Felipe Gonçalves Assis

Relatório de Estágio: Modelagem Eficiente em SystemC

Campina Grande - PB
Abril de 2014

Felipe Gonçalves Assis

Relatório de Estágio: Modelagem Eficiente em SystemC

Orientador: Marcos Ricardo Alcântara Morais

Universidade Federal de Campina Grande

Campina Grande - PB
Abril de 2014

Felipe Gonçalves Assis
Relatório de Estágio: Modelagem Eficiente em SystemC/ Felipe Gonçalves Assis. – Campina Grande - PB, Abril de 2014-
25 p. : il. (algumas color.) ; 30 cm.

Orientador: Marcos Ricardo Alcântara Morais

Relatório de Estágio – Universidade Federal de Campina Grande, Abril de 2014.

1. Hardware. 2. Modelagem. I. Marcos Ricardo Alcântara Morais. II. Universidade Federal de Campina Grande. III. Centro de Engenharia Elétrica e Informática. IV. Relatório de Estágio: Modelagem Eficiente em SystemC

CDU 621.38

Felipe Gonçalves Assis

Relatório de Estágio: Modelagem Eficiente em SystemC

Trabalho aprovado. Campina Grande - PB, 25.04.2014:

Professor Marcos Ricardo Alcântara Morais
Universidade Federal de Campina Grande
Orientador

Professor Avaliador
Universidade Federal de Campina Grande
Avaliador

Campina Grande - PB
Abril de 2014

Lista de ilustrações

Sumário

1 Introdução

Este trabalho tem como intuito descrever as atividades realizadas no âmbito profissional durante o Estágio Integrado, como parte indispensável para a formação acadêmica em Engenharia Elétrica.

O estágio foi realizado durante o período de 13/06/2013 a 11/04/2014 na Freescale Semicondutores do Brasil LTDA, precisamente no BSTC (Brazil Semiconductor Technological Center), com carga horária de 40 horas semanais e atendendo aos requisitos previstos na Resolução 01/2012 do Colegiado do Curso de Graduação de Engenharia Elétrica e em consonância com a Lei do Estágio (Lei 11.788/2008).

1.1 A Empresa

1.1.1 Freescale no Mundo

A Freescale atua em soluções para processamento embarcado nos mercados automotivo, de consumo, industrial e de redes. Seus produtos incluem microcontroladores, microprocessadores, sensores e circuitos integrados analógicos.

Fundada em 2004 como um *spin off* da Motorola Inc., a Freescale tem sua matriz localizada em Austin, Texas, nos Estados Unidos. Ela também possui filiais – operando em projeto, pesquisa, desenvolvimento, manufatura e vendas – em mais de 20 países (ver Figura 1), com um total de mais de 18.000 funcionários.

1.1.2 Freescale no Brasil

A Freescale Brasil iniciou suas operações em 1967, quando ainda atuava sob a direção da Motorola Inc., que apostou na capacidade intelectual e de inovação dos engenheiros locais.

A região de Campinas foi escolhida por sua localização estratégica e pela disponibilidade de mão de obra altamente especializada além da proximidade das melhores universidades do país.

Atualmente o BSTC conta com mais de 130 funcionários (95% engenheiros) contabilizando mais de 100 projetos entregues.

Desde outubro de 2007, a Freescale Brasil passou a operar em uma sede própria localizada no condomínio Technopark em Campinas.

