UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Chrystian de Sousa Guth

ANÁLISE DE *TIMING* ESTÁTICA E A AVALIAÇÃO DO IMPACTO DO ATRASO DAS INTERCONEXÕES EM CIRCUITOS DIGITAIS

Florianópolis - Santa Catarina

2013

Chrystian de Sousa Guth

ANÁLISE DE *TIMING* ESTÁTICA E A AVALIAÇÃO DO IMPACTO DO ATRASO DAS INTERCONEXÕES EM CIRCUITOS DIGITAIS

Trabalho de Conclusão de Curso submetido ao Curso de Bacharelado em Ciências da Computação para a obtenção do Grau de Bacharel em Ciências da Computação.
Orientador: M.Sc. Vinícius dos Santos Livramento
Coorientador: Prof. Dr. José Luís Almada Güntzel

Florianópolis - Santa Catarina

2013

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Guth, Chrystian
Análise de Timing Estática e a Avaliação do Impacto do Atraso das Interconexões em Circuitos Digitais / Chrystian Guth ; orientador, Vinícius dos Santos Livramento ; co-orientador, José Luís Almada Güntzel. - Florianópolis, SC, 2013.
268 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico. Graduação em Ciências da Computação.

Inclui referências

1. Ciências da Computação. 2. Análise de Timing Estática. 3. Standard Cell. 4. Automação de Projeto Eletrônico. 5. Atraso de Interconexões. I. dos Santos Livramento, Vinícius. II. Almada Güntzel, José Luís. III. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. IV. Título.

Chrystian de Sousa Guth

ANÁLISE DE *TIMING* ESTÁTICA E A AVALIAÇÃO DO IMPACTO DO ATRASO DAS INTERCONEXÕES EM CIRCUITOS DIGITAIS

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Bacharel em Ciências da Computação", e aprovado em sua forma final pelo Curso de Bacharelado em Ciências da Computação.

Florianópolis - Santa Catarina, 09 de Dezembro 2013.

Prof. Dr. Renato Cislaghi
Coordenador

Banca Examinadora:

M.Sc. Vinícius dos Santos Livramento
Orientador

Prof. Dr. José Luís Almada Güntzel
Coorientador

Dr. Renan Alves Fonseca

À minha família.

AGRADECIMENTOS

À minha mãe, Ieda, pelo amor, apoio e dedicação que nunca faltaram. Também aos meus irmãos, Ralf e Elis Regina, pela força e confiança nesses 4 anos de graduação.

Agradeço à minha namorada Lígia pelo amor, compreensão e paciência, principalmente nos últimos meses em que me dediquei a este trabalho.

Ao meu orientador, Vinícius dos Santos Livramento, pela confiança, dedicação e aprendizado proporcionado desde o início de 2011, em que fui seu assistente, até a conclusão deste trabalho. Agradeço também por sua excelente orientação e rigor exigido, os quais foram fundamentais para o sucesso deste trabalho de conclusão de curso.

Ao meu coorientador, professor José Luís Almada Güntzel, pela grande colaboração, que muito contribuiu para a conclusão deste trabalho.

Ao membro da banca, Renan Alves Fonseca, pelo tempo dedicado para uma revisão rigorosa e pelas sugestões que contribuíram com este trabalho.

Aos colegas da graduação André Camargo e Cláudio Dettoni, Gabriel Gava, Lucas Pereira, Renan Netto e demais colegas do ECL que de alguma forma participaram deste trabalho.

Ao CNPq pelo custeio parcial da execução deste trabalho, com bolsa na modalidade de iniciação tecnológica (Processo número: 182980/2013-8).

RESUMO

Análise de *timing* estática (*STA: Static Timing Analysis*) é a técnica mais utilizada para estimar o atraso de circuitos digitais durante o fluxo de síntese física. Com o advento das tecnologias CMOS (Complementary Metal-Oxide Semiconductor) nanométricas, o atraso das interconexões passou a ser dominante em relação ao atraso das portas lógicas e por este motivo, não pode mais ser desprezado. A técnica de Elmore, baseada no primeiro momento da resposta ao impulso é amplamente utilizada para se calcular os atrasos das interconexões, porém, pode ser imprecisa por desconsiderar o efeito de *resistive shielding*. Algumas técnicas modificam a técnica de Elmore, a fim de contornar o problema do efeito de *resistive shielding*, obtendo resultados mais precisos, mantendo um baixo custo computacional. A consideração do efeito de *resistive shielding* requer a implementação de uma técnica para obtenção da capacitância efetiva em cada segmento da interconexão, impactando também no atraso da porta lógica que a interconexão está ligada na saída (*driver*). A ferramenta de STA implementada neste trabalho realiza o cálculo dos atrasos das interconexões gerando resultados que são, em média, $4,28\%$ mais otimistas do que aqueles gerados pela ferramenta Synopsys PrimeTime, porém com tempo de execução cerca de 8 vezes menor.

Palavras-chave: Automação de Projeto Eletrônico (EDA), Biblioteca *Standard Cell*, Análise de *Timing* Estática *(STA)*, *Complementary Metal-Oxide Semiconductor*

ABSTRACT

Static timing analysis is the most used technique to calculate the critical path in digital circuits during the standard cell design flow. Since feature size of CMOS devices are reducing, the interconnect delay becomes much more significant than before. Elmore delay model, based on the first moment of the impulse response is widely used to compute the interconnect delay, but, the technique doesn't consider the effect called resistive shielding. Some techniques modify the Elmore delay model in order to workaround the problem caused by the resistive shielding effect, getting more acurately results, keeping a low computational cost. The consideration of the resistive shielding effect requires the implementation of a technique to obain the effective capacitance value in each interconnect segment. This effective capacitance value impacts in the driver delay and slew. The STA tool implemented in this work does the interconnect delay calculation, getting results that are, in average 4.28% optimistics than those that are obtained by an industrial tool, with 8 times less runtime.

Keywords: Electronic Design Automation (EDA), Standard Cell Library, Static Timing Analysis, Complementary Metal-Oxide Semiconductor

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

LISTA DE SÍMBOLOS

$d_{j \to i}$ O *delay* do *timing arc* que vai do pino v_j até o pino v_i 61

$slew_{j \to i}$
O *slew* do *timing arc* que vai do pino v_j até o pino v_i 61

$iD_{i \to k}$ O atraso de propagação na interconexão que liga o pino v_i até o pino v_k. No modelo de capacitância concentrada, $iD_{i \to k} = 0$ 61

$iS_{i \to k}$ Degradação do *slew* através da interconexão que liga v_i em v_k 61

$fanouts(i)$
Conjunto dos pinos que são destino da interconexão para qual v_i é *driver* 61

r_i É o *required time* no *timing point* v_i. O *required time* é o tempo máximo que o valor de a_i pode assumir para que a restrição de desempenho seja respeitada. Se v_i é uma saída primária do circuito, então $r_i = T$, onde $f = \frac{1}{T}$ é a frequência mínima de operação do circuito digital 61

$slack_i$ Folga de tempo no ponto v_i, ou seja, quanto o *arrival time* pode atrasar neste ponto, de modo que o período máximo continue sendo respeitado. Se em um determinado ponto do circuito o *slack* é negativo, então o caminho em questão está violando a restrição de atraso máximo do sistema 61

SUMÁRIO

1 INTRODUÇÃO

Este capítulo tem por objetivo, apresentar uma visão geral sobre o fluxo de projeto *standard cell* e a importância da análise de *timing* estática no desenvolvimento de circuitos digitais. Serão apresentadas também a motivação e a justificativa deste trabalho.

1.1 FLUXO DE PROJETO *STANDARD CELL*

O crescimento da complexidade dos circuitos digitais contemporâneos[1] e a necessidade de um *time-to-market* (tempo de entrega ao mercado) curto faz com que o projeto de tais circuitos adote o fluxo *standard cell* (Figura 1).

No fluxo *standard cell* as portas lógicas são caracterizadas e validadas previamente em uma dada tecnologia, originando as chamadas "células". Essas células [2] são catalogadas com suas diversas características elétricas em uma biblioteca, podendo ser reutilizadas em diversos projetos que usem a mesma tecnologia. O reuso amortiza o custo dos projetos inseridos neste nodo tecnológico e possibilita um *time-to-market* mais curto.

O fluxo *standard cell* pode ser subdividido em etapas, e ao decorrer dessas etapas, a análise de *timing* pode ser requisitada milhares de vezes. De acordo com Bhasker e Chadha (2009), essas são algumas das etapas importantes no fluxo *standard cell*:

- **Síntese:** Responsável por criar uma representação em nível de portas lógicas, a partir de uma descrição no nível de transferência entre registradores (*RTL: Register Transfer Level*). A partir desta etapa, a análise de *timing* estática (*STA: Static Timing Analysis*) é utilizada, para estimar as características *temporais* do circuito;

- **Otimização Lógica:** Responsável por minimizar a lógica do circuito sintetizado. A análise de *timing* estática pode ser realizada antes desta etapa, para verificar os caminhos de maior atraso, também chamados de caminhos críticos. Se a análise de *timing* for realizada depois desta etapa, o objetivo é identificar

[1] Um processador para *desktop* desenvolvido no ano de 2008 tem cerca de 731 milhões de transistores, excluindo a área de memória (INTEL, 2008).

[2] Célula é a instância de *layout* para a implementação física de uma porta lógica.

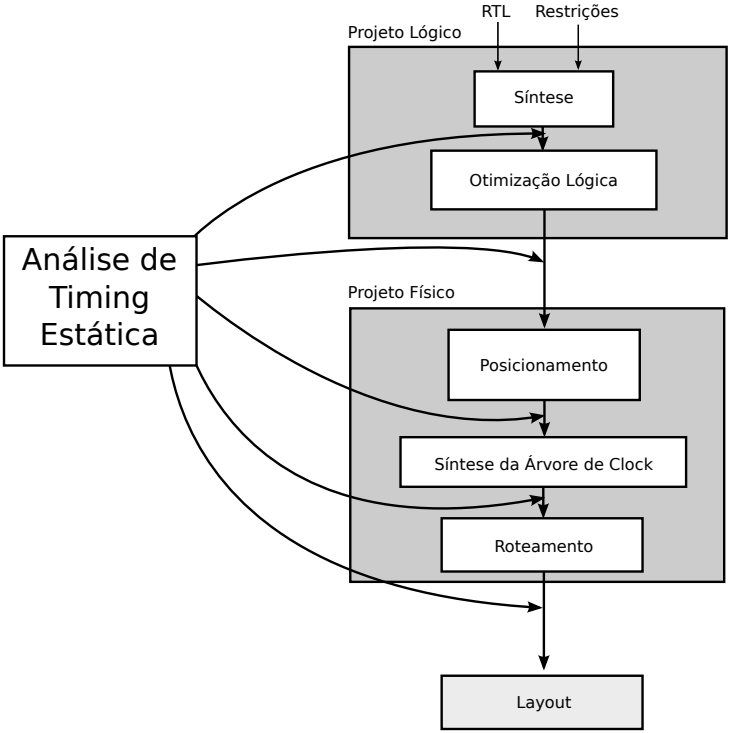


Figura 1 – Fluxo de projeto *Standard Cell*. Adaptado de (BHASKER; CHADHA, 2009).

quais caminhos ainda precisam ser otimizados ou identificar os caminhos críticos;

- **Posicionamento:** Define a localização espacial dos *layouts* das células. Antes dessa etapa, modelos de interconexão ideais são adotados, pois ainda não se possui as informações necessárias de posicionamento. Uma maneira alternativa para se modelar as interconexões é utilizar um modelo de *wireload*, que estima o tamanho das interconexões de acordo com o seu número de destinos, ou *fanouts*;

- **Síntese da Árvore de *Clock*:** No início da síntese física, as árvores dos relógios são consideradas como ideais, ou seja, não possuem atraso de propagação. O objetivo desta etapa é mini-

mizar o *clock skew*, que é a diferença entre os tempos de chegada do sinal de relógio nas entradas dos registradores. A análise de *timing* estática é importante nesta etapa para avaliar essas diferenças nos tempos de chegada.

- **Roteamento:** Responsável por criar as conexões entre as diferentes células incluídas no projeto, utilizando as diferentes camadas de metal. As mudanças nas topologias nesta etapa necessitam diversas avaliações das informações temporais.

Os projetos de circuitos digitais no fluxo *standard cell* são realizados visando, além das funcionalidades requisitadas, a operação em uma frequência especificada. Por isso, diversas otimizações são efetuadas ao longo do fluxo, para que tais funcionalidades consigam ser realizadas na frequência definida. Nas primeiras etapas de um projeto no fluxo *standard cell*, apenas as questões relacionadas à funcionalidade do projeto são verificadas, pois ainda não estão disponíveis informações detalhadas referentes ao comportamento elétrico do circuito. Nas etapas posteriores, as informações temporais precisam ser avaliadas com precisão, para que as etapas de otimização garantam a satisfação dos requisitos do projeto.

1.2 MOTIVAÇÃO

No fluxo *standard cell*, a partir da descrição RTL, uma série otimizações são realizadas ao decorrer de suas etapas. Como consequência dessas otimizações, as topologias das interconexões se alteram, levando à necessidade, ao decorrer do fluxo, de diversas avaliações de suas informações temporais. Kahng et al. (2013) trata do problema de *gate sizing* utilizando diversas modelagens para os atrasos das interconexões, bem como seu impacto na propagação do *slew* do circuito.

Em diversos sistemas projetados atualmente, entre 50% a 70% do ciclo de relógio é "consumido" pelo atraso de propagação de suas interconexões (CONG et al., 1996). Nas tecnologias com alta escala de integração (*VLSI: Very-large-scale integration*) atuais, onde diversas otimizações tem por objetivo reduzir a resistência dos *drivers*, as interconexões passam a ser cada vez mais impactantes no desempenho do circuito digital.

Durante as otimizações nas etapas iniciais do fluxo (*pre-layout*), a análise de *timing* é requisitada milhares de vezes, sendo assim necessário que a ferramenta de análise de *timing* tenha o melhor desem-

penho possível. Como as informações relacionadas ao aspecto físico do circuito, como posicionamento (WANG; YANG; SARRAFZADEH, 2000) e roteamento (RYZHENKO; BURNS, 2012) nas etapas iniciais precisam ser aproximadas, a ferramenta de análise de *timing* fornece estimativas pessimistas sobre o *timing* do circuito.

Já nas etapas finais (*pós-layout*), a análise de *timing* precisa ser a mais precisa possível. Porém, a modelagem dos elementos dos circuitos digitais torna-se mais complexa, diminuindo o desempenho da ferramenta.

Como as informações de *timing* precisam ser avaliadas centenas ou milhares de vezes durante os processos de otimização, ferramentas de análise de *timing* eficientes e escaláveis precisam ser desenvolvidas e aperfeiçoadas para acompanhar a evolução da tecnologia *CMOS* (*Complementary Metal-Oxide Semiconductor*).

1.3 JUSTIFICATIVA

Diversas otimizações são realizadas no decorrer do fluxo de projeto *standard cell* e o uso de ferramentas para a automação de projeto eletrônico (*EDA: Electronic Design Automation*) é indispensável em suas diferentes etapas. A inexistência de ferramentas de análise de *timing* estática precisas de domínio público e a restrição no acesso à ferramentas industriais (devido ao alto custo de suas licenças) resultam em um problema de infraestrutura de pesquisa. Assim, este trabalho tem como resultado uma alternativa de ferramenta de análise de *timing* para projetistas de circuitos digitais, bem como uma infraestrutura realista e precisa para desenvolvedores de ferramentas, que necessitam da análise de *timing* em alguma etapa do fluxo de projeto *standard cell*.

1.4 OBJETIVOS

1.4.1 Objetivo Geral

Este trabalho tem por objetivo o projeto, validação, avaliação e documentação de uma ferramenta de análise de *timing* estática voltada para o fluxo *standard cell*.

1.4.2 Objetivos Específicos

1. Avaliação e análise experimental do modelo de interconexão com capacitância concentrada, desprezando-se o impacto das resistências;

2. Avaliação e análise experimental da técnica de Elmore para cálculo do atraso das interconexões baseando-se em um modelo de interconexão RC concentrado;

3. Avaliação e análise experimental da técnica para cálculo do atraso de interconexões utilizando a abordagem de capacitância efetiva;

4. Construção de uma ferramenta de análise de *timing* estática incluindo as funcionalidades descritas nos objetivos 1, 2 e 3, bem como sua validação empírica perante uma ferramenta de análise de *timing* industrial.

1.5 ESCOPO

Este trabalho aborda o problema da análise de *timing* estática utilizando técnicas para estimação dos atrasos das interconexões. A análise de *timing* é realizada propagando os atrasos de cada porta lógica em ordem topológica, a fim de estimar o desempenho do circuito. Os modelos de atraso (*delay*) e *slew* utilizados neste trabalho são os mesmos utilizados no *fluxo standard cell*[3].

As interconexões serão modeladas de duas formas:

- **Modelo da capacitância concentrada**, impactando apenas nos *atrasos* de seus *drivers*;

- **Modelo RC concentrado**[4], apresentando também, seus próprios atrasos como impacto no atraso do circuito.

Não faz parte do escopo deste trabalho a consideração dos tempos de *setup* e *hold* das células sequenciais, como os registradores. Eles serão modelados pelo *timing arc*[5] da entrada de relógio até a saída.

[3] O cálculo dos atrasos das portas lógicas será melhor apresentado na Seção 2.1.

[4] Este modelo pode ser chamado de modelo RC distribuído em alguns trabalhos científicos, como na competição de *sizing* do *ISPD (International Symposium on Physical Design)* de 2013 (OZDAL et al., 2013).

[5] O conceito de *timing arc* será apresentado na Seção 2.1.

1.6 ORGANIZAÇÃO DESTE TRABALHO

Este trabalho está organizado da seguinte forma:

O Capítulo 2 apresenta os conceitos básicos essenciais para o entendimento do presente trabalho.

No Capítulo 3 é apresentada uma revisão bibliográfica acerca das técnicas utilizadas para cálculo do atraso das interconexões e da capacitância efetiva.

Já o Capítulo 4 trata da análise de *timing*, apresentando seus algoritmos e particularidades na implementação.

O Capítulo 5 apresenta os experimentos realizados utilizando a ferramenta implementada neste trabalho.

Finalmente, as conclusões e algumas perspectivas de trabalhos futuros são apresentadas no Capítulo 6.

2 CONCEITOS FUNDAMENTAIS DE CIRCUITOS DIGITAIS

Este capítulo apresenta os conceitos básicos relacionados à temporização e modelagem de circuitos digitais, essenciais para o entendimento do presente trabalho. A Seção 2.1 apresenta as características temporais das portas, assim como os modelos de atraso adotados no fluxo *standard cell*. Os modelos de interconexões e as suas características temporais serão apresentados nas Seções 2.2 e 2.3, respectivamente.

2.1 CARACTERÍSTICAS TEMPORAIS DAS PORTAS LÓGICAS

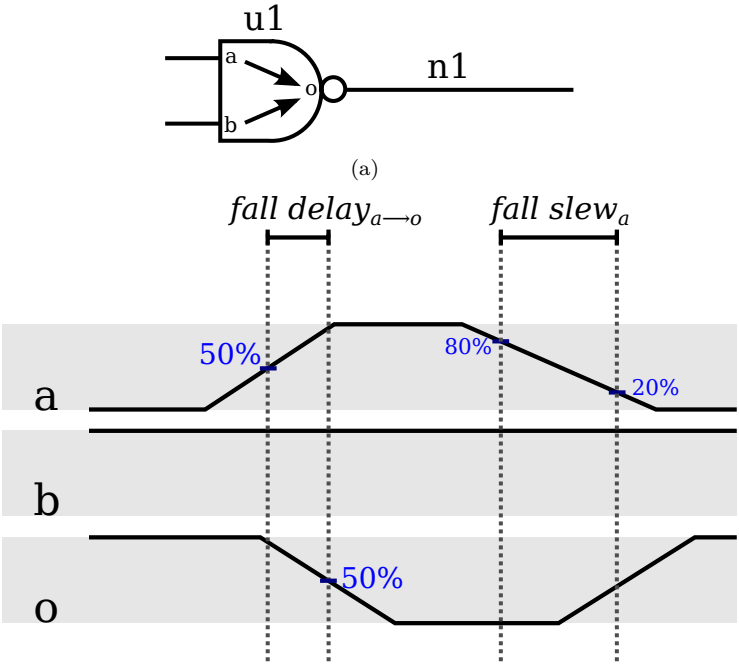


Figura 2 – (a) Uma porta lógica *CMOS* $u1$, de função **NAND**, com duas entradas é *driver* da interconexão *n1*. (b) Algumas características temporais (*delay* e *slew*) da porta lógica $u1$.

As características temporais do circuito são derivadas das características temporais de suas partes, quais sejam, as portas lógicas e as interconexões que o compõem. Para as portas lógicas, as informações a seguir são relevantes (LIVRAMENTO, 2013):

- ***Timing Arc* (Arco de Tempo):** é um conceito utilizado para associar um pino de entrada de uma porta com a saída dessa mesma porta. Uma porta *NAND* de duas entradas, como a apresentada na Figura 2(a) possui dois *timing arcs*: um entre a entrada a e a saída ($a \rightarrow o$) e outro entre a entrada b e a saída ($b \rightarrow o$). Para elementos sequenciais, como os registradores, normalmente consideram-se como *timing arcs* as conexões entre o sinal de relógio e as saídas. O arco é chamado *positive unate* se uma transição de subida (descida) na entrada causa uma transição de subida (descida) na saída. Se uma transição de subida (descida) da entrada causa uma transição de descida (subida) na saída, o arco é chamado *negative unate* (BHASKER; CHADHA, 2009).

- ***Delay* (Atraso de Propagação):** é o tempo que o sinal em um pino de saída o leva para atingir um limiar[1] de sua transição total, devido a uma mudança no sinal em um pino de entrada. Se a transição em o for do nível lógico 0 para 1, o atraso é chamado de atraso de subida (*rise delay*), caso o contrário, é chamado de atraso de descida (*fall delay*) (Figura 2).

- ***Slew* (Tempo de Transição):** é o tempo que um sinal leva para transicionar de uma porcentagem do valor de referência (V_{dd}) à outra (BHASKER; CHADHA, 2009)[2]. Se a transição for de um valor for de uma porcentagem menor para uma maior, ela é chamada de transição de subida (*rise slew*), caso contrário, trata-se de ua transição de descida (*fall slew*) (Figura 2).

- **Propagação do *Slew*:** é a política utilizada para propagação dos *slews* das entradas até as saídas das portas lógicas. A estratégia geralmente adotada é a de propagar para o pino de saída da porta lógica o maior dentre os *slews* associados aos *timing arcs*.

[1] Este limiar geralmente é definido nas bibliotecas de célula como sendo 50% do Vdd.

[2] Nas bibliotecas de células, essas porcentagens geralmente são definidas como 20% e 80% ou 10% e 90%

- ***Driver:*** é a porta lógica (ou o pino de saída de uma porta lógica) que gera o sinal para uma interconexão. Cada interconexão possui apenas um *driver*.

2.1.1 Modelo de atraso adotado em fluxo *standard cell*

Nas bibliotecas *standard cell* atuais, modelos de atrasos não-lineares[3] são fornecidos para os *timing arcs* das células disponíveis. Esses modelos, que geralmente são obtidos através de simulações em nível elétrico, são armazenados na forma de *lookup tables*, como a da Figura 3. Uma *lookup table* descreve o *delay* ou o *slew* de uma célula em função de dois fatores: o *slew* na entrada do *timing arc* (colunas), e a capacitância de saída (*load*) (linhas).

Utilizando a *lookup table* da Figura 3 para estimar o *delay* de um dos *timing arcs* de uma célula *CMOS* e supondo que o *slew* na entrada deste *timing arc* seja de 8.0, e a capacitância vista na saída seja 0.1, obtém-se que $delay = 3.49$, pois 3.49 é o valor endereçado pelos índices da função (*slew* e *load*). Caso os valores de *slew* ou *load* não existam na tabela, uma interpolação linear é realizada. Da mesma forma, o cálculo do *slew* do *timing arc* é realizado com base na *lookup table* específica para o *slew*.

2.2 MODELOS DE INTERCONEXÃO

Modelos de interconexão devem ser adotados de acordo com a etapa que o projeto se encontra no fluxo. Nas etapas iniciais, ou de *pre-layout*, ainda não há informações sobre o posicionamento e sobre o roteamento. Assim, as interconexões recebem modelos simplistas, possibilitando que as otimizações necessárias sejam realizadas, sem degradação no desempenho, para que as informações reais dos parasitas sejam apuradas. Nas etapas mais próximas da síntese física, ou *pos-layout*, as interconexões são modeladas em função de suas capacitâncias e resistências, com o intuito de fornecer uma simulação mais precisa possível.

Esta seção tem por objetivo, apresentar alguns modelos de representação de interconexões, suas vantagens e desvantagens. Também será apresentado o formato de representação de parasitas mais utilizado no projeto de circuitos digitais.

[3] Conhecidos na indústria por *NLDM* (*Non-Linear Delay Model*)

```
rise_delay (delay_table) {
 load (0.0, 0.1, 0.2, 0.4, 0.8, 1.6, 3.2) ;
 input_slew (0.5, 3.0, 5.0, 8.0, 14.0, 20.0, 30.0, 50.0) ;
 values (
   1.17, 1.82, 2.26, 2.76, 3.48, 4.04, 4.82, 6.12,
   1.69, 2.34, 2.86, 3.49, 4.41, 5.11, 6.06, 7.58,
   2.21, 2.86, 3.38, 4.12, 5.22, 6.05, 7.16, 8.90,
   3.25, 3.90, 4.42, 5.20, 6.60, 7.67, 9.08, 11.23,
   5.33, 5.98, 6.50, 7.28, 8.84, 10.30, 12.24, 15.14,
   9.50, 10.15, 10.67, 11.45, 13.01, 14.57, 17.15, 21.33,
   17.83, 18.48, 19.00, 19.78, 21.34, 22.90, 25.50, 30.70
  );
}
```

Figura 3 – Uma *lookup table* para atraso de subida (*rise delay*) de um arco de *timing*. As linhas são endereçadas por *load* (capacitância de saída da porta lógica) e as colunas por *input slew* (*slew* aplicado na entrada do *timing arc*). Adaptada de (OZDAL et al., 2013).

2.2.1 Modelo RC Distribuído (*Distributed RC Model*)

Uma interconexão pode ser representada idealmente como uma linha distribuída (Figura 4): a linha de comprimento L é dividida em segmentos de tamanho ΔL, com $\Delta L \to 0$, e cada segmento é representado por um valor de resistência r e um valor de capacitância c. Assim, a resistência e a capacitância total da linha são $r \times L$ e $c \times L$, respectivamente. O cálculo dos atrasos no modelo RC distribuído implica na resolução de equações diferenciais, as quais possuem soluções complexas. Uma solução numérica seria realista, porém resulta em um custo computacional muito elevado, tornando inviável sua adoção em fluxo *standard cell*. Para tal objetivo, utilizam-se modelos de interconexão simplificados.

