

Definition at line 39 of file Router.h.

Referenced by Router().

5.16.5.4 list< sc_fifo< Flit > * > Router::fifoSignal

Definition at line 54 of file Router.h.

5.16.5.5 list< sc_signal< Flit > * > Router::flitSignal

Definition at line 56 of file Router.h.

Referenced by connect(), and Router().

5.16.5.6 IB * Router::ib

Definition at line 44 of file Router.h.

Referenced by Router().

5.16.5.7 IC * Router::ic

Definition at line 45 of file Router.h.

Referenced by checkForActivity(), and Router().

5.16.5.8 sc_bv<N_SIZE> Router::ID

Definition at line 63 of file Router.h.

5.16.5.9 sc_fifo_in< Flit > Router::in

Definition at line 41 of file Router.h.

Referenced by Router().

5.16.5.10 IRS * Router::irs

Definition at line 49 of file Router.h.

Referenced by Router().

5.16.5.11 ME * Router::me

Definition at line 52 of file Router.h.

5.16.5.12 sc_in<N_SIZE> Router::nID[NUMBER_OF_DIRECTIONS]

Definition at line 44 of file Router.h.

5.16.5.13 OC * Router::oc

Definition at line 48 of file Router.h.

Referenced by Router().

5.16.5.14 ODS * Router::ods

Definition at line 46 of file Router.h.

Referenced by Router().

5.16.5.15 OFC * Router::ofc

Definition at line 50 of file Router.h.

Referenced by Router().

5.16.5.16 ORS * Router::ors

Definition at line 47 of file Router.h.

Referenced by Router().

5.16.5.17 sc_fifo_out< Flit > Router::out

Definition at line 42 of file Router.h.

Referenced by Router().

5.16.5.18 Position * Router::position

Definition at line 52 of file Router.h.

Referenced by Router().

5.16.5.19 double Router::totalPower

Definition at line 59 of file Router.h.

Referenced by orion_router_power(), and Router().

The documentation for this struct was generated from the following files:

- socin/Router.h
- socmein/Router.h
- socmein_cluster/Router.h
- socin/Router.cpp
- socmein/Router.cpp

- socmein_cluster/Router.cpp

5.17 RouterMatrix Class Reference

`#include <RouterMatrix.h>`

Public Member Functions

- RouterMatrix (unsigned int x_size, unsigned int y_size)
- ~RouterMatrix ()
- void print (void)
- void draw (int x, int y)
- void setRouter (unsigned int x, unsigned int y, Router *router)
- Router * getRouter (unsigned int x, unsigned int y)
- unsigned int x_size (void)
- unsigned int y_size (void)
- unsigned int routerCounter (void)

Public Attributes

- map< unsigned int, map < unsigned int, Router * > > XY
- map< unsigned int, map < unsigned int, Link * > > x_links [2]
- map< unsigned int, map < unsigned int, Link * > > y_links [2]
- map< unsigned int, map < unsigned int, Link * > > local_links [2]

Private Attributes

- unsigned int _x_size
- unsigned int _y_size
- unsigned int _routerCounter

5.17.1 Detailed Description

Definition at line 26 of file RouterMatrix.h.

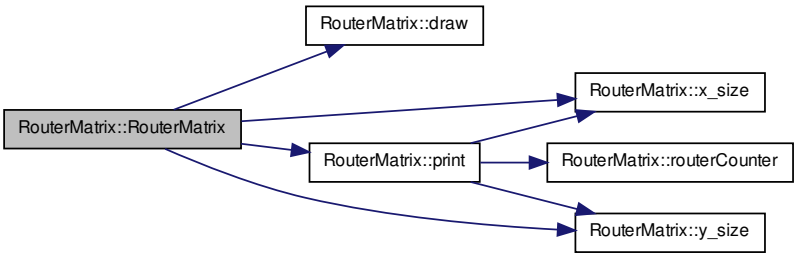
5.17.2 Constructor & Destructor Documentation

5.17.2.1 RouterMatrix::RouterMatrix (unsigned int *x_size,* unsigned int *y_size*)

Definition at line 5 of file RouterMatrix.cpp.

References draw(), E, E_TO_W, FROM_L, L, local_links, N, N_TO_S, print(), S, S_TO_N, TO_L, W, W_TO_E, x_links, x_size(), XY, y_links, and y_size().

Here is the call graph for this function:



5.17.2.2 RouterMatrix::~RouterMatrix ( )

Definition at line 65 of file RouterMatrix.cpp.

References _x_size, _y_size, LinkReport::area, RouterReport::area, E_TO_W, FROM_L, local_links, N_TO_S, LinkReport::power, RouterReport::power, S_TO_N, TO_L, W_TO_E, x_links, XY, and y_links.

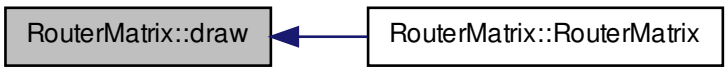
5.17.3 Member Function Documentation

5.17.3.1 void RouterMatrix::draw (int *x*, int *y*)

Definition at line 160 of file RouterMatrix.cpp.

Referenced by RouterMatrix().

Here is the caller graph for this function:



5.17.3.2 Router* RouterMatrix::getRouter (unsigned int *x*, unsigned int *y*)

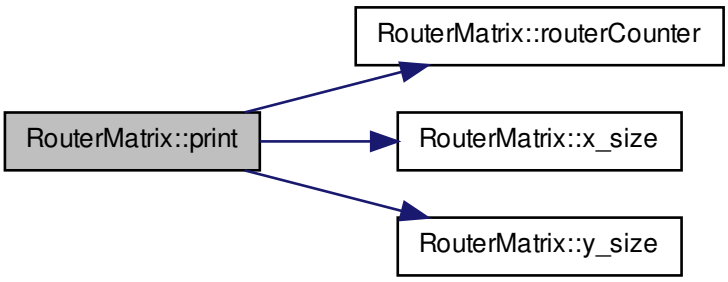
5.17.3.3 void RouterMatrix::print (void)

Definition at line 155 of file RouterMatrix.cpp.

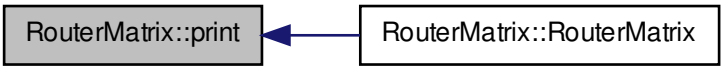
References routerCounter(), x_size(), and y_size().

Referenced by RouterMatrix().

Here is the call graph for this function:



Here is the caller graph for this function:



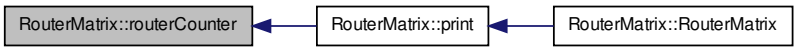
5.17.3.4 unsigned int RouterMatrix::routerCounter (void)

Definition at line 178 of file RouterMatrix.cpp.

References _routerCounter.

Referenced by print().

Here is the caller graph for this function:



5.17.3.5 void RouterMatrix::setRouter (unsigned int *x*, unsigned int *y*, Router * *router*)

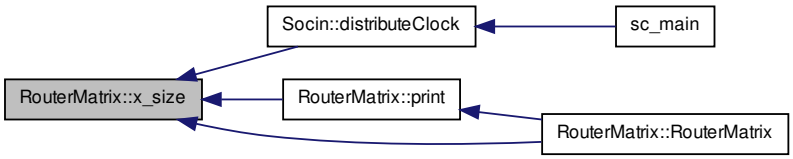
5.17.3.6 unsigned int RouterMatrix::x_size (void)

Definition at line 176 of file RouterMatrix.cpp.

References _x_size.

Referenced by Socin::distributeClock(), print(), and RouterMatrix().

Here is the caller graph for this function:



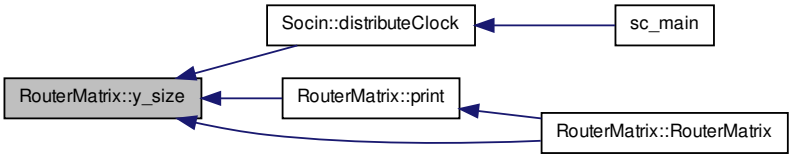
5.17.3.7 unsigned int RouterMatrix::y_size (void)

Definition at line 177 of file RouterMatrix.cpp.

References _y_size.

Referenced by Socin::distributeClock(), print(), and RouterMatrix().

Here is the caller graph for this function:



5.17.4 Member Data Documentation

5.17.4.1 unsigned int RouterMatrix::_routerCounter [private]

Definition at line 29 of file RouterMatrix.h.

Referenced by routerCounter().

5.17.4.2 unsigned int RouterMatrix::_x_size [private]

Definition at line 27 of file RouterMatrix.h.

Referenced by x_size(), and ~RouterMatrix().

5.17.4.3 unsigned int RouterMatrix::_y_size [private]

Definition at line 28 of file RouterMatrix.h.

Referenced by y_size(), and ~RouterMatrix().

5.17.4.4 map<unsigned int, map<unsigned int, Link* > > RouterMatrix::local_links[2]

Definition at line 35 of file RouterMatrix.h.

Referenced by RouterMatrix(), sc_main(), and ~RouterMatrix().

5.17.4.5 map<unsigned int, map<unsigned int, Link* > > RouterMatrix::x_links[2]

Definition at line 33 of file RouterMatrix.h.

Referenced by RouterMatrix(), and ~RouterMatrix().

5.17.4.6 map<unsigned int, map<unsigned int, Router* > > RouterMatrix::XY

Definition at line 32 of file RouterMatrix.h.

Referenced by Socin::distributeClock(), RouterMatrix(), and ~RouterMatrix().

5.17.4.7 map<unsigned int, map<unsigned int, Link* > > RouterMatrix::y_links[2]

Definition at line 34 of file RouterMatrix.h.

Referenced by RouterMatrix(), and ~RouterMatrix().

The documentation for this class was generated from the following files:

- common/RouterMatrix.h
- common/RouterMatrix.cpp

5.18 RouterReport Class Reference

`#include <Router.h>`

Public Member Functions

- RouterReport ()
- RouterReport ()
- RouterReport ()

Public Attributes

- double power
- double area

5.18.1 Detailed Description

Definition at line 25 of file Router.h.

5.18.2 Constructor & Destructor Documentation

5.18.2.1 RouterReport::RouterReport ( ) [inline]

Definition at line 31 of file Router.h.

5.18.2.2 RouterReport::RouterReport ( ) [inline]

Definition at line 32 of file Router.h.

5.18.2.3 RouterReport::RouterReport ( ) [inline]

Definition at line 32 of file Router.h.

5.18.3 Member Data Documentation

5.18.3.1 double RouterReport::area

Definition at line 29 of file Router.h.

Referenced by Router::orion_router_area(), and RouterMatrix::∼RouterMatrix().

5.18.3.2 double RouterReport::power

Definition at line 28 of file Router.h.

Referenced by Router::orion_router_power(), and RouterMatrix::∼RouterMatrix().

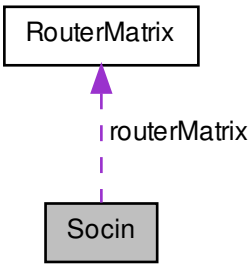
The documentation for this class was generated from the following files:

- socin/Router.h
- socmein/Router.h
- socmein_cluster/Router.h

5.19 Socin Struct Reference

`#include <Socin.h>`

Collaboration diagram for Socin:



Public Types

- typedef Socin SC_CURRENT_USER_MODULE

Public Member Functions

- Socin (sc_module_name name, unsigned int x_size, unsigned int y_size)
- ~Socin ()
- void print (void)
- void distributeClock ()

Public Attributes

- sc_in< bool > clk
- RouterMatrix * routerMatrix

5.19.1 Detailed Description

Definition at line 14 of file Socin.h.

5.19.2 Member Typedef Documentation

5.19.2.1 typedef Socin Socin::SC_CURRENT_USER_MODULE

Definition at line 20 of file Socin.h.

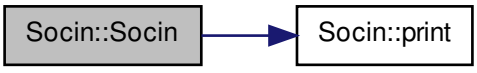
5.19.3 Constructor & Destructor Documentation

5.19.3.1 Socin::Socin (sc_module_name *name*, unsigned int *x_size*, unsigned int *y_size*)

Definition at line 3 of file Socin.cpp.

References print(), and routerMatrix.

Here is the call graph for this function:



5.19.3.2 Socin::~Socin ( )

Definition at line 12 of file Socin.cpp.

References routerMatrix.

5.19.4 Member Function Documentation

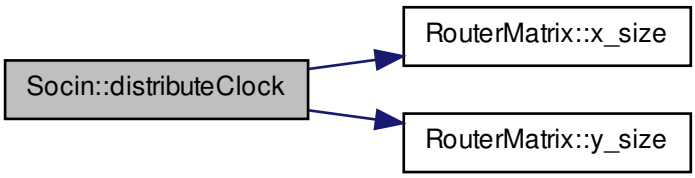
5.19.4.1 void Socin::distributeClock ( )

Definition at line 17 of file Socin.cpp.

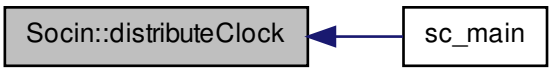
References clk, routerMatrix, RouterMatrix::x_size(), RouterMatrix::XY, and RouterMatrix::y_size().

Referenced by sc_main().

Here is the call graph for this function:



Here is the caller graph for this function:



5.19.4.2 void Socin::print (void)

Definition at line 26 of file Socin.cpp.