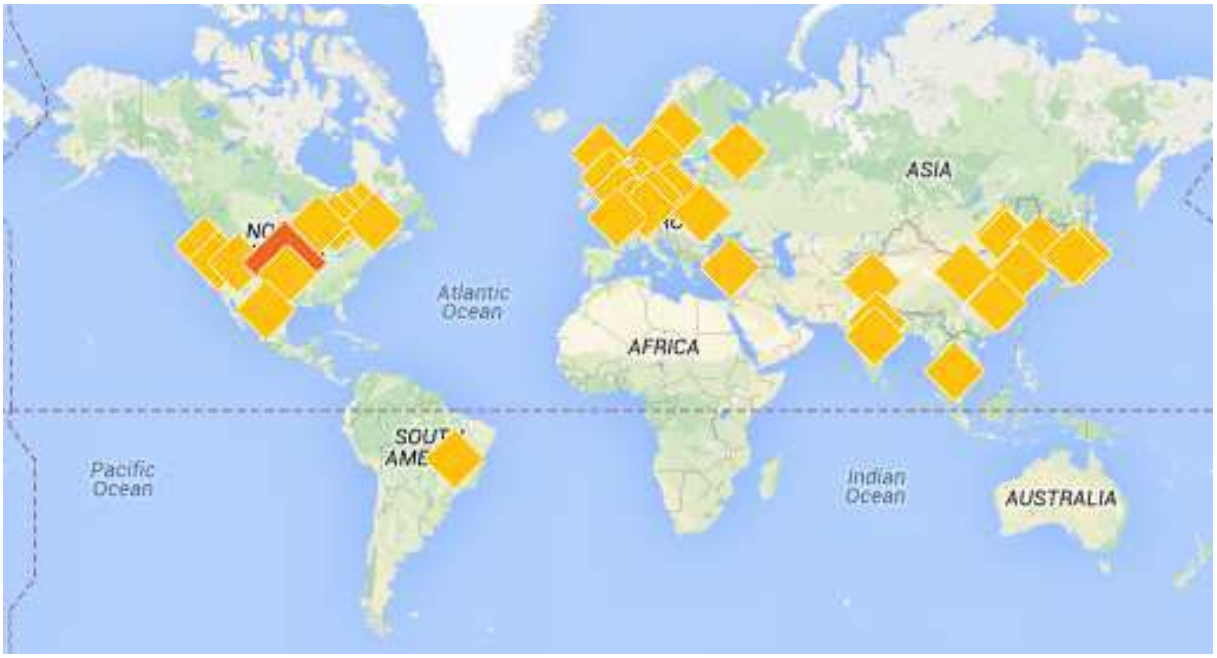

Figura 1 – Filiais da Freescale (figura retirada do *site* oficial).

O BSTC atua sobretudo como uma *Design House*, desenvolvendo projetos de *IP-cores* e SoCs (*System on a Chip*). Os chips projetados são produzidos em *foundries* fora do país e comercializados globalmente.

Baseado no histórico de desempenho do *Design Center* brasileiro, alguns dos principais projetos mundiais da Freescale são desenvolvidos localmente e sempre com excelentes resultados. Baseado nesses resultados, desde 2007 o Design Center do Brasil passou a ter importância estratégica na empresa adquirindo o status de Premier Design Center.

A área comercial da empresa também tem presença significativa na América do Sul, tendo o Brasil como base de operações. A equipe comercial conta com gerentes de vendas e engenheiros de aplicação que juntos, oferecem aos clientes soluções completas de suporte e acompanhamento que se inicia no conceito do projeto, passa pelo desenvolvimento propriamente dito do mesmo e culmina com o fornecimento dos dispositivos semicondutores via canais competentes de distribuição.

2 Tarefa Proposta

2.1 Área de Atuação

As atividades de estágio foram realizadas no contexto da equipe de verificação de IPs digitais.

Os projetos na moderna indústria de semicondutores se caracterizam por uma alta complexidade. Ao mesmo tempo, os custos de um erro de projeto são muito altos. Em algumas áreas, como em aplicações automotivas, médicas ou espaciais, certas falhas são inaceitáveis. Mesmo no mercado de consumo, a larga escala de produção necessária para tornar os produtos economicamente viáveis também significa que o custo de uma refabricação é freqüentemente inaceitável.

Isso torna a área de verificação de *hardware* particularmente relevante, tipicamente consumindo a maior parte dos recursos de um projeto, e por isso mesmo alvo de pesquisa e desenvolvimento, tanto na academia como na indústria.

Um dos métodos de verificação consiste em simular um módulo sintetizável escrito em uma *linguagem de descrição de hardware* (HDL, na sigla em inglês) e comparar seu comportamento ao de um *modelo de referência*, um componente de *software* codificando a especificação do módulo. Daí a relevância da modelagem de *hardware* para verificação.

Modelos e simulações também são úteis durante o trabalho de especificação de um IP, para *exploração arquitetural*, ou seja, o estudo do efeito de diferentes decisões arquiteturais na funcionalidade e eficiência do módulo. Além disso, modelos também são úteis em aplicações, quando se queira simular um sistema que tem o IP como elemento.

Três características, freqüentemente conflitantes, são desejáveis de um modelo:

1. Precisão: O modelo deve refletir a especificação com suficiente detalhamento;
2. Abstração: É preferível que o modelo esteja codificado em alto nível;
3. Eficiência: É desejável que se possam executar simulações complexas em pouco tempo;

A propriedade de abstração deriva sua desejabilidade das necessidades de correção e mantenibilidade. Códigos em alto nível tendem a conter menos erros e são mais fáceis de depurar e refatorar.

Abstração e eficiência não são tão antagônicas. Freqüentemente, elevar o nível de abstração aumenta as possibilidades de otimização. Isso é particularmente notável na modelagem de *hardware*.

Talvez o método mais promovido atualmente de elevar os níveis de abstração e eficiência de modelos de *hardware* seja a *Modelagem Baseada em Transações*, ou *Modelagem em Nível de Transação* (TLM, na sigla em inglês), associada historicamente à biblioteca C++ de modelagem e simulação denominada SystemC.

A ideia geral em TLM consiste em abstrair detalhes não-essenciais da comunicação entre módulos, permitindo que o modelo se concentre em suas características funcionais essenciais. Detalhes do *timing* dos sinais são comumente abstraídos, parcial ou totalmente.

