IP-Checksum Incremental Update Method Proposal for Efficient Use of Energy in Wireless Environments
Proposta do Cálculo do Update do Checksum para uso Eficiente de Energia em Ambientes Wireless

Domingos S. S. Carneiro, Paulo V. A. Pinheiro, Pedro H. Prudêncio, Daniel N. S. Cavalcante,
Diego V. S. Sousa, Rudy M. Braquehais, Thially V. P. Marrocos, Marcial P. Fernandez
Laboratório de Redes de Comunicação e Segurança (LARCES)
Universidade Estadual do Ceará (UECE)
Av. Paranjana 1700 – Campus do Itapery – 60740-903 – Fortaleza – CE – Brasil
Email: {domingos,paulovap,pedro,danielcavalcante,diegosousa,rudy,thially,marcial}@larces.uece.br

***Abstract*— The use of Wireless Mesh Networks has growing use in areas without telecommunication infrastructure. One of the main problems is the network performance due to the low capacity of the processors used in intermediary mesh nodes. We propose a hardware implementation of IP Checksum incremental update in FPGA (Field Programmable Gate Array) or ASIC (Application Specific Integrated Circuits). Then, this implementation is compared with the kernel Linux TCP/IP implementation.**

***Resumo*—O uso das Redes *Mesh* está crescendo em áreas sem infraestrutura de telecomunicação. Um de seus principais problemas é a performance de rede devido aos processadores de baixa capacidade usados em nós intermediários. Neste trabalho, é proposta uma implementação em hardware do Recálculo Incremental do *IP Checksum* em FPGA (Field Programmable Gate Array) ou ASIC (Application Specific Integrated Circuits). Então, esta implementação é comparada com a implementação em software da pilha TCP/IP do kernel Linux.**

***Keywords*— mesh networks, checksum, FPGA**
***Palavras Chave*—redes mesh, checksum, FPGA**

I. Introdução

O crescimento das comunicações sem-fio ocasionou um surgimento de um novo tipo de redes: redes *mesh* sem-fio (*wireless mesh networks*) [1]. Estas redes são heterogêneas, possuindo dispositivos que utilizam diferentes tecnologias sem-fio (IEEE 802.11, WiMAX, redes de celulares, Bluetooth). Os roteadores destas redes normalmente trabalham com várias tecnologias, possibilitando assim a comunicação entre dispositivos de diferentes tecnologias indiretamente. Por exemplo, um dispositivo que trabalhe com IEEE 802.11 não pode se comunicar diretamente com um dispositivo WiMAX; para realizar esta comunicação as informações passam através do roteador. Nestas redes, os roteadores são geralmente dispositivos com baixa mobilidade, enquanto os clientes são dispositivos móveis.

A maioria dos tipos de redes sem-fio têm baixo alcance, não sendo possível comunicar dispositivos que estejam separados por uma distância considerável. As Redes *Mesh* podem ser utilizadas para contornar este problema, de maneira semelhante à maneira pela qual dispositivos de tecnologias diferentes se comunicam. Os pacotes passam por vários nós da rede antes de chegarem a seu destino, e cada nó é responsável por encaminhar pacotes dos outros nós.

Já existem redes de acesso público onde os próprios usuários cuidam da manutenção dos roteadores, tais como Seattle Wireless, Champaign-Urbana Community Wireless Network (CUWiN), San Francisco BAWUG, Roofnet system do MIT, dentre outras. Várias empresas já vendem soluções prontas para Redes Mesh, tais como MeshNetworks, Tropos Networks, Radiant Networks, Firetide, BelAir Networks, Strix Systems [2].

Um pré-requisito para se entender os problemas das Redes *Mesh* abordados neste trabalho é entender como funciona cada tipo de Rede Mesh. A seguir uma listagem dos tipos e suas principais características:

1) Redes Mesh de Infraestrutura/Backbone: Redes de tecnologias possivelmente diferentes comunicam-se através de uma rede de roteadores sem-fio centrais com baixa mobilidade. Estas redes possuem pontos de convergência que são ligados aos roteadores da infraestrutura.

2) Redes Mesh de Clientes: Uma rede P2P (peer to peer) onde os próprios dispositivos clientes roteiam as informações.

3) Redes Mesh Híbridas: Redes onde os dois tipos anteriores coexistem.