2.2.2 Modelo de Capacitância Concentrada (*Lumped C Model*)

O Modelo de capacitância concentrada é geralmente utilizado nas etapas iniciais do projeto, pois se trata de um modelo simples com fácil simulação. Quando a resistência da interconexão é desprezível, de-

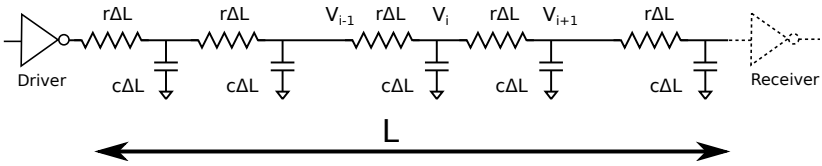


Figura 4 – Modelo RC Distribuído. Obtida de (RABAEY; CHANDRAKASAN; NIKOLIC, 2008).

vido o fato de que a resistência do *driver* é substancialmente maior que a resistência total da interconexão, ou quando as informações parasitas ainda não foram obtidas com detalhe, o fio pode ser representado como um capacitor C, que corresponde à capacitância total da interconexão. Seu atraso de propagação é desconsiderado, já que o fio não possui resistências. Seu único impacto no desempenho é a sua contribuição na capacitância vista pelo *driver* (RABAEY; CHANDRAKASAN; NIKOLIC, 2008).

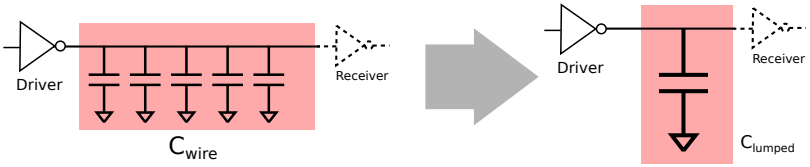


Figura 5 – Modelo de Capacitância Concentrada. Adaptada de (RABAEY; CHANDRAKASAN; NIKOLIC, 2008).

2.2.3 Modelo RC Concentrado (*Lumped RC Model*)

O modelo RC concentrado é amplamente adotado no fluxo *standard cell* para modelagem das interconexões. No modelo RC concentrado, concentra-se toda a resistência de cada segmento da interconexão em um único resistor R e similarmente, combina-se a capacitância total em um único capacitor C. A rede resistor-capacitor é normalmente representada como uma árvore RC (Figura 6). De acordo com Rabaey, Chandrakasan e Nikolic (2008), uma árvore RC possui as seguintes propriedades:

- A rede tem apenas um nodo de entrada, chamado de **fonte** (*source*);
- Todos os capacitores são entre um nodo e o terra;
- A rede não possui *loops* resistivos, por isso é chamada de Árvore.

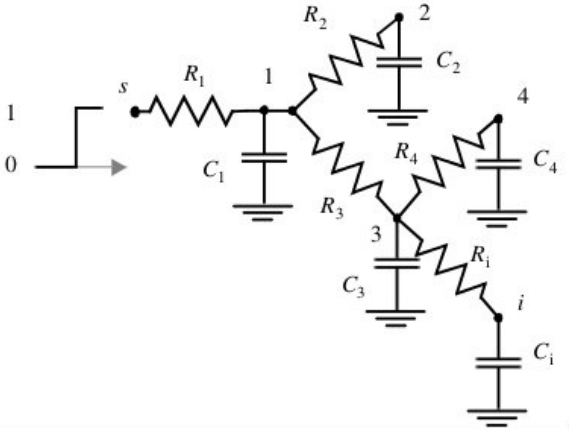


Figura 6 – Uma árvore RC. Obtida de (RABAEY; CHANDRAKASAN; NIKOLIC, 2008).

2.2.4 Extração de Elementos Parasitas no Projeto de Circuitos Digitais

Quando se tem as informações de capacitância e resistência totais de uma interconexão, C_{wire} e R_{wire} respectivamente, em uma fase de *pré-layout*, é necessário criar uma topologia para este fio, uma vez que o atraso da interconexão depende de como ela está estruturada. Existem três topologias (Figura 7) que podem ser utilizadas a fim de representar a interconexão (BHASKER; CHADHA, 2009):

- **Árvore de melhor caso (*Best-case tree*):** (Figura 7-a) Assume-se que cada pino de destino é fisicamente adjacente ao *driver*. Assim, nenhuma resistência estará no caminho entre *driver* e destino, e todos os pinos de destino atuarão como *load* na saída da interconexão.

- **Árvore balanceada (*Balanced tree*):** (Figura 7-b) Na árvore balanceada, todos os pinos de destino se encontram na mesma distância do *driver*, e o caminho para cada destino corresponde a mesma quantidade de capacitância e resistência que os outros caminhos.

- **Árvore de pior caso (*Worst-case tree*):** (Figura 7-c) Neste caso, todos os destinos se encontram no fim da interconexão. Assim, cada pino de destino vê a resistência e a capacitância total da interconexão.

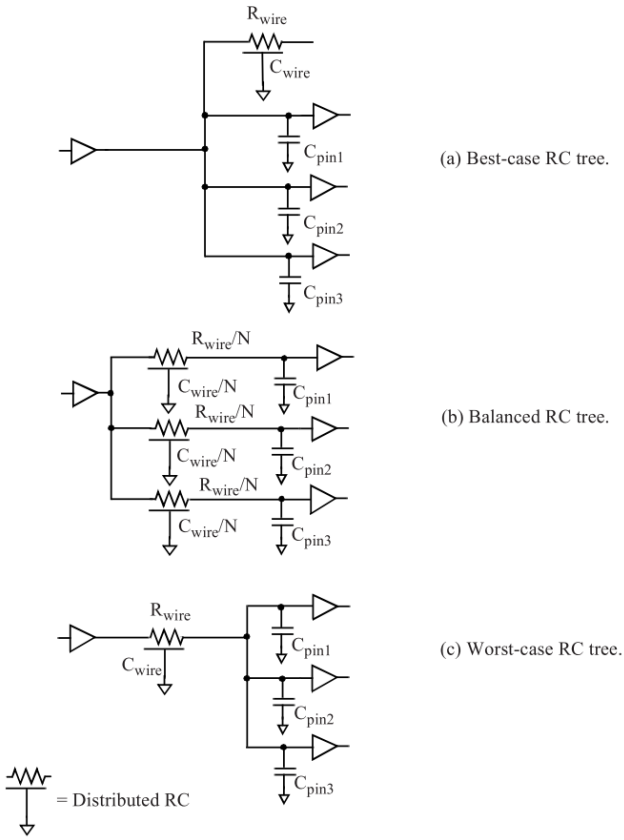


Figura 7 – Representações utilizadas para as árvores RC em um contexto de *pre-layout*. Obtida de (BHASKER; CHADHA, 2009).

Em projetos de circuitos digitais, as capacitâncias parasitas são geralmente descritas no formato SPEF [4] (*Standard Parasitic Exchange Format*) definido pelo *IEEE* (*Institute of Electrical and Electronics Engineers*). O formato SPEF é um padrão feito para garantir a interoperabilidade entre ferramentas de automação de projeto eletrônico (*EDA: Electronic Design Automation*). Os parasitas podem ser representados em diferentes níveis de sofisticação, desde o simplista modelo de capacitância concentrada, até uma representação mais precisa de Árvores RC.

Um exemplo de interconexão descrita no formato SPEF (IEEE, 1999) pode ser visualizado nas Figuras 8(a) e 8(b). A linha 1 no código SPEF da Figura 8(b) apresenta o nome da interconexão (`inp1`) e o valor de sua capacitância total (5.4). As linhas 2, 3 e 4 indicam que existe uma conexão entre uma entrada primária `inp1`, indicado por `*P inp1 I`, e a entrada de um pino interno *a* da porta `u1`, indicado por `*I u1:a I`. Da linha 6 até a linha 9 são representadas as capacitâncias da árvore RC.

A representação de um capacitor num arquivo SPEF se dá pelo formato:

```
[Número] [Nome] [Capacitância]
```

De maneira semelhante, os resistores, como pode ser visto nas linhas 11 até 13, são descritos no formato:

```
[Número] [Capacitor Fonte] [Capacitor Destino] [Resistência]
```

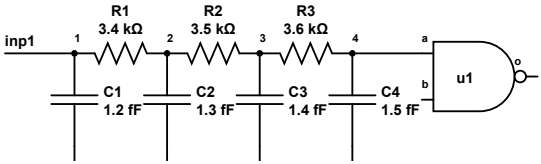
O valor $*END$ (linha 14) é utilizado para determinar o fim da descrição de uma interconexão.

2.3 CARACTERÍSTICAS TEMPORAIS DAS INTERCONEXÕES

A Figura 9 ilustra as três principais contribuições das interconexões, sobre o atraso do circuito:

- **Capacitância Vista Pelo *Driver*:** É necessário modelar a carga capacitiva a ser carregada pelo *driver* da interconexão com o objetivo de se obter a informação de *load*, a qual é utilizada no cálculo do *delay* e *slew* dos *timing arcs* das portas lógicas,

[4] Existem outros formatos como SPF (*standard parasitic format*), DSPF (*detailed standard parasitic format*), RSPF (*reduced standard parasitic format*) e SBPF (*Synopsys binary parasitic format*).



(a)

```
1  *D_NET inp1 5.4
2  *CONN
3  *P inp1 I
4  *I u1:a I
5  *CAP
6  1 inp1 1.2
7  2 inp1:1 1.3
8  3 inp1:2 1.4
9  4 u1:a 1.5
10 *RES
11 1 inp1 inp1:1 3.4
12 2 inp1:1 inp1:2 3.5
13 3 inp1:2 u1:a 3.6
14 *END
```

(b)

Figura 8 – (a) Interconexão RC obtida do circuito *simple* da competição de *sizing* do ISPD. (b) SPEF referente à Figura 8(a).

como visto anteriormente. Nesta capacitância é incluído também o impacto causado pelos pinos de destino da interconexão [5]. Na fase *pré-layout*, essa estimativa é realizada somando a capacitância total da interconexão com a capacitância de cada pino de destino dela. Porém, ao se tratar de interconexões com característica resistiva, o uso da abordagem de capacitância concentrada

[5] Um pino de destino de uma interconexão é um pino que se liga na interconexão, que não é o pino *driver*. Por exemplo, na Figura 9(a), os pinos de destino da interconexão são o segundo pino de entrada da porta $u2$, o pino de entrada da porta $u3$ e o pino d do *flip-flop* $f1$.

é impreciso. Para que os modelos de atraso não-lineares, que dependem do valor de capacitância de saída, sejam utilizados para os *drivers* diretamente, é necessário o uso de uma abordagem conhecida como **Capacitância Efetiva (C_{eff})**. Tal abordagem tenta encontrar um valor de capacitância que pode ser utilizado como carga equivalente, em termos de *timing*, para a saída do *driver* (BHASKER; CHADHA, 2009). Algumas técnicas serão abordadas no Capítulo 3.

- **Atraso da Interconexão:** Além do impacto local nos *delays* e *slews* de seus *drivers*, as interconexões exercem impacto global no circuito, com seu próprio atraso de propagação (Figura 9(b)), devido a sua característica resistiva. Com a alta frequência de operação dos circuitos digitais atuais e o dimensionamento dos transistores para escalas nanométricas, os atrasos das interconexões, que antes não eram significativos, hoje chegam consumir de 50% a 70% do ciclo do relógio, e esta porcentagem tende a aumentar na medida que os transistores diminuem (CONG et al., 1996). Uma das métricas mais populares para se calcular o atraso em interconexões é o atraso de Elmore (*Elmore Delay*) (ELMORE, 1948), pela simplicidade e razoável correlação com os atrasos reais. Esta técnica será apresentada com mais detalhes na Seção 3.2.

- **Degradação do *Slew*:** O cálculo do *slew* é crucial para determinar a precisão de uma avaliação de *timing* em um circuito digital (ZHOU et al., 2007). Os *delays* dos *timing arcs* dependem do *slew* de entrada e do *slew* de saída. Quando um sinal se propaga por uma interconexão, seu *slew* (i.e., sua declividade) sofre uma degradação devido ao efeito resistivo da mesma (Figura 9(b)). A não-modelagem desta degradação pela interconexão, acarreta em erros de até 50% (SHEEHAN, 2002). A abordagem para degradação do *slew* utilizada neste trabalho será apresentada na Seção 3.3.2.

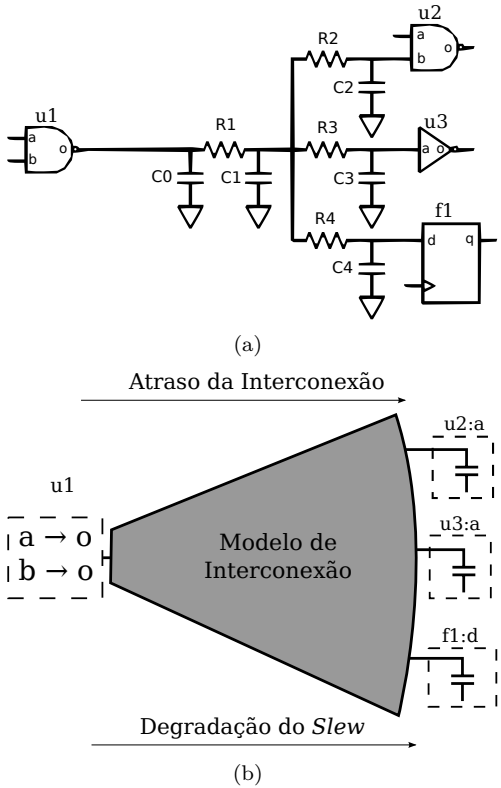


Figura 9 – (a) Um circuito composto por três portas lógicas ($u1$, $u2$ e $u3$), uma célula sequencial ($f1$) e uma interconexão em forma de árvore RC, que liga a saída de $u1$ às entradas de $u2$, $u3$ e $f1$; (b) São apresentadas as modelagens para os *timing arcs* da porta lógica $u1$; O modelo da interconexão é abstraído, recebendo um valor de capacitância efetiva. As setas indicam que a interconexão oferece um atraso e uma degradação no *slew*. Cada destino da interconexão é representado como um valor de capacitância de seus pinos de entrada.

3 CÁLCULO DAS CARACTERÍSTICAS TEMPORAIS DA INTERCONEXÃO

Este capítulo tem por objetivo apresentar uma técnica utilizada para o cálculo das características temporais das interconexões, necessário para a estimativa de *timing* global dos circuitos digitais. Na Seção 3.1 será apresentado um modelo computacional para as interconexões. A Seção 3.2 apresentará uma revisão bibliográfica mostrando algumas técnicas para cálculo do atraso das interconexões, bem como a técnica de Elmore e a técnica escolhida para ser implementada no presente trabalho, que será detalhada na Seção 3.3.

3.1 REPRESENTAÇÃO DAS INTERCONEXÕES

Para que o atraso de uma interconexão seja estimado com precisão, um modelo de grafo (Figura 10) pode ser utilizado para representar o fio em termos de capacitâncias e resistências.

No modelo de grafo $I(C, R)$ utilizado, o conjunto dos vértices é composto pelos nodos internos da interconexão, que representam cada capacitor. As arestas do grafo modelam os resistores, e cada resistor conecta um par de capacitores. Sendo assim:

- $\mathbf{C} = \{c | c$ é um capacitor da rede RC$\}$
- $\mathbf{R} = \{(c, d) |$ existe um resistor que conecta os capacitores c e $d\}$

3.2 CÁLCULO DO ATRASO DAS INTERCONEXÕES

Diversas técnicas são empregadas no cálculo do atraso das interconexões. Uma vez que o cálculo real dos atrasos das interconexões possui custo muito elevado para ser realizado para milhares de interconexões em centenas de vezes, modelos aproximados geralmente são utilizados no fluxo *standard cell*. A avaliação assintótica da forma de onda (*AWE: Asymptotic Waveform Evaluation*) (PILLAGE; ROHRER, 1990) é uma técnica amplamente utilizada para geração de modelos de ordem reduzida, realizando uma aproximação na função de transferência via aproximação de Padé. Outras técnicas como PRIMA (Passive Reduced-Order Interconnect Macromodeling Algorithm) (ODABASIOGLU; CELIK; PILEGGI, 1997) possuem uma alta complexidade

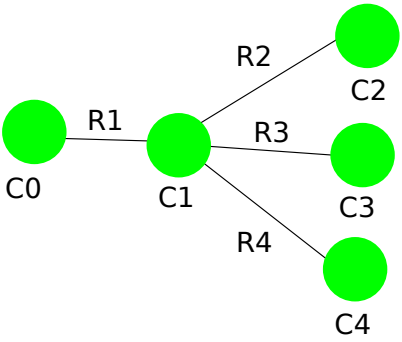


Figura 10 – O grafo correspondente à interconexão da Figura 9(a), com cinco vértices e quatro arestas.

computacional, sendo muito lentas para serem utilizadas no contexto de uma técnica de otimização, como *gate sizing* (KAHNG et al., 2013).

A técnica de Elmore (1948) é uma técnica baseada no primeiro momento da resposta ao impulso amplamente utilizada no cálculo dos atrasos das interconexões. A popularidade da técnica de Elmore deve-se aos fatores que seguem:

- Boa correlação com os atrasos reais nos nodos mais afastados do *driver* (KASHYAP; ALPERT; DEVGAN, 2000);
- Utiliza uma fórmula fechada, que envolve apenas as resistências e capacitâncias do circuito (HOROWITZ, 1983);
- Provê um limite superior provado para o atraso real de qualquer árvore RC (GUPTA et al., 1997);
- É aditiva, ou seja, o atraso do nodo A até o nodo C passando pelo nodo B é a soma dos atrasos entre A e B e entre B e C (KASHYAP; ALPERT; DEVGAN, 2000).

De acordo com Rabaey, Chandrakasan e Nikolic (2008), em um nodo c_i da árvore RC, o atraso de Elmore (τ_i) pode ser facilmente calculado como:

$$\tau_i = \sum_{k=1}^{N} C_k R_{ik} \tag{3.1}$$

Onde N é o número de capacitores da árvore RC, C_k é o valor de capacitância do nodo c_k e R_{ik} é a resistência compartilhada entre os caminhos $s \rightarrow i$ e $s \rightarrow k$ (s é o nodo fonte), ou seja:

$$R_{ik} = \sum R_j \Rightarrow (R_j \in [caminhos(s \rightarrow i) \cap caminhos(s \rightarrow k)]) \quad (3.2)$$

Na topologia da Figura 9(a), $C0$ é o nodo fonte. Assim, o atraso de Elmore para o nodo $C4$ é:

$$\tau_4 = C_1\ R_1 + C_2\ R_1 + C_3\ R_1 + C_4(R_1 + R_4) \quad (3.3)$$

A técnica de Elmore pode ser implementada também em sua forma recursiva. O algoritmo para cálculo do atraso de Elmore recebe como entrada o grafo $I(C, R)$ da interconexão e é executado após a inicialização das capacitâncias totais *downstream* (C_{total_i}) de cada nodo interno.

1. **Inicialização das capacitâncias totais *downstream*:** Os nodos internos são numerados de 1 até n em ordem topológica, sendo n o tamanho do conjunto de vértices. Assim, o passo de inicialização de cada $c_i \in C$ acontece em ordem topológica reversa, seguindo a Equação 3.4;

$$C_{total_i} = C_i + \sum_{j \in filhos(i)} C_{total_j} \quad (3.4)$$

   Sendo que C_i é o valor de capacitância do nodo c_i. O conjunto $filhos(i)$ é o conjunto de capacitores que estão interligados diretamente com o capacitor c_i através de um resistor R, que tenham um nível topológico maior que este[1]. Analogamente, o $pai(i)$ é um capacitor que precede c_i[2] e se conecta com ele, também, através de um resistor.

2. **Cálculo dos atrasos utilizando a técnica de Elmore:** O atraso de Elmore em cada nodo c_i da interconexão é calculado recursivamente, somando o atraso no pai de c_i com o valor da resistência que liga c_i ao seu pai multiplicado pela capacitância total *downstream* de c_i, como mostrado na Equação 3.5.

[1] Caso c_i seja um nodo terminal, seu conjunto $filhos(i)$ é vazio.
[2] Se c_i não for o nodo fonte da árvore.

$$\tau_i = \tau_{pai(i)} + R(pai(i), i) \times C_{total_i} \tag{3.5}$$

A função $R(i, j)$ retorna o valor da resistência que liga dois capacitores c_i e $c_j \in C$. O atributo C_{total_i} é a capacitância total *downstream* de um nodo c_i. Assim, o cálculo do atraso em cada nodo da interconexão compõe o atraso da interconexão partindo do *driver* até cada pino de destino.

A técnica de Elmore para atraso de interconexões fornece boas aproximações quando o efeito conhecido como *resistive shielding* não é tão alto. Este efeito acontece devido ao fato de que as resistências alteram o tempo que as capacitâncias levam para serem carregadas ou descarregadas. O efeito de *resistive shielding* faz com que o atraso do *driver* de uma interconexão seja menor que o atraso dele considerando a capacitância concentrada da interconexão. Similarmente, o efeito faz com que o atraso da interconexão seja menor que o atraso de Elmore utilizando o valor de capacitância total para cada segmento. Considere a interconexão da Figura 9(a), no caso extremo em que $R4 = \infty$, o capacitor $C4$ nunca seria carregado, e portanto, o atraso da interconexão não deveria levar em consideração o valor do capacitor $C4$.

Algumas adaptações na técnica de Elmore foram propostas para que o cálculo do atraso das interconexões capturem também o efeito do *resistive shielding* utilizando a abordagem da capacitância efetiva em cada nodo da interconexão. Na Figura 11, as transições na saída de um *driver* são comparadas ao se utilizar a abordagem de capacitância efetiva (linha pontilhada) e de capacitância concentrada (linha contínua). Pode-se observar que para o valor escolhido de capacitância efetiva, o sinal leva o mesmo tempo para atingir o ponto médio da curva ($V_{dd} = 50\%$) que quando o *driver* está conectado diretamente à carga real da árvore RC (linha tracejada). Note a diferença neste ponto em relação à curva de capacitância concentrada, mostrando a imprecisão de se utilizar este modelo em certos casos.

Como a abordagem de capacitância efetiva está relacionada à consideração do efeito de *resistive shielding*, ao utilizá-la em cada segmento da interconexão, é possível obter-se um atraso na interconexão mais preciso do que o atraso de Elmore, mesmo sem considerar momentos de maior ordem da resposta ao impulso.

Kashyap, Alpert e Devgan (2000) propuseram uma técnica para calcular o atraso da interconexão levando em conta o efeito de *resistive shielding*. Com a mesma complexidade da técnica de Elmore, a

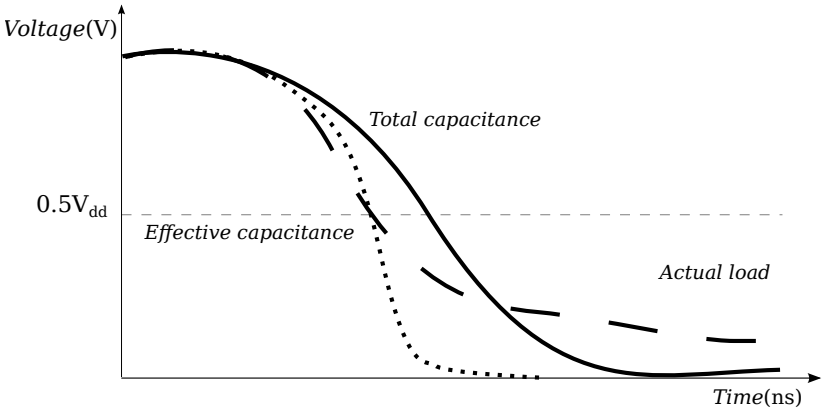


Figura 11 – Formas de onda na saída de uma porta lógica em função da abordagem utilizada para cálculo da capacitância. Obtida de (BHASKER; CHADHA, 2009).

técnica proposta para cálculo de atraso em uma árvore RC calcula também o valor de capacitância efetiva. Porém, Kashyap, Alpert e Devgan (2000) não consideravam o *driver* da interconexão como sendo uma porta lógica *CMOS*. Como consequência, sua aproximação para o *slew* na entrada da árvore RC era imprecisa. Como o cálculo da capacitância efetiva de uma árvore depende do *slew* que incide nesta, e o *slew* depende da capacitância vista pelo *driver*, Puri, Kung e Drumm (2002) propuseram uma técnica que leva em consideração o impacto da capacitância no *slew* do driver, e também, do *slew* no cálculo da capacitância efetiva. Esta técnica será apresentada na Seção 3.3 e foi a técnica implementada neste trabalho.

3.3 TÉCNICA DE Puri, Kung e Drumm (2002) PARA O CÁLCULO DA CAPACITÂNCIA EFETIVA E DEGRADAÇÃO DO *SLEW*

O objetivo desta seção é apresentar a técnica para cálculo das informações referentes às características temporais das interconexões que foi escolhida para ser implementada neste trabalho.

Devido ao fato de que o valor de capacitância efetiva de uma interconexão depende do *slew* incidente nesta, que por sua vez, depende do valor de capacitância efetiva, Puri, Kung e Drumm (2002)

propuseram uma técnica iterativa para simular esta interdependência. A técnica em questão obtém o atraso de Elmore com capacitância efetiva para a interconexão, bem como a degradação do *slew* e o valor de capacitância utilizado no cálculo do *delay* e *slew* do *driver*.

A seguir serão apresentadas as técnicas para cálculo da capacitância efetiva e degradação do *slew*, bem como o algoritmo implementado para realização desses cálculos.

3.3.1 Cálculo da Capacitância Efetiva

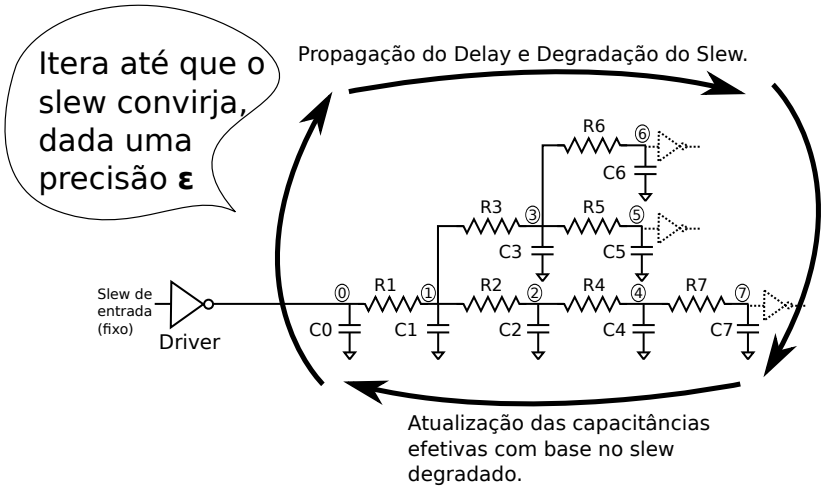


Figura 12 – Visão geral da técnica iterativa para o cálculo do atraso da interconexão, capacitância efetiva e degradação do *slew*. Adaptada de (PURI; KUNG; DRUMM, 2002).