Isso, contudo, não é útil na modelagem de blocos cuja funcionalidade essencial está intrinsecamente associada ao *timing* dos eventos, como no caso de temporizadores e controladores de E/S.

Em (2), Schnieringer e Brand descrevem como se pode escrever modelos eficientes em SystemC preservando a precisão de ciclo de clock. As técnicas se aplicam na verdade a qualquer infraestrutura de simulação baseada em eventos discretos.

A atividade realizada no estágio foi basicamente demonstrar a aplicabilidade dessas técnicas na modelagem de um IP real no portfólio do BSTC, mais especificamente, a eTPU (1).

A ideia partiu da troca de experiências inicial ocorrida nos primeiros dias de estágio, quando se questionou a impossibilidade de uma modelagem mais eficiente e em maior nível de abstração sem comprometer a precisão de ciclo de clock essencial à funcionalidade de certos IPs.

Os benefícios dessa abordagem podem ser aproveitados não só para a verificação, mas sobretudo para exploração arquitetural em projetos futuros. Também se espera que clientes possam aproveitar os modelos rápidos no desenvolvimento de aplicações.

3 Materiais e Métodos

Para avaliar as possibilidades e limitações das técnicas propostas, escreveram-se modelos alternativos de blocos de um IP já desenvolvido: a eTPU (1), os quais foram validados contra seu código HDL já verificado. Mais especificamente, queria-se julgar:

1. Um modelo correto e eficiente é de fato viável?

2. Quão complicada é a depuração de um modelo assim escrito?

3. Quão difícil é modificar o modelo em face de pequenas alterações na especificação?

Em essência, queria-se entender como as técnicas propostas afetariam a dinâmica de um projeto real.

3.1 SystemC

A ferramenta básica de trabalho foi a biblioteca C++ de modelagem e simulação denominada SystemC (3), a qual permite a escrita de modelos e simulações baseadas em eventos discretos.

3.2 A eTPU

A eTPU(1) (*Enhanced Time Processing Unit*) é um controlador de E/S com *core* e memória próprios, que permite a realização de tarefas complexas de temporização e gerenciamento de E/S independentemente da CPU. Ela encontra aplicação em sistemas onde se exigem baixa latência e alta precisão. Exemplos são comunicação serial e controle de motor.

Seu diagrama de blocos pode ser visto na Figura 2.

Da eTPU foram modelados primeiramente seus canais, e em seguida o contador de ângulo (EAC, *eTPU Angle Counter*). O canal foi escolhido como primeiro objeto de modelagem por sua simplicidade. Demonstrado o sucesso desse primeiro modelo, o EAC foi experimentado por oferecer várias sutilezas de *timing*, tornando o processo de depuração mais desafiador.