Um dos grandes problemas com Redes *Mesh*, principalmente com as redes Clientes e Híbridas, é que seus roteadores têm baixa capacidade de processamento, e precisam economizar energia. Neste trabalho, é proposta a utilização de implementações em hardware usando ASIC para a execução das tarefas de roteamento. Como estudo de caso, compara-se o número de ciclos de máquina, com implementações em software e em hardware, da atualização incremental do

checksum no cabeçalho IP (*IP Checksum Incremental Update*). O algoritmo de checksum incremental utilizado pelo protocolo IP na atualização do campo de cabeçalho TTL foi implementado na linguagem descritiva de hardware Verilog, e tal implementação foi comparada com a implementação do mesmo algoritmo presente no *kernel* do Linux (software).

Este trabalho é organizado da seguinte forma: na seção II, é revisado o algoritmo checksum, normalizado pelas [3], [4], [5], [6]; na seção III, é apresentada a implementação em software do algoritmo supracitado; na seção IV, é apresentada a mesma implementação, porém em hardware (Verilog); na seção V, é feita uma comparação entre as duas implementações; e na seção VI é apresentada a conclusão.

II. Algoritmo de Checksum

Esta seção explica os algoritmos do cálculo e do recálculo incremental do checksum. O algoritmo consiste em juntar pares de bytes do cabeçalho para formarem números de 16 bits e calcular a soma **complemento para um** (one's complement, CP1)[7], [8], [9] destes.

A. Cálculo do checksum

Ao se fazer o checksum de uma sequência de bytes (A, B, C, D, ..., Y, Z), cada par de bytes (a,b) é convertido para um inteiro 256*a+b, representado pela notação [a,b]. No caso de bytes ímpares, considera-se um byte "virtual" com valor 0 (tabela I).

A soma **complemento para um** dos inteiros [*,*] é calculada. Para efeito de cálculo considera-se o espaço do cabeçalho destinado ao checksum como sendo 0. Ao final dos cálculos, a negação binária (negação no CP1) do valor da soma é colocada no lugar destinado ao checksum. A tabela I clarifica esta operação.

TABLE I
CHECKSUM DE UMA SEQUÊNCIA DE BYTES (SOMA CP1)

Sequência de bytes	Soma
A, B, C, D, ..., W, X, Y, Z	[A,B]+'[C,D]+'...+'[W,X]+'[Y,Z]
A, B, C, D, ..., W, X, Y	[A,B]+'[C,D]+'...+'[W,X]+'[Y,0]

Para se fazer a checagem, a mesma soma é computada, levando-se em conta o campo de checksum. Então o resultado de 0 (todos os bits 1, ou todos os bits 0 no CP1) indica o sucesso no cálculo[3].

B. Propriedades Matemáticas

O cálculo do checksum possui algumas propriedades matemáticas [3]:

- Comutativa e Associativa;
- Independência na ordem dos bytes;
- Soma paralela.

Estas propriedades facilitam os cálculos, minimizando as exceções. Além destas propriedades existem também algumas técnicas adicionais para melhorar a velocidade do cálculo e uma delas é o recálculo incremental do checksum.

C. Recálculo incremental do checksum (incremental update)

Para evitar o recálculo completo do checksum quando houver uma pequena alteração no cabeçalho IP, foi criado um método para recálculo incremental do checksum [3], [4], [5], [6].

Pequenas alterações no cabeçalho IP, normalmente no campo de TTL, não necessitam do cálculo completo de checksum. Realizar esta operação teria um alto custo computacional. Assim, para estes casos é usado um algoritmo que recalcula o checksum em função somente dos bits modificados.

Uma das equações para o cálculo do novo checksum consiste em pegar a soma CP1 do novo campo (C') em função do antigo (C). Estes valores podem ser alterados também por fragmentação de IP e alteração da rota da fonte [3], [4], [5], [6]:

$$C' = C + (-m) + m' = C + (m' - m) \tag{1}$$

Na proposição em questão os valores m e m' serão, respectivamente, a configuração inicial e final de um campo de 16-bits. A equação apresentada acima não é útil para um uso direto, pois C e C' não se referem ao atual checksum do campo. Uma equação que complementa a de cima é mostrada abaixo [3], [4], [5], [6]:

$$HC' = \sim (\sim HC + \sim m + m') \tag{2}$$

Onde HC' é o novo campo checksum, HC é o antigo campo. Um exemplo usando esta equação pode ser visto na figura 1, onde HC = 0xDD2F, m = 0x5555 e m' = 0x3285: .

```
HC' = HC + m + ~m'
    = 0xDD2F + 0x5555 + ~0x3285
    = 0xFFFF (sucesso)
```

Fig. 1. recálculo do checksum

D. Recálculo em máquinas complemento para dois

Em uma máquina **complemento para dois** (*two's complement*, CP2) [7], [8], [9], a soma CP1 pode ser computada transformando-se os números a serem somados em CP2, somando-os, e convertendo estes números de volta. A subtração ocorre de maneira semelhante. O restante do algoritmo permanece o mesmo.