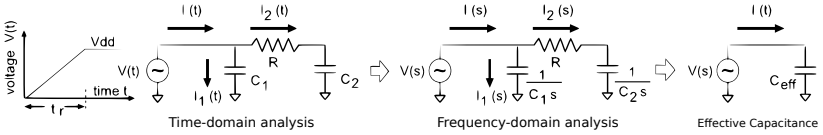


Figura 13 – Cálculo da capacitância efetiva utilizando rampa de entrada. Obtida de (PURI; KUNG; DRUMM, 2002).

Considere uma rede π $C_1 - R - C_2$ alimentada por uma fonte de tensão $V(t)$, como a ilustrada na Figura 13. Seja $I(t)$ a corrente total fornecida pela fonte de tensão $V(t)$, $I_1(t)$ a corrente através de C_1 e $I_2(t)$ a corrente através de $R - C_2$. De acordo com Puri, Kung e Drumm (2002), realizando uma análise no domínio da frequência, obtemos que:

$$I(s) = I_1(s) + I_2(s) \tag{3.6}$$

Se

$$I_1(s) = \frac{V(s)}{1/C_1 s} \tag{3.7}$$

$$I_2(s) = \frac{V(s)}{R + 1/C_2 s} \tag{3.8}$$

Então:

$$I(s) = \frac{V(s)}{1/C_1 s} + \frac{V(s)}{R + 1/C_2 s} \tag{3.9}$$

ou

$$I(s) = V(s)\left(C_1 s + \frac{C_2 s}{1 + RC_2 s}\right) \tag{3.10}$$

Agora, considerando que a fonte de tensão $V(t)$ é uma rampa com tempo de subida t_r, $V(t)$ é dado por:

$$V(t) = \begin{cases} \frac{V_{dd}}{t_r} \times t & \text{se } t < t_r \\ V_{dd} & \text{caso contrário} \end{cases} \tag{3.11}$$

E no domínio da frequência:

$$V(s) = \frac{V_{dd}}{t_r} \times \frac{1}{s^2} \times (1 - e^{-st_r}) \tag{3.12}$$

Substituindo a $V(s)$ da Equação 3.10 pela Equação 3.12 e voltando ao domínio do tempo obtém-se:

$$I(t) = \frac{V_{dd}}{t_r}((C_1 + C_2) - C_2 e^{-\frac{t}{RC_2}}) \quad \text{para } t < t_r \tag{3.13}$$

Em termos de *timing*, na capacitância efetiva, a carga transferida Q é a mesma que da rede π, no ponto médio da curva (tempo que a curva atinge $50\%V_{dd}$). A carga transferida Q é a integral da corrente $I(t)$ com o tempo indo de 0 até $t_r/2$:

$$Q = \int_0^{t_r/2} I(t)\mathrm{d}t = \int_0^{t_r/2} \frac{V_{dd}}{t_r}((C_1+C_2) - C_2 e^{-\frac{t}{RC_2}})\mathrm{d}t \tag{3.14}$$

A carga transferida Q para carregar a capacitância efetiva da rede π (C_{eff}) até 50% de V_{dd} é dada, também, por $\frac{C_{eff}V_{dd}}{2}$. Ao igualar as duas equações de transferência de carga, obtemos:

$$C_{eff} = C_1 + C_2(1 - \frac{2RC_2V_{dd}}{t_r}(1 - e^{-\frac{t_r}{2RC_2}})) \tag{3.15}$$

Assim, $C_{eff} = C_1 + C_2 \times K$, onde K é o fator de *shielding*, definido por:

$$K = 1 - 2x(1 - e^{-\frac{1}{2x}}), \quad \text{onde } x = \frac{RC_2}{t_r} \tag{3.16}$$

3.3.2 Degradação do *Slew* Através da Árvore RC

O *slew* é degradado na árvore RC para obtenção de coeficientes mais precisos para os fatores de *shielding*. Considere um segmento de uma árvore RC, mostrado na Figura 14, que é alimentado por uma fonte de tensão $V(t) = \frac{V_{dd}}{t_r} \times t$, onde t_r é o *slew* da rampa de entrada. Conforme (PURI; KUNG; DRUMM, 2002), a tensão de saída neste caso, quando $t = t_r$, pode ser derivada como:

$$\frac{V_{dd}}{t_r}(t_r - RC_2 + RC_2 e^{-\frac{t_r}{RC_2}}) \tag{3.17}$$

Conforme a Figura 14:

$$\frac{V_1}{t_r} = \frac{V_{dd}}{t'_r} \tag{3.18}$$

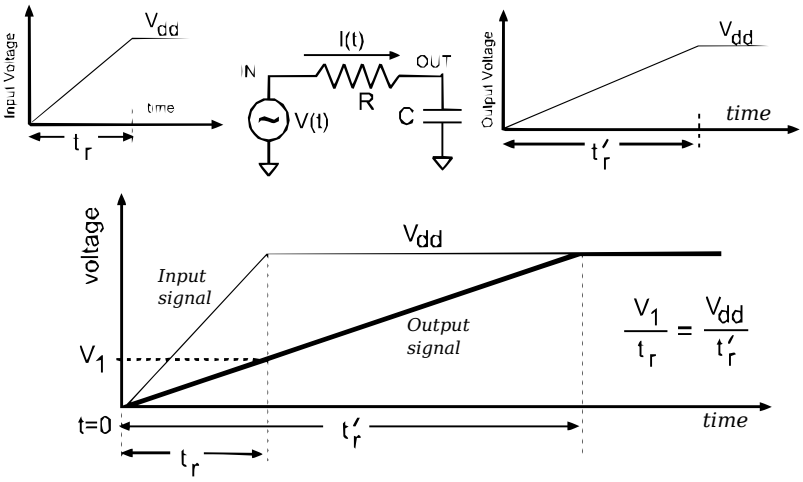


Figura 14 – Degradação no *slew* em um segmento de uma árvore RC. Obtida de (PURI; KUNG; DRUMM, 2002).

Onde t'_r é o *slew* na saída. Substituindo V_1 da Equação 3.18 pela Equação 3.17, obtemos o valor de *slew* na saída, em função do *slew* aplicado na entrada:

$$t'_r = \frac{t_r}{1 - x(1 - e^{-\frac{1}{x}})} \quad \text{onde } x = \frac{RC_2}{t_r} \tag{3.19}$$

Generalizando para uma árvore RC qualquer [3] e utilizando o valor de capacitância efetiva, o $slew_i$ no nodo c_i é definido pela equação:

$$slew_i = \frac{slew_j}{1 - \frac{R_i\ C_{eff_i}}{slew_j}(1 - e^{-\frac{slew_j}{R_i\ C_{eff_i}}})} \tag{3.20}$$

Onde $slew_j$ é o *slew* na entrada de v_i, vindo pelo nodo pai v_j e R_i é o valor do resistor que conecta c_i com c_j

[3] Representada conforme apresentado na Seção 3.1.

3.3.3 O Algoritmo de Puri, Kung e Drumm (2002)

Dado o grafo de uma árvore RC[4], o algoritmo apresentado nessa seção calcula os valores de capacitância efetiva e *slew* em cada nodo interno da interconexão. Para o atraso da interconexão, o método implementa a técnica de Elmore utilizando os valores de capacitância efetiva, ao invés dos valores de capacitância total *downstream*, simulando o efeito de *resistive shielding*.

A capacitância efetiva é denotada em cada nodo c_i por C_{eff_i} e o *slew* por $slew_i$. O valor do *slew* aplicado no nodo fonte da árvore RC, denotado por $slew_1$, é exatamente o valor do *slew* na saída do *driver* desta interconexão, o qual é função do *slew* na entrada da porta lógica *driver* e da capacitância efetiva vista na saída. Este valor de *slew* será refinado iterativamente para se estimar o valor de capacitância efetiva da interconexão.

O algoritmo para cálculo iterativo da capacitância efetiva de uma interconexão, bem como seu atraso e a degradação no *slew*, conforme (PURI; KUNG; DRUMM, 2002), ocorre em cinco passos:

1. **Inicialização:**
   (a) A capacitância efetiva C_{eff_i} de cada nodo c_i da Árvore RC é incializada com o valor de capacitância total *downstream* de c_i, ou seja $C_{eff_i} = C_{total_i}$;
   (b) O *slew* no nodo fonte da árvore RC $slew_1$ é calculado utilizando o modelo de atraso da porta lógica *driver*, considerando a capacitância concentrada da árvore RC (i.e., $\sum_{i=1}^{N} C_i$): $slew_1 = f(C_{total_1})$.

2. **Atualização dos *slews* em ordem topológica:**
   (a) Atraso τ_i do nodo fonte c_1 para cada nodo c_i da árvore é calculado utilizando a técnica de Elmore (Equação 3.5), substituindo C_{eff_i} por C_{total_i}, para simular o efeito de *resistive shielding*;
   (b) A degradação do *slew* em cada nodo c_i é calculada utilizando a Equação 3.20.

3. **Atualização das capacitâncias efetivas em ordem topológica reversa:**

[4] Com os nodos numerados de 1 a n em ordem topológica, onde n é o tamanho do conjunto de vértices e o nodo c_1 é o nodo fonte da árvore RC.

(a) A capacitância efetiva (C_{eff_i}) de cada nodo c_i é calculada como a soma da capacitância do nodo c_i e todas as capacitâncias dos nodos filhos:

$$C_{eff_i} = C_i + \sum_{j \in filhos(i)} K_j \times C_{tot_j} \tag{3.21}$$

Onde K_j é o fator de *shielding*, definido por:

$$K_j = 1 - \frac{2R_j C_{eff_j}}{slew_i}(1 - e^{-\frac{slew_i}{2R_j C_{eff_j}}}) \tag{3.22}$$

Onde R_j é o valor da resistência que conecta o nodo c_j ao seu pai, no caso, c_i.

4. **Atualização do *Slew* do *Driver*:** O *slew* no nodo fonte $slew_1$ é calculado diretamente, utilizando o C_{eff_1} atual ;

5. **Iteração:** Os passos de 2 até 4 são repetidos até que $slew_1$ convirja, dada uma precisão ε .

Na implementação apresentada neste trabalho, o ε foi definido como sendo 1% e na maioria dos casos observados, cerca de 5 iterações são necessárias para realizar o cálculo da capacitância efetiva (PURI; KUNG; DRUMM, 2002). Como cada iteração do algoritmo percorre a lista em ordem topológica (direta e reversa), a complexidade assintótica de pior caso de cada iteração do algoritmo é de $O(n)$ onde n é o número de nodos da árvore, ao passo que a complexidade do algoritmo é $O(c.n)$, onde c é o número de iterações. Entretanto, como o número de iterações é na grande maioria dos casos menor que 5 (E portanto c é muito menor que n), assume-se que o crescimento no tempo de execução tem comportamento linear.

4 ANÁLISE DE *TIMING* ESTÁTICA

O objetivo deste capítulo é apresentar a análise de *timing* estática (*STA: Static Timing Analysis*), bem como os conceitos importantes referentes à esta técnica, juntamente com o algoritmo de STA.

Análise de *timing* estática, ou *static timing analysis* (GUNTZEL, 2000) (BHASKER; CHADHA, 2009), é uma das técnicas utilizadas para se estimar o atraso crítico de circuitos digitais. A análise de *timing* é chamada de estática quando ela não realiza simulação e portanto, independe de estímulos de entrada, considerando apenas a topologia do circuito. É um processo completo e exaustivo (BHASKER; CHADHA, 2009) que verifica as mais diversas informações de *timing* em um circuito, como os *delays*, *slews*, *slacks* (folgas), *required times* (tempos requeridos) e diversas violações de restrições de projeto.

Dada a descrição do projeto usando alguma linguagem de descrição de hardware (*HDL: Hardware Description Language*), restrições de projeto e uma biblioteca de células, o objetivo da análise de *timing* é apresentar informações temporais em todos os pontos do circuito e apontar as possíveis violações (Figura 15). Essas informações são utilizadas para avaliar se o projeto sob verificação pode operar na velocidade estipulada, ou seja, se o circuito final poderá funcionar com segurança na frequência de relógio escolhida, sem que existam violações nas restrições de projeto.

O fluxo básico de uma ferramenta de análise de *timing* é:

1. **Leitura dos arquivos de entrada:** Nesta etapa, os arquivos referentes às bibliotecas de célula, descrição do circuito juntamente com as restrições do projeto são lidos e suas informações são armazenadas em estruturas de dados, que serão consultadas na geração do modelo de grafo e na atualização das informações temporais;

2. **Geração do grafo de *timing*:** Responsável por implementar o modelo de grafo de *timing*. As estruturas de dados utilizadas na implementação do modelo de grafo têm impacto direto no desempenho da ferramenta de *timing*.

3. **Atualização de informações temporais:** Etapa onde a propagação dos atrasos através dos *timing arcs*, bem como a avaliação do cumprimento ou não das restrições de desempenho são realizados.

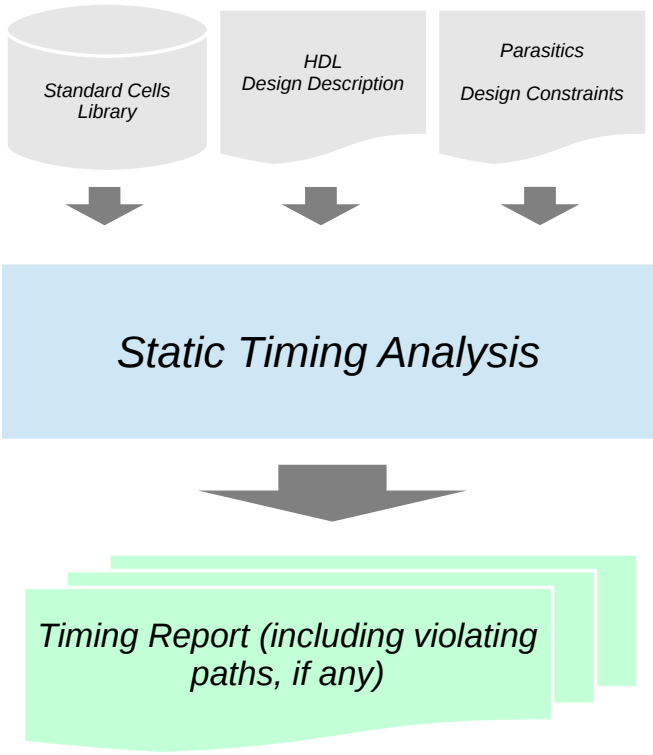


Figura 15 – Análise de *timing* estática. Adaptado de (BHASKER; CHADHA, 2009).

Na Seção 4.1 será apresentado o modelo de grafo utilizado para modelar os circuitos digitais na análise de *timing*. A Seção 4.2 mostrará a nomenclatura utilizada para as diversas informações temporais relevantes na análise de *timing* estática. Finalmente, as informações particulares sobre a implementação da ferramenta construída neste trabalho serão apresentadas na Seção 4.3.

4.1 REPRESENTAÇÃO DE CIRCUITOS DIGITAIS

Um circuito combinacional pode ser representado por um grafo de *timing*. Neste grafo, as portas lógicas e os pinos de entrada e saí-

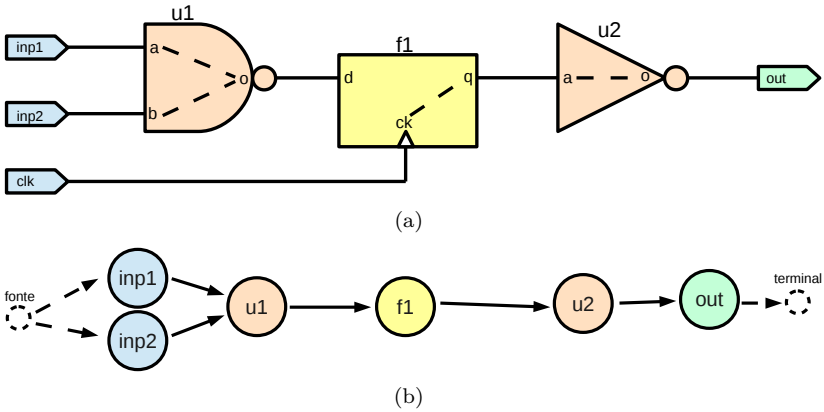


Figura 16 – (a) Circuito *simple* retirado do banco de *benchmarks* da competição de *sizing* do ISPD; (b) Grafo correspondente ao circuito da letra (a).

das primárias são os vértices e as interconexões são as arestas, como mostrado na Figura 16 (BHASKER; CHADHA, 2009).

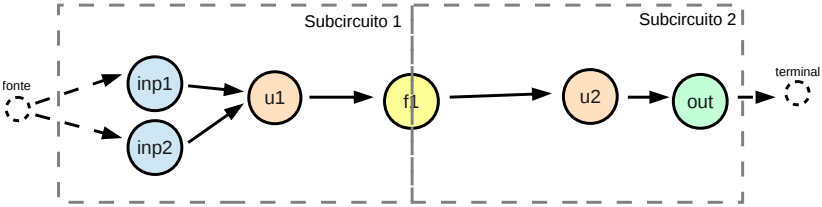


Figura 17 – Grafo de *timing* dividido em dois sub-circuitos devido à existência de uma célula sequencial.

Um circuito que consiste de células combinacionais e sequenciais (*flip-flops* e *latches*) pode ser representado como um conjunto de blocos combinacionais, divididos pelos *latches* (Figura 17). Assim, a entrada de uma célula sequencial pode ser tratada como uma saída primária do circuito, e a saída dessa pode ser tratada como uma entrada primária de outro circuito (Figura 18).

Em um contexto de projeto com o fluxo *standard cell*, é interessante que o grafo modele também os *timing arcs* das portas lógicas.

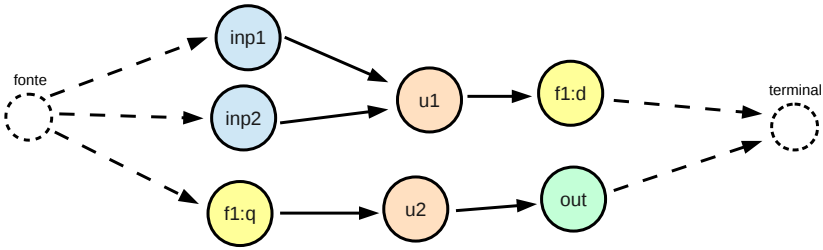


Figura 18 – Grafo de *timing* com célula sequencial atuando como entrada e saída primária do circuito.

Assim, alternativamente, os vértices do grafo de *timing* representam os pinos de entrada e saída das portas lógicas, entradas e saídas primárias e as arestas representam os *timing arcs* e as interconexões. Um grafo representando o modelo escolhido pode ser visualizado na Figura 19.

A nomenclatura usada no grafo direcionado $G(V, E)$ deste segundo modelo, adotado no presente trabalho, é a seguinte:

- V = { $v_i | v_i$ é um *timing point* (pino de *timing*), que pode ser a entrada ou saída de uma porta lógica, aqui referenciado como pino. Um *timing point* pode também representar uma entrada ou saída primária do circuito. }

- I = { $(v_i, v_j) | v_i, v_j \in V$ e (v_i, v_j) é uma interconexão do circuito, que conecta v_i em v_j. v_i é um pino de saída de uma porta lógica ou uma entrada primária, e v_j pode ser a entrada de uma porta lógica ou uma saída primária. }

- A = { $(v_i, v_j) | v_i, v_j \in V$ e (v_i, v_j) é um *timing arc*. Portanto, v_i e v_j são pinos de entrada e saída (respectivamente) de uma mesma porta lógica. }

- Por fim, o conjunto das arestas E = I $\cup$ A.

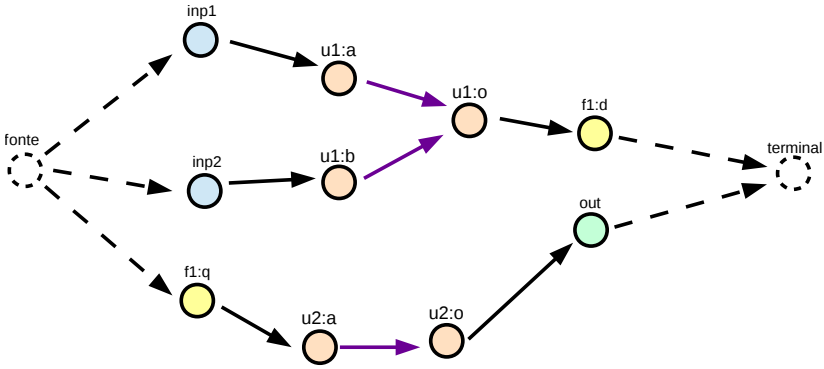


Figura 19 – Grafo de *timing* com representação dos *timing points*, *timing arcs* e interconexões.

Nos vértices, ou *timing points*, são armazenadas as informações temporais para os pinos do circuito, tais como os *arrival times*, *slews* e *slacks* que serão apresentadas na Seção 4.2.

4.2 CÁLCULO DO PIOR ATRASO DO CIRCUITO

O cálculo do pior atraso do circuito é realizado propagando os *arrival times* (tempos de chegada) das portas lógicas em ordem topológica através de um método conhecido como *PERT/CPM (Program Evaluation and Review Technique / Critical Path Method)*. Para o entendimento das políticas de propagação dos atrasos na avaliação do desempenho de um circuito, os termos a seguir são importantes:

- **Caminho:** uma sequência de vértices (*timing points*) tal que, para cada um de seus vértices há uma aresta (*timing arc* ou interconexão) para o próximo vértice da sequência. O primeiro *timing point* da sequência é uma entrada primária e o último é uma saída primária;
- $\boldsymbol{inputs(i)}$: conjunto de *timing points* que se ligam com v_i através de um *timing arc*. Todo $v_j \in input(i)$ é necessariamente um pino de entrada de uma porta lógica, e v_i é um pino de saída;
- $\boldsymbol{a_i}$: *arrival time*, ou tempo de chegada no pino v_i. O *arrival time* é definido pelo atraso do caminho parcial que inicia em uma entrada primária e termina em v_i;

- $\boldsymbol{slew_i}$: o *slew* no pino v_i;
- $\boldsymbol{d_{j \to i}}$: o *delay* do *timing arc* que vai do pino v_j até o pino v_i;
- $\boldsymbol{slew_{j \to i}}$: o *slew* do *timing arc* que vai do pino v_j até o pino v_i;
- $\boldsymbol{iD_{i \to k}}$: o atraso de propagação na interconexão que liga o pino v_i até o pino v_k. No modelo de capacitância concentrada, $iD_{i \to k} = 0$;
- $\boldsymbol{iS_{i \to k}}$: degradação do *slew* através da interconexão que liga v_i em v_k;
- $\boldsymbol{fanouts(i)}$: conjunto dos pinos que são destino da interconexão para qual v_i é *driver*;
- $\boldsymbol{r_i}$: é o *required time* no *timing point* v_i. O *required time* é o tempo máximo que o valor de a_i pode assumir para que a restrição de desempenho seja respeitada. Se v_i é uma saída primária do circuito, então $r_i = T$, onde $f = \frac{1}{T}$ é a frequência mínima de operação do circuito digital;
- $\boldsymbol{slack_i}$: folga de tempo no ponto v_i, ou seja, quanto o *arrival time* pode atrasar neste ponto, de modo que o período máximo continue sendo respeitado. Se em um determinado ponto do circuito o *slack* é negativo, então o caminho em questão está violando a restrição de atraso máximo do sistema.

Na análise de *timing* estática, os piores atrasos de cada porta lógica são propagados visitando-se o grafo direcionado em ordem topológica. Para cada $v_i \in V$ que são pinos de saída de portas lógicas, os *arrival times*, bem como os *slews* são determinados de modo a respeitar as seguintes restrições:

$$a_i = \max_{\forall v_j \in inputs(i)} (a_j + d_{j \to i}) \tag{4.1}$$

$$slew_i = \max_{\forall v_j \in inputs(i)} (slew_{j \to i}) \tag{4.2}$$

Se $v_i \in V$ é um pino de entrada de uma porta lógica e $v_j \in V$ é o *driver* da interconexão que conecta v_j em v_i, o *arrival time* e o *slew* em v_i são definidos por:

$$a_i = a_j + iD_{j \to i} \tag{4.3}$$

$$slew_i = slew_j + iS_{j \to i} \tag{4.4}$$

Após a propagação dos *arrival times* em todos os pinos, é necessário realizar a propagação dos tempos requeridos, o que é feito percorrendo-se o grafo em ordem topológica reversa, a fim de obterem-se os valores dos *slacks*. Em um pino de saída v_i de uma porta lógica, o tempo requerido pode ser obtido facilmente, observando o menor dos tempos requeridos dentre os seus *fanouts* e suas interconexões, ou seja:

$$r_i = \min_{\forall v_j \in fanouts(i)} (r_j - iD_{i \to j}) \tag{4.5}$$

Para se propagar o *required time* do pino v_j de saída de uma porta lógica para uma entrada v_i, utiliza-se o valor de *delay* do arco que liga v_i em v_j, o qual já foi calculado previamente:

$$r_i = r_j - d_{i \to j} \tag{4.6}$$

A partir dos *required times* e *arrival times*, podemos determinar os *slacks* nos diversos pontos do circuito, através da equação:

$$slack_i = r_i - a_i \tag{4.7}$$

Se em algum ponto v_i, $slack_i = 0$, então v_i se encontra em um caminho crítico. Se $slack_i < 0$, então v_i se encontra em um caminho que viola a restrição de desempenho.

O **pior slack** é definido como o menor valor de *slack* entre as saídas primárias. O valor **total de slack negativo** é o somatório dos módulos dos *slacks* negativos das saídas primárias.

4.3 IMPLEMENTAÇÃO DA FERRAMENTA DE STA

Esta seção apresentará as estratégias utilizadas para o desenvolvimento da ferramenta de análise de *timing*. Serão ilustradas as principais estruturas de dados, modelos de grafo, e serão apresentados também os algoritmos implementados na ferramenta desenvolvida.