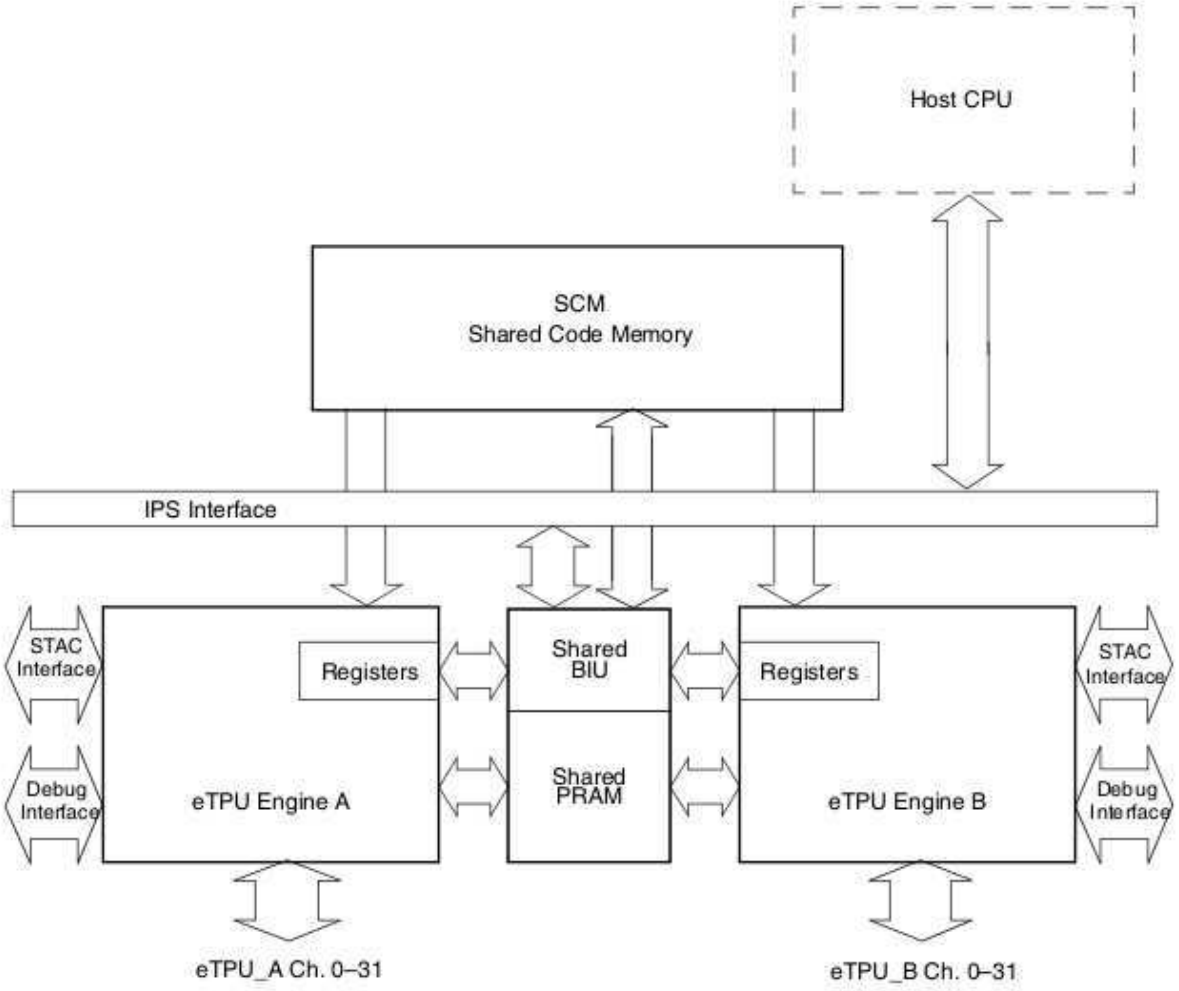

Figura 2 – Diagrama de blocos da eTPU (retirado de (1)).

3.2.1 Canais da eTPU

A eTPU possui duas *engines*, cada uma com 32 canais independentes. O diagrama de blocos de uma *engine* está na Figura 3.

Os canais têm acesso dois contadores, TCR1 e TCR2, os quais podem ser comparados por comparadores de igualdade ou de maior-ou-igual a valores programados para produzir eventos de *match*.

Cada canal tem um sinal (um bit) de entrada. Diferentes transições desse sinal podem ser detectadas.

Cada canal também tem um sinal (um bit) de saída.

Os canais podem ser programados para responder autonomamente a variadas combinações de até dois eventos de transição e dois *matches*. Respostas podem incluir capturas dos contadores TCR1 e TCR2, ações sobre o sinal de saída, e a geração de sinais de interrupção para a *microengine* da eTPU.