Pode-se demonstrar que o método de soma em máquinas CP2 pode ser simplificado [3], [4], [5]. Apenas é realizada uma soma CP2 dos valores a serem operados. Então, o *carry* (bit de *overflow*) desta soma é somado ao resultado. Este é o método utilizado no *kernel* do Linux (seção III).

III. Implementação em software do checksum no *kernel* do Linux

A detecção de erros é uma funcionalidade essencial no processamento de pacote em todos os níveis da pilha TCP/IP, principalmente devido ao baixíssimo acoplamento entre suas camadas. A preocupação com a possibilidade de ocorrência de erro está diretamente relacionada à confiabilidade do

meio onde os quadros trafegam. Quase sempre os erros são atribuídos a ruídos no meio físico ou ocorrência de falhas na memória dos roteadores. Desta forma o checksum é introduzido nos dados trafegados, para aumentar a confiabilidade da rede.

Sempre que um segmento TCP ou UDP (utilizando o protocolo IPv4[10]) é recebido, é realizado o cálculo do seu checksum como definido em [11] e descrito na seção II. Para esta verificação é acrescido ao segmento um pseudo-cabeçalho contendo os endereços IP das máquinas de origem e de destino, o número do protocolo e tamanho do segmento. No caso dos pacotes IPv4 a mesma verificação é realizada, porém somente com os dados presentes no cabeçalho do pacote, pois presume-se que o protocolo da camada superior já verifique o restante do conteúdo. A principal finalidade desta verificação é evitar que um pacote tenha seu endereço ou controle alterado causando instabilidades no roteamento. Sendo, ainda, este cálculo repetido a cada roteador pelo qual passar, pois os pacotes IP têm pelo menos o seu campo TTL decrementado a cada salto[12]. Utilizando o algoritmo [4] descrito na subseção II-C, o custo deste recálculo é imensamente reduzido. E, logo abaixo da camada de rede, os protocolos do nível de enlace também possuem campos com checksum, normalmente códigos de verificação de redundância cíclica (CRC). Seria nesta camada onde, teoricamente, deveriam concentrar-se todas as verificações da ocorrência de erros, pois esta recebe seus quadros da camada física, sujeita às ações de ruídos, e os copia para a memória. Contudo a pilha TCP/IP não exige nenhuma garantia quanto ao recebimento de pacotes livres de erros do nível de enlace.

Com essa notória redundância de verificação novos padrões, como IPv6[13], foram sugeridos de forma a minimizar este *overhead*. São modificadas as presunções sobre o funcionamento das outras camadas de forma a diminuir a complexidade dos protocolos. Porém também é conhecida a dificuldade em adotar estes novos padrões [14]; sendo assim, o atual padrão em uso continuará exigindo uma considerável parcela dos recursos de processamento dos dispositivos de rede[15].

A. Uso do checksum na pilha TCP/IP do Linux

A implementação em software do algoritmo de atualização do checksum utilizada para comparação é a mesma presente no código do *kernel* do Linux[16]. A escolha deste sistema operacional deve-se principalmente por se tratar de um código livre e pela vasta quantidade de versões para as mais diversas arquiteturas (como Powerpc, Intel, System-on-a-chip, dentre outras), além de ser considerado o sistema mais próximo das especificações nas RFCs.

Os protocolos da subcamada de enlace mais populares, como o ethernets 10/100/1000 bits, 802.11, Token Ring etc. são implementados em hardware, desta forma o cálculo do CRC é realizado diretamente pelo dispositivo, restando ao sistema operacional somente encarregar-se de ler os pacotes de rede, que chegam após serem processados pela placa de rede, e fornecer aos *drivers* destas placas os *buffers* com os pacotes a serem transmitidos.