4.3.1 O Modelo de Grafo Adotado

As estruturas de dados utilizadas para armazenar os elementos do grafo são essencialmente listas ordenadas topologicamente. Em uma lista ordenada topologicamente, dado um elemento i, à esquerda necessariamente se encontram os elementos de mesmo ou menor nível lógico, e à direita, de nível igual ou maior, como mostrado na Figura 20. Da mesma maneira, os *timing arcs* e as interconexões também são ordenados topologicamente, em suas respectivas listas. Com essa escolha, o algoritmo de análise de *timing* estática passa a ser apenas de uma varredura em ordem, na lista de *timing points*, atualizando a informação de *timing* acumulada para cada vértice do grafo.

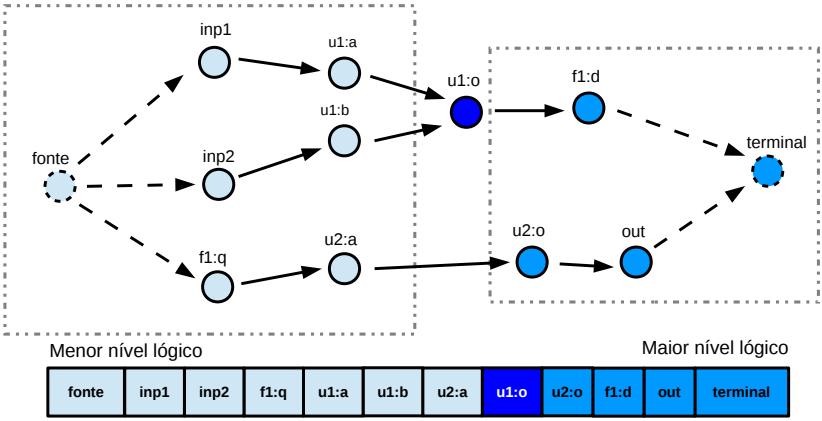


Figura 20 – Na lista ordenada, observando o elemento *u1:o*, os elementos de menor ou de igual nível lógico (*fonte, inp1, inp2, f1:q, u1:a, u1:b, u2:a*) se encontram à esquerda, e os de maior ou igual (*u2:o, f1:d, out, terminal*) se encontram à direita.

4.3.2 Algoritmo de Análise de *Timing* Estática

Com o modelo de grafo definido e implementado em suas devidas estruturas de dados, a análise de *timing* estática é realizada atualizando as informações de *timing* de cada nodo do grafo de *timing*, em ordem topológica, como mostrado no algoritmo 1.

No algoritmo 1 é apresentada a rotina de análise de *timing*. Os

Algoritmo 1: Análise de *timing* estática.

Entrada: Grafo de *timing* $G(V, E)$
Saída: Informações de *timing* para os elementos do grafo (*timing points*, interconexões e *timing arcs*)

```
1  para todo v_i ∈ V em ordem topológica hacer
2      se v_i é um pino de entrada então
3          C_eff ← calcular_C_eff();
4          d_{i→o} ← delay_biblioteca(s_i, C_eff);
5          slew_{i→o} ← slew_biblioteca(slew_i, C_eff);
6          propagar_atrasos();
7      senão se v_i é um pino de saída ou é uma entrada
       primária então
8          propagar_para_fanouts();
9      fim
10 fin
11 para todo cada v_i ∈ V em ordem topológica reversa
   hacer
12     atualiza_folgas(v_i);
13 fin
```

vértices são processados topologicamente, propagando os atrasos calculados de cada *timing arc* das portas lógicas para suas saídas.

Para os pinos de entrada das portas lógicas, o procedimento realizado compreende da Linha 3 até a Linha 6. Na Linha 3 é realizado o cálculo da capacitância efetiva, que posteriormente é utilizada para se obter os valores de *delay* e *slew* nos arcos que partem de v_i (Linhas 4 e 5). A seguir, na Linha 6, representada pela rotina *propagar_atrasos*(), os *arrival times* e *slews* são propagados para o pino de saída utilizando as Equações 4.1 e 4.2, respectivamente.

Já para os pinos de saída, a rotina *propagar_para_fanouts*() é executada, e corresponde à propagação dos *arrival times* e *slews* através das interconexões, utilizando as Equações 4.3 e 4.4.

Após todos os *arrival times* serem calculados, as folgas são obtidas propagando-se os *required times* em ordem topológica reversa, como foi apresentado na Seção 4.2 pelas Equações 4.5, 4.6 e 4.7, correspondendo ao procedimento *atualiza_folgas*().

5 EXPERIMENTOS

Esse capítulo tem por objetivo descrever os experimentos realizados neste trabalho e apresentar os resultados obtidos.

Na Seção 5.1 será apresentada a metodologia e infraestrutura utilizadas para a realização dos experimentos.

Na Seção 5.2 é apresentado o primeiro experimento, que trata da validação da ferramenta perante o *PrimeTime*, utilizando o modelo de interconexões de capacitância concentrada.

Na Seção 5.3 será apresentada a maneira como o *PrimeTime* modela o circuito, e como as informações de *timing* são calculadas.

E por fim, a Seção 5.4 tem por objetivo mostrar os resultados obtidos com a técnica de cálculo de capacitância efetiva e degradação do *slew* implementadas neste trabalho. A organização dos experimentos realizados a fim de validar a ferramenta pode ser observada na Tabela 1.

Driver	Interconexão		Tabela
	Atraso	Degradação do *Slew*	
C_{total}	**SEM**	**SEM**	2
C_{eff}	Elmore (C_{eff})	(PURI; KUNG; DRUMM, 2002)	4
C_{eff}	Elmore (C_{eff})	**SEM**	6
C_{total}	Elmore (C_{total})	(PURI; KUNG; DRUMM, 2002)	5

Tabela 1 – Técnicas validadas nos experimentos e as respectivas tabelas que apresentam os resultados obtidos.

5.1 METODOLOGIA E INFRAESTRUTURA EXPERIMENTAL

Para realizar a avaliação das técnicas abordadas neste trabalho, uma ferramenta para análise de *timing* na linguagem de programação C++ foi implementada. A ferramenta realiza a análise de *timing* e considera dois possíveis modelos de interconexão: o modelo da capacitância concentrada e o modelo RC concentrado.

Como parte dos experimentos é realizada comparando as informações calculadas pela ferramenta implementada com as informações reportadas pelo *PrimeTime*, o erro percentual (EP_t) e o erro médio percentual absoluto ($EMPA$) (Equações 5.1 e 5.2) foram adotados como métricas para estimar a qualidade das informações de *timing* reporta-

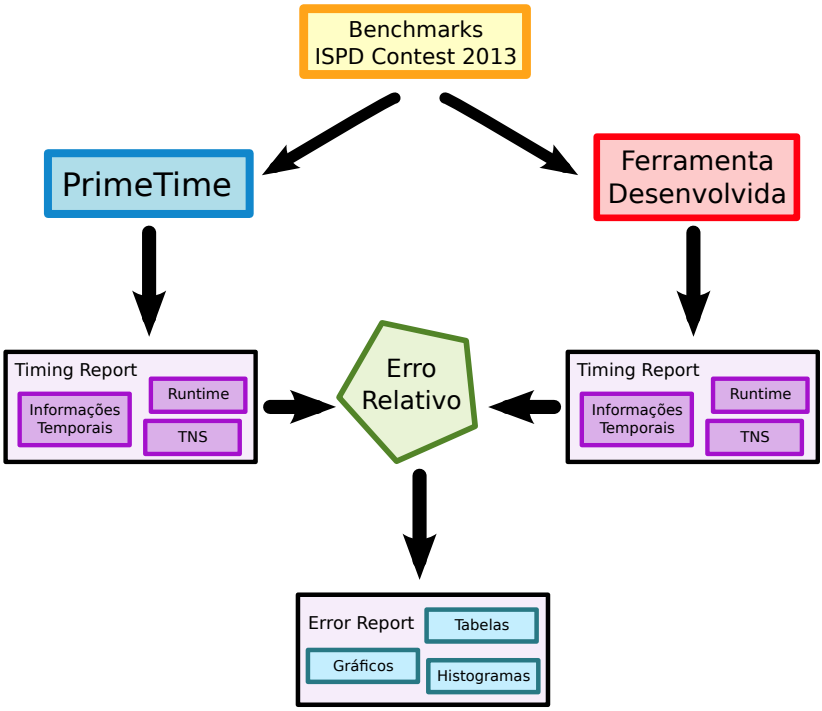


Figura 21 – Fluxo utilizado na validação da ferramenta implementada neste trabalho perante a ferramenta industrial *PrimeTime*.

das pela ferramenta implementada neste trabalho (Figura 21).

$$EP_t = \frac{(A_t - P_t)}{A_t} \times 100 \tag{5.1}$$

$$EMPA = \frac{\sum_{t=1}^{n} |EP_t|}{n} \tag{5.2}$$

O erro percentual é calculado para cada uma das informações comparadas com o *PrimeTime* utilizando a equação 5.1, sendo que A_t é a informação obtida pelo *PrimeTime* e P_t é a informação calculada pela ferramenta implementada. Tais informações usadas para fim de validação da ferramenta foram:

- ***TNS (Total Negative Slack)*:** O somatório de *slack* negativo nas saídas primárias.

- ***Violating POs:*** Número de saídas primárias violando a restrição de desempenho mínimo.

- ***Runtime (s):*** Tempo de execução, em segundos, para realizar uma análise de *timing* em um circuito, desconsiderando o tempo constante de inicialização da ferramenta.

- ***Critical Path:*** Valor do caminho crítico do circuito.

Os resultados dos experimentos serão apresentados posteriormente por meio de gráficos, tabelas e histogramas.

Este trabalho utilizou como base a infraestrutura disponibilizada pela competição de *gate sizing* discreto do ISPD de 2013, a qual fornece:

- Um conjunto de 8 circuitos da competição do ISPD de 2013:

  1. ***usb_phy***: com 511 células combinacionais, 98 células sequenciais, 15 entradas e 19 saídas primárias;
  2. ***pci_bridge32***: com 27316 células combinacionais, 3359 células sequenciais, 160 entradas e 201 saídas primárias;
  3. ***fft***: com 30297 células combinacionais, 1984 células sequenciais, 1026 entradas e 1026 saídas primárias;
  4. ***cordic***: com 40371 células combinacionais, 1230 células sequenciais, 34 entradas e 64 saídas primárias;
  5. ***des_perf***: com 103842 células combinacionais, 8802 células sequenciais, 234 entradas e 201 saídas primárias;
  6. ***edit_dist***: com 125000 células combinacionais, 5661 células sequenciais, 2562 entradas e 12 saídas primárias;
  7. ***matrix_mult***: com 30297 células combinacionais, 1984 células sequenciais, 3202 entradas e 1600 saídas primárias;
  8. ***netcard***: com 884427 células combinacionais, 97831 células sequenciais, 1836 entradas e 10 saídas primárias.

- Uma biblioteca *standard cell* realista, composta por onze células combinacionais de diversas funções lógicas e um célula sequencial;

- Uma ferramenta de análise de *timing* estática PrimeTime ® da empresa Synopsys (2012) para comparação de resultados;

Os circuitos são compostos por descrições no formato Verilog, capacitâncias parasitas e resistências descritas no formato IEEE SPEF (*Standard Parasitic Exchange Format*) (IEEE, 1999), e restrições de *timing* descritas no formato SDC (*Synopsys Design Constraints*).

5.2 VALIDAÇÃO DO MODELO DE CAPACITÂNCIA CONCENTRADA PERANTE FERRAMENTA INDUSTRIAL

Este experimento tem por objetivo validar a ferramenta de *STA* desenvolvida perante a ferramenta industrial *PrimeTime*, utilizando a abordagem de capacitância concentrada para modelar as interconexões. Para uma comparação justa, a ferramenta industrial foi também configurada para utilizar este modelo. Para tanto, utilizou-se um computador *desktop* com processador *Intel Core i7*, de 4 núcleos, e 4GB de *RAM*, e os resultados deste experimento são apresentados na Tabela 2.

Lumped Capacitance Interconnect Model

BENCHMARK	TNS (ps)	Viol. POs	Critical Path (ps)	Runtime (s)
usb_phy	0,00E+00	0	3,40E+02	0,00
pci_bridge32	2,08E+03	46	1,05E+03	0,02
fft	0,00E+00	0	1,51E+03	0,03
cordic	2,98E+04	185	3,16E+03	0,03
des_perf	0,00E+00	0	9,24E+02	0,09
edit_dist	0,00E+00	0	2,92E+03	0,11
matrix_mult	0,00E+00	0	2,04E+03	0,14
netcard	2,60E+06	11925	3,11E+03	24,83
Média	**3,29E+05**	**1519,5**	**1,88E+03**	**3,16**
EMPA	**0,00**	**0,00**	**0,00**	**-**

Tabela 2 – Comparação das informações de *timing* calculadas pela ferramenta implementada *versus* informações fornecidas pelo *PrimeTime*, utilizando o modelo de interconexões de capacitância concentrada.

As células marcadas correspondem aos valores que são menores que os obtidos na ferramenta comercial. A penúltima linha apresenta a média dos valores calculados para cada coluna da tabela. A última linha mostra o $EMPA$ para cada uma das informações mostradas nas colunas. Os valores de $EMPA$ valendo $0{,}00\%$ indicam que a ferramenta calcula os mesmos valores que a ferramenta industrial para as

informações comparadas. A média de *runtime* obtida é 6,92 vezes menor que a média da ferramenta industrial, sendo 50,05 vezes menor para os 7 primeiros circuitos (excluindo o *netcard*). No circuito *usb_phy*, a diferença de *runtime* é de 20 vezes e nos outros circuitos (exceto o *netcard*) a diferença tem valor médio de 52,71 vezes com baixo desvio padrão (6,48).

5.3 ANÁLISE DE *TIMING* ESTÁTICA EM FERRAMENTA INDUSTRIAL

Esta seção tem por objetivo apresentar o modelo (Figura 22) utilizado para cálculo do desempenho na ferramenta industrial *PrimeTime* e apresentar algumas informações de *timing* relevantes, obtidas para cada circuito da competição de *sizing* do ISPD.

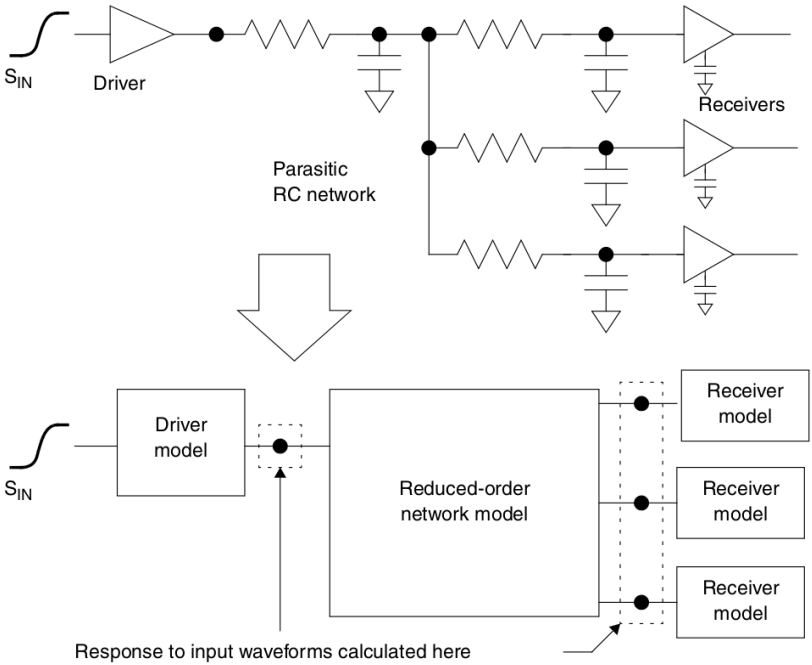


Figura 22 – Modelagem utilizada no *PrimeTime* para o *driver*, interconexão e destinos. Obtida de (SYNOPSYS, 2013).

5.3.1 Modelagem de *Driver* (*Driver model*)

No *PrimeTime*, o *driver* é modelado utilizando uma rampa de tensão em série com um resistor (um modelo Thèvenin). O resistor ajuda a suavizar a rampa de tensão, para que a forma de onda resultante seja similar à forma de onda do *driver* real, que está conectado na interconexão.

De acordo com (SYNOPSYS, 2013), o modelo de *driver* tem três parâmetros:

- A resistência do *driver* R_d;
- O tempo de início da rampa t_Z;
- A duração da rampa Δt.

O *PrimeTime* escolhe os parâmetros de modo que a forma de onda de saída seja o mais próximo possível da simulação. O modelo de *driver* simplificado é construído para cada *timing arc* de cada porta lógica do circuito.

5.3.2 Modelagem do Destino (*Receiver model*)

Cada destino da interconexão é representado como um valor de capacitância, que corresponde à capacitância do pino em que a interconexão está ligada.

5.3.3 Modelagem da Interconexão (*Reduced-order network model*)

Um modelo reduzido de interconexão é uma representação simplificada de uma interconexão, com as mesmas características de resposta da interconexão original, a qual pode ter centenas de capacitâncias e resistências. O *PrimeTime* utiliza a redução de Arnoldi (ODABASIOGLU; CELIK; PILEGGI, 1997) (SILVEIRA et al., 1999) para criar um modelo reduzido. Através do modelo reduzido, a ferramenta escolhe valores para a C_{eff}, de modo que o atraso do *driver* seja igual ao atraso do *driver* ligado à interconexão original. Por falta de fontes na bibliografia, não foi possível implementar este método de ajuste da C_{eff}, impossibilitando uma comparação mais justa. Desta forma,

não foi possível identificar o algoritmo para o cálculo do atraso das interconexões no *PrimeTime*.

5.3.4 Resultados Obtidos

PrimeTime

BENCHMARK	*TNS (ps)*	*Viol. POs*	*Critical Path (ps)*	*Runtime (s)*
usb_phy	4,85E+04	57	1,18E+03	0,24
pci_bridge32	9,97E+06	3034	6,25E+03	2,71
fft	2,46E+07	1983	1,33E+04	3,65
cordic	1,45E+07	1206	1,62E+04	4,60
des_perf	1,26E+07	1648	1,15E+04	11,26
edit_dist	3,20E+07	3416	1,13E+04	13,27
matrix_mult	1,62E+07	2852	1,24E+04	18,98
netcard	2,60E+06	11925	3,11E+03	271,97
Média	1,41E+07	3265,1	9,40E+03	40,84

Tabela 3 – Valores obtidos pelo *PrimeTime* no *benchmark* experimental utilizado neste trabalho.

A Tabela 3 mostra os valores obtidos pelo *PrimeTime* aplicando a STA na infraestrutura experimental. A última linha da tabela mostra as médias para cada informação obtida.

5.4 VALIDAÇÃO DA TÉCNICA IMPLEMENTADA PERANTE FERRAMENTA INDUSTRIAL

Esta seção tem por objetivo comparar a qualidade das informações de *timing* obtidas pela ferramenta de análise de *timing* implementada neste trabalho com as informações reportadas pelo *PrimeTime*. Utilizando as técnicas apresentadas na Seção 3.3 (i. e., capacitância efetiva, atraso de interconexões e degradação de *slew*), a análise de *timing* estática foi aplicada nos circuitos de teste e suas soluções foram comparadas com as fornecidas pela ferramenta industrial. Tal comparação foi realizada com base nas métricas apresentada na Seção 5.1. Os resultados deste experimento podem ser vistos na Tabela 4.

A penúltima linha apresenta a média dos valores de cada coluna

C_{eff} + Elmore (C_{eff}) + Slew Degradation

BENCHMARK	TNS (ps)	Viol. POs	Critical Path (ps)	Runtime (s)
usb_phy	4,91E+04	59	1,19E+03	0,01
pci_bridge32	9,80E+06	3002	6,29E+03	0,31
fft	2,34E+07	1983	1,22E+04	0,45
cordic	1,27E+07	1206	1,55E+04	0,58
des_perf	1,12E+07	1648	1,08E+04	1,08
edit_dist	2,95E+07	3311	1,11E+04	1,79
matrix_mult	1,38E+07	2831	1,11E+04	2,15
netcard	2,17E+06	8944	3,00E+03	12,82
Média	**1,28E+07**	**2873,00**	**8,89E+03**	**2,40**
EMPA	**8,81**	**4,17**	**4,48**	**-**

Tabela 4 – Valores obtidos pela ferramenta implementada neste trabalho nos circuitos da competição de *sizing* do ISPD.

e a última linha apresenta o $EMPA$ para cada uma das informações em relação ao *PrimeTime*, que foram apresentados na Tabela 3. Com esse experimento, conclui-se que a ferramenta desenvolvida neste trabalho fornece informações de *timing* próximas às reportadas pelo *PrimeTime*[1], com um tempo de execução $17,02$ vezes menor.

As células marcadas apresentam os valores que são otimistas em relação do *PrimeTime* (i. e., que são menores que os obtidos pelo *PrimeTime*), correspondendo a 29 dos 36 valores obtidos.

O erro de menor valor absoluto para o *TNS* é de $1,34\%$ e o de maior é $16,7\%$ nos circuitos *usb_phy* e *netcard*, respectivamente, sendo que no segundo, o erro reflete em uma aproximação otimista, e no primeiro, pessimista.

Na média, a análise de *timing* na ferramenta desenvolvida é otimista em relação ao *PrimeTime*, de acordo com o grande número de células marcadas. É possível observar também que os maiores erros são obtidos nos maiores circuitos e os menores erros, nos menores circuitos.

No experimento mostrado na Tabela 5, o modelo de capacitância concentrada foi utilizado para modelar a carga vista pelo *driver*. Para o atraso das interconexões, a técnica de Elmore com capacitância concentrada foi utilizada. Já para a degradação do *slew*, foi utilizada a técnica descrita no Capítulo 3. Como esperado, os modelos utiliza-

[1] $EMPA = 8,21$ e $4,48$ para *TNS* e *critical path*, respectivamente.

C_{total} + Elmore (C_{total}) + Slew Degradation				
BENCHMARK	**TNS (ps)**	**Viol. POs**	**Critical Path (ps)**	**Runtime (s)**
usb_phy	6,25E+04	61	1,34E+03	0,00
pci_bridge32	1,16E+07	3070	6,57E+03	0,09
fft	2,79E+07	1983	1,38E+04	0,12
cordic	1,54E+07	1207	1,72E+04	0,14
des_perf	1,25E+07	1648	1,13E+04	0,35
edit_dist	3,50E+07	3508	1,24E+04	0,44
matrix_mult	1,62E+07	2851	1,19E+04	0,56
netcard	1,07E+07	36934	3,37E+03	6,55
Média	**1,62E+07**	**6407,75**	**9,72E+03**	**1,03**
EMPA	**48,32**	**27,59**	**6,48**	**-**

Tabela 5 – Experimentos utilizando o modelo capacitância concentrada para carga de saída dos *drivers*, técnica de Elmore para computar os atrasos das interconexões, e degradação do *slew* conforme apresentado no Capítulo 3.

dos neste experimento refletem em uma aproximação pessimista para o atraso do circuito[2], já que a técnica de Elmore pura[3] foi aplicada no cálculo dos atrasos das interconexões. A técnica obteve $0,13\%$ e $311,79\%$ de erro para TNS nos circuitos *matrix_mult* e *netcard*, respectivamente. Já para *critical path*, os erros obtidos vão de $1,94\%$ até $13,85\%$, nos circuitos *des_perf* e *usb_phy*, respectivamente.

A importância do cálculo da degradação do *slew* pode ser visualizado na Tabela 6. Os erros obtidos neste experimento ($40,80\%$ para TNS e $21,21\%$ para critical path) mostram resultados muito otimistas em relação ao *PrimeTime*, quando a técnica apresentada no Capítulo 3 é aplicada, sem considerar a degradação do *slew* nos destinos das interconexões.

5.4.1 Relação entre C_{eff} e C_{total}

Este experimento tem por objetivo justificar os baixos erros obtidos pelo experimento apresentado na Tabela 5. Foram obtidas as

[2]Informação obtida do baixo número de células marcadas (total de 4, com exceção das células de *runtime*).

[3]Sem utilizar a abordagem de capacitância efetiva.

C_{eff} + Elmore (C_{eff}) + No Slew Degradation

BENCHMARK	TNS (ps)	Viol. POs	Critical Path (ps)	Runtime (s)
usb_phy	3,14E+04	57	9,51E+02	0,00
pci_bridge32	7,46E+06	2847	5,48E+03	0,29
fft	1,88E+07	1983	1,03E+04	0,40
cordic	9,32E+06	1204	1,27E+01	0,48
des_perf	8,67E+06	1648	9,22E+03	0,96
edit_dist	2,02E+07	3139	9,11E+03	1,54
matrix_mult	9,07E+06	2639	9,28E+03	1,89
netcard	1,41E+05	1033	2,63E+03	12,81
Média	**9,21E+06**	**1818,75**	**5,87E+03**	**0,79**
EMPA	**40,80**	**14,16**	**29,21**	**-**

Tabela 6 – Experimentos utilizando o modelo capacitância efetiva para carga de saída dos *drivers*, técnica de Elmore utilizando as capacitâcias efetivas de cada nodo interno das interconexões, para computar seus atrasos. Neste experimento, a degradação do *slew* não foi considerada.

relações C_{eff}/C_{total} médias para todos os circuitos, os considerando em três configurações diferentes:

- **Min.:** configuração de menor consumo de *leakage*[4];
- **Normal:** configuração padrão, conforme descrita em *verilog*;
- **Max.:** configuração de maior consumo de *leakage*;

Na Tabela 7, observa-se que nos circuitos testados, o valor de C_{eff} médio fica muito próximo do valor de C_{total} na configuração de maior consumo de *leakage*. Nessa configuração, as portas lógicas possuem maiores *slews* em relação às configurações de menor *leakage* e *leakage* padrão. Por exemplo, no circuito *pci_bridge*, a mudança da configuração *Min.* para *Max.* aumenta em 43% o *slew* médio do circuito, e no *matrix_mult*, 93%. Como $C_{eff} = C_1 + C_2 \times K$, sendo K definido na Equação 3.16, o crescimento no valor do *slew* implica no crescimento do valor K, fazendo com que C_{eff} fique mais próxima de $C_1 + C_2$, que por sua vez, é o valor C_{total}. Este experimento explica o

[4] *Leakage* se trata de uma parcela da potência estática que é dissipada pelos transistores *CMOS* mesmo quando estão desligados.