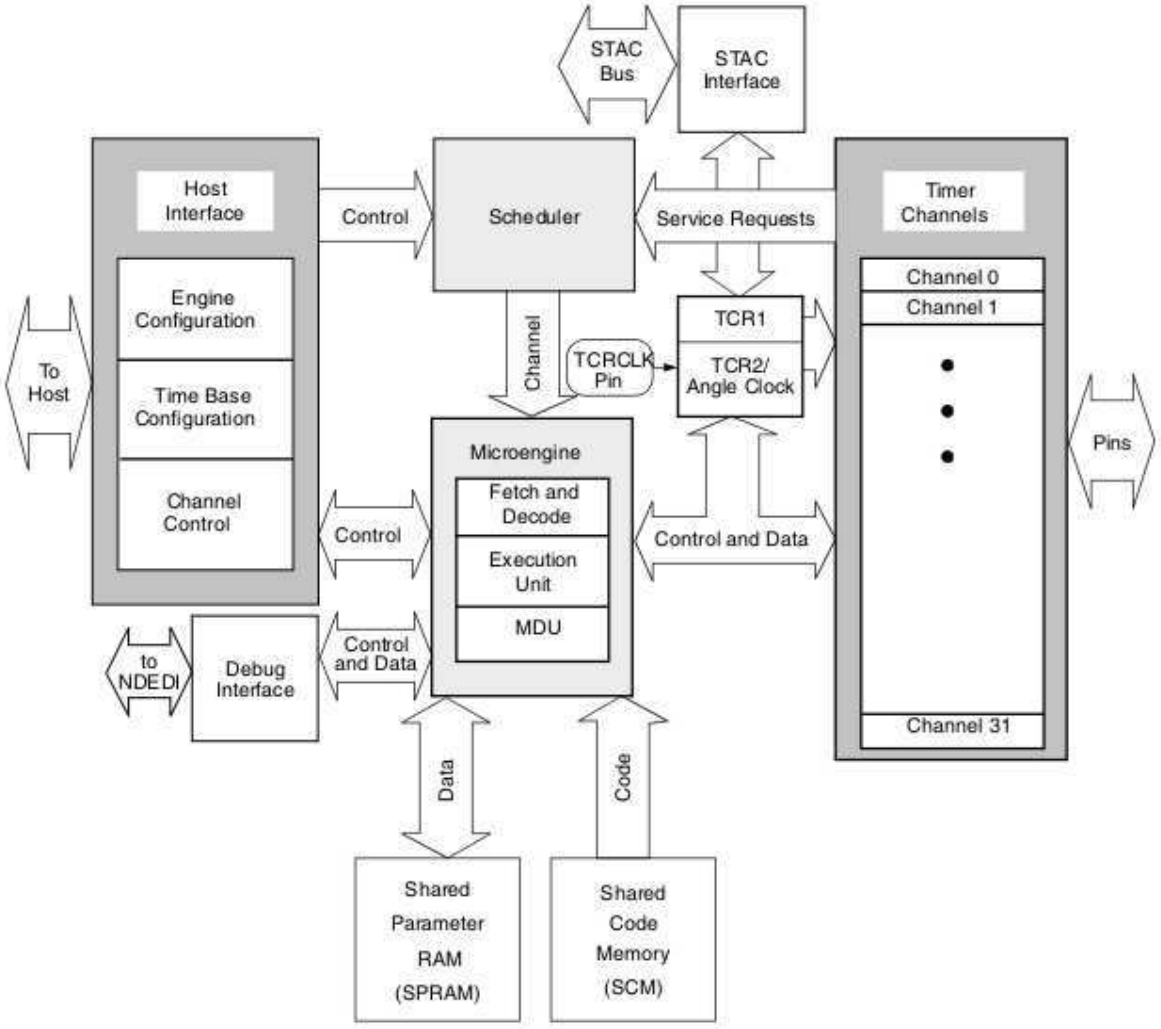

Figura 3 – Diagrama de blocos de uma *engine* da eTPU (retirado de (1)).

3.2.2 EAC

O contador de ângulo da eTPU (EAC) é uma lógica para estimação de ângulo (fase) de uma roda dentada a partir da detecção dos dentes. Essencialmente, serve para a implementação de um PLL digital.

A taxa de incremento do contador é atualizada por *software*, enquanto um *hardware* dedicado implementa a lógica para incremento do contador com período, em média, fracionário, e para automaticamente acelerar ou parar a contagem adequadamente em casos de aceleração e desaceleração (ou seja, quando os dentes são detectados antes ou depois do esperado).

Alguns outros recursos também são disponibilizados, como o tratamento automático de dentes faltando.

3.3 Princípios de Modelagem

A inspiração básica para o trabalho foi (2). Aqui descrevemos nossa visão das ideias propostas naquele trabalho.

Conforme enunciado em (2), o objetivo básico é economizar na geração de eventos, e minimizando o número de subrotinas chamadas ao longo de uma simulação. Na nossa implementação, isso é alcançado por duas vias:

1. A codificação no modelo não só de seu estado presente, mas também de seu futuro.

2. Sistemas síncronos são representados num estilo não-convencional, que evita a geração de eventos quando o sistema está estacionário;

Isso deverá ser melhor compreendido com exemplos concretos dados mais adiantes. Antes, examinaremos os conceitos fundamentais codificados na implementação.

Ao longo dos trabalhos, observou-se um processo contínuo de amadurecimento segundo o qual ideias intuitivas foram sendo traduzidas em padrões de codificação, que forma sendo reconhecidos conscientemente e gradativamente formalizados em uma biblioteca de propósito geral utilizada na modelagem.

São essas ideias amadurecidas que apresentamos aqui. A organização lógica da apresentação não reflete a ordem cronológica dos desenvolvimentos.

3.4 Clocks e Timers

O elemento mais fundamental da biblioteca é o conceito de um clock abstrato.

Para nós, um *clock abstrato*, ou simplesmente clock, é simplesmente um conjunto de instantes futuros, chamados de *tiques*. Os intervalos fechados à esquerda e abertos à direita entre tiques adjacentes são chamados *ciclos*. Um clock pode ser visto equivalentemente como uma partição da linha do tempo em ciclos.

Estendendo o conceito de clock, temos o conceito de *timer*. Para nós, um timer é simplesmente um clock munido de uma função que atribui a cada ciclo um valor, dito a *contagem* do timer. O tipo desse valor é, a princípio, arbitrário. Os tiques de um timer podem ser vistos como os instantes em que sua contagem pode mudar.

Os tiques de um clock abstrato podem representar, por exemplo, as bordas ativas de um clock real, ou os instantes em que a contagem de um prescaler volta a 0. Os tiques não precisam ser igualmente espaçados. Um clock com ciclos de tamanho variável ocorre na modelagem do EAC.

Timers abstratos representam uma previsão para o futuro de um sinal. Contadores cíclicos são o objeto típico a ser modelado por um timer.