No caso dos protocolos da camada de transporte, a computação do campo checksum é realizada em sua grande maioria ainda em software. A função `csum_tcpudp_magic` encarrega-se de montar o pseudo-cabeçalho e agregá-lo ao segmento a ser verificado para então realizar a soma. Apesar de romper com a independência das camadas, algumas placas de rede mais novas também são capazes de atualizar e verificar este campo, assim liberando a CPU para outras tarefas. Esta funcionalidade também é conhecida como *offload* da soma de verificação e [6] descreve uma implementação em hardware do checksum. Quando um *buffer* deve ser transmitido, ele tem incluído um ponteiro para o espaço do campo no segmento onde deverá ser escrito pela placa o resultado do checksum. E, ao receber um segmento, esta mesma placa sinaliza com um *flag* o resultado da verificação. No entanto, um resultado negativo não necessariamente invalida o quadro para então descartá-lo; este comportamento poderia interferir no algoritmo de controle de fluxo no caso do TCP.

Devido à reduzida quantidade de bytes no cabeçalho dos pacotes IP a serem somados, o *kernel* realiza toda a soma em software, mesmo quando disponível uma placa capaz de fazê-la. O *kernel* possui duas funções com este propósito. A primeira função, `ip_compute_csum`, é uma função de propósito geral, ou seja, ela simplesmente recebe um *buffer* como entrada e soma seus bytes; esta função também é chamada pela função `csum_tcpudp_magic`. A segunda é a função `ip_fast_csum`, que é otimizada para o cabeçalho IP. Estas duas funções são implementadas na linguagem assembly específica de cada arquitetura suportada pelo *kernel*. Vale lembrar que a camada de rede possui um papel de destaque na comutação de pacotes. A principal função é o encaminhamento de pacotes, sendo que, ao encaminhar um pacote, este tem seu campo TTL decrementado de 1 e descartado caso o TTL chegue a 0. Desta forma evita-se que pacotes perdidos permaneçam vagando pela rede indefinidamente. Porém, graças ao algoritmo descrito na subseção II-C, não é necessário calcular novamente o checksum, mas somente atualizá-lo. Este recálculo do checksum é feito diretamente na função `ip_decrease_ttl`; além de recalcular o checksum, é, como o nome sugere, decrementado o campo TTL. Esta função é implementada na linguagem C e abaixo segue o código (figura 2):

```
static inline
int ip_decrease_ttl(struct iphdr *iph)
{
   u32 check = iph->check;
   check += _constant_htons(0x0100);
   iph->check = check + (check>=0xFFFF);
   return --iph->ttl;
}
```

Fig. 2. Código da função ip_decrease_ttl

.

Esta função recebe como único parâmetro um cabeçalho IP e retorna o novo valor decrementado do campo TTL. Nos três primeiros passos a função atualiza a soma para então

modificar o TTL. A primeira instrução armazena o antigo valor do checksum em uma variável de 32 bits. Em seguida o byte mais significativo é acrescido de um bit, correspondendo à diferença entre o antigo e novo valor do TTL. Assim, caso haja algum *overflow*, um bit será somado ao checksum. Por fim, a função decrementa o campo TTL e retorna seu valor. Um fluxograma desta operação é mostrado na figura 3. Esta função de incremento do checksum não possui implementação em hardware.

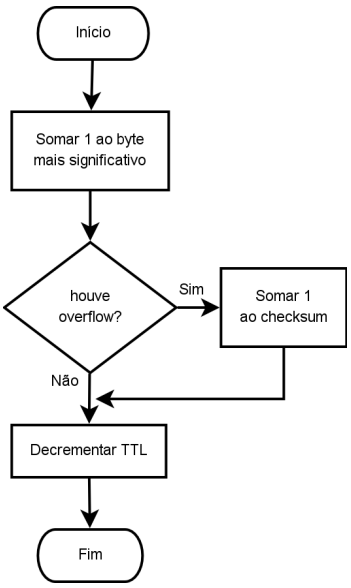

Fig. 3. Fluxograma da função ip_decrease_ttl

B. Quantidade de ciclos de máquina do incremento do checksum

Para se calcular a quantidade de instruções e ciclos de máquina necessários para executar a função de incremento, esta foi isolada em um programa aplicativo, permitindo um melhor controle do momento em que seria executada e do cálculo de tempo. Havia, ainda, a preocupação quanto ao fato de que, executando no modo usuário, o processo poderia perder a CPU, pelo fato de o sistema operacional escalonar algum outro processo. Mesmo o código sendo muito pequeno, o processo foi executado com prioridade máxima, para diminuir qualquer interferência durante sua execução. A figura 4 mostra o código do processo executado:

.