C_{eff} / C_{total}

BENCHMARK	Size		
	Min.	Normal	Max.
usb_phy	0,934	0,934	0,996
pci_bridge32	0,949	0,949	0,999
fft	0,945	0,946	0,995
cordic	0,940	0,940	0,999
des_perf	0,966	0,966	0,999
edit_dist	0,925	0,927	0,996
matrix_mult	0,938	0,939	0,996
Média	**0,942**	**0,943**	**0,997**

Tabela 7 – Relação C_{eff}/C_{total} média por circuito.

baixo erro obtido ao utilizar-se o modelo de capacitância concentrada para estimar o valor da capacitância vista pelo *driver* (Tabela 5).

Foram obtidas também as relações C_{eff}/C_{total} para todas as interconexões dos circuitos *pci_bridge32* e *matrix_mult* e os histogramas com as distribuições de frequências para estas relações são apresentados nas Figuras 24 e 25. No experimento apresentado na Figura 24 são mostradas as relações C_{eff}/C_{total} nas interconexões do circuito *pci_bridge32*, divididas em três partes. Na primeira em verde, se encontram as frequências das relações C_{eff}/C_{total} nas interconexões com menor valor de resistência; a segunda parte, em vermelho, mostra a distribuição das frequências nas interconexões com valor médio de resistência; e a última parte, em azul, mostra as frequências para a última parte das interconexões, ou seja, as com maior valor de resistência total. Note que na Figura 24(a) todos os valores de frequência são mostrados, e na Figura 24(b), apenas os valores de frequência menores que 200 são mostrados. Este mesmo experimento foi também realizado para o circuito *matrix_mult* e seu resultado pode ser visualizado na Figura 25.

Analisando os histogramas apresentados nas Figuras 24 e 25, observa-se o impacto do efeito de *resistive shielding*. Nas interconexões com menor valor resistivo, o valor de C_{eff} é mais próximo de C_{total} na maioria dos casos. Quanto maior for o valor de resistências das interconexões, maior o efeito de *resistive shielding*, fazendo com que as relações C_{eff}/C_{total} assumam mais valores menores que 1.

5.4.2 Erro de *Arrival Times* nas Saídas Primárias

Este experimento avalia os erros nos *arrival times* obtidos pela ferramenta implementada neste trabalho em relação aos calculados pelo *PrimeTime*, nos circuitos *pci_bridge32* e *matrix_mult*. Este experimento complementa o experimento apresentado na Seção 5.4.1 na escolha do modelo a ser utilizado para as interconexões.

Como observado na Tabela 7, quando o circuito está na configuração de maior *leakage*, o valor de C_{eff} se aproxima de C_{total} ($C_{eff}/C_{total} = 0.99$), possibilitando o uso da abordagem de capacitância concentrada para representar as interconexões, refletindo em um resultado semelhante ao da abordagem de capacitância efetiva, porém com um *runtime* cerca de 3.5 vezes menor. A distribuição das frequências dos erros percentuais obtidos neste experimento podem ser visualizados na Figura 23.

Kahng et al. (2013) realizaram um experimento semelhante comparando quatro técnicas para o cálculo do atraso das interconexões e duas para a degradação do *slew*. No contexto de uma técnica de *gate sizing*, a ferramenta de análise de *timing* utilizada tem grande impacto no *runtime* da otimização. Tal experimento influenciou na escolha das técnicas para atraso da interconexão e degradação do *slew* (D2M (ALPERT; DEVGAN; KASHYAP, 2000) e PERI (KASHYAP et al., 2002), respectivamente).

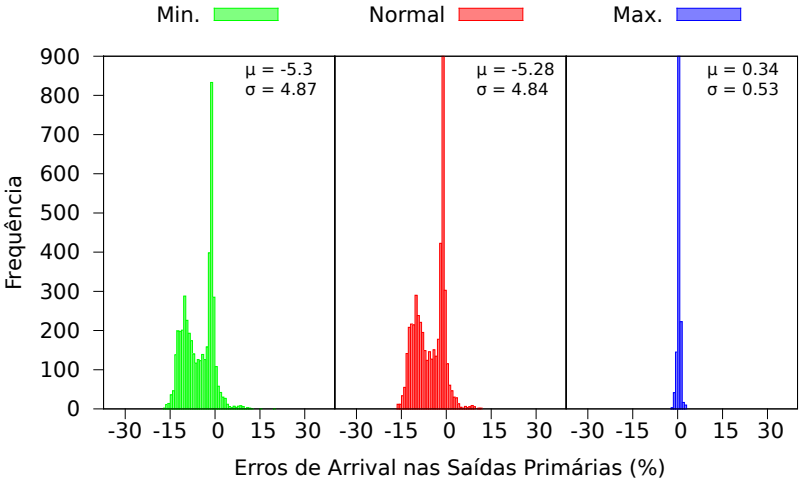
5.4.3 Nível Lógico *versus* Erro Relativo

Neste experimento os erros relativos percentuais dos *arrival times* de saída de cada porta lógica dos circuitos *pci_bridge32* e *matrix_mult* em relação aos *arrival times* reportados pelo *PrimeTime* foram analisados ao decorrer dos níveis lógicos dos circuitos.

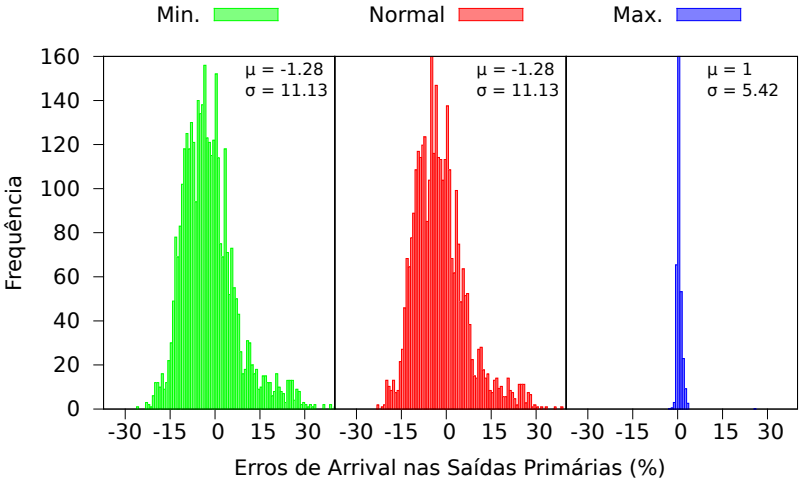
As Figuras 26 e 27 mostram os erros distribuídos pelos níveis lógicos nos circuitos *pci_bridge32* e *matrix_mult*, respectivamente. Em azul, estão apresentados os pontos de cada *arrival time* de saída. Na curva em vermelho, são mostrados os erros para os *arrival times* de saída das portas pertencentes ao caminho crítico. Na curva em verde, são mostrados os erros referentes às portas pertencentes ao caminho com o maior número de portas.

Ao analisar os gráficos das Figuras 26 e 27, observa-se que tanto os erros grandes, quanto os erros pequenos dos primeiros níveis lógicos, ao serem propagados, são estabilizados, convergindo para um número

próximo de -7%, o que se reflete também nos erros dos caminhos críticos e maiores caminhos.

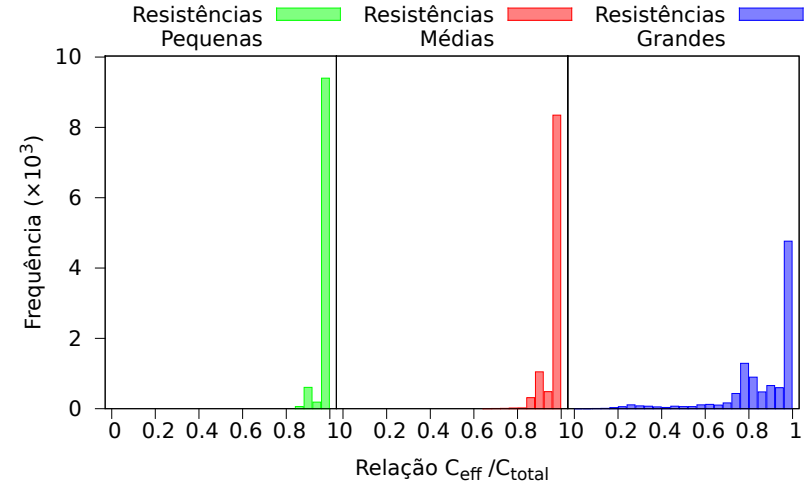


(a)

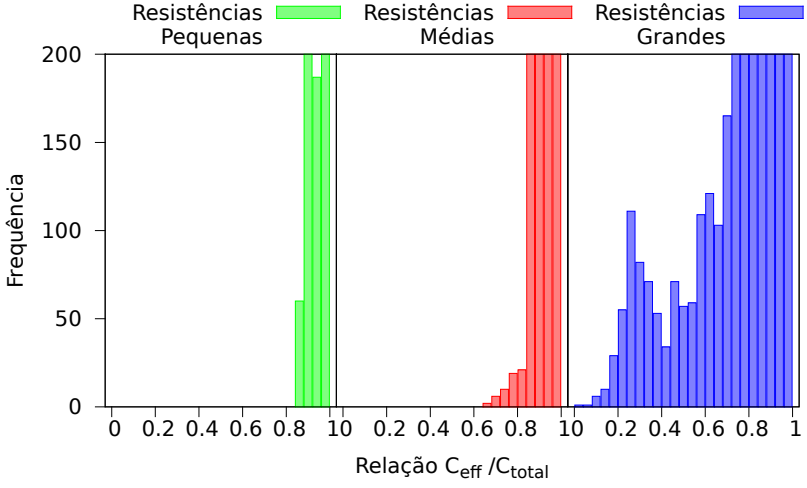


(b)

Figura 23 – Distribuição das frequências dos erros percentuais calculados nas saídas primárias dos circuitos: (a) *matrix_mult*; (b) *pci_bridge*32. Na primeira parte os erros foram calculados considerando a configuração de menor consumo de *leakage*. Na segunda parte, considerando a configuração padrão. E na terceira, considerando a configuração de maior consumo de *leakage*. μ e σ representam a média e desvio padrão das amostras, respectivamente.

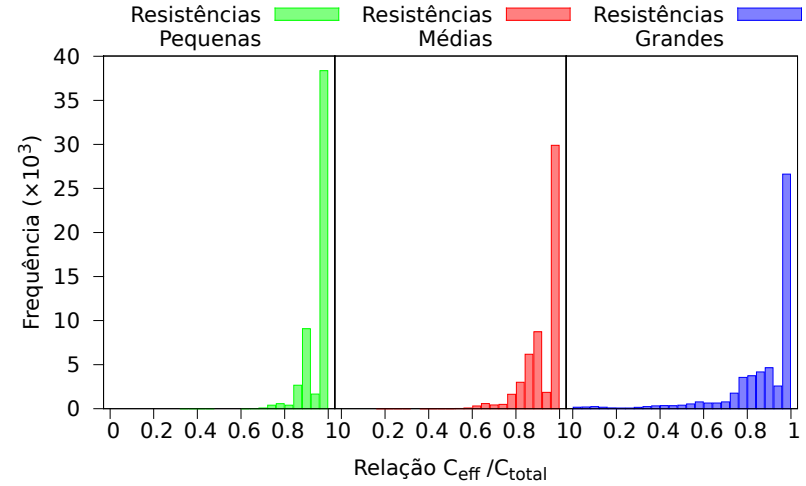


(a)

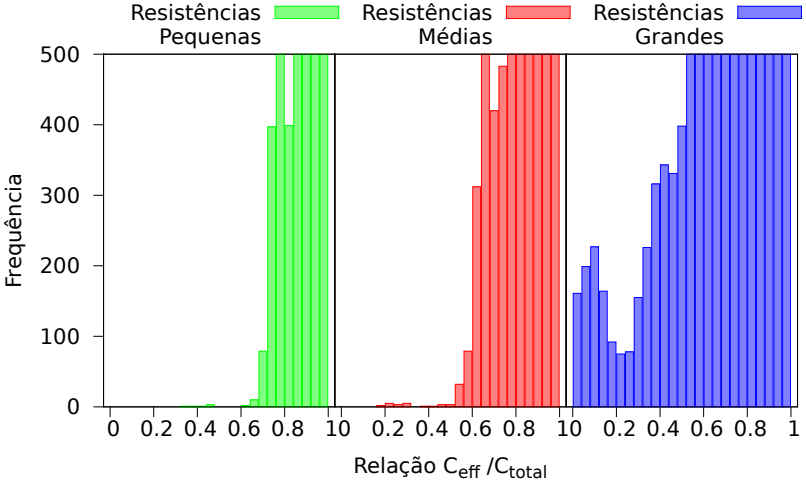


(b)

Figura 24 – Distribuição das frequências das relações C_{eff}/C_{total} das interconexões do circuito *pci_bridge32*. Na primeira parte, as frequências são das interconexões com menor valor de resistência total, na segunda parte, das com valor de resistência total médio, e na terceira, das com maiores valores de resistência total.



(a)



(b)

Figura 25 – Distribuição das frequências das relações C_{eff}/C_{total} das interconexões do circuito *matrix_mult*. Na primeira parte, as frequências são das interconexões com menor valor de resistência total, na segunda parte, das com valor de resistência total médio, e na terceira, das com maiores valores de resistência total.

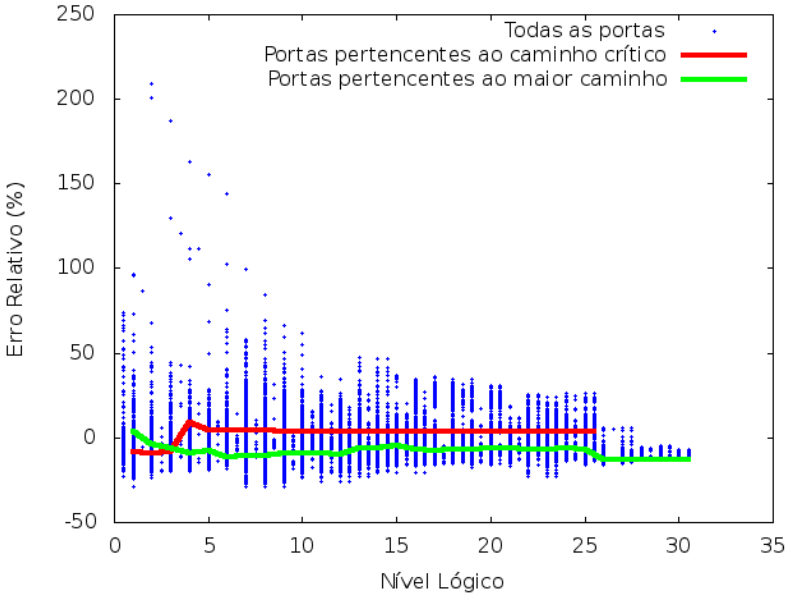


Figura 26 – Erro relativo dos *arrival times* em relação aos resultados obtidos pelo *PrimeTime*, ao decorrer dos níveis lógicos, no *benchmark pci_bridge*32. O *arrival time* utilizado na comparação é o *arrival time* no *timing point* de saída de cada porta lógica. Em azul, cada ponto representa uma porta lógica. Em vermelho, é a curva referente às portas lógicas pertencentes ao caminho crítico. A curva em verde, é referente às portas lógicas pertencentes ao maior caminho, ou seja, ao caminho com maior número de portas.

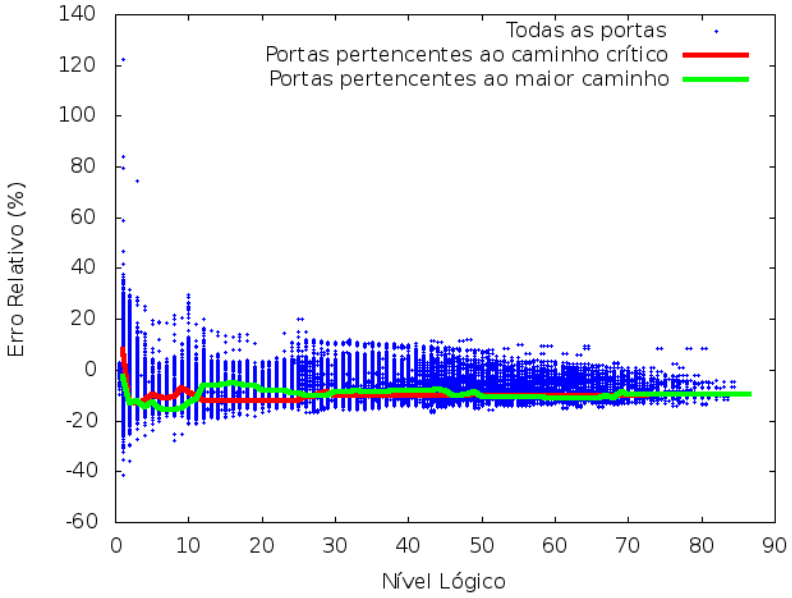


Figura 27 – Erro relativo dos *arrival times* em relação aos resultados obtidos pelo *PrimeTime*, ao decorrer dos níveis lógicos, no *benchmark matrix_mult*. O *arrival time* utilizado na comparação é o *arrival time* no *timing point* de saída de cada porta lógica. Em azul, cada ponto representa uma porta lógica. Em vermelho, é a curva referente às portas lógicas pertencentes ao caminho crítico. A curva em verde, é referente às portas lógicas pertencentes ao maior caminho, ou seja, ao caminho com maior número de portas.

6 CONCLUSÃO

Foram avaliados os impactos das interconexões no contexto da análise de *timing* estática utilizando uma infraestrutura experimental realista.

A consideração do atraso das interconexões no fluxo *standard cell* é de muita importância e a avaliação desses atrasos deve ser eficiente e precisa. Neste trabalho foi possível observar a importância da avaliação dos atrasos das interconexões e também, que a desconsideração da degradação do *slew* através das interconexões pode obter atrasos muito otimistas para o circuito, acarretando erros de cerca de 20% no valor do caminho crítico para os circuitos da competição de *sizing* do ISPD.

A abordagem da capacitância efetiva para interconexões implica na consideração do efeito de *resistive shielding*, o qual impacta na qualidade do cálculo do atraso do circuito. A ferramenta de análise de *timing* desenvolvida neste trabalho implementa a técnica de Puri, Kung e Drumm (2002) para o cálculo da capacitância efetiva, atraso das interconexões e degradação do *slew*. Tal ferramenta apresentou ser cerca de **17,02 vezes mais rápida** que o *PrimeTime*, obtendo resultados para *TNS* e *critical path* que subestimam em cerca de $8,85\%$ e $4,48\%$ respectivamente, os reportados pela ferramenta industrial.

A relação C_{eff}/C_{total} nos circuitos da competição de *sizing* do ISPD de 2013 mostrou-se na média, próxima de 1. A partir dessa informação, o modelo de capacitância concentrada para calcular o atraso dos *drivers*, juntamente com a técnica de Elmore com C_{total} e a técnica de Puri, Kung e Drumm (2002) para degradação do *slew* foi avaliada, apresentando estimativas pessimistas em $10,76\%$ para *TNS* nos circuitos testados (exceto o *netcard*) e $6,48\%$ para *critical path*, sendo que o tempo de execução é cerca de **3 vezes menor** que o da técnica considerando a C_{eff}.

6.1 TRABALHOS FUTUROS

Diversos trabalhos futuros podem ser realizados a fim de complementar a ferramenta avaliada neste trabalho, tais como:

- Investigação detalhada dos erros obtidos pela técnica implementada neste trabalho, quando comparada à ferramenta industrial;
- Avaliação da eficiência da técnica implementada no contexto de

uma técnica de otimização de fluxo *standard cell*, como por exemplo, *gate sizing*;

- Avaliação da técnica implementada utilizando bibliotecas *standard cell* e circuitos comerciais.

REFERÊNCIAS BIBLIOGRÁFICAS

ALPERT, C. J.; DEVGAN, A.; KASHYAP, C. A two moment rc delay metric for performance optimization. In: ACM. Proceedings of the 2000 international symposium on Physical design. [S.l.], 2000. p. 69–74.

BHASKER, J.; CHADHA, R. Static Timing Analysis for Nanometer Designs: A Pratical Approach. 1. ed. [S.l.]: Springer, 2009.

CONG, J. et al. Performance optimization of vlsi interconnect layout. Integr. VLSI J., Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 21, n. 1-2, p. 1–94, nov. 1996. ISSN 0167-9260. Disponível em: <http://dx.doi.org/10.1016/S0167-9260(96)00008-9>.

ELMORE, W. C. The transient response of damped linear networks with particular regard to wideband amplifiers. Journal of Applied Physics, AIP, v. 19, n. 1, p. 55–63, 1948. Disponível em: <http://link.aip.org/link/?JAP/19/55/1>.

GUNTZEL, J. L. A. Functional Timing Analysis of VLSI Circuits Containing Complex Gates. Tese (Doutorado) — Universidade Federal do Rio Grande do Sul, RS-Brazil, 2000.

GUPTA, R. et al. The Elmore Delay as a Bound for RC Trees with Generalized Input Signals. v. 16, n. 1, p. 95–104, 1997.

HOROWITZ, J. R. P. P. M. Signal delay in rc tree networks. IEEE transactions on computer-aided design, v. 2, n. 3, p. 202–211, 1983.

IEEE. Ieee standard for integrated circuit (ic) delay and power calculation system. IEEE Std 1481-1999, p. i–390, 1999.

INTEL. Intel(R) Core(TM) i7-920 Processor Specifications. 2008. Disponível em: <http://ark.intel.com/products/37147/Intel-Core-i7-920-Processor-8M-Cache-2_66-GHz-4_80-GTs-Intel-QPI>. Acesso em: 26/10/2013.

KAHNG, A. B. et al. High-performance gate sizing with a signoff timer. In: ACM. International Conference on Computer-Aided Design (ICCAD). [S.l.]: ACM, 2013.

KASHYAP, C. V.; ALPERT, C. J.; DEVGAN, A. An effective capacitance based delay metric for rc interconnect. In: IEEE PRESS. `Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design`. [S.l.], 2000. p. 229–235.

KASHYAP, C. V. et al. Peri: a technique for extending delay and slew metrics to ramp inputs. In: ACM. `Proceedings of the 8th ACM/IEEE international workshop on Timing issues in the specification and synthesis of digital systems`. [S.l.], 2002. p. 57–62.

LIVRAMENTO, V. dos S. `Sizing Discreto Baseado em Relaxação Lagrangeana para Minimização de Leakage em Circuitos Digitais`. Dissertação (Mestrado), 2013.

ODABASIOGLU, A.; CELIK, M.; PILEGGI, L. T. Prima: passive reduced-order interconnect macromodeling algorithm. In: IEEE COMPUTER SOCIETY. `Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design`. [S.l.], 1997. p. 58–65.

OZDAL, M. M. et al. An improved benchmark suite for the ispd-2013 discrete cell sizing contest. In: `Proceedings of ACM International Symposium on Physical Design`. [S.l.: s.n.], 2013. p. 168–170.

PILLAGE, L. T.; ROHRER, R. A. Asymptotic waveform evaluation for timing analysis. `Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on`, IEEE, v. 9, n. 4, p. 352–366, 1990.

PURI, R.; KUNG, D. S.; DRUMM, A. D. Fast and accurate wire delay estimation for physical synthesis of large asics. In: `Proceedings of the 12th ACM Great Lakes symposium on VLSI`. New York, NY, USA: ACM, 2002. (GLSVLSI '02), p. 30–36. ISBN 1-58113-462-2. Disponível em: <http://doi.acm.org/10.1145/505306.505314>.

RABAEY, J. M.; CHANDRAKASAN, A.; NIKOLIC, B. `Digital Integrated Circuits`. 3rd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2008. ISBN 0132219107, 9780132219105.

RYZHENKO, N.; BURNS, S. Standard cell routing via boolean satisfiability. In: `Proceedings of the 49th Annual Design Automation Conference`. New York, NY, USA: ACM, 2012. (DAC '12), p. 603–612. ISBN 978-1-4503-1199-1. Disponível em: <http://doi.acm.org/10.1145/2228360.2228470>.

SHEEHAN, B. N. Osculating thevenin model for predicting delay and slew of capacitively characterized cells. In: ACM. Proceedings of the 39th annual Design Automation Conference. [S.l.], 2002. p. 866–869.

SILVEIRA, L. M. et al. A coordinate-transformed arnoldi algorithm for generating guaranteed stable reduced-order models of rlc circuits. Computer Methods in Applied Mechanics and Engineering, Elsevier, v. 169, n. 3, p. 377–389, 1999.

SYNOPSYS. Synopsys PrimeTime User's Manual. 2012. Disponível em: <http://www.synopsys.com>. Acesso em: 01/12/2012.

SYNOPSYS. Synopsys PrimeTime SI Version H-2013.06 User Guide. 2013. Disponível em: <http://www.synopsys.com>. Acesso em: 31/10/2013.

WANG, M.; YANG, X.; SARRAFZADEH, M. Dragon2000: standard-cell placement tool for large industry circuits. In: IEEE PRESS. Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design. [S.l.], 2000. p. 260–263.

ZHOU, Y. et al. A more effective ceff for slew estimation. In: IEEE. Integrated Circuit Design and Technology, 2007. ICICDT'07. IEEE International Conference on. [S.l.], 2007. p. 1–4.

ANEXO A – Artigo sobre o TCC

Análise de *Timing* Estática e a Avaliação do Impacto do Atraso das Interconexões em Circuitos Digitais

Chrystian de Sousa Guth[1]

[1]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC)
Campus Universitário – Trindade – Florianópolis – SC – Brasil

csguth@inf.ufsc.br

Abstract. *Static timing analysis is the most used technique to estimate the delay in digital circuits during the Standard Cell flow. As the circuits must be represented in two different parts (the logic gates and interconnects), an efficient technique must be implemented to calculate the interconnect delay. The STA tool implemented in this work do the interconnect delay calculation getting the results that are, in average 4.28% optimistics than those that are obtained by a industrial tool, with a 8 times inferior runtime.*

Resumo. *Análise de timing estática (STA: Static Timing Analysis) é a técnica mais utilizada para estimar o atraso de circuitos digitais durante o fluxo de síntese física. Como os circuitos devem ser modelados em duas partes (portas lógicas e interconexões), uma técnica eficiente deve ser implementada para se estimar os atrasos das interconexões. A ferramenta de STA implementada neste trabalho realiza o cálculo dos atrasos das interconexões gerando resultados que são, em média, 4,28% mais otimistas do que aqueles gerados pela ferramenta Synopsys PrimeTime, porém com tempo de execução cerca de 8 vezes menor.*

1. Introdução

O crescimento da complexidade dos circuitos digitais contemporâneos[1] e a necessidade de um *time-to-market* (tempo de entrega ao mercado) curto faz com que o projeto de tais circuitos adote o fluxo *standard cell* (Figura 1).