Timers não produzem eventos de mudança de valor a cada tique, e sim quando a previsão que codificam muda. Essa é a primeira técnica de economia de eventos.

Similarmente, clocks só produzem eventos de mudança de valor quando mudam os tiques futuros que preveem.

O processo de modelar um sinal como um timer pode ser visto assim: Seu estado é representado não só pelos atributos de uma classe, mas também pelo tempo de simulação. Enquanto as mudanças de valor se traduzem simplesmente por avanços no tempo, não precisamos gerar evento.

3.5 Notificadores

Notificadores são classes auxiliares que podem ser programadas para agendarem eventos de acordo com um clock ou um timer. Para clocks, o notificador básico gera eventos para o próximo tique, a partir do instante em que é solicitado. Para timers, o notificador básico gera eventos para o próximo *match*, ou seja, o próximo tique no qual o timer atinge um dado valor.

Notificadores encapsulam o trabalho de reagendar os eventos quando o clock ou timer muda. Graças a eles, coisas como *clock gating* e reprogramação de temporizadores podem ser simuladas sem nenhuma sobrecarga no código do modelo.

3.6 Sinais Síncronos

Sinais síncronos são a principal aplicação de clocks abstratos nos modelos desenvolvidos. Um sinal síncrono pode ser comparado a um `sc_signal` de SystemC. Enquanto um `sc_signal`, após uma escrita, só muda de valor no delta seguinte da simulação, um sinal síncrono só muda de valor no ciclo seguinte do clock ao qual está associado, gerando eventos de mudança de valor no tique correspondente.

Um sinal síncrono possui as seguintes interessantes características formais:

1. Seu valor é função do tempo simulado. Um sinal síncrono só muda de valor quando o tempo simulado avança, nunca instantaneamente nem quando a simulação avança apenas um delta.

2. Seus eventos de mudança de valor são sempre temporizados. Ele é sempre agendado para um tempo simulado futuro, nunca instantaneamente ou para o próximo delta.

Note como são análogas às propriedades de um `sc_signal`, cujo valor é função do delta da simulação e cujos eventos de mudança de valor são sempre notificados para o próximo delta.

Sinais síncronos permitem uma estratégia alternativa de modelagem de sistemas síncronos.

Tradicionalmente, modelamos sistemas síncronos gerando eventos para cada borda ativa do clock. Todos as processos são ativados por esses eventos. Usamos `sc_signal`s

para deferir as escritas para o próximo delta de simulação, mas, como só lemos o sinal na borda ativa seguinte, é só aí que essa mudança é efetiva.

Sinais síncronos encapsulam esse princípio ao só gerarem o evento de mudança no ponto em que ele é efetivo, quando são amostrados pela lógica síncrona.

Podemos então fazer os processos serem sensíveis não ao evento do clock, mas aos sinais de entrada, exatamente como na modelagem de lógica combinacional. Se os sinais de entrada forem síncronos, suas mudanças só ativarão os processos nas bordas ativas do clock, como é esperado.

A vantagem é que, quando as entradas não mudam, os processos não precisam ser ativados. Um sistema estacionário não produz eventos, e cálculos redundantes não precisam ser executados a cada borda de clock. Essa é a segunda técnica de economia de eventos.

A Listagem 3.1 ilustra o princípio. O sinal síncrono `value` é dividido por dois a cada pulso de clock. Ao atingir o ponto fixo de 0, a escrita deixa de gerar eventos de mudança e o *callback* `update()` deixa de ser chamado.

Diferentes tipos de clock representando diferentes padrões de tiques podem ser conectados à porta `clk`. O usuário também não precisa se preocupar com mudanças no valor do clock ao longo da simulação, como um *gating*, por exemplo.

A linha em destaque é o que mais caracteriza esse estilo de modelagem. Tradicionalmente, o bloco seria sensível a algum evento de clock, e eventos seriam gerados indefinidamente, sempre ativando o *callback* `update()`, mesmo sem efeito sobre o estado do sistema.

A Listagem 3.2 ilustra a aplicação do princípio de codificação do futuro de um modelo. Após a inicialização, o sinal `value` muda de valor sempre que o timer conectado à porta `timer` atinge a contagem 42. A contagem do timer é do tipo `unsigned`.

Novamente, diferentes tipos de timer, com diferentes padrões de tiques e de contagem, podem ser conectados à porta `timer`. O usuário também não precisa se preocupar com mudanças no timer ao longo da simulação, como um reset, por exemplo.

Os únicos eventos gerados aqui serão para os *matches* (sempre em 42) e para as mudanças no sinal `value` que ocorrem em seguida. O timer de entrada, claro, também pode gerar eventos, mas só ao ser reconfigurado. Como dito, o usuário nunca lida diretamente com esses eventos, eles apenas servem para que os notificadores recalculem os eventos agendados.