Como o *kernel* possui seus próprios tipos definidos, a função foi alterada para utilizar os tipos primitivos do C, preservando a quantidade de bits nos tipos correspondentes. A estrutura `struct iphdr` original possui tantos campos quanto um cabeçalho IP, os quais foram, porém, suprimidos, já que não foram utilizados neste código.

```
struct iphdr {
   unsigned short check;
   unsigned char ttl;
};

static
int ip_decrease_ttl{struct iphdr *iph}
{
   unsigned int check = iph->check;
   check += 0x0100;
   iph->check = check + (check>=0xFFFF);
   return --iph-ttl;
}

int main()
{
   struct iphdr header = ( 0xB057, 16 );
   ip_decrease_ttl(&header);
   return 0;
}
```

Fig. 4. Código do Processo

IV. Implementação do Recálculo do Checksum em Hardware

A. Projeto e Desenvolvimento de Hardware

Com o intuito de alcançar as metas deste projeto, em particular a construção de tal plataforma dinamicamente reconfigurável, um conjunto de sistemas (de hardware e de software) foi utilizado, bem como uma metodologia de desenvolvimento de projetos.

Nas próximas subseções serão apresentados todos os elementos cujo conhecimento é necessário à elaboração deste projeto: a arquitetura dos dispositivos usados, plataformas que lhes servem de suporte, ferramentas para desenvolvimento de sistemas embarcados e metodologia do projeto.

B. Etapas do processo de desenvolvimento

As etapas de desenvolvimento de um circuito lógico digital não são tão diferentes do desenvolvimento de sistemas embarcados. Uma estrutura é escrita em uma linguagem descritiva de hardware de alto nível (VHDL, Verilog, etc), o código é compilado e copiado para ser executado. O circuito também pode ser descrito como um esquemático digital, porém é menos popular e mais complexo que utilizar ferramentas baseadas em linguagens. Uma visão geral das etapas do processo de desenvolvimento de hardware para dispositivos lógico programáveis é mostrada na figura 5[17].

Os desenvolvedores de software tendem a ter uma linha de pensamento sequencial, mesmo quando estão desenvolvendo aplicações multitarefa (*multithreaded*). Diferentemente de um projeto de software, os projetistas de hardware precisam pensar e programar em paralelo. Todos os sinais são processados em paralelo, pois trafegam através de um caminho de execução próprio, em uma série de macro-células e interconexões até o destino do sinal de saída. Assim, as estruturas de descrição do hardware podem ser executadas ao mesmo tempo.

Geralmente, a etapa de início do projeto é seguida ou compartilhada com períodos de simulação funcional. Este é o momento onde um simulador é utilizado na execução

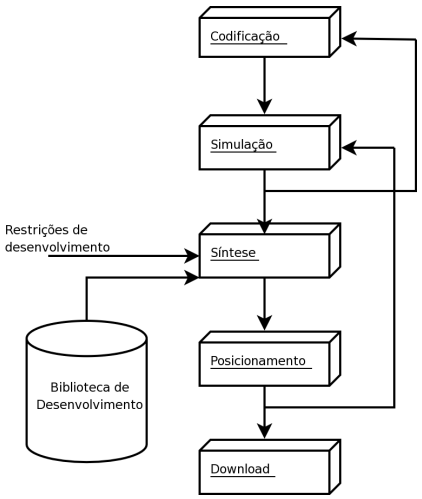

Fig. 5. Etapas de desenvolvimento

do projeto para confirmar a combinação de saídas corretas com determinadas entradas de teste. Assim, o projetista pode certificar-se de que a sua lógica de funcionamento está correta antes de ir ao estágio seguinte de desenvolvimento, embora isto ainda possa mudar no decorrer do projeto.

É feita uma representação funcional do hardware e então ocorre a compilação, a qual se dá em duas etapas; primeiramente, uma representação intermediária esquemática do projeto do hardware é produzida; dá-se o nome de síntese (*synthesis*) e o resultado é chamado de *netlist*. O *netlist* é armazenado em um formato padrão conhecido como EDIF (Electronic Design Interchange Format); trata-se de um dispositivo independente, pois seu conteúdo não depende de um FPGA ou ASIC[17].

A etapa seguinte, ainda no processo de tradução, é chamada de "*place & route*". Aqui, traçamos estruturas lógicas descritas no resultado da etapa anterior, as (*netlists*) em macro-células, interconexões e pinos reais de entrada e saída. Este processo é semelhante ao desenvolvimento de uma placa de circuito impresso, porém permite otimizações manuais ou automáticas das disposições. Como resultado temos um *bitstream*, que é uma série de dados binários que deverão ser carregados no FPGA, para que o projeto seja testado. Além disso, se for necessária uma implementação em ASIC, pode se gerar um *bitstream* padrão para fundição (*foundry*), como o GDSII.