Referenced by Socin().

Here is the caller graph for this function:

5.19.5 Member Data Documentation

5.19.5.1 sc_in<bool> Socin::clk

Definition at line 16 of file Socin.h.

Referenced by distributeClock(), and sc_main().

5.19.5.2 RouterMatrix* Socin::routerMatrix

Definition at line 17 of file Socin.h.

Referenced by distributeClock(), sc_main(), Socin(), and ~Socin().

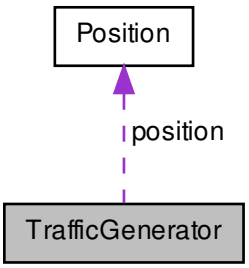
The documentation for this struct was generated from the following files:

- common/Socin.h
- common/Socin.cpp

5.20 TrafficGenerator Struct Reference

`#include <TrafficGenerator.h>`

Collaboration diagram for TrafficGenerator:



Public Types

- typedef TrafficGenerator SC_CURRENT_USER_MODULE

Public Member Functions

- TrafficGenerator (sc_module_name name, unsigned int x, unsigned int y)
- void generateTraffic ()
- void receiveTraffic ()
- void sendPacket (Packet *packet)
- void unicast (unsigned int size)
- void socinMulticast (unsigned int size)
- void socmeinMulticast (unsigned int size)

Public Attributes

- sc_in< bool > clk
- sc_fifo_in< Flit > in
- sc_fifo_out< Flit > out
- Position * position

5.20.1 Detailed Description

Definition at line 27 of file TrafficGenerator.h.

5.20.2 Member Typedef Documentation

5.20.2.1 typedef TrafficGenerator TrafficGenerator::SC_CURRENT_USER_MODULE

Definition at line 37 of file TrafficGenerator.h.

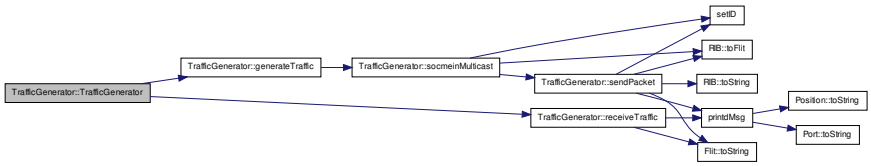
5.20.3 Constructor & Destructor Documentation

5.20.3.1 TrafficGenerator::TrafficGenerator (sc_module_name *name*, unsigned int *x*, unsigned int *y*)

Definition at line 3 of file TrafficGenerator.cpp.

References clk, generateTraffic(), position, and receiveTraffic().

Here is the call graph for this function:



5.20.4 Member Function Documentation

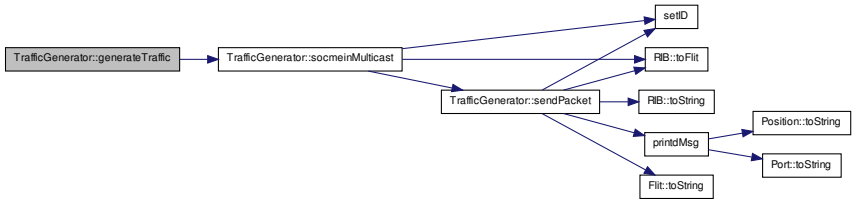
5.20.4.1 void TrafficGenerator::generateTraffic ( )

Definition at line 97 of file TrafficGenerator.cpp.

References socmeinMulticast().

Referenced by TrafficGenerator().

Here is the call graph for this function:



Here is the caller graph for this function:

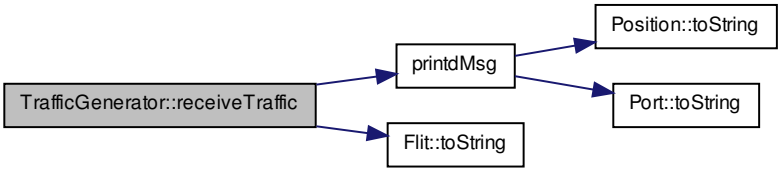
5.20.4.2 void TrafficGenerator::receiveTraffic ( )

Definition at line 104 of file TrafficGenerator.cpp.

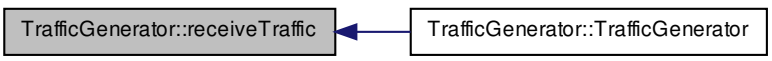
References dMsg, Flit::ID, in, position, printdMsg(), and Flit::toString().

Referenced by TrafficGenerator().

Here is the call graph for this function:



Here is the caller graph for this function:



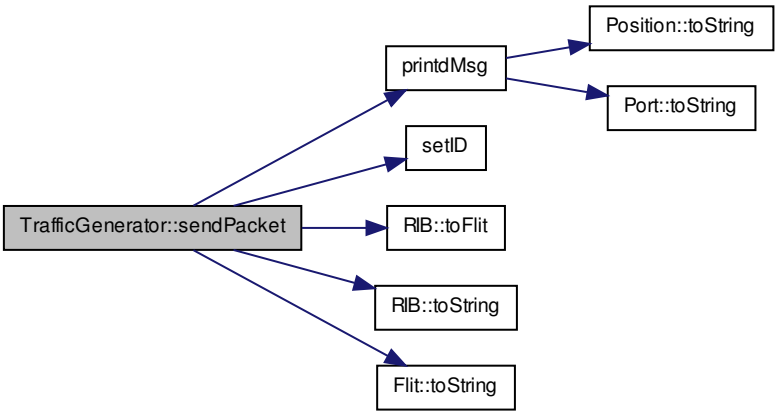
5.20.4.3 void TrafficGenerator::sendPacket (Packet * *packet*)

Definition at line 137 of file TrafficGenerator.cpp.

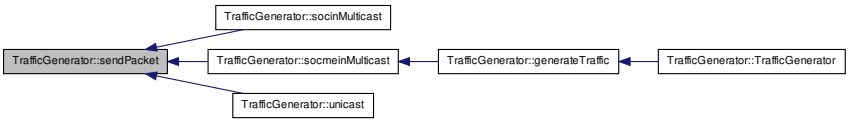
References Flit::bop, Flit::data, dMsg, Flit::eop, RIB::flit, Flit::ID, Packet::meaningfulPayload, out, Packet::payload, Packet::payloadSize, position, printdMsg(), Packet::rib, setID(), Packet::sX, Packet::sY, RIB::toFlit(), RIB::toString(), Flit::toString(), Position::x, and Position::y.

Referenced by socinMulticast(), socmeinMulticast(), and unicast().

Here is the call graph for this function:



Here is the caller graph for this function:

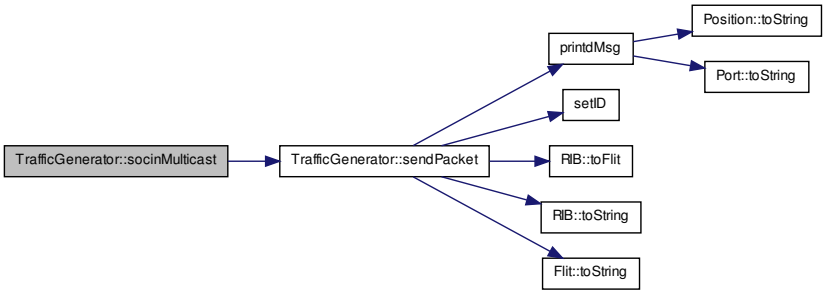


5.20.4.4 void TrafficGenerator::socinMulticast (unsigned int *size*)

Definition at line 22 of file TrafficGenerator.cpp.

References sendPacket().

Here is the call graph for this function:



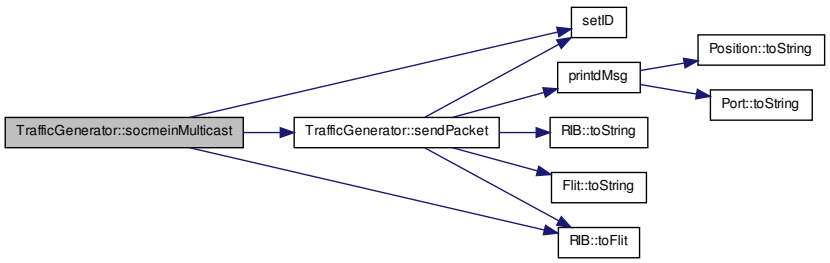
5.20.4.5 void TrafficGenerator::socmeinMulticast (unsigned int *size*)

Definition at line 57 of file TrafficGenerator.cpp.

References RIB::flit, Flit::ID, out, position, sendPacket(), setID(), RIB::toFlit(), Position::x, and Position::y.

Referenced by generateTraffic().

Here is the call graph for this function:



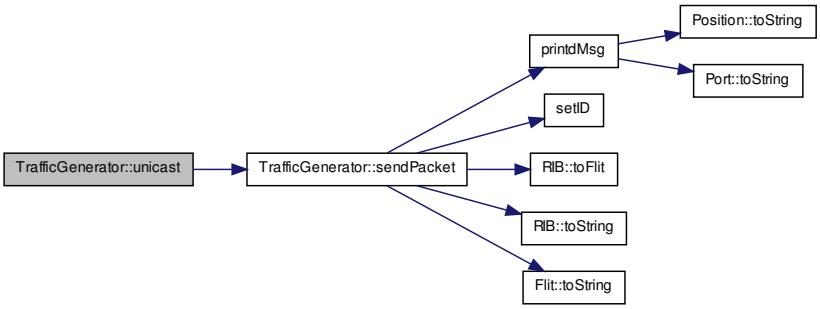
Here is the caller graph for this function:

5.20.4.6 void TrafficGenerator::unicast (unsigned int *size*)

Definition at line 11 of file TrafficGenerator.cpp.

References sendPacket().

Here is the call graph for this function:



5.20.5 Member Data Documentation

5.20.5.1 sc_in<bool> TrafficGenerator::clk

Definition at line 29 of file TrafficGenerator.h.

Referenced by TrafficGenerator().

5.20.5.2 sc_fifo_in<Flit> TrafficGenerator::in

Definition at line 31 of file TrafficGenerator.h.

Referenced by receiveTraffic(), and sc_main().

5.20.5.3 sc_fifo_out<Flit> TrafficGenerator::out

Definition at line 32 of file TrafficGenerator.h.

Referenced by sendPacket(), and socmeinMulticast().

5.20.5.4 Position* TrafficGenerator::position

Definition at line 34 of file TrafficGenerator.h.

Referenced by receiveTraffic(), sendPacket(), socmeinMulticast(), and TrafficGenerator().

The documentation for this struct was generated from the following files:

- common/TrafficGenerator.h
- common/TrafficGenerator.cpp

Chapter 6

File Documentation

6.1 common/config_2d.cpp File Reference

`#include "config.h"` Include dependency graph for config_2d.cpp:

Functions

- void printdMsg (string source, Position *position, Port *port)
- unsigned int setID ()
- void portInit ()

Variables

- string directionString [NUMBER_OF_DIRECTIONS]
- stringstream * dMsg
- Port portArray [NUMBER_OF_DIRECTIONS]

6.1.1 Function Documentation

6.1.1.1 void portInit ( )

Definition at line 37 of file config_2d.cpp.

References directionString, NUMBER_OF_DIRECTIONS, Port::port, and Port::portString.

Referenced by sc_main().

Here is the caller graph for this function:

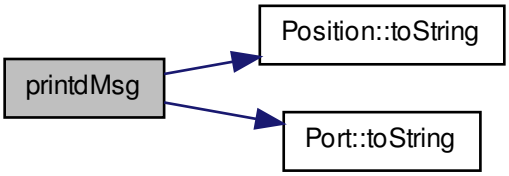
6.1.1.2 void printdMsg (string *source*, Position * *position*, Port * *port*)

Definition at line 12 of file config_2d.cpp.

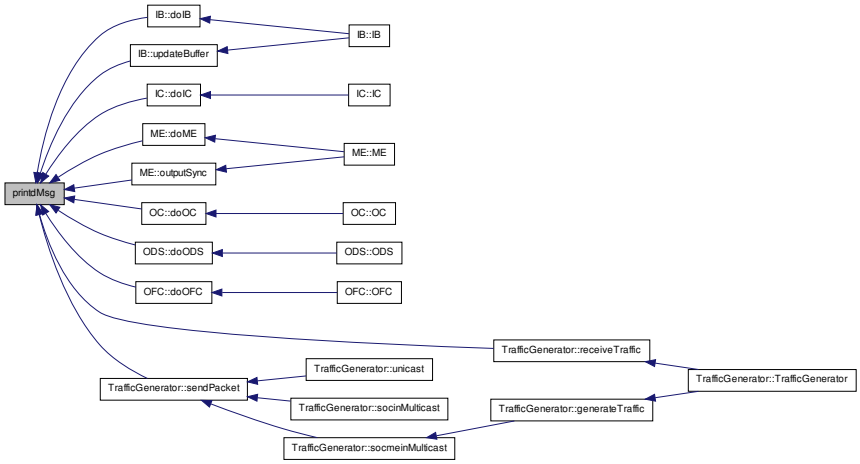
References dMsg, Position::toString(), and Port::toString().