No fluxo *standard cell* as portas lógicas são caracterizadas e validadas previamente em uma dada tecnologia, originando as chamadas "células". Essas células [2] são catalogadas com suas diversas características elétricas em uma biblioteca, podendo ser reutilizadas em diversos projetos que usem a mesma tecnologia. O reuso amortiza o custo dos projetos inseridos neste nodo tecnológico e possibilita um *time-to-market* mais curto.

Diversas otimizações são realizadas no decorrer do fluxo de projeto *standard cell* e o uso de ferramentas para a automação de projeto eletrônico (*EDA: Electronic design automation*) é indispensável em suas diferentes etapas. A inexistência de

[1]Um processador para *desktop* desenvolvido no ano de 2008 tem cerca de 731 milhões de transistores, excluindo a área de memória [Intel 2008].

[2]Célula é a instância de *layout* para a implementação física de uma porta lógica.

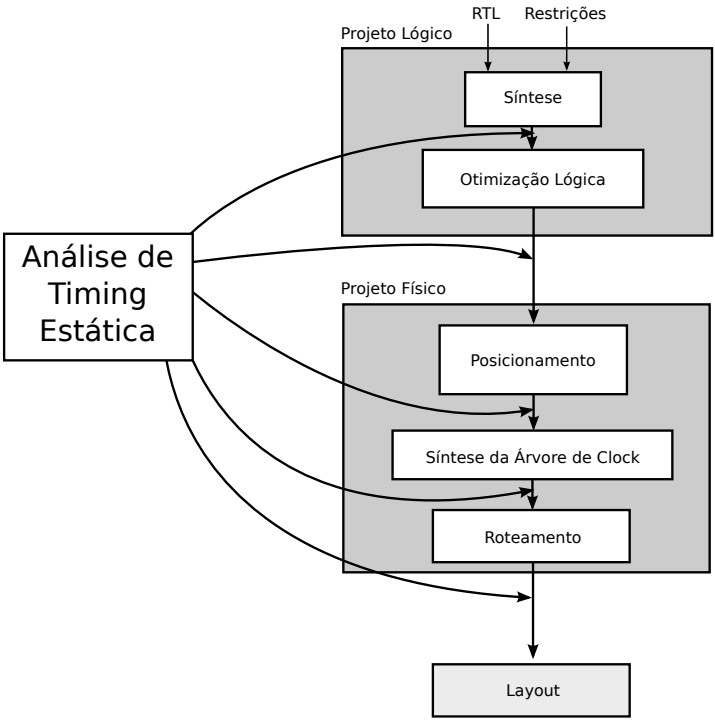


Figura 1. Fluxo de projeto *Standard Cell*. Adaptado de [Bhasker and Chadha 2009].

ferramentas de análise de *timing* estática precisas de domínio público e a restrição no acesso à ferramentas industriais (devido ao alto custo de suas licenças) resultam em um problema de infraestrutura de pesquisa. Assim, este trabalho tem como resultado uma alternativa de ferramenta de análise de *timing* para projetistas de circuitos digitais, bem como uma infraestrutura realista e precisa para desenvolvedores de ferramentas, que necessitam da análise de *timing* em alguma etapa do fluxo de projeto *standard cell*.

Este artigo se organiza da seguinte maneira: Na Seção 2 serão apresentados alguns conceitos básico para entendimento das técnicas que serão apresentadas posteriormente. A seguir, na Seção 3 será apresentada a técnica utilizada neste trabalho para cálculo das informações temporais dos circuitos digitais. Na Seção 4 será apresentada a análise de *timing* estática, bem como algumas particularidades no seu algoritmo. Finalmente, nas Seções 5 e 6 serão apresentados alguns resultados obtidos nos experimentos realizados e as conclusões finais, respectivamente.

2. Conceitos

As características temporais do circuito são derivadas das características temporais de suas partes, quais sejam, as portas lógicas e as interconexões que o compõem. O atraso das portas lógicas são obtidos das pré-caracterizações presentes nas bibliotecas de células. No que se refere às interconexões, elas geralmente são modeladas como árvores RC (Figura 2). Segundo [Rabaey et al. 2008], uma árvore RC possui três características: não possui *loops* resistivos; cada nodo possui uma capacitância com o *ground*; cada nodo possui um resistor que o conecta com o seu nodo pai.

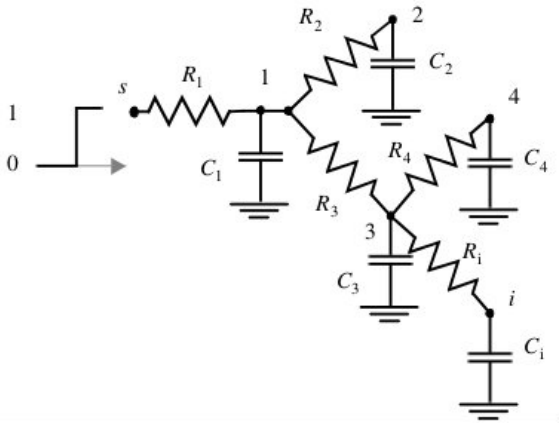


Figura 2. Uma árvore RC. Obtida de [Rabaey et al. 2008].

A Figura 3 ilustra as três principais contribuições das interconexões, sobre o atraso do circuito:

- **Capacitância Vista Pelo *Driver*:** É necessário modelar a carga capacitiva a ser carregada pelo *driver* da interconexão com o objetivo de se obter a informação de *load*, a qual é utilizada no cálculo do *delay* e *slew* dos *timing arcs* das portas lógicas, como visto anteriormente. Nesta capacitância é incluído também o impacto causado pelos pinos de destino da interconexão [3]. Na fase *pré-layout*, essa estimativa é realizada somando a capacitância total da interconexão com a capacitância de cada pino de destino dela. Porém, ao se tratar de interconexões com característica resistiva, o uso da abordagem de capacitância concentrada é impreciso. Para que os modelos de atraso não-lineares, que dependem do valor de capacitância de saída, sejam utilizados para os *drivers* diretamente, é necessário o uso de uma abordagem conhecida como **Capacitância Efetiva** (C_{eff}). Tal abordagem tenta encontrar um valor de capacitância que pode ser utilizado como carga equivalente, em termos de *timing*, para a saída do *driver* [Bhasker and Chadha 2009].

[3]Um pino de destino de uma interconexão é um pino que se liga na interconexão, que não é o pino *driver*. Por exemplo, na Figura 3(a), os pinos de destino da interconexão são o segundo pino de entrada da porta $u2$, o pino de entrada da porta $u3$ e o pino d do *flip-flop* $f1$.

- **Atraso da Interconexão:** Além do impacto local nos *delays* e *slews* de seus *drivers*, as interconexões exercem impacto global no circuito, com seu próprio atraso de propagação (Figura 3(b)), devido a sua característica resistiva. Com a alta frequência de operação dos circuitos digitais atuais e o dimensionamento dos transistores para escalas nanométricas, os atrasos das interconexões, que antes não eram significativos, hoje chegam consumir de 50% a 70% do ciclo do relógio, e esta porcentagem tende a aumentar na medida que os transistores diminuem [Cong et al. 1996]. Uma das métricas mais populares para se calcular o atraso em interconexões é o atraso de Elmore (*Elmore Delay*) [Elmore 1948], pela simplicidade e razoável correlação com os atrasos reais.
- **Degradação do *Slew*:** O cálculo do *slew* é crucial para determinar a precisão de uma avaliação de *timing* em um circuito digital [Zhou et al. 2007]. Os *delays* dos *timing arcs* dependem do *slew* de entrada e do *slew* de saída. Quando um sinal se propaga por uma interconexão, seu *slew* (i.e., sua declividade) sofre uma degradação devido ao efeito resistivo da mesma (Figura 3(b)). A não-modelagem desta degradação pela interconexão, acarreta em erros de até 50% [Sheehan 2002]. A abordagem para degradação do *slew* utilizada neste trabalho será apresentada na Seção 3.

3. Cálculo das Características Temporais da Interconexão

Para que o atraso de uma interconexão seja estimado com precisão, um modelo de grafo (Figura 4) pode ser utilizado para representar o fio em termos de capacitâncias e resistências.

No modelo de grafo $I(C, R)$ utilizado, o conjunto dos vértices é composto pelos nodos internos da interconexão, que representam cada capacitor. As arestas do grafo modelam os resistores, e cada resistor conecta um par de capacitores. Sendo assim:

- $\mathbf{C} = \{c | c$ é um capacitor da rede RC$\}$
- $\mathbf{R} = \{(c, d) |$ existe um resistor que conecta os capacitores c e $d\}$

A técnica de [Elmore 1948] é uma técnica baseada no primeiro momento da resposta ao impulso amplamente utilizada no cálculo dos atrasos das interconexões. De acordo com [Rabaey et al. 2008], em um nodo c_i da árvore RC, o atraso de Elmore (τ_i) pode ser facilmente calculado como:

$$\tau_i = \sum_{k=1}^{N} C_k R_{ik} \tag{1}$$

Dado o grafo de uma árvore RC[4], o algoritmo apresentado nessa seção calcula os valores de capacitância efetiva e *slew* em cada nodo interno da interconexão. Para o atraso da interconexão, o método implementa a técnica de Elmore utilizando os valores de capacitância efetiva, ao invés dos valores de capacitância total *downstream*, simulando o efeito de *resistive shielding*.

[4]Com os nodos numerados de 1 a n em ordem topológica, onde n é o tamanho do conjunto de vértices e o nodo c_1 é o nodo fonte da árvore RC.

A capacitância efetiva é denotada em cada nodo c_i por C_{eff_i} e o *slew* por $slew_i$. O valor do *slew* aplicado no nodo fonte da árvore RC, denotado por $slew_1$, é exatamente o valor do *slew* na saída do *driver* desta interconexão, o qual é função do *slew* na entrada da porta lógica *driver* e da capacitância efetiva vista na saída. Este valor de *slew* será refinado iterativamente para se estimar o valor de capacitância efetiva da interconexão.

O algoritmo para cálculo iterativo da capacitância efetiva de uma interconexão, bem como seu atraso e a degradação no *slew*, conforme [Puri et al. 2002], ocorre em cinco passos:

1. **Inicialização:**
   (a) A capacitância efetiva C_{eff_i} de cada nodo c_i da Árvore RC é inicializada com o valor de capacitância total *downstream* de c_i, ou seja $C_{eff_i} = C_{total_i}$;
   (b) O *slew* no nodo fonte da árvore RC $slew_1$ é calculado utilizando o modelo de atraso da porta lógica *driver*, considerando a capacitância concentrada da árvore RC (i.e., $\sum_{i=1}^{N} C_i$): $slew_1 = f(C_{total_1})$.
2. **Atualização dos *slews* em ordem topológica:**
   (a) Atraso τ_i do nodo fonte c_1 para cada nodo c_i da árvore é calculado utilizando a técnica de Elmore (Equação 1), substituindo C_{eff_i} por C_{total_i}, para simular o efeito de *resistive shielding*;
   (b) A degradação do *slew* em cada nodo c_i é calculada utilizando a Equação 2.

$$slew_i = \frac{slew_j}{1 - \frac{R_i\ C_{eff_i}}{slew_j}(1 - e^{-\frac{slew_j}{R_i\ C_{eff_i}}})} \tag{2}$$

3. **Atualização das capacitâncias efetivas em ordem topológica reversa:**
   (a) A capacitância efetiva (C_{eff_i}) de cada nodo c_i é calculada como a soma da capacitância do nodo c_i e todas as capacitâncias dos nodos filhos:

$$C_{eff_i} = C_i + \sum_{j \in filhos(i)} K_j \times C_{tot_j} \tag{3}$$

   Onde K_j é o fator de *shielding*, definido por:

$$K_j = 1 - \frac{2R_jC_{eff_j}}{slew_i}(1 - e^{-\frac{slew_i}{2R_jC_{eff_j}}}) \tag{4}$$

   Onde R_j é o valor da resistência que conecta o nodo c_j ao seu pai, no caso, c_i.
4. **Atualização do *Slew* do *Driver*:** O *slew* no nodo fonte $slew_1$ é calculado diretamente, utilizando o C_{eff_1} atual ;
5. **Iteração:** Os passos de 2 até 4 são repetidos até que $slew_1$ convirja, dada uma precisão ε .

Na implementação apresentada neste trabalho, o ε foi definido como sendo 1% e na maioria dos casos observados, cerca de 5 iterações são necessárias para realizar o cálculo da capacitância efetiva [Puri et al. 2002]. Como cada iteração do algoritmo percorre a lista em ordem topológica (direta e reversa), a complexidade assintótica de pior caso de cada iteração do algoritmo é de $O(n)$ onde n é o número de nodos da árvore, ao passo que a complexidade do algoritmo é $O(c.n)$, onde c é o número de iterações. Entretanto, como o número de iterações é na grande maioria dos casos menor que 5 (E portanto c é muito menor que n), assume-se que o crescimento no tempo de execução tem comportamento linear.

4. Análise de *Timing* Estática

O objetivo desta seção é apresentar a análise de *timing* estática (*STA: Static Timing Analysis*), bem como os conceitos importantes referentes à esta técnica, juntamente com o algoritmo de STA.

Análise de *timing* estática, ou *static timing analysis* [Guntzel 2000] [Bhasker and Chadha 2009], é uma das técnicas utilizadas para se estimar o atraso crítico de circuitos digitais. A análise de *timing* é chamada de estática quando ela não realiza simulação e portanto, independe de estímulos de entrada, considerando apenas a topologia do circuito. É um processo completo e exaustivo [Bhasker and Chadha 2009] que verifica as mais diversas informações de *timing* em um circuito, como os *delays*, *slews*, *slacks* (folgas), *required times* (tempos requeridos) e diversas violações de restrições de projeto.

Dada a descrição do projeto usando alguma linguagem de descrição de hardware (*HDL: Hardware Description Language*), restrições de projeto e uma biblioteca de células, o objetivo da análise de *timing* é apresentar informações temporais em todos os pontos do circuito e apontar as possíveis violações. Essas informações são utilizadas para avaliar se o projeto sob verificação pode operar na velocidade estipulada, ou seja, se o circuito final poderá funcionar com segurança na frequência de relógio escolhida, sem que existam violações nas restrições de projeto.

O fluxo básico de uma ferramenta de análise de *timing* é:

1. **Leitura dos arquivos de entrada:** Nesta etapa, os arquivos referentes às bibliotecas de célula, descrição do circuito juntamente com as restrições do projeto são lidos e suas informações são armazenadas em estruturas de dados, que serão consultadas na geração do modelo de grafo e na atualização das informações temporais;
2. **Geração do grafo de *timing*:** Responsável por implementar o modelo de grafo de *timing*. As estruturas de dados utilizadas na implementação do modelo de grafo têm impacto direto no desempenho da ferramenta de *timing*.
3. **Atualização de informações temporais:** Etapa onde a propagação dos atrasos através dos *timing arcs*, bem como a avaliação do cumprimento ou não das restrições de desempenho são realizados.

Na STA, o circuito pode ser representado como um grafo direcionado acíclico (*DAG: Directed Acyclic Graph*), onde o conjunto dos vértices representa os pinos de

saída e entrada das portas lógicas e o conjunto das arestas representa as interconexões e os *timing arcs*. Para melhorar a eficiência da ferramenta de STA, os vértices são armazenados em listas ordenadas topologicamente.

As estruturas de dados utilizadas para armazenar os elementos do grafo são essencialmente listas ordenadas topologicamente. Em uma lista ordenada topologicamente, dado um elemento i, à esquerda necessariamente se encontram os elementos de mesmo ou menor nível lógico, e à direita, de nível igual ou maior, como mostrado na Figura 5(b). Da mesma maneira, os *timing arcs* e as interconexões também são ordenados topologicamente, em suas respectivas listas. Com essa escolha, o algoritmo de análise de *timing* estática passa a ser apenas de uma varredura em ordem, na lista de *timing points*, atualizando a informação de *timing* acumulada para cada vértice do grafo.

5. Resultados

Essa seção tem por objetivo descrever os experimentos realizados neste trabalho e apresentar os resultados obtidos.

Como parte dos experimentos é realizada comparando as informações calculadas pela ferramenta implementada com as informações reportadas pelo *PrimeTime*, o erro percentual (EP_t) e o erro médio percentual absoluto ($EMPA$) (Equações 5 e 6) foram adotados como métricas para estimar a qualidade das informações de *timing* reportadas pela ferramenta implementada neste trabalho.

$$EP_t = \frac{(A_t - P_t)}{A_t} \times 100 \tag{5}$$

$$EMPA = \frac{\sum_{t=1}^{n} |EP_t|}{n} \tag{6}$$

O erro percentual é calculado para cada uma das informações comparadas com o *PrimeTime* utilizando a equação 5, sendo que A_t é a informação obtida pelo *PrimeTime* e P_t é a informação calculada pela ferramenta implementada. Tais informações usadas para fim de validação da ferramenta foram:

- ***TNS (Total Negative Slack)*:** O somatório de *slack* negativo nas saídas primárias.
- ***Violating POs:*** Número de saídas primárias violando a restrição de desempenho mínimo.
- ***Runtime (s):*** Tempo de execução, em segundos, para realizar uma análise de *timing* em um circuito, desconsiderando o tempo constante de inicialização da ferramenta.
- ***Critical Path:*** Valor do caminho crítico do circuito.

Os resultados dos experimentos serão apresentados posteriormente por meio de gráficos, tabelas e histogramas.

Este trabalho utilizou como base a infraestrutura disponibilizada pela competição de *gate sizing* discreto do ISPD de 2013, a qual fornece:

- Um conjunto de 8 circuitos da competição do ISPD de 2013:
    1. *usb_phy*: com 511 células combinacionais, 98 células sequenciais, 15 entradas e 19 saídas primárias;
    2. *pci_bridge*32: com 27316 células combinacionais, 3359 células sequenciais, 160 entradas e 201 saídas primárias;
    3. fft: com 30297 células combinacionais, 1984 células sequenciais, 1026 entradas e 1026 saídas primárias;
    4. *cordic*: com 40371 células combinacionais, 1230 células sequenciais, 34 entradas e 64 saídas primárias;
    5. des_perf: com 103842 células combinacionais, 8802 células sequenciais, 234 entradas e 201 saídas primárias;
    6. *edit_dist*: com 125000 células combinacionais, 5661 células sequenciais, 2562 entradas e 12 saídas primárias;
    7. *matrix_mult*: com 30297 células combinacionais, 1984 células sequenciais, 3202 entradas e 1600 saídas primárias;
    8. *netcard*: com 884427 células combinacionais, 97831 células sequenciais, 1836 entradas e 10 saídas primárias.
- Uma biblioteca *standard cell* realista, composta por onze células combinacionais de diversas funções lógicas e um célula sequencial;
- Uma ferramenta de análise de *timing* estática PrimeTime ® da empresa [Synopsys 2012] para comparação de resultados;

Os circuitos são compostos por descrições no formato Verilog, capacitâncias parasitas e resistências descritas no formato IEEE SPEF (*Standard Parasitic Exchange Format*) [IEEE 1999], e restrições de *timing* descritas no formato SDC (*Synopsys Design Constraints*).

5.1. Validação do Modelo de Capacitância Concentrada Perante Ferramenta Industrial

Este experimento tem por objetivo validar a ferramenta de *STA* desenvolvida perante a ferramenta industrial *PrimeTime*, utilizando a abordagem de capacitância concentrada para modelar as interconexões. Para uma comparação justa, a ferramenta industrial foi também configurada para utilizar este modelo. Para tanto, utilizou-se um computador *desktop* com processador *Intel Core i7*, de 4 núcleos, e 4GB de *RAM*, e os resultados deste experimento são apresentados na Tabela 1.

As células marcadas correspondem aos valores que são menores que os obtidos na ferramenta comercial. A penúltima linha apresenta a média dos valores calculados para cada coluna da tabela. A última linha mostra o $EMPA$ para cada uma das informações mostradas nas colunas. Os valores de $EMPA$ valendo $0,00\%$ indicam que a ferramenta calcula os mesmos valores que a ferramenta industrial para as informações comparadas. A média de *runtime* obtida é $6,92$ vezes menor que a média da ferramenta industrial, sendo $50,05$ vezes menor para os 7 primeiros circuitos (excluindo o *netcard*). No circuito *usb_phy*, a diferença de *runtime* é de 20 vezes e nos outros circuitos (exceto o *netcard*) a diferença tem valor médio de $52,71$ vezes com baixo desvio padrão $(6,48)$.

Lumped Capacitance Interconnect Model

BENCHMARK	TNS (ps)	Viol. POs	Critical Path (ps)	Runtime (s)
usb_phy	0,00E+00	0	3,40E+02	0,00
pci_bridge32	2,08E+03	46	1,05E+03	0,02
fft	0,00E+00	0	1,51E+03	0,03
cordic	2,98E+04	185	3,16E+03	0,03
des_perf	0,00E+00	0	9,24E+02	0,09
edit_dist	0,00E+00	0	2,92E+03	0,11
matrix_mult	0,00E+00	0	2,04E+03	0,14
netcard	2,60E+06	11925	3,11E+03	24,83
Média	**3,29E+05**	**1519,5**	**1,88E+03**	**3,16**
EMPA	**0,00**	**0,00**	**0,00**	**-**

Tabela 1. Comparação das informações de *timing* calculadas pela ferramenta implementada *versus* informações fornecidas pelo *PrimeTime*, utilizando o modelo de interconexões de capacitância concentrada.

5.2. Validação da Técnica Implementada Perante Ferramenta Industrial

Esta seção tem por objetivo comparar a qualidade das informações de *timing* obtidas pela ferramenta de análise de *timing* implementada neste trabalho com as informações reportadas pelo *PrimeTime*. Utilizando as técnicas apresentadas na Seção 3 (i. e., capacitância efetiva, atraso de interconexões e degradação de *slew*), a análise de *timing* estática foi aplicada nos circuitos de teste e suas soluções foram comparadas com as fornecidas pela ferramenta industrial. Tal comparação foi realizada com base nas métricas apresentada na Seção 5. Os resultados deste experimento podem ser vistos na Tabela 2.

A penúltima linha apresenta a média dos valores de cada coluna e a última linha apresenta o $EMPA$ para cada uma das informações em relação ao *PrimeTime*. Com esse experimento, conclui-se que a ferramenta desenvolvida neste trabalho fornece informações de *timing* próximas às reportadas pelo *PrimeTime*[5], com um tempo de execução $17,02$ vezes menor.

As células marcadas apresentam os valores que são otimistas em relação do *PrimeTime* (i. e., que são menores que os obtidos pelo *PrimeTime*), correspondendo a 29 dos 36 valores obtidos.

O erro de menor valor absoluto para o TNS é de $1,34\%$ e o de maior é $16,7\%$ nos circuitos *usb_phy* e *netcard*, respectivamente, sendo que no segundo, o erro reflete em uma aproximação otimista, e no primeiro, pessimista.

Na média, a análise de *timing* na ferramenta desenvolvida é otimista em relação ao *PrimeTime*, de acordo com o grande número de células marcadas. É possível observar também que os maiores erros são obtidos nos maiores circuitos e

[5] $EMPA = 8,21$ e $4,48$ para TNS e *critical path*, respectivamente.

C_{eff} + Elmore (C_{eff}) + Slew Degradation				
BENCHMARK	**TNS (ps)**	**Viol. POs**	**Critical Path (ps)**	**Runtime (s)**
usb_phy	4,91E+04	59	1,19E+03	0,01
pci_bridge32	9,80E+06	3002	6,29E+03	0,31
fft	2,34E+07	1983	1,22E+04	0,45
cordic	1,27E+07	1206	1,55E+04	0,58
des_perf	1,12E+07	1648	1,08E+04	1,08
edit_dist	2,95E+07	3311	1,11E+04	1,79
matrix_mult	1,38E+07	2831	1,11E+04	2,15
netcard	2,17E+06	8944	3,00E+03	12,82
Média	**1,28E+07**	**2873,00**	**8,89E+03**	**2,40**
EMPA	**8,81**	**4,17**	**4,48**	**-**

Tabela 2. Valores obtidos pela ferramenta implementada neste trabalho nos circuitos da competição de *sizing* do ISPD.

os menores erros, nos menores circuitos.

No experimento mostrado na Tabela 3, o modelo de capacitância concentrada foi utilizado para modelar a carga vista pelo *driver*. Para o atraso das interconexões, a técnica de Elmore com capacitância concentrada foi utilizada. Já para a degradação do *slew*, foi utilizada a técnica descrita na Seção 3. Como esperado, os modelos utilizados neste experimento refletem em uma aproximação pessimista para o atraso do circuito[6], já que a técnica de Elmore pura[7] foi aplicada no cálculo dos atrasos das interconexões. A técnica obteve $0,13\%$ e $311,79\%$ de erro para TNS nos circuitos *matrix_mult* e *netcard*, respectivamente. Já para *critical path*, os erros obtidos vão de $1,94\%$ até $13,85\%$, nos circuitos *des_perf* e *usb_phy*, respectivamente.

A importância do cálculo da degradação do *slew* pode ser visualizado na Tabela 4. Os erros obtidos neste experimento ($40,80\%$ para TNS e $21,21\%$ para critical path) mostram resultados muito otimistas em relação ao *PrimeTime*, quando a técnica apresentada na Seção 3 é aplicada, sem considerar a degradação do *slew* nos destinos das interconexões.

6. Conclusões

Foram avaliados os impactos das interconexões no contexto da análise de *timing* estática utilizando uma infraestrutura experimental realista.

A consideração do atraso das interconexões no fluxo *standard cell* é de muita importância e a avaliação desses atrasos deve ser eficiente e precisa. Neste trabalho foi possível observar a importância da avaliação dos atrasos das interconexões e também, que a desconsideração da degradação do *slew* através das interconexões

[6]Informação obtida do baixo número de células marcadas (total de 4, com exceção das células de *runtime*).

[7]Sem utilizar a abordagem de capacitância efetiva.

C_{total} + Elmore (C_{total}) + Slew Degradation

BENCHMARK	TNS (ps)	Viol. POs	Critical Path (ps)	Runtime (s)
usb_phy	6,25E+04	61	1,34E+03	0,00
pci_bridge32	1,16E+07	3070	6,57E+03	0,09
fft	2,79E+07	1983	1,38E+04	0,12
cordic	1,54E+07	1207	1,72E+04	0,14
des_perf	1,25E+07	1648	1,13E+04	0,35
edit_dist	3,50E+07	3508	1,24E+04	0,44
matrix_mult	1,62E+07	2851	1,19E+04	0,56
netcard	1,07E+07	36934	3,37E+03	6,55
Média	**1,62E+07**	**6407,75**	**9,72E+03**	**1,03**
EMPA	**48,32**	**27,59**	**6,48**	**-**

Tabela 3. Experimentos utilizando o modelo capacitância concentrada para carga de saída dos *drivers*, técnica de Elmore para computar os atrasos das interconexões, e degradação do *slew* conforme apresentado na Seção 3.