Ainda podemos contar com o auxílio de depuradores, geralmente fornecidos pela própria empresa desenvolvedora da ferramenta, visto que, para tal, é preciso um amplo conhecimento do comportamento dos chips.

Com o *bitstream* criado, será necessário transportá-lo, de algum modo, para o dispositivo. Há várias formas de fazê-lo, dependendo da tecnologia do chip utilizado. Estes dispositivos reconfiguráveis são como memórias. Utilizam as tecnologias PROM, EPROM, EEPROM e Flash. Estes últimos possuem as mesmas vantagens relacionadas à facilidade de programação e re-programação. Estes dispositivos assemelham-se aos microcontroladores, muitos suportando interfaces JTAG.

Há possibilidade ainda de o chip ser baseado em uma tecnologia de memória volátil, SRAM, com índices de memória temporários, o que traz vantagens e desvantagens. Como desvantagem temos a necessidade de recarregar todos os *bitstreams* sempre que houver uma restauração do sistema. Isto requer um tipo de memória adicional para armazená-lo. Como vantagem, temos a facilidade de atualização, pois o *bitstream* pode ser manipulado *on-the-fly* , facilitando a concepção do cenário de testes, otimizando todo o projeto de hardware [18].

C. Plataforma de Prototipação

As ferramentas de prototipação nos permitem criar sistemas baseados em circuitos lógico-digitais de maneira prática. O processo consiste em programar um FPGA ou outro dispositivo programável segundo a necessidade de sua aplicação. Para tanto, há várias plataformas que servem de suporte aos FPGAs, provendo conexões de entrada e saída que permitem interação das aplicações dos usuários entre processamento e meio externo.

Foi utilizada a plataforma *Integrated Software Environment* (ISE) da Xilinx, como ferramenta para o ambiente de projeto. Esta foi utilizada na estruturação e desenvolvimento do projeto, tanto na etapa inicial do projeto, quando os módulos foram todos criados pelo desenvolvedor, bem como na etapa de construção do dispositivo reconfigurável[19], [17].

D. Implementação em Verilog

A implementação do checksum em HDL é feita utilizando-se de 2 módulos feitos em Verilog. O primeiro deles, $one\_complement\_sum\_16$ (figura 6), realiza a soma **complemento para um** (*one's complement*, CP1) associando à saída sum a soma das entradas $numA$ e $numB$. O segundo módulo, $recalcipchecksum$ (figura 8), recalcula o checksum [3], [4], [5], [6] usando o módulo $one\_complement\_sum\_16$. Os módulos são descritos nas subseções a seguir.

```
module one_complement_sum_16(numA,numB,sum);
    input [15:0] numA;
    input [15:0] numB;
    wire [15:0] almostsum;
    output[15:0] sum;

    assign almostsum = numA + numA[15] + numB + numB[15];
    assign sum = almostsum - almostsum[15];
endmodule
```

Fig. 6. one_complement_sum_16

1) Implementação do Complemento para Um: Os cálculos feitos na implementação da figura 6 são os seguintes:

- Primeiro convertemos os dois números de entrada $numA$ e $numB$ para CP2 [7], [8], [9], depois atribuímos a sua soma ao resultado sum:

```
// convertemos os dois números para notação CP2
assign numA_CP2 = numA + numA[15];
assign numB_CP2 = numB + numB[15];

// somamos os dois números na nova notação
assign almostsum = numA_CP2 + numB_CP2;
```

```
// convertemos o resultado de volta
assign sum = almostsum - almostsum[15];
```

- É fácil perceber que podemos reescrever as 3 primeiras atribuições do código anterior da seguinte maneira:

```
assign almostsum = numA + numA[15] + numB + numB[15];
```

Chegando-se assim no comando utilizado na figura 6.