Referenced by IB::doIB(), IC::doIC(), ME::doME(), OC::doOC(), ODS::doODS(), OFC::doOFC(), ME::outputSync(), TrafficGenerator::receiveTraffic(), TrafficGenerator::sendPacket(), and IB::updateBuffer().

Here is the call graph for this function:



Here is the caller graph for this function:

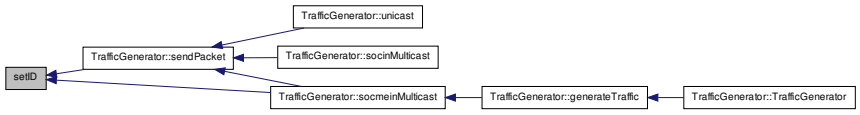


6.1.1.3 unsigned int setID ( )

Definition at line 26 of file config_2d.cpp.

Referenced by TrafficGenerator::sendPacket(), and TrafficGenerator::socmeinMulticast().

Here is the caller graph for this function:



6.1.2 Variable Documentation

6.1.2.1 string directionString[NUMBER_OF_DIRECTIONS]

Initial value:

```
 {"North",
                                                "East",
                                                "South",
                                                "West",
                                                "Local"
}
```

Definition at line 3 of file config_2d.cpp.

Referenced by ME::doME(), OC::doOC(), ODS::doODS(), Port::Port(), and portInit().

6.1.2.2 stringstream* dMsg

Definition at line 10 of file config_2d.cpp.

Referenced by IB::doIB(), IC::doIC(), ME::doME(), OC::doOC(), ODS::doODS(), OFC::doOFC(), ME::outputSync(), printdMsg(), TrafficGenerator::receiveTraffic(), TrafficGenerator::sendPacket(), and IB::updateBuffer().

6.1.2.3 Port portArray[NUMBER_OF_DIRECTIONS]

Definition at line 35 of file config_2d.cpp.

Referenced by Router::Router().

6.2 common/config_2d.h File Reference

Defines

- #define NUMBER_OF_DIRECTIONS 5

Enumerations

- enum en_directions { N, E, S, W, L }
- enum { EAST, WEST }
- enum { NORTH, SOUTH }

Functions

- void printdMsg (string source, Position *position, Port *port)
- void portInit ()
- unsigned int setID ()

Variables

- string directionString [NUMBER_OF_DIRECTIONS]
- Port portArray [NUMBER_OF_DIRECTIONS]
- stringstream * dMsg

6.2.1 Define Documentation

6.2.1.1 #define NUMBER_OF_DIRECTIONS 5

Definition at line 16 of file config_2d.h.

Referenced by Router::checkForActivity(), IC::doIC(), IRS::doIRS(), OC::doOC(), ODS::doODS(), ORS::doORS(), IRS::IRS(), ME::ME(), OC::OC(), ODS::ODS(), ORS::ORS(), ME::outputSync(), portInit(), and Router::Router().

6.2.2 Enumeration Type Documentation

6.2.2.1 anonymous enum

Enumerator:

EAST

WEST

Definition at line 30 of file config_2d.h.

6.2.2.2 anonymous enum

Enumerator:

NORTH

SOUTH

Definition at line 34 of file config_2d.h.

6.2.2.3 enum en_directions

Enumerator:

N

E

S

W

L

Definition at line 6 of file config_2d.h.

6.2.3 Function Documentation

6.2.3.1 void portInit ( )

Definition at line 37 of file config_2d.cpp.

References directionString, NUMBER_OF_DIRECTIONS, Port::port, and Port::portString.

Referenced by sc_main().

Here is the caller graph for this function:

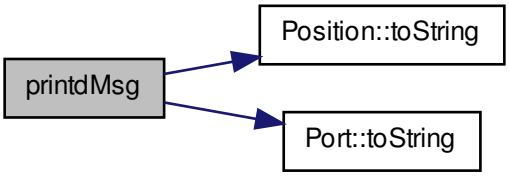
6.2.3.2 void printdMsg (string *source*, Position * *position*, Port * *port*)

Definition at line 12 of file config_2d.cpp.

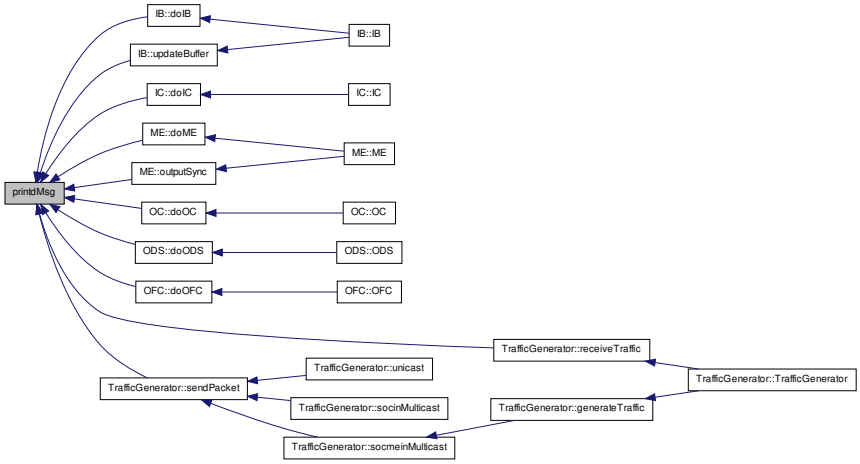
References dMsg, Position::toString(), and Port::toString().

Referenced by IB::doIB(), IC::doIC(), ME::doME(), OC::doOC(), ODS::doODS(), OFC::doOFC(), ME::outputSync(), TrafficGenerator::receiveTraffic(), TrafficGenerator::sendPacket(), and IB::updateBuffer().

Here is the call graph for this function:



Here is the caller graph for this function:

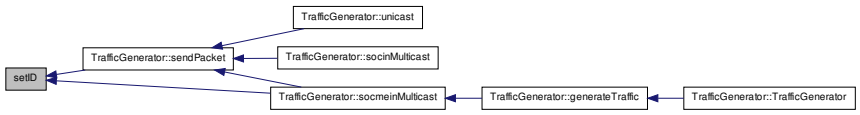


6.2.3.3 unsigned int setID ( )

Definition at line 26 of file config_2d.cpp.

Referenced by TrafficGenerator::sendPacket(), and TrafficGenerator::socmeinMulticast().

Here is the caller graph for this function:



6.2.4 Variable Documentation

6.2.4.1 string directionString[NUMBER_OF_DIRECTIONS]

Definition at line 3 of file config_2d.cpp.

Referenced by ME::doME(), OC::doOC(), ODS::doODS(), Port::Port(), and portInit().

6.2.4.2 stringstream* dMsg

Definition at line 10 of file config_2d.cpp.

Referenced by IB::doIB(), IC::doIC(), ME::doME(), OC::doOC(), ODS::doODS(), OFC::doOFC(), ME::outputSync(), printdMsg(), TrafficGenerator::receiveTraffic(), TrafficGenerator::sendPacket(), and IB::updateBuffer().

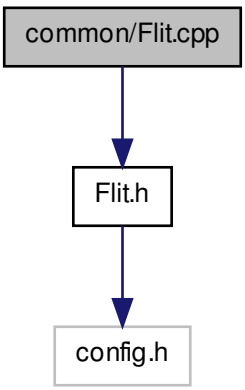
6.2.4.3 Port portArray[NUMBER_OF_DIRECTIONS]

Definition at line 35 of file config_2d.cpp.

Referenced by Router::Router().

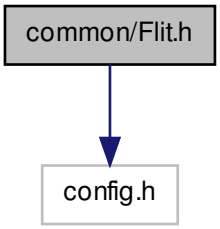
6.3 common/Flit.cpp File Reference

`#include "Flit.h"` Include dependency graph for Flit.cpp:

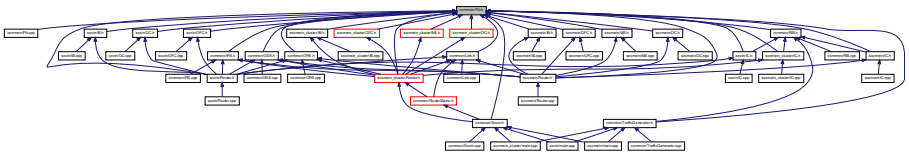


6.4 common/Flit.h File Reference

`#include "config.h"` Include dependency graph for Flit.h:



This graph shows which files directly or indirectly include this file:

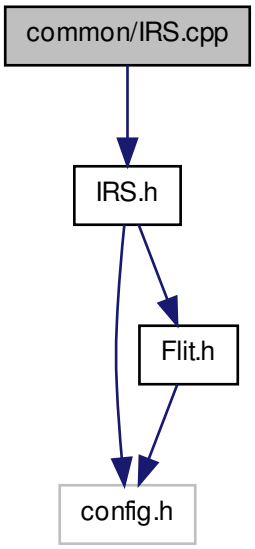


Classes

- class Flit

6.5 common/IRS.cpp File Reference

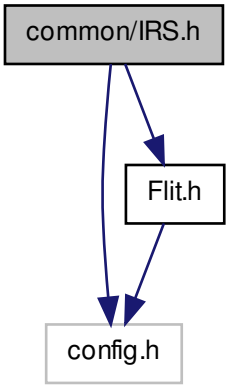
`#include "IRS.h"` Include dependency graph for IRS.cpp:



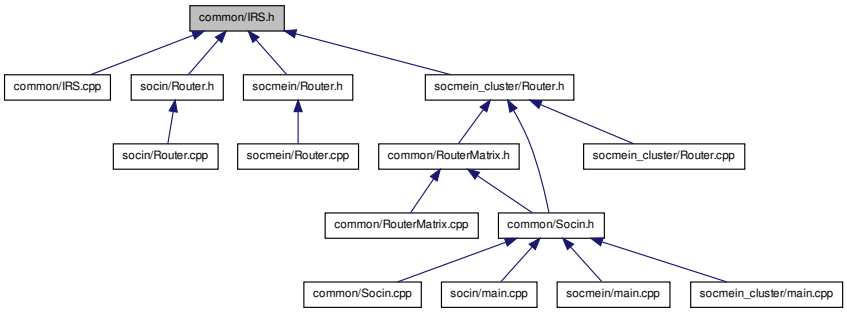
6.6 common/IRS.h File Reference

`#include "config.h"` `#include "Flit.h"` Include dependency graph for

IRS.h:



This graph shows which files directly or indirectly include this file:

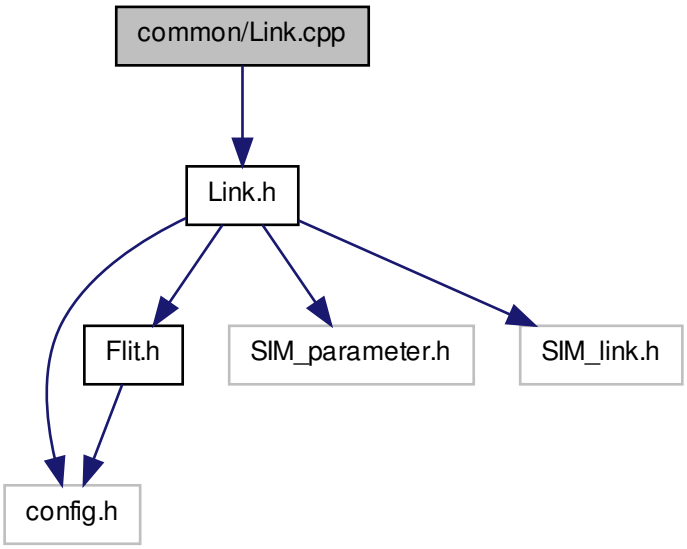


Classes

- struct IRS

6.7 common/Link.cpp File Reference

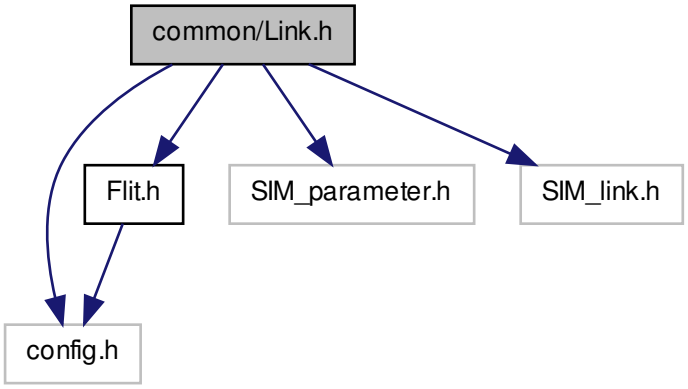
`#include "Link.h"` Include dependency graph for Link.cpp:



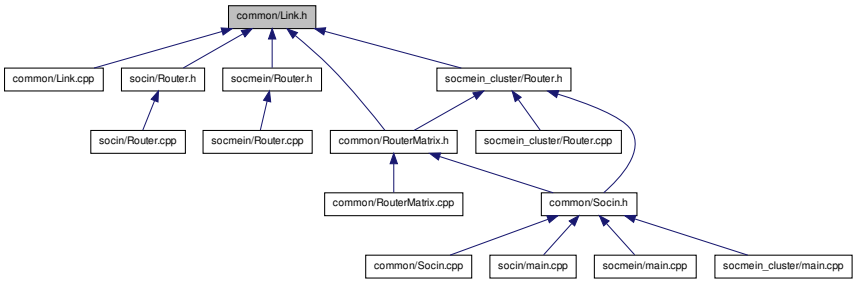
6.8 common/Link.h File Reference

`#include "config.h"` `#include "Flit.h"` `#include "SIM_parameter.h"` `#include "SIM_link.h"` Include dependency graph for -