C_{eff} + Elmore (C_{eff}) + No Slew Degradation

BENCHMARK	TNS (ps)	Viol. POs	Critical Path (ps)	Runtime (s)
usb_phy	3,14E+04	57	9,51E+02	0,00
pci_bridge32	7,46E+06	2847	5,48E+03	0,29
fft	1,88E+07	1983	1,03E+04	0,40
cordic	9,32E+06	1204	1,27E+01	0,48
des_perf	8,67E+06	1648	9,22E+03	0,96
edit_dist	2,02E+07	3139	9,11E+03	1,54
matrix_mult	9,07E+06	2639	9,28E+03	1,89
netcard	1,41E+05	1033	2,63E+03	12,81
Média	**9,21E+06**	**1818,75**	**5,87E+03**	**0,79**
EMPA	**40,80**	**14,16**	**29,21**	**-**

Tabela 4. Experimentos utilizando o modelo capacitância efetiva para carga de saída dos *drivers*, técnica de Elmore utilizando as capacitâcias efetivas de cada nodo interno das interconexões, para computar seus atrasos. Neste experimento, a degradação do *slew* não foi considerada.

pode obter atrasos muito otimistas para o circuito, acarretando erros de cerca de 20% no valor do caminho crítico para os circuitos da competição de *sizing* do ISPD.

A abordagem da capacitância efetiva para interconexões implica na consideração do efeito de *resistive shielding*, o qual impacta na qualidade do cálculo do atraso do circuito. A ferramenta de análise de *timing* desenvolvida neste trabalho

implementa a técnica de [Puri et al. 2002] para o cálculo da capacitância efetiva, atraso das interconexões e degradação do *slew*. Tal ferramenta apresentou ser cerca de **17,02 vezes mais rápida** que o *PrimeTime*, obtendo resultados para *TNS* e *critical path* que subestimam em cerca de $8,85\%$ e $4,48\%$ respectivamente, os reportados pela ferramenta industrial.

A relação C_{eff}/C_{total} nos circuitos da competição de *sizing* do ISPD de 2013 mostrou-se na média, próxima de 1. A partir dessa informação, o modelo de capacitância concentrada para calcular o atraso dos *drivers*, juntamente com a técnica de Elmore com C_{total} e a técnica de [Puri et al. 2002] para degradação do *slew* foi avaliada, apresentando estimativas pessimistas em $10,76\%$ para *TNS* nos circuitos testados (exceto o *netcard*) e $6,48\%$ para *critical path*, sendo que o tempo de execução é cerca de **3 vezes menor** que o da técnica considerando a C_{eff}.

6.1. Trabalhos Futuros

Diversos trabalhos futuros podem ser realizados a fim de complementar a ferramenta avaliada neste trabalho, tais como:

- Investigação detalhada dos erros obtidos pela técnica implementada neste trabalho, quando comparada à ferramenta industrial;
- Avaliação da eficiência da técnica implementada no contexto de uma técnica de otimização de fluxo *standard cell*, como por exemplo, *gate sizing*;
- Avaliação da técnica implementada utilizando bibliotecas *standard cell* e circuitos comerciais.

Referências

Bhasker, J. and Chadha, R. (2009). *Static Timing Analysis for Nanometer Designs: A Pratical Approach*. Springer, 1 edition.

Cong, J., He, L., Koh, C.-K., and Madden, P. H. (1996). Performance optimization of vlsi interconnect layout. *Integr. VLSI J.*, 21(1-2):1–94.

Elmore, W. C. (1948). The transient response of damped linear networks with particular regard to wideband amplifiers. *Journal of Applied Physics*, 19(1):55–63.

Guntzel, J. L. A. (2000). *Functional Timing Analysis of VLSI Circuits Containing Complex Gates*. PhD thesis, Universidade Federal do Rio Grande do Sul, RS-Brazil.

IEEE (1999). Ieee standard for integrated circuit (ic) delay and power calculation system. *IEEE Std 1481-1999*, pages i–390.

Intel (2008). Intel(r) core(tm) i7-920 processor specifications.

Puri, R., Kung, D. S., and Drumm, A. D. (2002). Fast and accurate wire delay estimation for physical synthesis of large asics. In *Proceedings of the 12th ACM Great Lakes symposium on VLSI*, GLSVLSI '02, pages 30–36, New York, NY, USA. ACM.

Rabaey, J. M., Chandrakasan, A., and Nikolic, B. (2008). *Digital Integrated Circuits*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition.

Sheehan, B. N. (2002). Osculating thevenin model for predicting delay and slew of capacitively characterized cells. In *Proceedings of the 39th annual Design Automation Conference*, pages 866–869. ACM.

Synopsys (2012). Synopsys primetime user's manual.

Zhou, Y., Li, Z., Kanj, R., Papa, D., Nassif, S., and Shi, W. (2007). A more effective ceff for slew estimation. In *Integrated Circuit Design and Technology, 2007. ICICDT'07. IEEE International Conference on*, pages 1–4. IEEE.

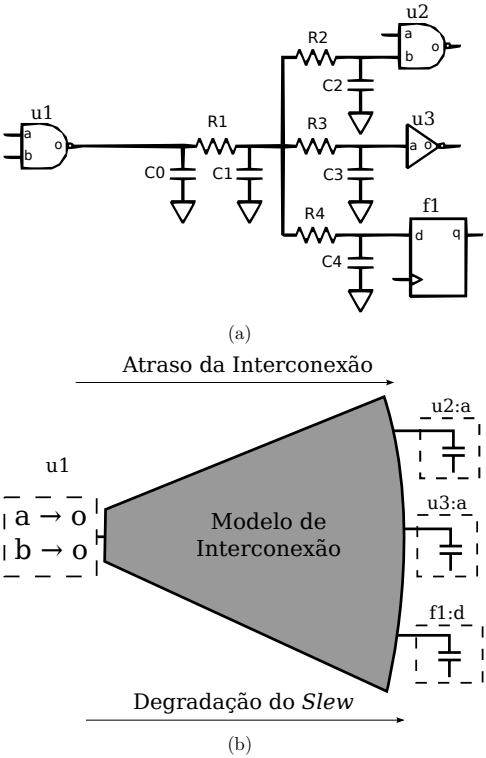


Figura 3. (a) Um circuito composto por três portas lógicas ($u1$, $u2$ e $u3$), uma célula sequencial ($f1$) e uma interconexão em forma de árvore RC, que liga a saída de $u1$ às entradas de $u2$, $u3$ e $f1$; (b) São apresentadas as modelagens para os *timing arcs* da porta lógica $u1$; O modelo da interconexão é abstraído, recebendo um valor de capacitância efetiva. As setas indicam que a interconexão oferece um atraso e uma degradação no *slew*. Cada destino da interconexão é representado como um valor de capacitância de seus pinos de entrada.

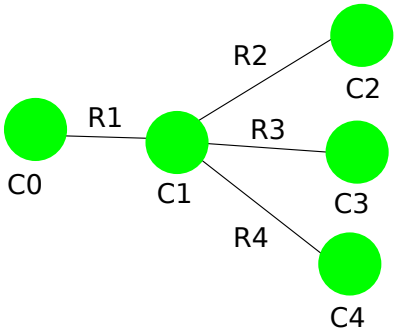


Figura 4. O grafo correspondente à interconexão da Figura 3(a), com cinco vértices e quatro arestas.

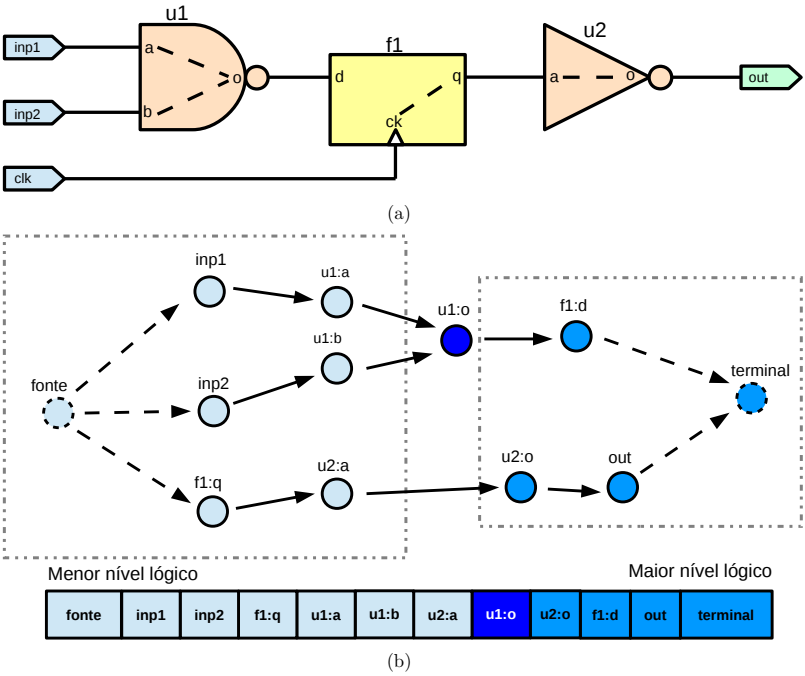


Figura 5. (a) Circuito *simple* retirado do banco de *benchmarks* da competição de *sizing* do ISPD; (b) Grafo correspondente ao circuito da letra (a).

ANEXO B – Código fonte da ferramenta em C++

```
#ifndef CEFF_RATIO_EXPERIMENT_H
#define CEFF_RATIO_EXPERIMENT_H

#include "timing_analysis.h"
#include <queue>
using std::priority_queue;

#include <ostream>
using std::ostream;

#include "transitions.h"
class Ceff_Ratio_Experiment
{

    struct ratio_t {
        double ratio;
        double total_resistance;
        double slew;

        bool operator >(const ratio_t & o) const
        {
            return true;
            return ratio > o.ratio;
        }
        friend ostream & operator << (ostream & out, ratio_t
             & r)
        {
            return out << r.ratio << "\t" << r.
                total_resistance;
        }
    };

    struct resistance_comparator {
        bool operator()(const ratio_t & a, const ratio_t & b
            )
        {
            return a.total_resistance > b.total_resistance;
        }
    };

    struct slew_comparator {
        bool operator()(const ratio_t & a, const ratio_t & b
            )
        {
            return a.slew > b.slew;
        }
    };

public:
    static void run_sorted_by_wire_size(Timing_Analysis::
        Timing_Analysis & ta);
```

```
47      static void run_sorted_by_slew(Timing_Analysis::
            Timing_Analysis & ta);
48      static void run_average_calculation(Timing_Analysis::
            Timing_Analysis & ta);
49
50  };
51
52  #endif // CEFF_RATIO_EXPERIMENT_H
```

Listing B.1 – ceff_ratio_experiment.h

```
1   #ifndef CIRCUITNETLIST_H_
2   #define CIRCUITNETLIST_H_
3
4   #include <string>
5   using std::string;
6
7   #include <ostream>
8   using std::ostream;
9
10  #include <vector>
11  using std::vector;
12
13  #include <map>
14  using std::map;
15
16  #include <utility>
17  using std::pair;
18
19  #include <iostream>
20  using std::cout;
21  using std::endl;
22
23  #include <queue>
24  using std::queue;
25
26  #include <cassert>
27
28  class Circuit_Netlist
29  {
30  public:
31
32      struct Logic_Gate {
33      string name;
34      string cellType;
35      vector<int> inNets;
36      int fanoutNetIndex;
37      bool inputDriver;
38      bool sequential;
39      bool primary_output;
40
```

```
41          Logic_Gate(const string name, const string cellType,
                 const unsigned inputs, int fanoutNetIndex,
                const bool inputDriver = false, const bool
                primary_output = false) :
42        name(name), cellType(cellType), inNets(inputs),
              fanoutNetIndex(fanoutNetIndex), inputDriver(
              inputDriver), primary_output(primary_output)
43        {};
44    };
45
46    struct Sink {
47      int gate;
48      string pinName;
49      Sink(const int gate, const string pinName) : gate(gate),
             pinName(pinName){};
50    };
51
52    struct Net {
53      string name;
54      int sourceNode;
55      string sourcePin;
56      vector<Sink> sinks;
57      bool dummyNet;
58
59      Net(const string name, const int sourceNode, const
            string sourcePin, const bool dummyNet = false) :
60        name(name), sourceNode(sourceNode), sourcePin(
              sourcePin), dummyNet(dummyNet)
61        {};
62
63      void addSink(const Sink sinkNode) { sinks.push_back(
            sinkNode); }
64    };
65
66  private:
67      friend ostream & operator<<( ostream & out, const
            Circuit_Netlist netlist)
68    {
69      out << "-- NET TOPOLOGY ("<< netlist.nets.size()<< ")"
            << endl;
70      for(size_t i = 0; i < netlist.nets.size(); i++)
71        out << "----- netT[" << i  << " = "<< netlist.
              netTopology[i] <<"] "<< netlist.nets[netlist.
              netTopology[i]] << endl;
72      out << "-- GATE TOPOLOGY" << endl;
73      for(size_t i = 0; i < netlist.gates.size(); i++)
74        out << "----- gateT["<<i<<" = "<< netlist.topology[i]
              <<"] " << netlist.gates[netlist.topology[i]] <<
              endl;
75      return out;
76    }
77      friend ostream & operator<<( ostream & out, const
```

```
            Circuit_Netlist::Net net)
78    {
79      out << net.name << (net.dummyNet? " dummyNet":"") << "
            from (" << net.sourceNode << ", " << net.sourcePin
            << ") : ";
80            for(size_t i = 0; i < net.sinks.size(); i++)
81        out << net.sinks[i] << " ";
82      return out;
83    }
84      friend ostream & operator<<( ostream & out, const
            Circuit_Netlist::Sink sink)
85    {
86      return out << "("<<sink.gate<<", "<<sink.pinName<<")";
87    }
88      friend ostream & operator<<( ostream & out, const
            Circuit_Netlist::Logic_Gate gate)
89    {
90      out << gate.cellType << " " << gate.name << (gate.
            inputDriver?" inputDriver":"") << endl;
91            for(size_t i = 0; i < gate.inNets.size(); i++)
92        out << "------ net["<<i<<"] = " << gate.inNets[i] <<
              endl;
93      return out;
94    }
95    map<string, int> gateNameToGateIndex;
96    map<string, int> netNameToNetIndex;
97      vector<Logic_Gate> gates;
98    vector<Net> nets;
99
100   vector<int> topology;
101   vector<int> netTopology;
102
103   vector<int> inverseTopology;
104   vector<int> inverseNetTopology;
105
106
107
108
109   int _numberOfGates;
110     int _timing_arcs;
111     int _timing_points;
112
113     int addGate(const string name, const string cellType,
            const int inputs, const bool isInputDriver = false,
            const bool primary_output = false);
114     int addNet(const string name, const int sourceNode,
            const string sourcePin);
115     int addNet(const string name);
116 public:
117     Circuit_Netlist():_numberOfGates(0), _timing_arcs(0),
            _timing_points(0){}
118     virtual ~Circuit_Netlist(){}
```

```
119
120     void addCellInst(const string name, const string cellType,
            vector<pair<string, string>> inputPinPairs, const
            bool isSequential = false, const bool isInputDriver =
            false, const bool primary_output = false);
121     void updateTopology();
122
123       size_t getNetsSize() const { return nets.size(); }
124       Net & getNet(const size_t & i) { return nets[i]; }
125       const Logic_Gate & getGateT(const size_t & i) const {
              return gates.at(topology.at(i)); }
126       int getTopologicIndex(const int & i) const { return (i
              ==-1?-1:inverseTopology.at(i)); }
127
128       size_t getGatesSize() const { return gates.size(); }
129       Logic_Gate & getGate(const size_t & i) { return gates[i
              ]; }
130       const Net & getNetT(const size_t & i) const { return
              nets.at(netTopology.at(i)); }
131       int get_net_topologic_index(const size_t & i) const {
              return inverseNetTopology.at(i); }
132
133     const vector<pair<int, string>> verilog() const;
134
135       int timing_arcs() const;
136       int timing_points() const;
137
138
139
140 };
141
142 #endif
```

Listing B.2 – circuit_netlist.h

```
1  #ifndef CONFIGURATION_H_
2  #define CONFIGURATION_H_
3
4  #include "spef_net.h"
5  #include "parser.h"
6
7  #include <string>
8  using std::string;
9
10 /*
11
12   This is a Configuration File
13   To switch between ISPD2012 (Lumped Capacitance Wire Delay
         Model) or ISPD2013 (Distributed RC Wire Delay Model)
         SPEF format
14
```

```
15 */
16
17 // STATIC METAPROGRAMMED IF
18 template<bool cond, class ThenType, class ElseType>
19 struct IF
20 {
21   typedef ThenType RET;
22 };
23 template<class ThenType, class ElseType>
24 struct IF<false, ThenType, ElseType>
25 {
26   typedef ElseType RET;
27 };
28
29
30 // User configurable {
31
32
33 class Traits
34 {
35 public:
36     static const bool ISPD_2012 = false;
37     static const double STD_THRESHOLD = 0.01;
38     static string ispd_contest_root;
39     static string ispd_contest_benchmark;
40     static string arrival_time_file_name;
41 };
42
43
44 // }
45
46
47
48
49
50
51
52
53
54
55
56
57
58 ////////// DON'T TOUCH!!!!
59 typedef IF<Traits::ISPD_2012, SpefParserISPD2012,
       SpefParserISPD2013>::RET SpefParser;
60 typedef IF<Traits::ISPD_2012, SpefNetISPD2012,
       SpefNetISPD2013>::RET SpefNet;
61 typedef IF<Traits::ISPD_2012, Parasitics2012, Parasitics2013
       >::RET Parasitics;
62
63
```

```
64 #endif
```

Listing B.3 – configuration.h

```
1  #ifndef DESIGNCONSTRAINTS_H_
2  #define DESIGNCONSTRAINTS_H_
3
4  #include <string>
5  using std::string;
6
7  #include "transitions.h"
8
9  #include <map>
10 using std::map;
11
12 #include <utility>
13 using std::make_pair;
14
15 #include <iostream>
16 using std::endl;
17 using std::cout;
18
19 class Design_Constraints
20 {
21     double _clock;
22     map<string, string> _driving_cells;
23     map<string, Transitions<double> > _input_transitions;
24     map<string, Transitions<double> > _input_delays;
25     map<string, Transitions<double> > _output_delays;
26     map<string, double> _output_loads;
27 public:
28
29     void clock(const double clock);
30     bool input_delay(const string input_name, const
           Transitions<double> delay);
31     bool output_delay(const string output_name, const
           Transitions<double> delay);
32     bool output_load(const string output_name, const double
           output_load);
33     bool driving_cell(const string input_name, const string
           driving_cell);
34     bool input_transition(const string input_name, const
           Transitions<double> transition);
35
36     double clock() const;
37     const Transitions<double> input_delay(const string
           input_name) const;
38     const Transitions<double> output_delay(const string
           output_name) const;
39     double output_load(const string output_name) const;
40     size_t output_loads_size() const;
```

```
41      const string driving_cell(const string input_name) const
            ;
42      const Transitions<double> input_transition(const string
            input_name) const;
43    /* data */
44  };
45
46  #endif
```

Listing B.4 – design_constraints.h

```
1   #ifndef EDGE_H
2   #define EDGE_H
3
4   #include <vector>
5   using std::vector;
6
7   #include <cassert>
8
9   #include <cstdlib>
10
11  namespace Timing_Analysis {
12
13      template<typename T>
14      class Edge {
15
16      protected:
17          Edge(T * from) : _from(from)
18          {
19
20          }
21
22          T * _from;
23          vector<T *> _to;
24
25          void set_fanout(const int i, T *tp)
26          {
27              if(_to.empty() && !i)
28                  _to.push_back(tp);
29              else
30                  _to[i] = tp;
31              assert(_to.size() == 1);
32          }
33
34          void add_fanout(T *tp)
35          {
36              _to.push_back(tp);
37          }
38
39
40      public:
```

```
41          T * from() const
42          {
43              return _from;
44          }
45
46          size_t fanouts_size() const
47          {
48              return _to.size();
49          }
50
51      };
52
53  }
54
55  #endif // EDGE_H
```

Listing B.5 – edge.h

```
1   #ifndef LIBERTYLIBRARY_H_
2   #define LIBERTYLIBRARY_H_
3
4   #include <utility>
5   using std::pair;
6   using std::make_pair;
7
8   #include <vector>
9   using std::vector;
10
11  #include <string>
12  using std::string;
13
14  #include <map>
15  using std::map;
16
17  #include <limits>
18  using std::numeric_limits;
19
20  #include <ostream>
21  using std::ostream;
22
23  #include <cassert>
24
25  #include "transitions.h"
26
27  // Look up table to store delay or slew functions
28  struct LibertyLookupTable {
29
30    // Look up table is indexed by the output load and the
        input transition values
31    // Example:
32    //   Let L = loadIndices[i]
```

```
33    //        T = transitionIndices[j]
34    //    Then, the table value corresponding to L and T will
          be:
35    //        table[i][j]
36    //
37    vector<double> loadIndices ;
38    vector<double> transitionIndices ;
39    vector<vector<double> > tableVals ;
40
41  };
42
43  ostream& operator<< (ostream& os, LibertyLookupTable& lut) ;
44
45  struct LibertyTimingInfo {
46
47    string fromPin ;
48    string toPin ;
49    string timingSense ; // "non_unate" or "negative_unate" or
          "positive_unate".
50    // Note that ISPD-13 library will have only negative-unate
          combinational cells. The clock arcs
51    // for sequentials will be non_unate (which can be ignored
          because of the simplified sequential
52    // timing model for ISPD-13).
53
54
55    LibertyLookupTable fallDelay ;
56    LibertyLookupTable riseDelay ;
57    LibertyLookupTable fallTransition ;
58    LibertyLookupTable riseTransition ;
59
60  } ;
61
62  ostream& operator<< (ostream& os, LibertyTimingInfo& timing)
        ;
63
64  struct LibertyPinInfo {
65
66    string name ; // pin name
67    double capacitance ; // input pin cap (not defined for
          output pins)
68    double maxCapacitance ; // the max load this pin can drive
69    bool isInput ; // whether the pin is input or output pin
70    bool isClock ; // whether the pin is a clock pin or not
71
72    LibertyPinInfo () : capacitance (0.0)
73    , maxCapacitance (std::numeric_limits<double>::max())
74    , isInput(true)
75    , isClock(false) {};
76
77  };
78
```

```
79
80 ostream& operator<< (ostream& os, LibertyPinInfo& pin) ;
81
82 struct LibertyCellInfo {
83
84   string name ; // cell name
85   string footprint ; // only the cells with the same
         footprint are swappable
86   double leakagePower ; // cell leakage power
87   double area ; // cell area (will not be a metric for ISPD
         -13)
88   bool isSequential ; // if true then sequential cell, else
         combinational
89   bool dontTouch ; // is the sizer allowed to size this cell
         ?
90   bool primaryOutput;
91
92   vector<LibertyPinInfo> pins ;
93   vector<LibertyTimingInfo> timingArcs ;
94
95   LibertyCellInfo () : leakagePower (0.0), area (0.0),
         isSequential (false), dontTouch(false), primaryOutput(
         false) {}
96
97 } ;
98
99 ostream& operator<< (ostream& os, LibertyCellInfo& cell) ;
100
101
102 class LibertyLibrary
103 {
104   double maxTransition;
105   vector< vector<LibertyCellInfo> > library;
106
107   map<string, int> footPrintToIndex;
108   map<string, int> cellOptionNumber; // ex cellOptionNumber[
         in01f01] = 0
109   map<string, int> cellToFootprintIndex;
110   // fazer um map de celltype para footprint
111
112 public:
113   LibertyLibrary(const double maxTransition = 0.0f);
114   virtual ~LibertyLibrary();
115
116
117   const pair<int, int> addCellInfo(const LibertyCellInfo &
         cellInfo); // return = [footprint index][option index]
118
119
120   const LibertyCellInfo & getCellInfo(const string &
         footPrint, const int & i) const;
121   const LibertyCellInfo & getCellInfo(const string &
```

```
      cellName) const;
122   const LibertyCellInfo & getCellInfo(const int &
      footPrintIndex, const int & optionIndex) const;
123   size_t number_of_options(const int footprint_index) const;
124
125   const pair<int, int> getCellIndex(const string &cellName)
      const;
126   double getMaxTransition() const ;
127
128
129
130   /* data */
131 };
132
133 enum Unateness {
134   NEGATIVE_UNATE, POSITIVE_UNATE, NON_UNATE
135 };
136
137 class LibertyLookupTableInterpolator
138 {
139 protected:
140     static const int DEFAULT_DECIMAL_PLACES;
141     void round(Transitions<double> & transitions, const int
          decimal_places);
142 public:
143   virtual double interpolate(const LibertyLookupTable & lut,
       const double load, const double transition) = 0;
144   virtual const Transitions<double> interpolate(const
      LibertyLookupTable & riseLut, const LibertyLookupTable
       & fallLut, const Transitions<double> load, const
      Transitions<double> transition, Unateness unateness =
      NEGATIVE_UNATE, bool is_input_driver = false) = 0;
145
146   /* data */
147 };
148
149 class LinearLibertyLookupTableInterpolator : public
      LibertyLookupTableInterpolator
150 {
151 public:
152   double interpolate(const LibertyLookupTable & lut, const
      double load, const double transition);
153   const Transitions<double> interpolate(const
      LibertyLookupTable & riseLut, const LibertyLookupTable
       & fallLut, const Transitions<double> load, const
      Transitions<double> transition, Unateness unateness =
      NEGATIVE_UNATE, bool is_input_driver = false);
154
155 };
156
157 #endif
```

Listing B.6 – liberty_library.h

```
#ifndef MULTI_FANOUT_EDGE_H
#define MULTI_FANOUT_EDGE_H
#include "edge.h"
#include "timing_point.h"

namespace Timing_Analysis
{

    template <class T>
    class Multi_Fanout_Edge : public Edge<T>
    {
    public:
        Multi_Fanout_Edge(T * from) : Edge<T>(from) {}
        T & to(const int i) const { return *Edge<T>::_to.at(
            i); }
    };

}

#endif // MULTI_FANOUT_EDGE_H
```

Listing B.7 – multi_fanout_edge.h

```
#ifndef PARSER_H_
#define PARSER_H_

#include <string>
using std::string;

#include <vector>
using std::vector;

#include <istream>
using std::istream;

#include <fstream>
using std::fstream;

#include <cassert>

#include <cstdlib>

#include "circuit_netlist.h"
#include "liberty_library.h"
#include "spef_net.h"
#include "design_constraints.h"