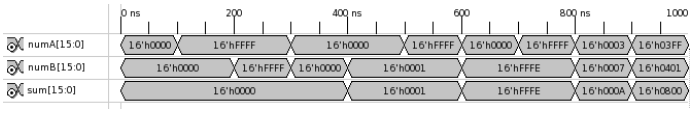

Fig. 7. Simulação do one_complement_sum_16

A figura 7 mostra uma instância de simulação do código usando o ISE Simulator da Xilinx. Os primeiros 4 testes mostram as 4 possibilidades de somas dos "zeros"da notação (0x0000+0x0000, 0x0000+0xFFFF, 0xFFFF+0x0000, 0xFFFF+0xFFFF). Os outros mostram somas com valores variados.

```
module recalcipchecksum( ttl_old, checksum_old,
        ttl_new, checksum_new );
    input [7:0] ttl_old;
    input [15:0] checksum_old;
    input [7:0] ttl_new;
    output [15:0] checksum_new;

    wire [15:0] partial_sum;
    wire [15:0] ttl_old_16;
    wire [15:0] ttl_new_16;

    // aqui o campo de TTL vai para o hibyte
    assign ttl_old_16[15:8]=ttl_old;
    assign ttl_old_16[7:0]=8'h00;
    assign ttl_new_16[15:8]=ttl_new;
    assign ttl_new_16[7:0]=8'h00;

    // C' = C + (-m) + m'
    one_complement_sum_16(
        checksum_old,
        ~ttl_old_16,
        partial_sum);
    one_complement_sum_16(
        partial_sum,
        ttl_new_16,
        checksum_new);
endmodule
```

Fig. 8. recalcipchecksum

2) Implementação do Recálculo de Checksum: O módulo principal (o de recálculo de checksum, figura 8) recebe como "parâmetros" os valores do antigo TTL, novo TTL e o antigo checksum. E faz a operação $C' = C + (-m) + m'$ abordada na seção II.

O checksum possui 16 bits, já que é calculado de 2 em 2 bytes, e o campo de TTL 8 bits. Se dividirmos o pacote IP em pedaços de tamanho 16 bits cada, o campo de TTL vai estar no byte mais alto de algum desses pedaços (juntamente com o número do protocolo do pacote, 0x06 para TCP por exemplo). Então, para realizarmos a operação $C' = C + (-m) + m'$, temos que atribuir aos bits mais significativos de m o valor do antigo TTL e preencher os bits restantes com 0. A mesma coisa deve ocorrer ao m' quando atribuirmos o TTL novo a ele. Isso é feito nas linhas:

```
assign ttl_old_16[15:8]=ttl_old;
assign ttl_old_16[7:0]=8'h00;
assign ttl_new_16[15:8]=ttl_new;
assign ttl_new_16[7:0]=8'h00;
```

O restante do código da figura 8 realiza a operação: $C' = C + (-m) + m'$.

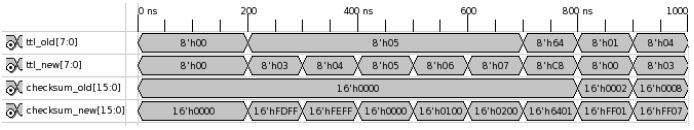

Fig. 9. Simulação do recalcipchecksum

A figura 9 mostra uma instância de simulação do código usando o ISE Simulator da Xilinx.

V. Comparação entre a implementação em C e em FPGA

Com o intuito de demonstrar que a implementação em hardware de funções para o processamento de pacotes IP em ambientes *wireless* é uma solução viável para economia de energia, estabelecemos uma linha comparativa que analisa a quantidade de ciclos necessários para o cálculo do incremento do checksum em software e em hardware.

Nos processadores modernos existe uma dificuldade em prever o tempo de execução de uma instrução devido ao cache de memória, agendamento de instruções e desvio de previsão. Para isso foi introduzida uma nova maneira de calcular os ciclos, através de um registro contador que se incrementa a cada ciclo de clock; com isso é possível determinar o tempo das instruções com extrema precisão[20].

Para calcular o número de ciclos de máquina, na arquitetura CISC, foi utilizada uma macro chamada `rdtscll`, encontrada em `asm/msr.h`. A sua finalidade é ler o valor no registrador e armazenar em uma variável de 64-bits. A preocupação com *overflow* não é necessária, visto que, em um processador 1 GHZ, o *overflow* ocorre a cada 4,2 segundos e o recálculo do checksum leva um tempo menor [20]. Para outras arquiteturas, o *kernel* do Linux possui a função `get_cycles` definida no arquivo `asm/timex.h` que calcula os ciclos de máquina para qualquer arquitetura; não sendo possível, retorna o valor zero[20]. Realizando testes com o programa descrito na subseção III-B, obteve-se 43 ciclos num processador CISC da família x86(AMD Athlon XP 2800+), enquanto em um processador RISC da familia PowerPC (MPC 659) levou apenas 6 ciclos.

Na implementação em Verilog do módulo `recalcipchecksum`, verificamos com as simulações realizadas no simulador ISE-Simulator da Xilinx que a variação dos níveis lógicos da saída do módulo em função das entradas variavam em intervalos de tempo menores que um ciclo de clock.