Num modelo tradicional, eventos seriam gerados para cada mudança de contagem do timer, em vez de apenas para as que interessam (quando ele atinge 42). O callback `update()` teria que ser reescrito para sempre verificar se o *match* foi atingido.

Observação: Note que, em geral, seja pelo uso de sinais síncronos ou timers, uma vez que deixamos de nos basear unicamente em eventos incondicionais de borda ativa de clock, as listas de sensibilidade se tornam parte importante da lógica do modelo.

Listagem 3.1 – Ilustração de modelagem alternativa de sistemas síncronos.

```
class automaton : public sc_core::sc_module
{
public:
    clock_in_port clk;

    synchronous_signal<int> value;

    SC_HAS_PROCESS(automaton);
    automaton(sc_core::sc_module_name name) :
        sc_core::sc_module(name),
        clk("clk"),
        value("value", 42) // Valor inicializado com 42.
    {
        SC_METHOD(update);
        sensitive << value; // <<<<<<<<<<<<<<<<<<<<<<<<<<<<

        value.clk.bind(clk);
    }

    static int f(int x)
    { return x / 2; }

    void update()
    {
        // value = f(value);
        value.write(f(value.read()));
    }
};
```

3.6.1 Reset Automático

Uma conveniência desenvolvida foram os sinais de reset automático. Esses sinais têm uma porta de entrada para o sinal de reset e resetam automaticamente para um valor configurado em sua construção. Seu uso sistemático evita a necessidade de reset explícito, suscetível a erros.

3.7 Composição de Clocks

Já vimos como o uso de clocks e timers abstratos está por trás das técnicas de modelagem eficiente que empregamos. Nesta seção apresentamos uma ideia que se mostrou útil na implementação de clocks.

O conceito de composição clocks aparece quando os tiques de um clock são naturalmente subconjunto dos tiques de outro. Equivalentemente, os ciclos de um clock são

Listagem 3.2 – Ilustração de codificação do futuro de um modelo.

```
class toggler : public sc_core::sc_module
{
public:
    timer_in_port<unsigned> timer;

    synchronous_signal<bool> value;
    match_notifier<unsigned> noti;

    SC_HAS_PROCESS(toggler);
    toggler(sc_core::sc_module_name name) :
        sc_core::sc_module(name),
        timer("timer"),
        value("value", false),
        noti("noti")
    {
        SC_METHOD(update);
        sensitive << noti.event();

        value.clk.bind(timer);
        noti.timer.bind(timer);
    }

    void update()
    {
        // value = !value;
        value.write(!value.read());
        noti.notify(42);
    }
};
```

reuniões dos ciclos de outro.

Para modelar esse tipo de situação, há uma certa representação de clocks que é particularmente conveniente.

A ideia é que numeramos os ciclos de um clock sequencialmente. A numeração exata é arbitrária. O essencial é que ciclos sucessivos correspondam a ciclos sucessivos.

A representação consiste então de um par de funções: $\mathbf{c}(\cdot)$ e $\mathbf{t}[\cdot]$. A primeira associa a cada instante de tempo t um *índice de ciclo* $\mathbf{c}(t)$. A segunda é na verdade determinada pela primeira, associando a cada índice de ciclo c o tique $\mathbf{t}[c]$ correspondente a seu início. Em outras palavras,

$$\mathbf{t}[c] = \min(\mathbf{c}^{-1}(c)).$$

Em particular, $\mathbf{t}[\cdot]$ é uma inversa à direita de $\mathbf{c}(\cdot)$.

Note que, a partir dessas duas funções, podemos, por exemplo, construir a função

"próximo tique" como

$$\texttt{next\_tick}(t) = \mathbf{t}[\mathbf{c}(t) + 1].$$

A ideia da composição é que os índices de ciclo de um clock (o *clock interno*) servem como a variável tempo de outro (o *clock externo*). Se a função índice de ciclo do clock interno é representada por $\mathbf{c}(\cdot)$ e a do externo por $\mathsf{k}(\cdot)$, a do *clock composto* é simplesmente $\mathsf{k}(\mathbf{c}(\cdot))$.

Conseqüentemente, se a função tique do clock interno é $\mathbf{t}[\cdot]$ e a do externo $\mathsf{c}[\cdot]$, a do clock composto é simplesmente $\mathbf{t}[\mathsf{c}[\cdot]]$.

Em outras palavras, o índice de ciclo no instante t é $\mathsf{k}(\mathbf{c}(t))$, e o tique correspondente ao ciclo de índice k é $\mathbf{t}[\mathsf{c}[k]]$.

Duas funções índice de ciclo $\mathbf{c_1}(\cdot)$ e $\mathbf{c_2}(\cdot)$ representam o mesmo clock se e somente se $\mathbf{c_1}(t) - \mathbf{c_2}(t)$ independe de t.