Link.h:



This graph shows which files directly or indirectly include this file:

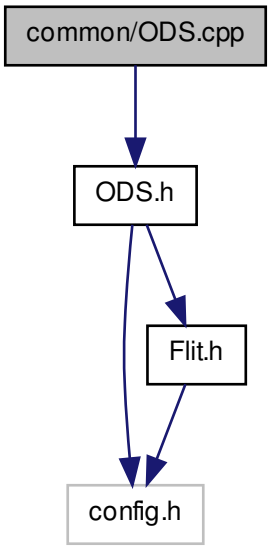


Classes

- class LinkReport
- class Link

6.9 common/ODS.cpp File Reference

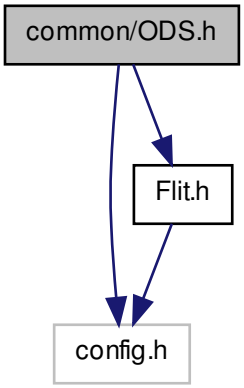
`#include "ODS.h"` Include dependency graph for ODS.cpp:



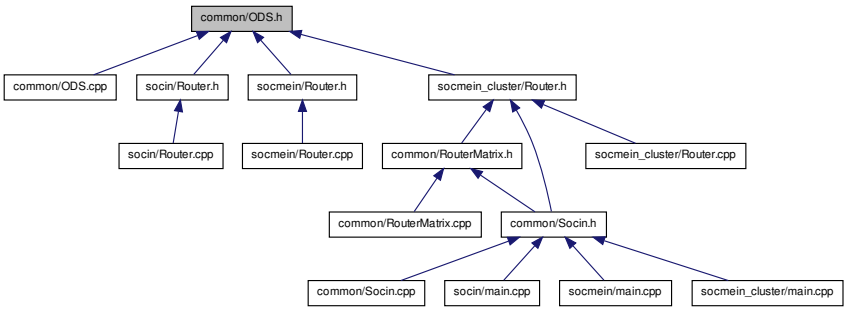
6.10 common/ODS.h File Reference

`#include "config.h"` `#include "Flit.h"` Include dependency graph for

ODS.h:



This graph shows which files directly or indirectly include this file:

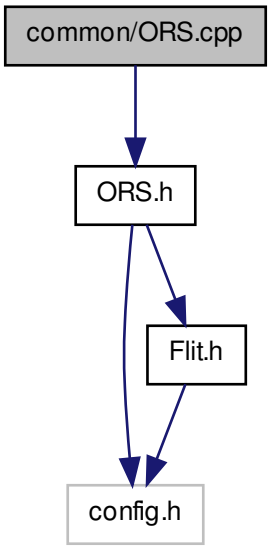


Classes

- struct ODS

6.11 common/ORS.cpp File Reference

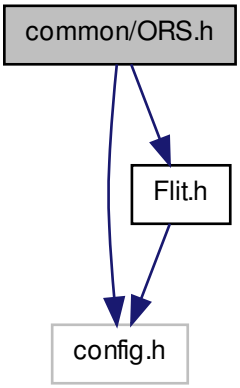
`#include "ORS.h"` Include dependency graph for ORS.cpp:



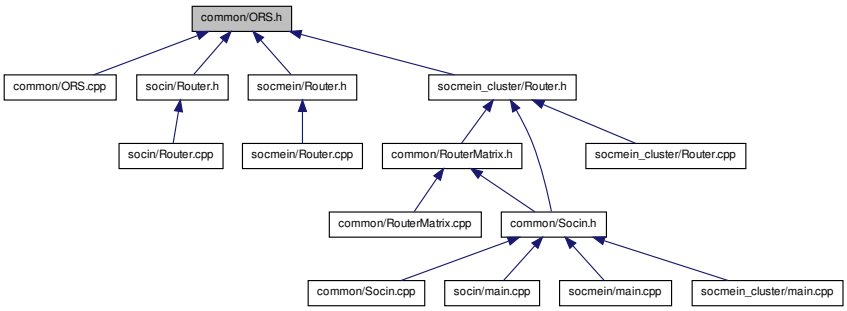
6.12 common/ORS.h File Reference

`#include "config.h"` `#include "Flit.h"` Include dependency graph for

ORS.h:



This graph shows which files directly or indirectly include this file:

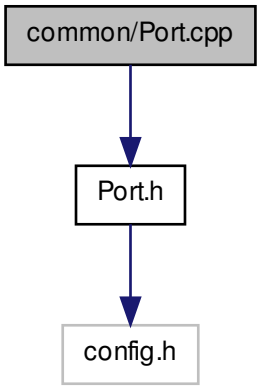


Classes

- struct ORS

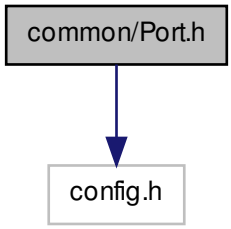
6.13 common/Port.cpp File Reference

`#include "Port.h"` Include dependency graph for Port.cpp:

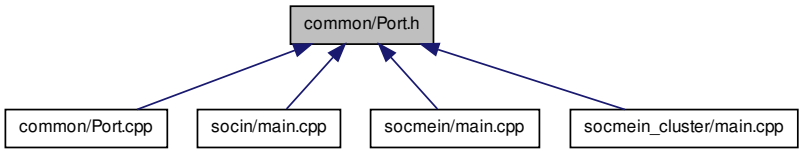


6.14 common/Port.h File Reference

`#include "config.h"` Include dependency graph for Port.h:



This graph shows which files directly or indirectly include this file:

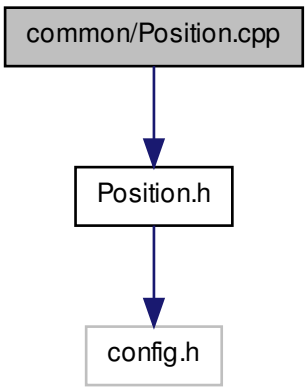


Classes

- class Port

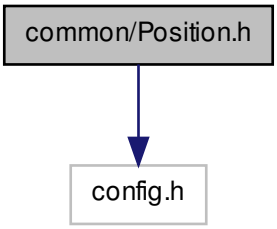
6.15 common/Position.cpp File Reference

`#include "Position.h"` Include dependency graph for Position.cpp:

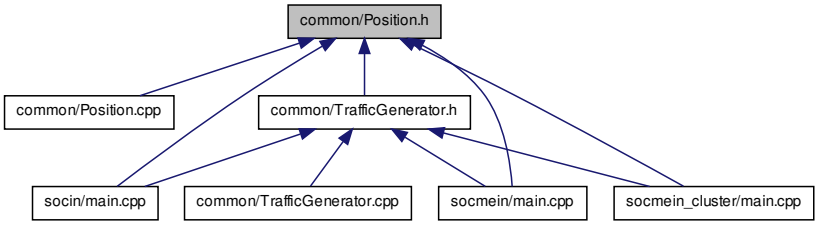


6.16 common/Position.h File Reference

`#include "config.h"` Include dependency graph for Position.h:



This graph shows which files directly or indirectly include this file:

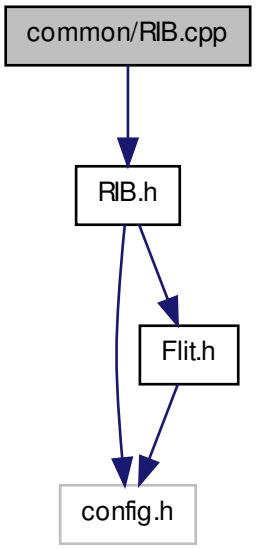


Classes

- class Position

6.17 common/RIB.cpp File Reference

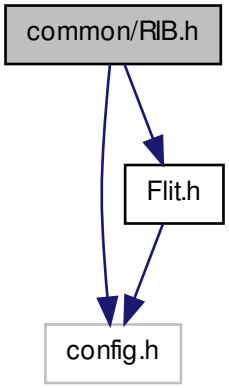
`#include "RIB.h"` Include dependency graph for RIB.cpp:



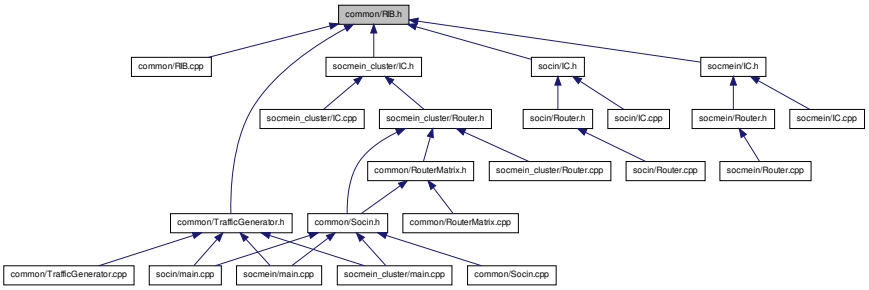
6.18 common/RIB.h File Reference

`#include "config.h"` `#include "Flit.h"` Include dependency graph for

RIB.h:



This graph shows which files directly or indirectly include this file:



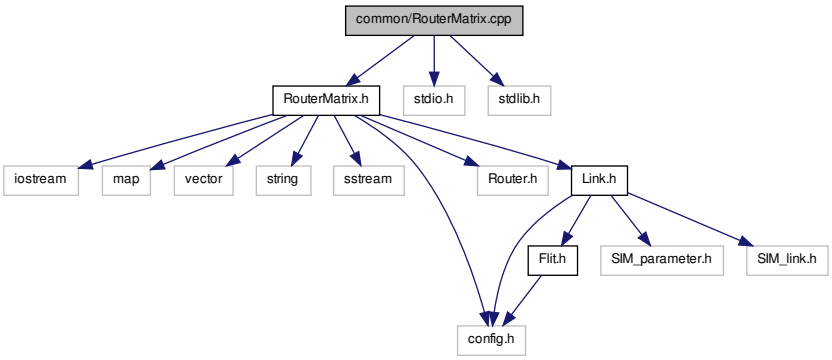
Classes

- class RIB

6.19 common/RouterMatrix.cpp File Reference

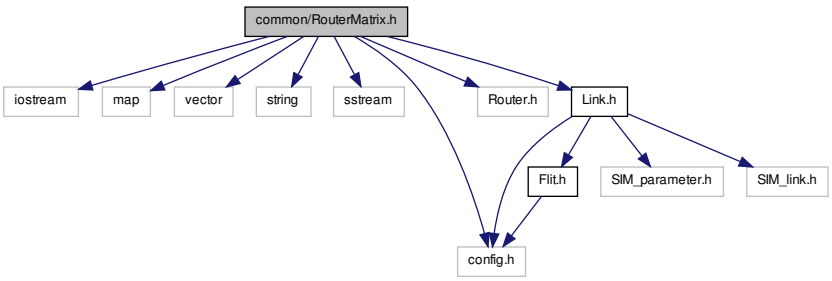
```
#include "RouterMatrix.h"   #include <stdio.h>   #include
```

<stdlib.h> Include dependency graph for RouterMatrix.cpp:

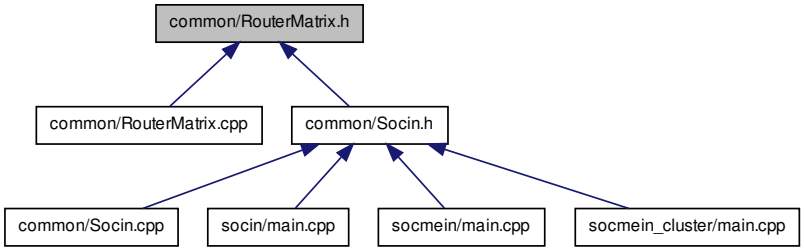


6.20 common/RouterMatrix.h File Reference

#include <iostream> #include <map> #include <vector> #include <string> #include <sstream> #include "config.h" #include "Router.h" #include "Link.h" Include dependency graph for RouterMatrix.h:



This graph shows which files directly or indirectly include this file:



Classes

- class RouterMatrix

Enumerations

- enum { W_TO_E, E_TO_W }
- enum { S_TO_N, N_TO_S }
- enum { TO_L, FROM_L }

6.20.1 Enumeration Type Documentation

6.20.1.1 anonymous enum

Enumerator:

W_TO_E

E_TO_W

Definition at line 14 of file RouterMatrix.h.

6.20.1.2 anonymous enum

Enumerator:

S_TO_N

N_TO_S

Definition at line 18 of file RouterMatrix.h.