Em [21] é descrita uma técnica apurada para estimação de gastos de energia de um algoritmo; neste trabalho também é mencionado que o gasto de energia é normalmente proporcional ao número de ciclos de máquina. Através disso podemos concluir que a implementação em hardware gastará menos energia por precisar de menos ciclos de máquina.

Como a implementação em hardware leva 6 vezes menos ciclos de máquina (levando em consideração o processador PowerPC), ela poderá funcionar a um clock 6 vezes menor para alcançar o mesmo *wire speed*. Isso vai fazer com que seja dissipada menos energia com calor.

VI. Conclusão e trabalhos futuros

Com a evolução da ubiquidade dos dispositivos de comunicação móvel, há a demanda pela melhor utilização dos recursos de energia por parte dos seus componentes eletrônicos integrantes destes dispositvos. Concluímos, com base no que foi apresentado, que, com a diminuição da quantidade de ciclos gastos com o recálculo do checksum, obteve-se uma considerável economia de energia da CPU para o processamento de pacotes IP.

Como trabalho futuro, planeja-se fazer uma análise mais apurada do impacto do número de ciclos de máquina no gasto de energia de um processador. Planeja-se, também, comparar implementações em software e em hardware de outras funções de roteamento, tais como busca em tabela de encaminhamento (*forwarding table lookup*) e classificação de pacotes (*packet classification*), analisando seu impacto em gasto de energia ou velocidade em se tratando de redes *mesh*.

Agradecimentos

Este trabalho foi realizado com recursos da Pactec S/A e FUNCAP.

Referências

[1] I. Akyildiz, X. Wang, and W. Wang, "Wireless Mesh Networks: A Survey," *Computer Networks*, vol. 47, no. 4, pp. 445–487, 2005.

[2] R. Bruno, M. Conti, and E. Gregori, "Mesh networks: commodity multihop ad hoc networks," *Communications Magazine, IEEE*, vol. 43, no. 3, pp. 123–131, 2005.

[3] R. Braden, D. Borman, and C. Partridge, "RFC1071: Computing the Internet checksum," *Internet RFCs*, 1988.

[4] T. Mallory and A. Kullberg, "RFC1141: Incremental updating of the Internet checksum," *Internet RFCs*, 1990.

[5] A. Rijsinghani, "RFC1624: Computation of the Internet Checksum via Incremental Update," *Internet RFCs*, 1994.

[6] J. Touch and B. Parham, "RFC1936: Implementing the Internet Checksum in Hardware," *Internet RFCs*, 1996.

[7] M. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann Publishers San Francisco, CA, 2004.

[8] I. Koren, *Computer arithmetic algorithms*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1993.

[9] K. Hwang, *Computer Arithmetic: Principles, Architecture and Design*. John Wiley & Sons, Inc. New York, NY, USA, 1979.

[10] J. Postel, "RFC0791: Internet Protocol," *Internet RFCs*, 1981.

[11] ——, "RFC0793: Transmission Control Protocol," *Internet RFCs*, 1981.

[12] A. Tanenbaum, *Computer Networks*. Prentice Hall PTR, 2002.

[13] S. Deering and R. Hinden, "RFC 2460," *Internet Protocol, Version*, vol. 6, 1998.

[14] C. Kaminsky, D. Mariz, D. Sadok, and S. Fernandez, "Arquiteturas de Rede para a próxima geração da internet," *SBRC*, pp. 123–172, 2005.

[15] K. Salah and K. El-Badawi, "Evaluating system performance in Gigabit networks," *Local Computer Networks, 2003. LCN'03. Proceedings. 28th Annual IEEE International Conference on*, pp. 498–505, 2003.

[16] C. Benvenutti, *Understanding Linux Network Internals*. O'Reilly, 2006.

[17] J. Lockwood, N. Naufel, J. Turner, and D. Taylor, "Reprogrammable network packet processing on the field programmable port extender (FPX)," *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pp. 87–93, 2001.

[18] X. Inc., "Virtex-II Pro Platform FPGA: Complete Data Sheet," 2005.

[19] M. Barr, "Programmable logic: What's it to ya," *Embedded Systems Programming*, pp. 75–84, 1999.

[20] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*. O'Reilly, 2005.

[21] R. Jayaseelan, T. Mitra, and X. Li, "Estimating the Worst-Case Energy Consumption of Embedded Software," *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)-Volume 00*, pp. 81–90, 2006.