4 Resultados

As técnicas descritas no capítulo anterior foram aplicadas primeiro na modelagem dos canais da eTPU e depois na do EAC.

4.1 Canal

Um canal da eTPU é uma máquina de estados quase sempre estacionária. O estado só muda em resposta a eventos bem específicos: Transições no sinal de entrada, *matches* e comandos da *microengine*. Graças ao estilo de modelagem de sistemas síncronos descrito acima, o código de atualização só é executado na ocorrência desses eventos.

A isso se soma a modelagem dos TCR1 e TCR2 como timers, de modo que o seu incremento costumeiro também não gera eventos. Eventos de *matches* são agendados para os instantes esperados, e atualizados em caso de reprogramação.

Para validar o modelo, ele foi enxertado no ambiente de simulação utilizado na verificação funcional. Monitores e comparadores específicos foram usados para comparar o modelo alternativo contra o código RTL já verificado que, no caso, serviu de modelo de referência. Os estímulos existentes foram aproveitados, e todos os testes de regressão foram executados.

Apesar do sucesso, avalia-se que, quando se trata de validar tal modelo, os estímulos típicos da verificação, justamente por serem ricos em eventos concorrentes, podem não ser suficientes para obter uma boa cobertura.

Um erro que poderia passar despercebido, por exemplo, é a falta de um sinal de entrada na lista de sensibilidade de um processo. Esse erro não seria cometido num estilo tradicional de modelagem, onde os processos são invariavelmente ativados a cada ciclo de clock. Esse erro poderia ser evitado se pudéssemos garantir que um processo só lê sinais aos quais é sensível (algo como o `always@(*)` de Verilog-2001).

Maiores investigações sobre o assunto ainda não foram feitas.

4.2 EAC

O estado do EAC, assim como o do canal, também muda apenas em resposta a eventos esporádicos. No caso, esses eventos são principalmente a detecção de um dente, o timeout da espera por um dente, e comandos da *microengine*.

Foi aqui onde se elaborou a ideia de composição de clocks. Os tiques gerados para contagem de ângulo são modelados como a composição de três clocks: o clock básico da

eTPU, o gerador de base de tempo para o EAC e, finalmente, a lógica de clock de período fracionário.

Este último é um exemplo de clock com espaçamento variável entre tiques. Apesar dessa aparente peculiaridade, todos os clocks que modelamos seguem as mesmas equações:

$$\mathbf{c}(t) = \left\lfloor \frac{t - t_0}{T} \right\rfloor$$

$$\mathbf{t}[c] = \lceil t_0 + cT \rceil$$

A única particularidade do EAC é que os parâmetros t_0 e T são fracionários.

Esses parâmetros não aparecem explicitamente na interface do modelo, que está escrita no vocabulário da especificação. Internamente é feito o mapeamento entre o estado dos registradores e os parâmetros. (O parâmetro T é comum aos dois vocabulários.)

À época do final do estágio, o modelo foi apenas previamente validado por testes direcionados, demonstrando todas as funcionalidades básicas.

5 Considerações Finais

Apesar de o projeto de estágio ter sido predominantemente individual, a oportunidade foi largamente aproveitada para o aprendizado sobre os desafios enfrentados na área de verificação de *hardware* em uma empresa de relevância no setor de semicondutores.

Ao mesmo tempo, a atividade foi largamente construtiva por ter permitido a aplicação de conhecimentos adquiridos durante a graduação no contexto de um projeto comercial real, com todas as complexidades implicadas. Foi necessário, por exemplo, desenvolver infraestrutura para suportar clocks dinâmicos. Esse tipo de necessidade levou ao desenvolvimento de biblioteca e métodos muito mais gerais do que talvez fossem concebidos num contexto puramente acadêmico.

Particularmente relevantes às atividades de estágio foram os conhecimentos adquiridos na disciplina de Arquitetura de Sistemas Digitais e em projetos realizados em paralelo durante o curso, mais especificamente, o projeto ALTATV, que forneceu a familiaridade com modelagem em SystemC, e o projeto Brazil-IP, fonte dos conhecimentos na área específica de verificação.

A empresa concedente teve papel determinante, tanto na definição inicial da tarefa, como no apoio no uso de ferramentas e na compreensão dos blocos modelados.

Referências

1 FREESCALE SEMICONDUCTOR, INC. *Enhanced Time Processing Unit (eTPU) Preliminary Reference Manual*. [S.l.], 2004.

2 SCHNIERINGER, M.; BRAND, K. *SystemC: Key modeling concepts besides TLM to boost your simulation performance*. `http://www.design-reuse.com/articles/17877/systemc-tlm.html`. Acessado em 22.04.2014.

3 International Standard, IEEE Standard for Standard SystemC Language Reference Manual. [S.l.]: IEEE Computer Society, New York, USA, Janeiro 2012.