6.20.1.3 anonymous enum

Enumerator:

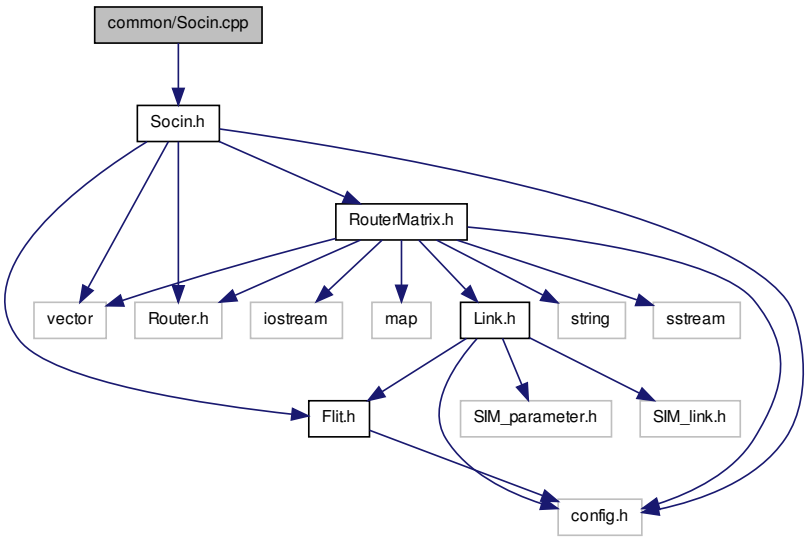
TO_L

FROM_L

Definition at line 22 of file RouterMatrix.h.

6.21 common/Socin.cpp File Reference

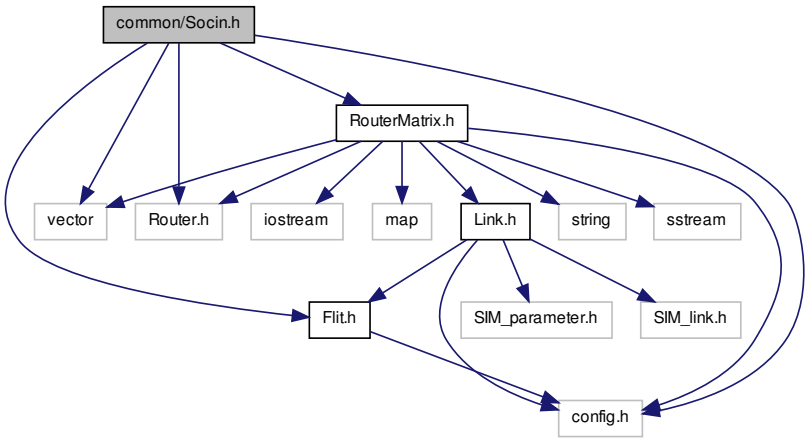
`#include "Socin.h"` Include dependency graph for Socin.cpp:



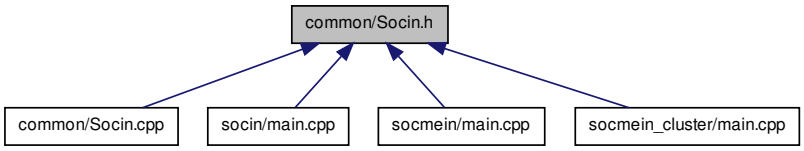
6.22 common/Socin.h File Reference

`#include <vector> #include "config.h" #include "Flit.h"` × `#include "Router.h" #include "RouterMatrix.h"` Include depen-

dency graph for Socin.h:



This graph shows which files directly or indirectly include this file:



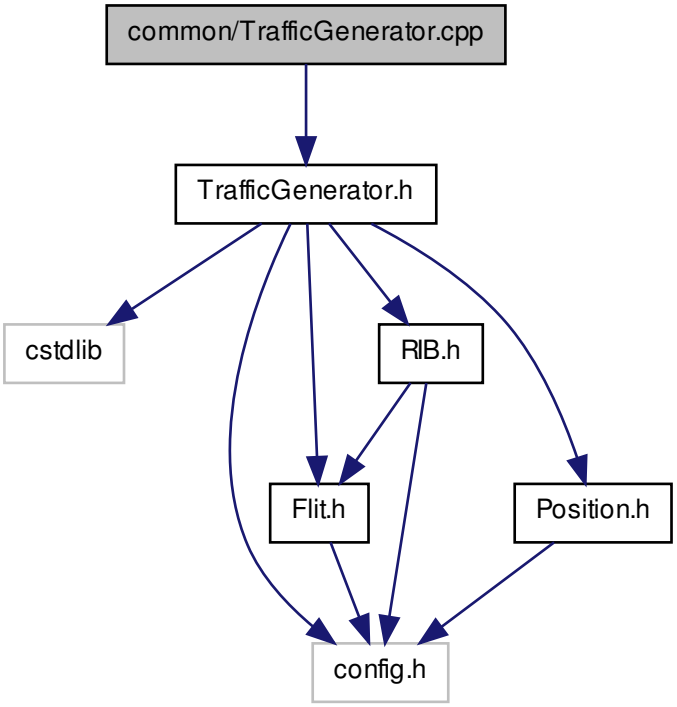
Classes

- struct Socin

6.23 common/TrafficGenerator.cpp File Reference

`#include "TrafficGenerator.h"` Include dependency graph for Traffic-

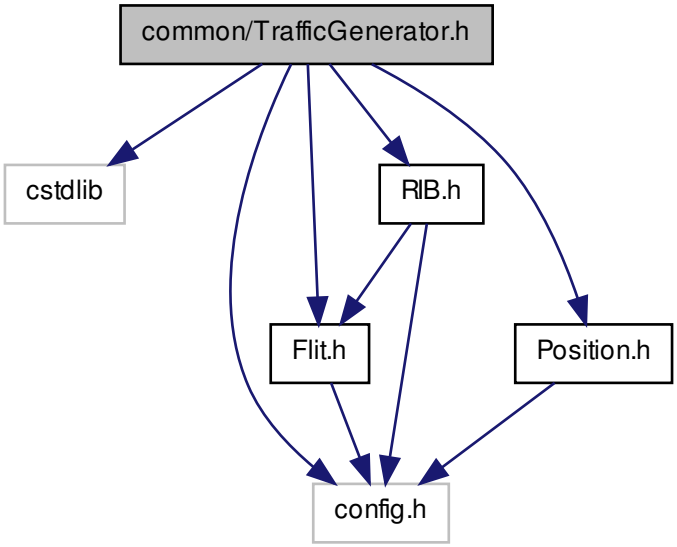
Generator.cpp:



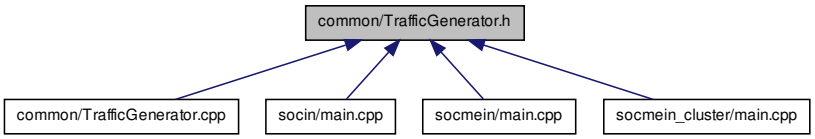
6.24 common/TrafficGenerator.h File Reference

`#include <cstdlib> #include "config.h" #include "Flit.h"` ×
`#include "RIB.h" #include "Position.h"` Include dependency graph for

TrafficGenerator.h:



This graph shows which files directly or indirectly include this file:

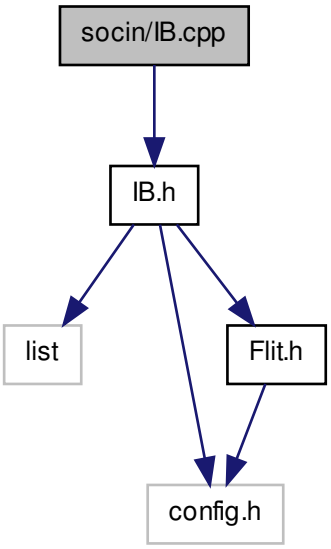


Classes

- class Packet
- struct TrafficGenerator

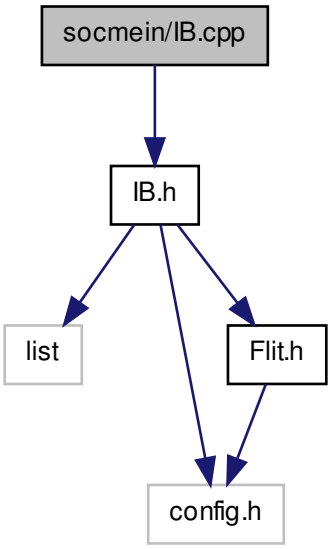
6.25 socin/IB.cpp File Reference

`#include "IB.h"` Include dependency graph for IB.cpp:



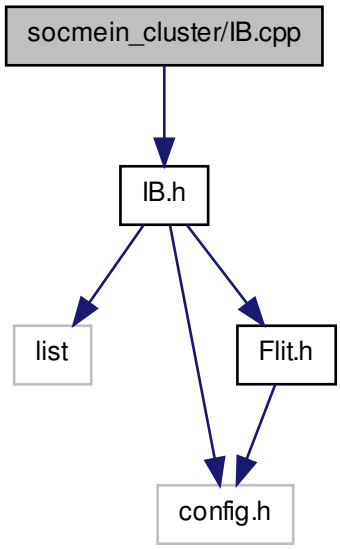
6.26 socmein/IB.cpp File Reference

`#include "IB.h"` Include dependency graph for IB.cpp:



6.27 socmein_cluster/IB.cpp File Reference

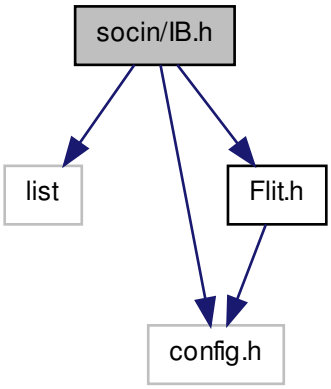
`#include "IB.h"` Include dependency graph for IB.cpp:



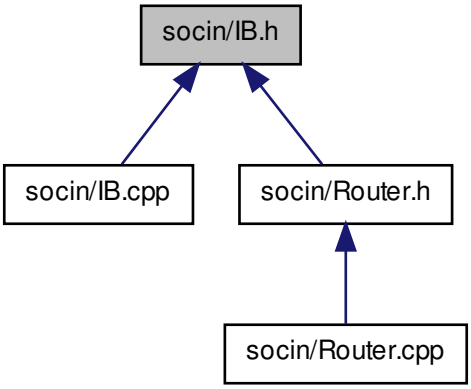
6.28 socin/IB.h File Reference

`#include <list>` `#include "config.h"` `#include "Flit.h"` Include

dependency graph for IB.h:



This graph shows which files directly or indirectly include this file:

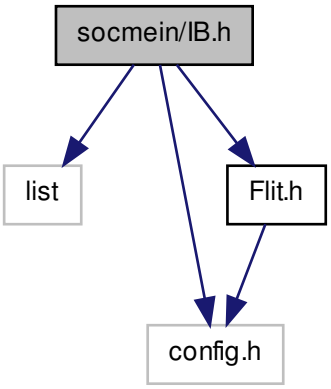


Classes

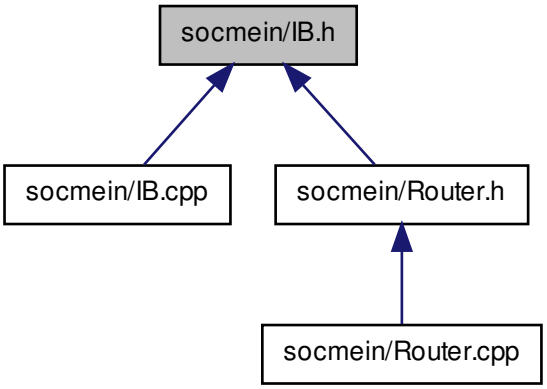
- struct IB

6.29 socmein/IB.h File Reference

`#include <list>` `#include "config.h"` `#include "Flit.h"` Include dependency graph for IB.h:



This graph shows which files directly or indirectly include this file:

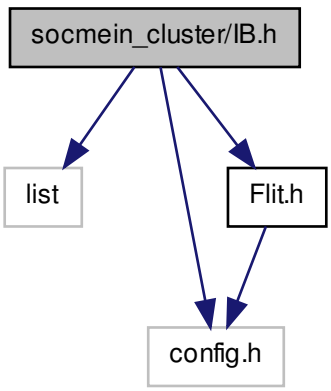


Classes

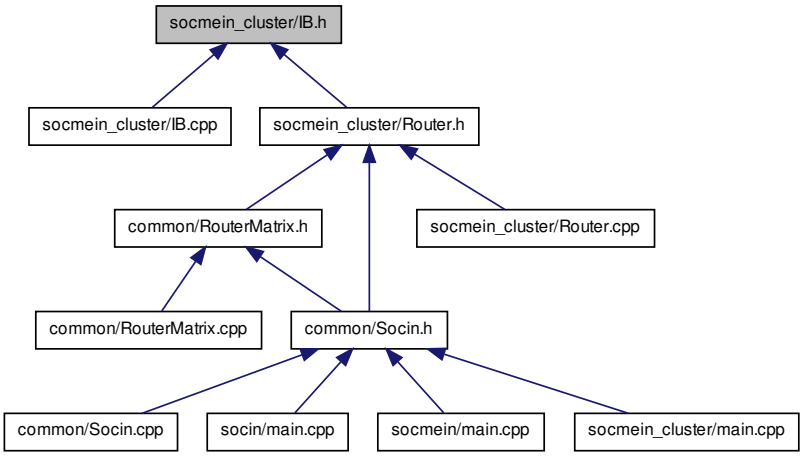
- struct IB

6.30 socmein_cluster/IB.h File Reference

`#include <list>` `#include "config.h"` `#include "Flit.h"` Include dependency graph for IB.h:



This graph shows which files directly or indirectly include this file:

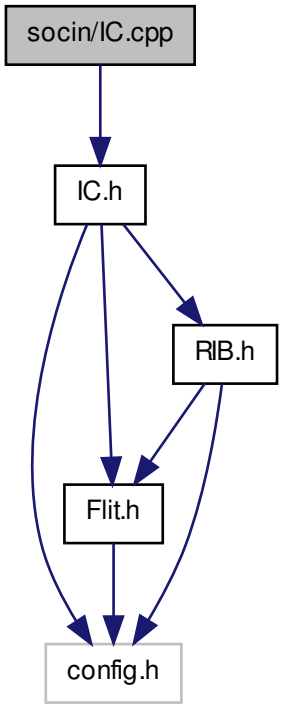


Classes

- struct IB

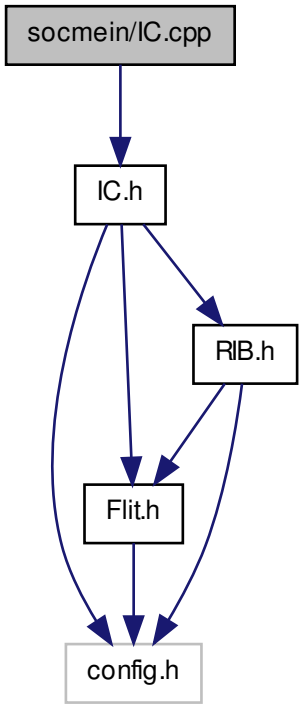
6.31 socin/IC.cpp File Reference

`#include "IC.h"` Include dependency graph for IC.cpp:



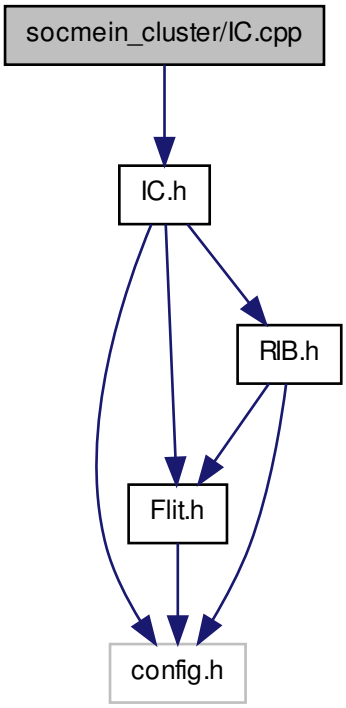
6.32 socmein/IC.cpp File Reference

`#include "IC.h"` Include dependency graph for IC.cpp:



6.33 socmein_cluster/IC.cpp File Reference

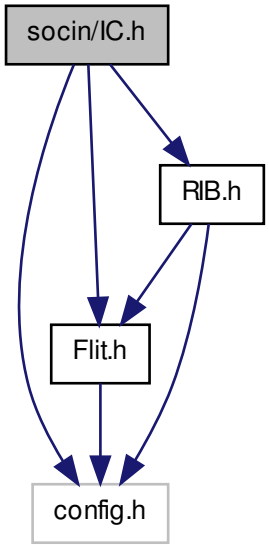
`#include "IC.h"` Include dependency graph for IC.cpp:



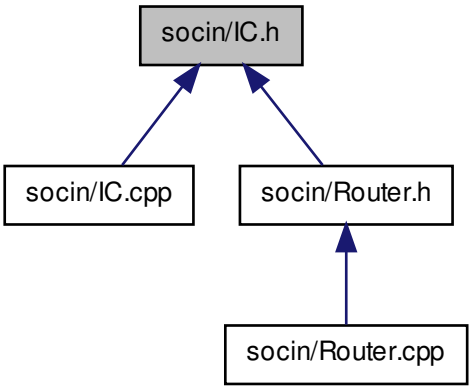
6.34 socin/IC.h File Reference

`#include "config.h"  #include "Flit.h"  #include "RIB.h"` ×

Include dependency graph for IC.h:



This graph shows which files directly or indirectly include this file:

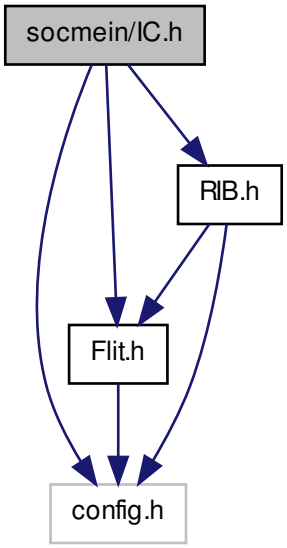


Classes

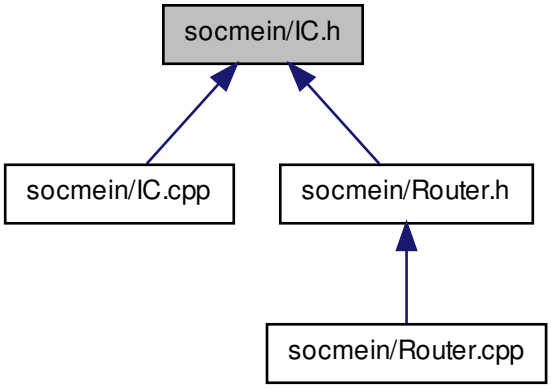
- struct IC

6.35 socmein/IC.h File Reference

`#include "config.h"` `#include "Flit.h"` `#include "RIB.h"` ×
Include dependency graph for IC.h:



This graph shows which files directly or indirectly include this file:



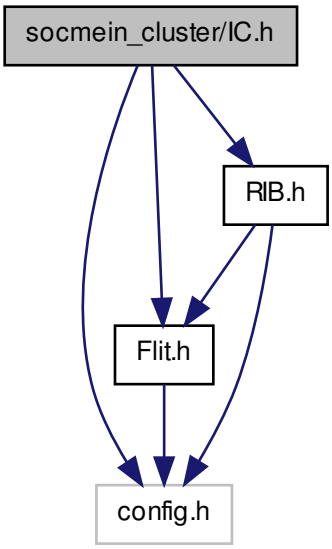
Classes

- struct IC

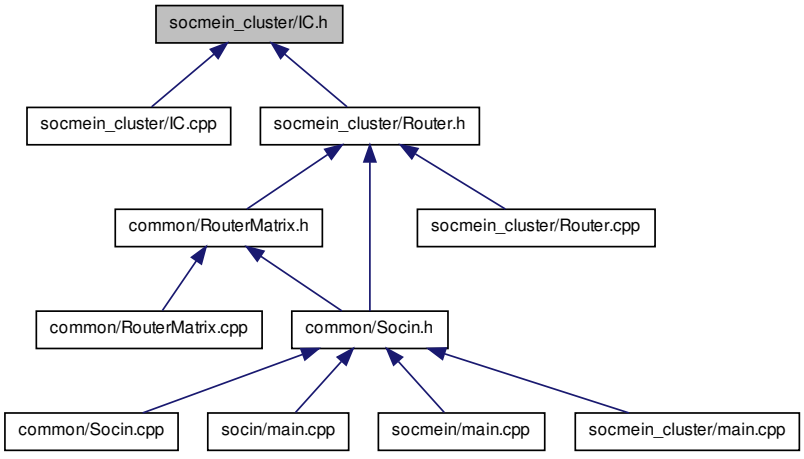
6.36 socmein_cluster/IC.h File Reference

```
#include "config.h"  #include "Flit.h"  #include "RIB.h" ×
```

Include dependency graph for IC.h:



This graph shows which files directly or indirectly include this file:

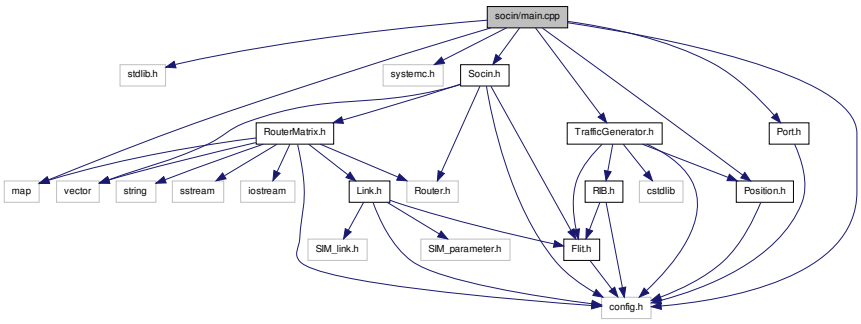


Classes

- struct IC

6.37 socin/main.cpp File Reference

`#include <stdlib.h> #include <map> #include "systemc.h" #include "config.h" #include "Socin.h" #include "TrafficGenerator.h" #include "Position.h" #include "Port.h"` Include dependency graph for main.cpp:



Functions

- int sc_main (int argc, char ∗argv[])

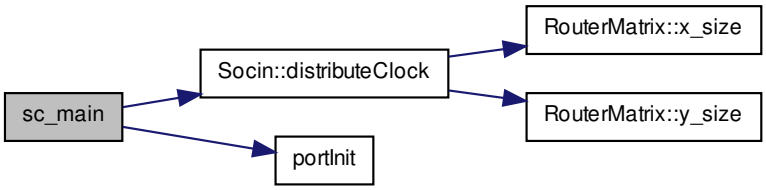
6.37.1 Function Documentation

6.37.1.1 int sc_main (int *argc,* char ∗ *argv[]*)

Definition at line 14 of file main.cpp.

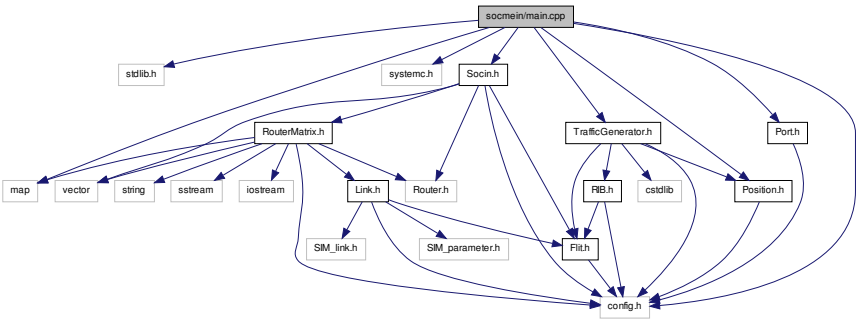
References Socin::clk, Socin::distributeClock(), FROM_L, TrafficGenerator::in, RouterMatrix::local_links, portInit(), Socin::routerMatrix, and TO_L.

Here is the call graph for this function:



6.38 socmein/main.cpp File Reference

`#include <stdlib.h> #include <map> #include "systemc.h" #include "config.h" #include "Socin.h" #include "TrafficGenerator.h" #include "Position.h" #include "Port.h"` Include dependency graph for main.cpp:



Functions

- int sc_main (int argc, char ∗argv[])

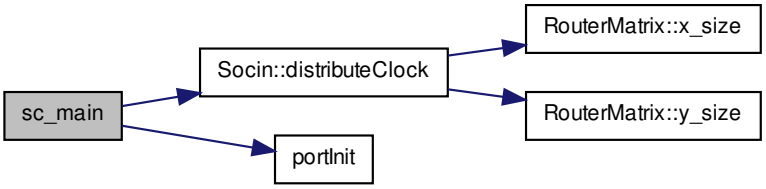
6.38.1 Function Documentation

6.38.1.1 int sc_main (int *argc,* char ∗ *argv[]*)

Definition at line 14 of file main.cpp.

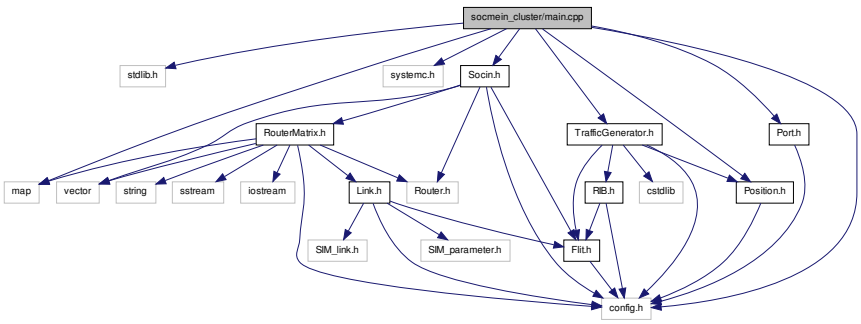
References Socin::clk, Socin::distributeClock(), FROM_L, TrafficGenerator::in, RouterMatrix::local_links, portInit(), Socin::routerMatrix, and TO_L.

Here is the call graph for this function:



6.39 socmein_cluster/main.cpp File Reference

`#include <stdlib.h> #include <map> #include "systemc.h" #include "config.h" #include "Socin.h" #include "TrafficGenerator.h" #include "Position.h" #include "Port.h"` Include dependency graph for main.cpp:



Functions

- int sc_main (int argc, char *argv[])

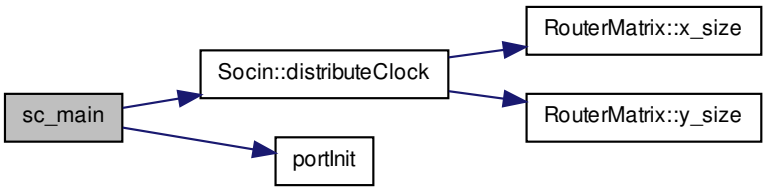
6.39.1 Function Documentation

6.39.1.1 int sc_main (int *argc,* char * *argv[]*)

Definition at line 14 of file main.cpp.

References Socin::clk, Socin::distributeClock(), FROM_L, TrafficGenerator::in, RouterMatrix::local_links, portInit(), Socin::routerMatrix, and TO_L.

Here is the call graph for this function:



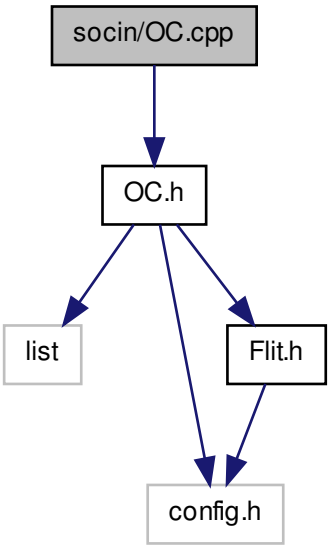
6.40 socin/main.h File Reference

6.41 socmein/main.h File Reference

6.42 socmein_cluster/main.h File Reference

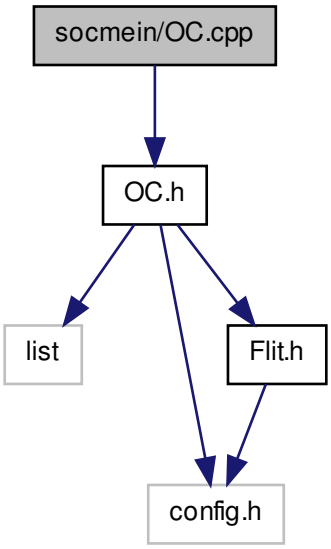
6.43 socin/OC.cpp File Reference

`#include "OC.h"` Include dependency graph for OC.cpp:



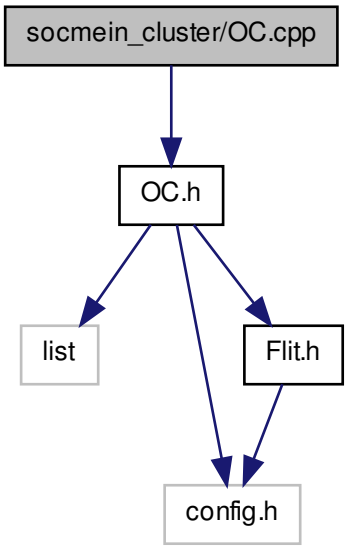
6.44 socmein/OC.cpp File Reference

`#include "OC.h"` Include dependency graph for OC.cpp:



6.45 socmein_cluster/OC.cpp File Reference

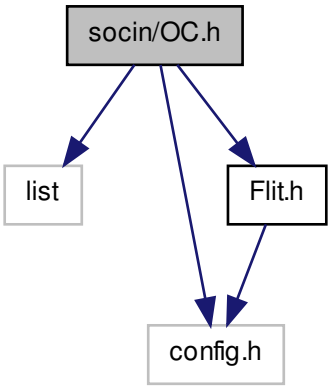
`#include "OC.h"` Include dependency graph for OC.cpp:



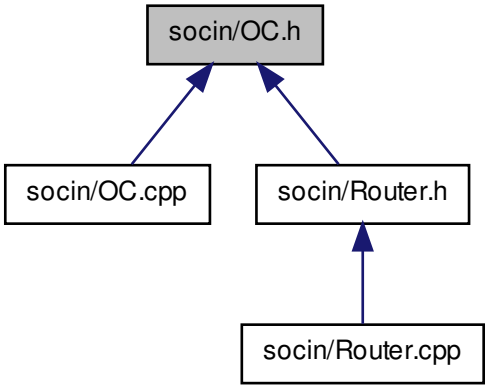
6.46 socin/OC.h File Reference

`#include <list>` `#include "config.h"` `#include "Flit.h"` Include

dependency graph for OC.h:



This graph shows which files directly or indirectly include this file:

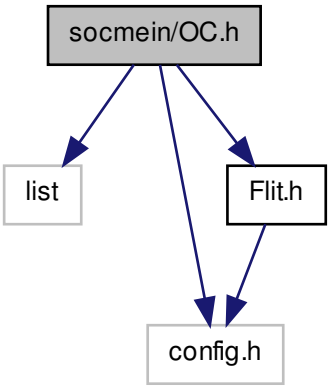


Classes

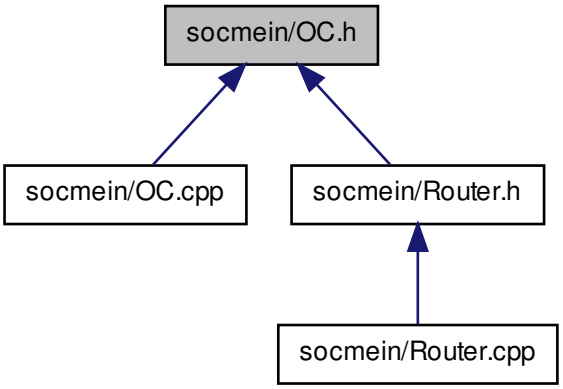
- struct OC

6.47 socmein/OC.h File Reference

`#include <list>` `#include "config.h"` `#include "Flit.h"` Include dependency graph for OC.h:



This graph shows which files directly or indirectly include this file:

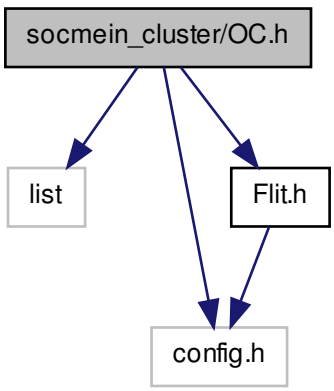


Classes

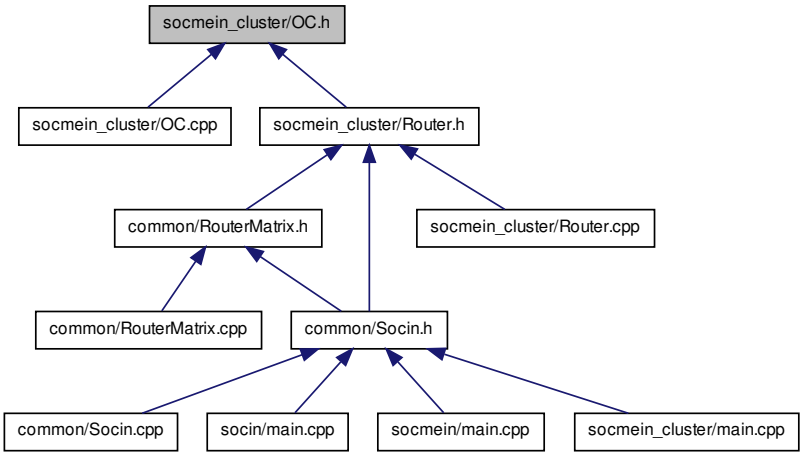
- struct OC

6.48 socmein_cluster/OC.h File Reference

`#include <list>` `#include "config.h"` `#include "Flit.h"` Include dependency graph for OC.h:



This graph shows which files directly or indirectly include this file:

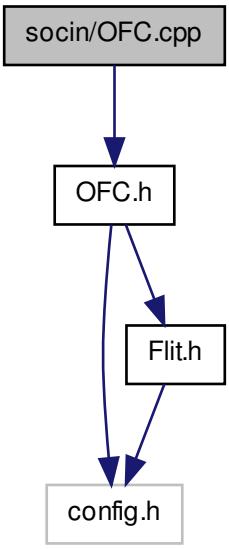


Classes

- struct OC

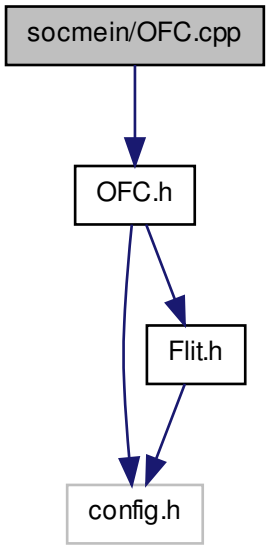
6.49 socin/OFC.cpp File Reference

`#include "OFC.h"` Include dependency graph for OFC.cpp:



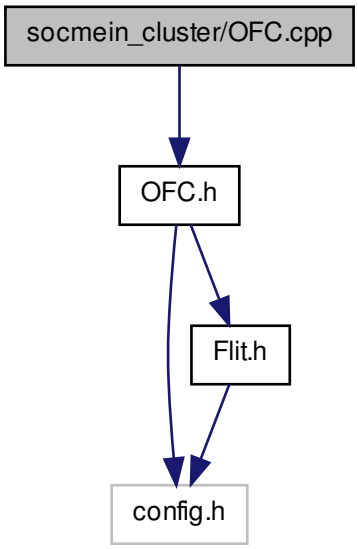
6.50 socmein/OFC.cpp File Reference

`#include "OFC.h"` Include dependency graph for OFC.cpp:



6.51 socmein_cluster/OFC.cpp File Reference

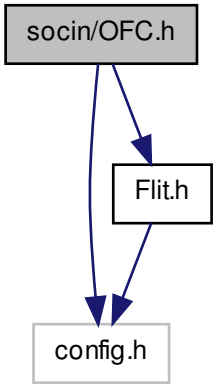
`#include "OFC.h"` Include dependency graph for OFC.cpp:



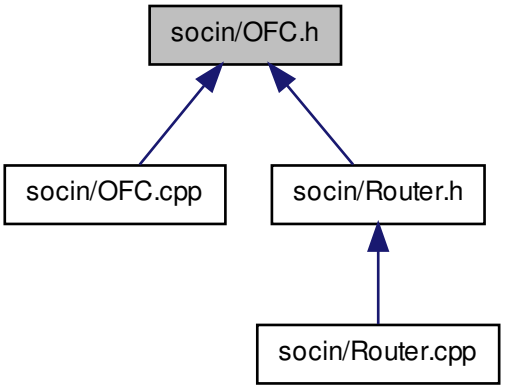
6.52 socin/OFC.h File Reference

`#include "config.h"` `#include "Flit.h"` Include dependency graph for

OFC.h:



This graph shows which files directly or indirectly include this file:

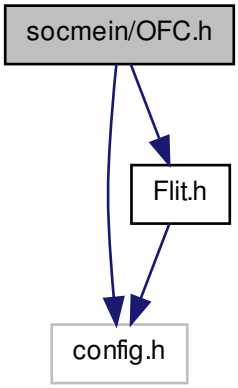


Classes

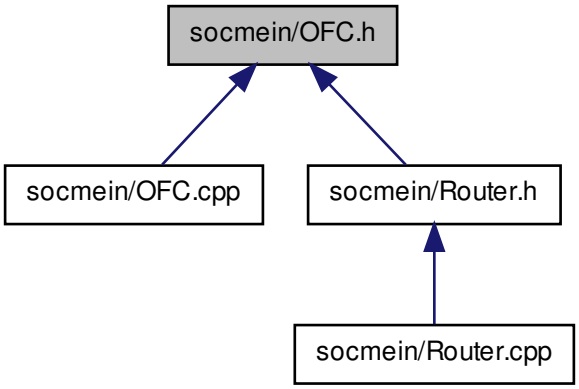
- struct OFC

6.53 socmein/OFC.h File Reference

`#include "config.h"` `#include "Flit.h"` Include dependency graph for OFC.h:



This graph shows which files directly or indirectly include this file:

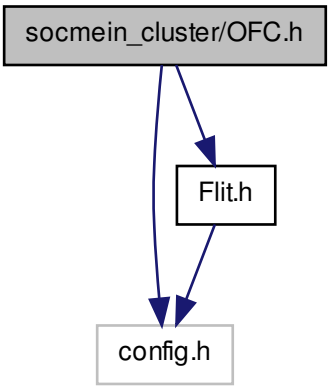


Classes

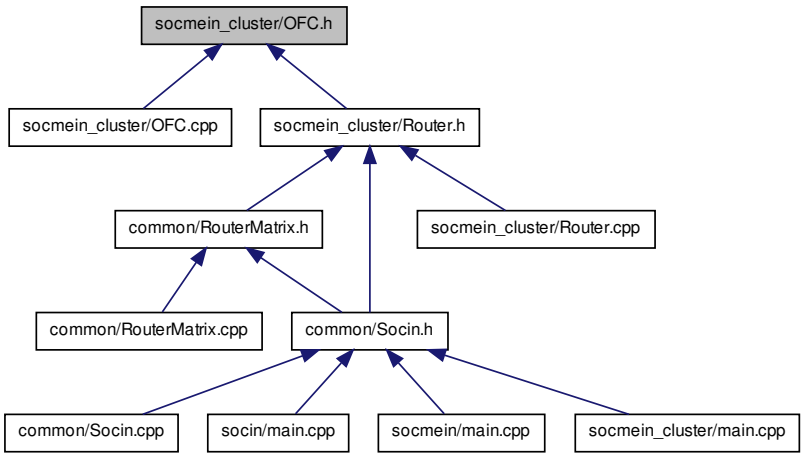
- struct OFC

6.54 socmein_cluster/OFC.h File Reference

`#include "config.h"` `#include "Flit.h"` Include dependency graph for OFC.h:



This graph shows which files directly or indirectly include this file:

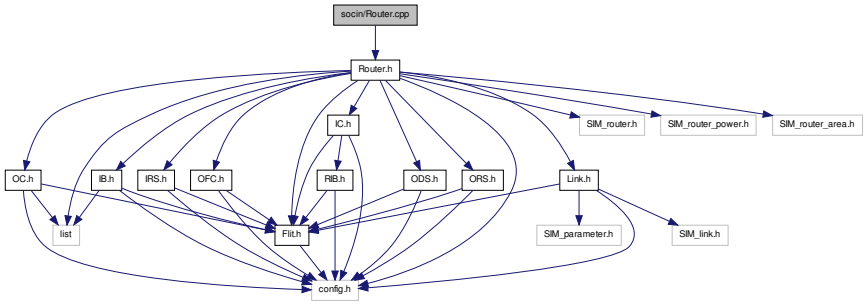


Classes

- struct OFC

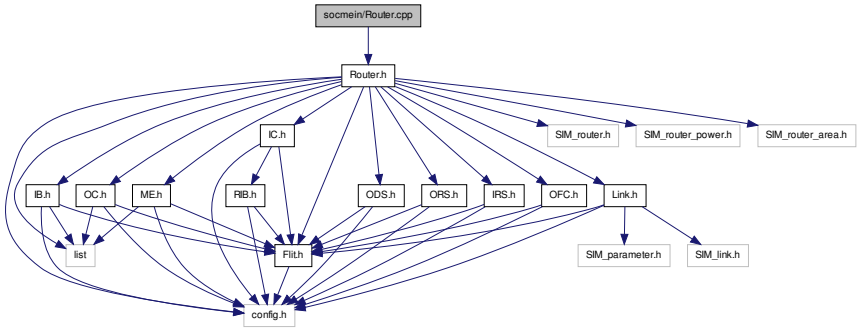
6.55 socin/Router.cpp File Reference

`#include "Router.h"` Include dependency graph for Router.cpp:



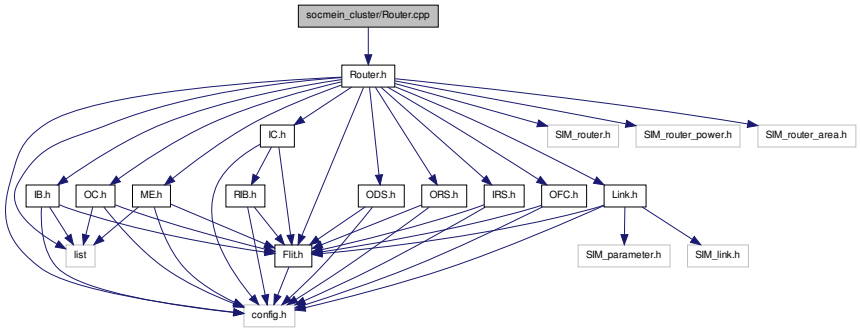
6.56 socmein/Router.cpp File Reference

`#include "Router.h"` Include dependency graph for Router.cpp:



6.57 socmein_cluster/Router.cpp File Reference

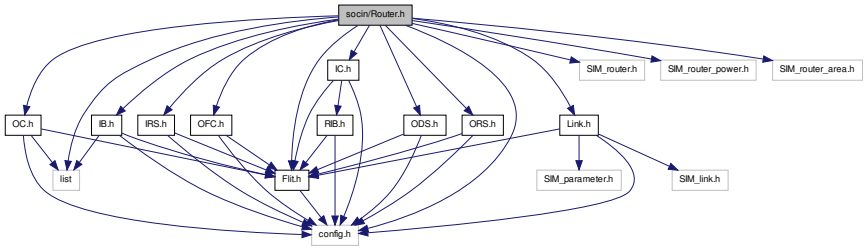
`#include "Router.h"` Include dependency graph for Router.cpp:



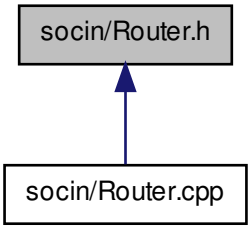
6.58 socin/Router.h File Reference

`#include <list>` `#include "config.h"` `#include "Flit.h"` × `#include "Link.h"` `#include "IB.h"` `#include "IC.h"` `#include "ODS.h"` `#include "ORS.h"` `#include "OC.h"` `#include "IRS.h"` `#include "OFC.h"` `#include "SIM_router.h"` `#include "SIM_router_power.h"` `#include "SIM_router_area.h"` Include dependency

graph for Router.h:



This graph shows which files directly or indirectly include this file:



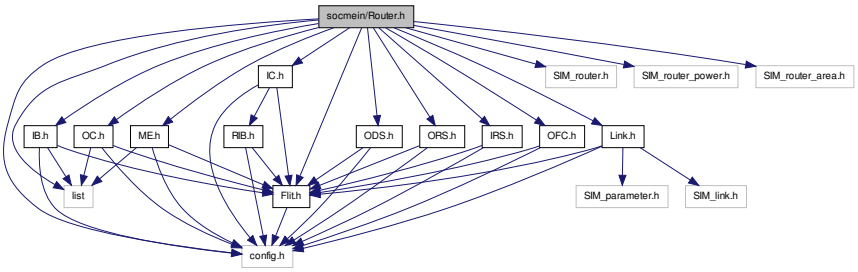
Classes

- class RouterReport
- struct Router

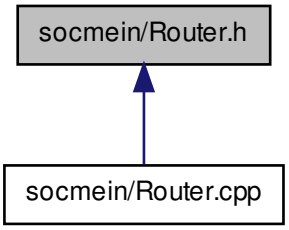
6.59 socmein/Router.h File Reference

```
#include <list>  #include "config.h"  #include "Flit.h" ×
#include "Link.h" #include "IB.h" #include "IC.h" #include
"ODS.h" #include "ORS.h" #include "OC.h" #include "IRS.h"
#include "OFC.h"  #include "ME.h"  #include "SIM_router.h"
#include "SIM_router_power.h"  #include "SIM_router_area.-
```

h" Include dependency graph for Router.h:



This graph shows which files directly or indirectly include this file:



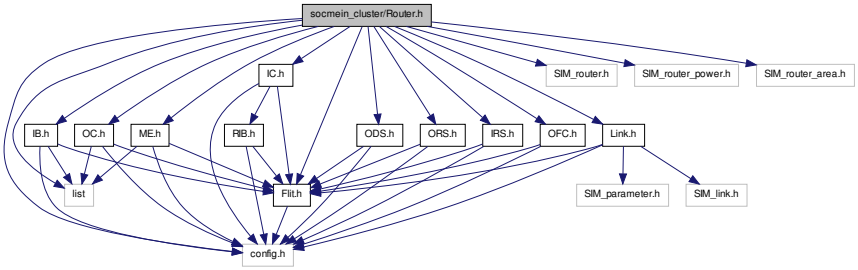
Classes

- class RouterReport
- struct Router

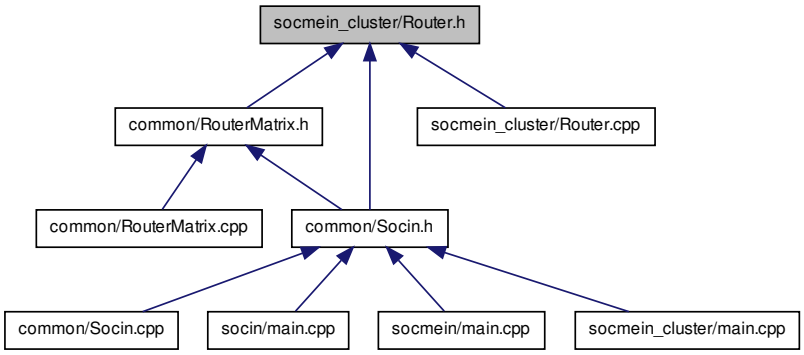
6.60 socmein_cluster/Router.h File Reference

```
#include <list>  #include "config.h"  #include "Flit.h" ×
#include "Link.h" #include "IB.h" #include "IC.h" #include
"ODS.h" #include "ORS.h" #include "OC.h" #include "IRS.h"
#include "OFC.h"  #include "ME.h"  #include "SIM_router.h"
#include "SIM_router_power.h"  #include "SIM_router_area.-
```

h" Include dependency graph for Router.h:



This graph shows which files directly or indirectly include this file:

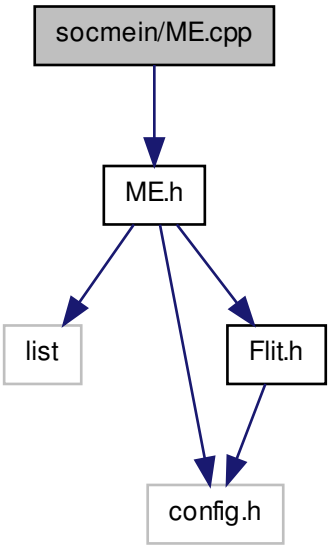


Classes

- class RouterReport
- struct Router

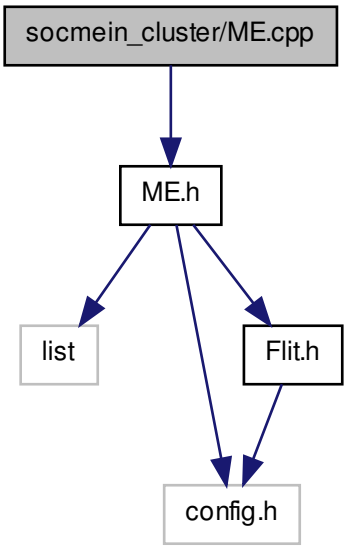
6.61 socmein/ME.cpp File Reference

`#include "ME.h"` Include dependency graph for ME.cpp:



6.62 socmein_cluster/ME.cpp File Reference

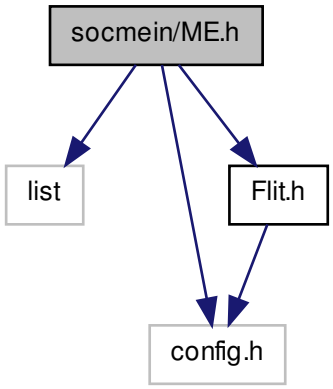
`#include "ME.h"` Include dependency graph for ME.cpp:



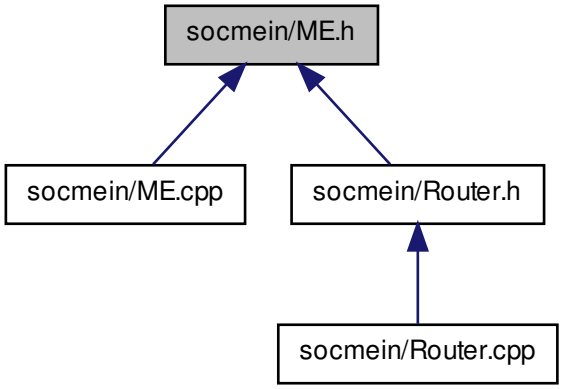
6.63 socmein/ME.h File Reference

`#include <list>` `#include "config.h"` `#include "Flit.h"` Include

dependency graph for ME.h:



This graph shows which files directly or indirectly include this file:

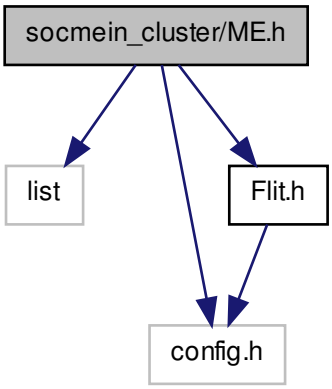


Classes

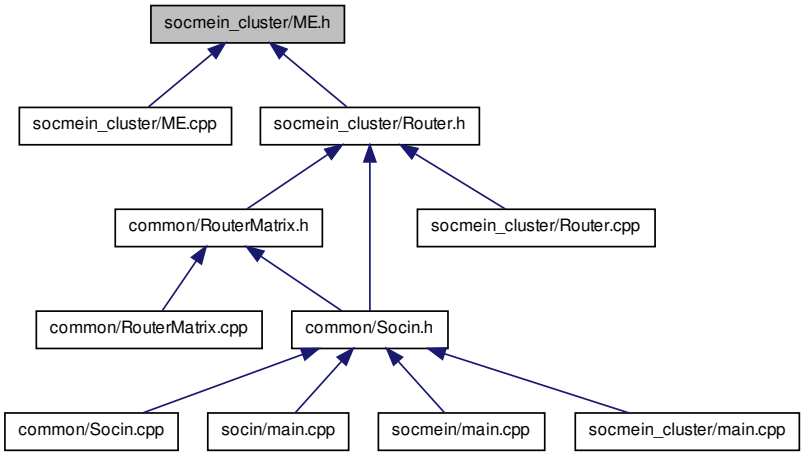
- struct ME

6.64 socmein_cluster/ME.h File Reference

`#include <list>` `#include "config.h"` `#include "Flit.h"` Include dependency graph for ME.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct ME