```

```
25 class Parser
26 {
27 protected:
28   bool isSpecialChar(const char & c);
29   bool readLineAsTokens(istream& is, vector<string>& tokens,
         bool includeSpecialChars = false);
30   fstream is;
31 public:
32   Parser();
33   virtual ~Parser();
34 };
35
36
37 class Prime_Time_Output_Parser : public Parser
38 {
39 public:
40   struct Pin_Timing {
41     string pin_name;
42     Transitions<double> slack;
43     Transitions<double> slew;
44     Transitions<double> arrival_time;
45     friend ostream & operator <<( ostream & out, const
         Pin_Timing & pin)
46     {
47       return out << pin.pin_name << " " << pin.slack << " "
           <<  pin.slew << " " << pin.arrival_time;
48     }
49   };
50   struct Port_Timing {
51     string port_name;
52     Transitions<double> slack;
53     Transitions<double> slew;
54         Transitions<double> arrival_window;
55
56     friend ostream & operator <<( ostream & out, const
         Port_Timing & port)
57     {
58       return out << port.port_name << " " << port.slack << "
           " << port.slew;
59     }
60   };
61   class Prime_Time_Output{
62     friend class Prime_Time_Output_Parser;
63     vector<Pin_Timing> _pins;
64     vector<Port_Timing> _ports;
65   public:
66         size_t pins_size() const { return _pins.size(); }
67         size_t ports_size() const { return _ports.size(); }
68     const Pin_Timing & pin(const size_t i) const { return
         _pins.at(i); }
69     const Port_Timing & port(const size_t i) const { return
         _ports.at(i); }
```

```
70    };
71      Prime_Time_Output_Parser(){}
72    const Prime_Time_Output parse_prime_time_output_file(const
          string filename);
73  };
74
75
76  class VerilogParser : public Parser
77  {
78    static const string SEQUENTIAL_CELL;
79    static const string INPUT_DRIVER_CELL;
80    static const string PRIMARY_OUTPUT_CELL;
81    static const string CLOCK_NET;
82
83    // Read the module definition
84    bool read_module(string& moduleName);
85
86    // Read the next primary input.
87    // Return value indicates if the last read was successful
          or not.
88    bool read_primary_input(string& primaryInput);
89
90    // Read the next primary output.
91    // Return value indicates if the last read was successful
          or not.
92    bool read_primary_output(string& primaryInput);
93
94    // Read the next net.
95    // Return value indicates if the last read was successful
          or not.
96    bool read_wire(string& wire);
97
98    // Read the next cell instance.
99    // Return value indicates if the last read was successful
          or not.
100   bool read_cell_inst(string& cellType, string& cellInstName
          , vector<std::pair<string, string> >& pinNetPairs);
101   bool read_assign(pair<string, string> & assignment);
102 public:
103   const Circuit_Netlist readFile(const string filename);
104
105
106   virtual ~VerilogParser()
107   {
108
109   };
110 };
111
112 // See test_lib_parser () function in parser_helper.cpp for
        an
113 // example of how to use this class.
114 class LibertyParser : public Parser
```

```
115 {
116
117   void _skip_lut_3D () ;
118   void _begin_read_lut (LibertyLookupTable& lut) ;
119   void _begin_read_timing_info (string pinName,
          LibertyTimingInfo& cell) ;
120   void _begin_read_pin_info (string pinName, LibertyCellInfo
          & cell, LibertyPinInfo& pin) ;
121   void _begin_read_cell_info (string cellName,
          LibertyCellInfo& cell) ;
122     // Read the default max_transition defined for the
            library.
123   // Return value indicates if the last read was successful
          or not.
124   // This function must be called in the beginning before
          any read_cell_info function call.
125   bool read_default_max_transition (double& maxTransition) ;
126
127
128   // Read the next standard cell definition.
129   // Return value indicates if the last read was successful
          or not.
130   bool read_cell_info (LibertyCellInfo& cell) ;
131 public:
132
133   const LibertyLibrary readFile(const string filename);
134
135 };
136
137
138 class SpefParserISPD2013 : public Parser
139 {
140   bool read_connections(SpefNetISPD2013 & net);
141   void read_capacitances(SpefNetISPD2013 & net);
142   void read_resistances(SpefNetISPD2013 & net);
143   bool read_net_data(SpefNetISPD2013& spefNet);
144 public:
145   const Parasitics2013 readFile(const string filename);
146
147   /* data */
148 };
149
150 class SpefParserISPD2012 : public Parser
151 {
152   bool read_net_cap(string & net, double & cap);
153 public:
154   const Parasitics2012 readFile(const string filename);
155
156   /* data */
157 };
158
159 class SDCParser : public Parser
```

```
160 {
161   // The following functions must be issued in a particular
          order
162   // See test_sdc_parser function for an example
163
164   // Read clock definition
165   // Return value indicates if the last read was successful
          or not.
166   bool read_clock(string& clockName, string& clockPort,
          double& period);
167
168   // Read input delay
169   // Return value indicates if the last read was successful
          or not.
170   bool read_input_delay(string& portName, double& delay);
171
172   // Read driver info for the input port
173   // Return value indicates if the last read was successful
          or not.
174   bool read_driver_info(string& inPortName, string&
          driverSize, string& driverPin, double&
          inputTransitionFall, double& inputTransitionRise);
175
176   // Read output delay
177   // Return value indicates if the last read was successful
          or not.
178   bool read_output_delay(string& portName, double& delay);
179
180   // Read output load
181   // Return value indicates if the last read was successful
          or not.
182   bool read_output_load(string& outPortName, double& load);
183 public:
184   const Design_Constraints readFile(const string filename);
185 };
186
187 #endif
```

Listing B.8 – parser.h

```
1  #ifndef SINGLE_FANOUT_EDGE_H
2  #define SINGLE_FANOUT_EDGE_H
3
4  #include "edge.h"
5
6  namespace Timing_Analysis {
7
8      template<class T>
9      class Single_Fanout_Edge : public Edge<T>
10     {
11     public:
```

```
12          Single_Fanout_Edge(T * from, T * to) : Edge<T>(from)
                {
13              Edge<T>::set_fanout(0, to);
14          }
15          T & to(void) const { return *Edge<T>::_to.at(0);  }
16
17      };
18
19  }
20
21  #endif // SINGLE_FANOUT_EDGE_H
```

Listing B.9 – single_fanout_edge.h

```
1   #ifndef SLEW_DEGRADATION_EXPERIMENT_H
2   #define SLEW_DEGRADATION_EXPERIMENT_H
3
4   #include "timing_analysis.h"
5
6   #include <utility>
7   using std::pair;
8   using std::make_pair;
9
10  #include <queue>
11  using std::queue;
12
13  class Slew_Degradation_Experiment
14  {
15      static bool nearly_equals(const Transitions<double> a,
            const Transitions<double> b);
16      static const double EPSILON;
17  public:
18      static void run(Timing_Analysis::Timing_Analysis &ta);
19  };
20
21  #endif // SLEW_DEGRADATION_EXPERIMENT_H
```

Listing B.10 – slew_degradation_experiment.h

```
1   /*
2    * SpefNet.h
3    *
4    *  Created on: Jun 4, 2013
5    *      Author: chrystian
6    */
7
8   #ifndef SPEFNET_H_
9   #define SPEFNET_H_
10
11  #include <string>
```

```
12 using std::string;
13
14 #include <vector>
15 using std::vector;
16
17 #include <map>
18 using std::map;
19
20 #include <ostream>
21 using std::ostream;
22
23 #include <iostream>
24 using std::endl;
25
26 #include <queue>
27 using std::queue;
28
29 #include "transitions.h"
30
31 #include <cassert>
32
33 class SpefNetISPD2012
34 {
35 public:
36   string netName;
37   double netLumpedCap;
38     double total_resistance;
39
40     SpefNetISPD2012():netName("DEFAULT_NET_NAME"),
         netLumpedCap(0), total_resistance(0){}
41 };
42
43 class SpefNetISPD2013
44 {
45 public:
46   struct Resistor
47   {
48     int node1;
49     int node2;
50     double value;
51         Resistor(const int & node1, const int & node2, const
              double & value) : node1(node1), node2(node2),
             value(value){}
52         int getOtherNode(const int & node) const { return (
             node == node1 ? node2 : node1); }
53
54   };
55   struct Capacitor
56   {
57     int node;
58     double value;
59         Capacitor(const int & node, const double & value) :
```

```
                node(node), value(value) {}
60    };
61    struct Node
62    {
63      int nodeIndex;
64      string name;
65      vector<int> resistors;
66      double capacitance;
67          Node(const int & index, const string & name) :
                nodeIndex(index), name(name), capacitance(1e-6)
                { }
68    };
69  private:
70    vector<Node> nodes;
71    vector<Resistor> resistors;
72    vector<Capacitor> capacitors;
73    map<string, int> nodeMap;
74      int addNode(const string & name);
75
76  public:
77      SpefNetISPD2013(){}
78      virtual ~SpefNetISPD2013(){}
79    void addResistor(const string & node1, const string &
          node2, const double & value);
80    void addCapacitor(const string & node, const double &
          value);
81
82    friend ostream& operator<<(ostream & out, const
          SpefNetISPD2013 & descriptor);
83      size_t nodesSize() const {return nodes.size();}
84      const Node & getNode(const unsigned & i) const { return
            nodes.at(i);}
85      int getNodeIndex(const string & name) const;
86      size_t resistorsSize() const { return resistors.size();
            }
87      const Resistor & getResistor(const unsigned & i) const {
             return resistors.at(i); }
88
89      void set(string name, double lumpedCapacitance, double
            total_resistance);
90    string netName;
91    double netLumpedCap;
92      double total_resistance;
93
94
95
96
97  };
98
99  typedef map<string, SpefNetISPD2012> Parasitics2012;
100 typedef map<string, SpefNetISPD2013> Parasitics2013;
101
```

```
102 #endif /* SPEFNET_H_ */
```

Listing B.11 – spef_net.h

```
1  #ifndef TIMER_H
2  #define TIMER_H
3
4  #include <sys/time.h>
5  #include <cstdlib>
6
7  #include <string>
8  using std::string;
9  #include <cassert>
10 #include <ostream>
11 using std::ostream;
12
13 class Timer
14 {
15     struct timeval _start_time;
16     struct timeval _stop_time;
17
18
19 public:
20
21     class Result
22     {
23         friend class Timer;
24         float _time;
25         string _unity;
26
27
28          public:
29         void set(float time, string unity){
30             _time = time;
31             _unity = unity;
32         }
33         friend ostream & operator << (ostream & out, const
               Timer::Result & result)
34         {
35             return out << result._time << " " << result.
                   _unity;
36         }
37
38         float time() const { return _time; }
39     };
40
41 private:
42     Timer::Result _execution_time;
43 public:
44     Timer();
45     virtual ~Timer();
```

```
46
47
48      void start();
49      void end();
50
51      const Timer::Result &value(const double time_definition)
            ;
52
53      static const double MICRO;
54      static const string micro;
55      static const double MILI;
56      static const string mili;
57      static const double SECOND;
58      static const string second;
59  };
60
61  #endif // TIMER_H
```

Listing B.12 – timer.h

```
1   //
        ////////////////////////////////////////////////////////////////
2   //
3   //
4   //  Timing Analysis Interface helper class to interface with
5   //  the timer.
6   //
7   //  This code is provided for description purposes only. The
        contest
8   //  organizers cannot guarantee that the provided code is
        free of
9   //  bugs or defects. !!!! USE THIS CODE AT YOUR OWN RISK
        !!!!!
10  //
11  //
12  //  The contestants are free to use these functions as-is or
        make
13  //  modifications. If the contestants choose to use the
        provided
14  //  code, they are responsible for making sure that it works
        as
15  //  expected.
16  //
17  //  The code provided here has no real or implied warranties
        .
18  //
19  //
20  //
        ////////////////////////////////////////////////////////////////
```

```
21
22 #ifndef _TIMERINTERFACE_H_
23 #define _TIMERINTERFACE_H_
24 #include <vector>
25 #include <string>
26 #include <fstream>
27 #include <iostream>
28 #include <sstream>
29 #include <dirent.h>
30 #include <cassert>
31 #include <cstdlib>
32
33 using std::cout;
34 using std::endl;
35
36 class TimerInterface {
37   // This class contains functions for the timing analysis
         interface.
38   // To use any function belonging to this class, call
         TimerInterface::<function_name>(<argument_list>);
39
40   // Declarations
41   // LOOK AT THIS PUBLIC SECTION - FUNCTION IMPLEMENTATIONS
         BELOW IN THIS FILE
42  public:
43   // Status (outside of this class, you must use
         TimerInterface::Status to define variables of this
         type)
44   enum Status { TIMER_NOT_STARTED = 0,    // Timing analysis
         has not been started
45                 TIMER_BUSY,              // Timer is busy (
                     is reading design or performing timing
                     analysis)
46                 TIMER_FINISHED_SUCCESS,  // Timing analysis
                     finished successfully
47                 TIMER_FINISHED_ERROR,    // Error occured
                     during timing analysis
48                 TIMER_INTERFACEERROR     // Error indicating
                      that the program could not get timer
                     status (could not read status file)
49   };
50
51   // Get timer status
52   // Inputs: contest root directory (string)
53   //         benchmark name (string)
54   // Return: status (see enum Status above)
55   static Status getTimerStatus(const std::string &
         contest_root, const std::string &benchmark);
56
57   // Write sizes and run timing analysis in blocking mode
58   // 1. Write sizes
59   // 2. Starts timing analysis
```

```
60    // 3. Waits for timing analysis to be completed
61    // Inputs: vector of pairs where first value is instance
          name (string) and second value is cell name (string)
62    //         contest root directory (string)
63    //         benchmark name (string)
64    //         polling time (number of seconds that the
          function should wait before polling timer status to
          check whether timer is done)
65    // Return: timer status
66    static Status runTimingAnalysisBlocking(const std::vector<
          std::pair<std::string, std::string> > &sizes, const
          std::string &contest_root, const std::string &
          benchmark, const unsigned pollingTime);
67
68    // Start timing analysis in non-blocking mode
69    // 1. Write sizes
70    // 2. Starts timing analysis and returns (does not wait
          for timing analysis to be completed)
71    // Inputs: vector of pairs where first value is instance
          name (string) and second value is cell name (string)
72    //         contest root directory (string)
73    //         benchmark name (string)
74    // Return: timer status
75    static Status startTimingAnalysisNonBlocking(const std::
          vector<std::pair<std::string, std::string> > &sizes,
          const std::string &contest_root, const std::string &
          benchmark);
76
77    // Wait for given number of seconds (useful function if
          you want to wait before checking timer status after
          calling startTimingAnalysisNonBlocking)
78    // Input: seconds to wait
79    static void wait(int seconds);
80
81
82
83    // PRIVATE SECTION
          ------------------------------------------------------------------------
84    // DO NOT LOOK AT THIS PRIVATE SECTION, YOU SHOULD ONLY
          LOOK AT FUNCTIONS DEFINED IN PUBLIC SECTION
85  private:
86    // Get timer status (helper function for isTimerDone)
87    // Input:  vector of file names (returned by getFiles)
88    // Return: string indicating timer status
89    static std::string getTimerStatusString(const std::vector<
          std::string> &files);
90
91    // Checks if a file exists (returns true if it does, false
          otherwise)
92    // Input:  filename including path (string)
93    // Return: true if the file exists and is readable, false
```

```
        otherwise
94      static bool doesFileExist(const std::string &file);
95
96      // Get a list of files from given directory (used by
            getTimerStatus to check if timer is done)
97      // Input: directory name (string)
98      // Output: vector of file names (strings), argument passed
             by reference
99      // Return: true if directory could be read, false
            otherwise
100     static bool getFiles(std::vector<std::string> &files,
            const std::string &dir);
101
102     // Remove a file from the given directory (helper function
             used by startTimingAnalysis)
103     // Inputs: name of the file without directory name (string
            )
104     //         directory name (string)
105     // Return: true if file was removed successfully, false
            otherwise
106     static bool removeFile(const std::string &dir, const std::
            string &file);
107
108     // Write sizes to a file for timing analysis call
109     // Inputs: vector of pairs where first value is instance
            name (string) and second value is cell name (string)
110     //         contest root directory (string)
111     //         benchmark name (string)
112     // Return: true if sizes were written successfully to .int
            .sizes file, false otherwise
113     static bool writeSizesForTimer(const std::vector<std::pair
            <std::string, std::string> > &sizes, const std::string
             &contest_root, const std::string &benchmark);
114
115     // Start timing analysis (does not wait for it to finish)
116     // Input:  contest root directory (string)
117     //         benchmark name (string)
118     // Return: true if successfully wrote command to start
            timing analysis, false otherwse
119     static bool startTimingAnalysis(const std::string &
            contest_root, const std::string &benchmark);
120
121     // Run timing analysis in blocking mode
122     // 1. Starts timing analysis
123     // 2. Waits for timing analysis to be completed
124     // Input:  contest root directory (string)
125     //         benchmark name (string)
126     //         polling time (number of seconds that the
            function should wait before polling timer status to
            check whether timer is done)
127     // Return: timer status
128     static Status runTimingAnalysisBlocking(const std::string
```

```
        &contest_root, const std::string &benchmark, const
        unsigned pollingTime);
129
130     // Start timing analysis in non-blocking mode
131     // 1. Starts timing analysis and returns (does not wait
        for timing analysis to be completed)
132     // Input:  contest root directory (string)
133     //         benchmark name (string)
134     // Return: timer status
135     static Status startTimingAnalysisNonBlocking(const std::
        string &contest_root, const std::string &benchmark);
136
137     // END PRIVATE SECTION
        ------------------------------------------------------------------------
138 }; // END class TimerInterface
139
140
141
142 #endif // _TIMERINTERFACE_H_
```

Listing B.13 – timer_interface.h

```
1  #ifndef TIMING_ANALYSIS_H_
2  #define TIMING_ANALYSIS_H_
3
4  #include <vector>
5  using std::vector;
6
7  #include <string>
8  using std::string;
9
10 #include <ostream>
11 using std::ostream;
12
13 #include <map>
14 using std::map;
15
16 #include <utility>
17 using std::pair;
18
19 #include <fstream>
20 using std::fstream;
21
22 #include <stack>
23 using std::stack;
24
25 #include <set>
26 using std::set;
27
28 #include <queue>
```

```
29 using std::priority_queue;
30
31 #include <cstdlib>
32
33 #include "timer_interface.h"
34
35 #include "transitions.h"
36 #include "circuit_netlist.h"
37 #include "wire_delay_model.h"
38 #include "liberty_library.h"
39 #include "design_constraints.h"
40
41 #include "configuration.h"
42
43 #include "parser.h"
44
45
46 #include "timing_net.h"
47 #include "timing_point.h"
48 #include "timing_net.h"
49
50 namespace Timing_Analysis
51 {
52
53     class ita_comparator {
54
55     public:
56         bool operator()(Timing_Point * a, Timing_Point * b);
57     };
58
59     class Option
60   {
61         friend class Timing_Analysis;
62         int _footprint_index;
63         int _option_index;
64         bool _dont_touch;
65
66     public:
67         Option():_footprint_index(-1), _option_index(-1),
                _dont_touch(false){}
68         Option(const int footprintIndex, const int
                optionIndex) : _footprint_index(footprintIndex),
                 _option_index(optionIndex), _dont_touch(false)
                {}
69         int footprint_index() const;
70         int option_index() const;
71         bool is_dont_touch() const;
72   };
73
74
75     class Timing_Net;
76     class Timing_Arc;
```

```
77      class Timing_Analysis
78    {
79
80
81          vector<Timing_Point> _points;
82          vector<Timing_Arc> _arcs;
83          vector<Timing_Net> _nets;
84          vector<Option> _options;
85          map<int, double> _PO_loads;
86          map<string, int> _pin_name_to_timing_point_index;
87          vector<pair<size_t, size_t> >
                _gate_index_to_timing_point_index;
88
89          vector<pair<int, string> > _verilog;
90          vector<pair<string, string> > _sizes;
91          vector<bool> _dirty;
92
93          const LibertyLibrary * _library;
94          const Parasitics * _parasitics;
95          LibertyLookupTableInterpolator * _interpolator;
96
97          Transitions<double> _target_delay;
98          Transitions<double> _max_transition;
99          Transitions<double> _critical_path;
100         Transitions<double> _total_negative_slack;
101         Transitions<double> _worst_slack;
102         Transitions<double> _slew_violations;
103         Transitions<double> _capacitance_violations;
104
105         unsigned _total_violating_POs;
106         int _first_PO_index;
107
108
109         void initialize_timing_data();
110
111
112         // PRIVATE GETTERS
113         const Transitions<double> calculate_timing_arc_delay
                (const Timing_Arc & timing_arc, const
                Transitions<double> transition, const
                Transitions<double> ceff);
114
115         // STATIC TIMING ANALYSIS
116         void update_timing(const int timing_point_index);
117         void update_slacks(const int timing_point_index);
118
119     // TOPOLOGY INIT
120         const pair<size_t, size_t> create_timing_points(
                const int i,const Circuit_Netlist::Logic_Gate &
                gate,const pair<int, int> cellIndex, const
                LibertyCellInfo & cellInfo);
121         void number_of_timing_points_and_timing_arcs(int &
```

```
            numberOfTimingPoints, int & numberOfTimingArcs,
            const Circuit_Netlist & netlist, const
            LibertyLibrary * lib);
122         void create_timing_arcs(const pair<size_t, size_t>
            tpIndexes, const bool is_pi , const bool is_po )
            ;
123
124     // PRIMETIME CALLING
125         void get_sizes_vector();
126
127     // OUTPUT METHODS
128         void write_sizes_file(const string filename);
129
130     public:
131         Timing_Analysis(const Circuit_Netlist & netlist,
            const LibertyLibrary * lib, const Parasitics *
            _parasitics, const Design_Constraints * sdc);
132         virtual ~Timing_Analysis();
133
134
135         //
136
137         void call_prime_time();
138         void full_timing_analysis();
139         void incremental_timing_analysis(int gate_number,
            int new_option);
140         void update_timing_points(const Timing_Point *
            output_timing_point);
141
142     // GETTERS
143         size_t number_of_gates() const { return _options.
            size(); }
144
145         size_t timing_points_size() { return _points.size();
             }
146         const Timing_Point & timing_point( const int i ) {
            return _points.at(i); }
147
148         size_t timing_arcs_size() { return _arcs.size(); }
149         const Timing_Arc & timing_arc( const int i ) {
            return _arcs.at(i); }
150
151         size_t timing_nets_size() { return _nets.size(); }
152         const Timing_Net & timing_net( const int i ) {
            return _nets.at(i); }
153
154         double pin_capacitance(const int timing_point_index)
             const;
155         double pin_load(const int timing_point_index) const;
156         const Option & option(const int gate_number);
157         const LibertyCellInfo & liberty_cell_info(const int
            gate_index) const;
```

```
158
159         size_t number_of_options(const int gate_index);
160
161         Transitions<double> total_negative_slack() const {
                return _total_negative_slack; }
162         Transitions<double> worst_slack() const { return
                _worst_slack; }
163         Transitions<double> target_delay() const { return
                _target_delay; }
164         unsigned total_violating_POs() const { return
                _total_violating_POs; }
165         Transitions<double> critical_path() const { return
                _critical_path; }
166         Transitions<double> capacitance_violations() const {
                 return _capacitance_violations; }
167
168         set<int> timing_points_in_longest_path();
169         set<int> timing_points_in_critical_path();
170
171         bool has_timing_violations();
172
173
174
175         int first_PO_index() const { return _first_PO_index;
                }
176
177     // SETTERS
178         bool option(const int gate_index, const int option);
179
180         void set_all_gates_to_max_size();
181         void set_all_gates_to_min_size();
182
183         // DEBUG
184         bool validate_with_prime_time();
185         void print_info();
186         void print_circuit_info();
187         void report_timing();
188         void print_effective_capacitances();
189
190         void write_timing_file(const string filename);
191         bool check_timing_file(const string timing_file);
192
193         pair<pair<int, int>, pair<Transitions<double>,
                Transitions<double> > > check_ceffs(double
                precision);
194
195
196     };
197
198 };
199
200 #endif
```

Listing B.14 – timing_analysis.h

```
1  #ifndef TIMING_ARC_H
2  #define TIMING_ARC_H
3
4  #include "single_fanout_edge.h"
5  #include "transitions.h"
6  #include "timing_point.h"
7
8  namespace Timing_Analysis
9  {
10
11     class Timing_Point;
12     class Timing_Arc : public Single_Fanout_Edge<
           Timing_Point>
13     {
14 //        friend class Timing_Analysis;
15         Transitions<double> _delay;
16         Transitions<double> _slew;
17         int _arc_number;
18         int _gate_number;
19
20     public:
21         Timing_Arc(Timing_Point * from, Timing_Point * to,
               const int arcNumber, const int gate_number) :
               Single_Fanout_Edge<Timing_Point>(from, to),
               _delay(0.0f, 0.0f), _slew(0.0f, 0.0f),
               _arc_number(arcNumber), _gate_number(gate_number
               ) {}
22         virtual ~Timing_Arc(){}
23
24
25         void clear();
26
27
28         // GETTERS
29         Transitions<double> delay() const { return _delay; }
30         Transitions<double> slew() const { return _slew; }
31
32         void delay(const Transitions<double> & delay) {
               _delay = delay; }
33         void slew(const Transitions<double> & slew) { _slew
               = slew; }
34
35         int arc_number() const { return _arc_number; }
36         int gate_number() const { return _gate_number; }
37
38
39
40
```

```
41          friend ostream & operator<<(ostream & out, const
                Timing_Arc & ta);
42
43      };
44
45  }
46
47  #endif // TIMING_ARC_H
```

Listing B.15 – timing_arc.h

```
1   #ifndef TIMING_NET_H
2   #define TIMING_NET_H
3
4   #include "wire_delay_model.h"
5   #include "multi_fanout_edge.h"
6   #include "timing_point.h"
7
8   namespace Timing_Analysis {
9
10      class Timing_Point;
11      class Timing_Net : public Multi_Fanout_Edge<Timing_Point
            >
12      {
13          friend class Timing_Analysis;
14          friend class Timing_Point;
15          string _name;
16          WireDelayModel * _wire_delay_model;
17
18      public:
19          Timing_Net(const string & name, Timing_Point * from,
                WireDelayModel * wire_delay_model)
20          :Multi_Fanout_Edge<Timing_Point>(from), _name(name),
                _wire_delay_model(wire_delay_model)
21          {
22          }
23          virtual ~Timing_Net(){}
24
25          const string name() const;
26          friend ostream & operator<<(ostream & out, const
                Timing_Net & tn);
27
28
29
30         WireDelayModel * wire_delay_model() {return
               _wire_delay_model;}
31      };
32  }
33
34  #endif // TIMING_NET_H
```

Listing B.16 – timing_net.h

```
#ifndef TIMING_POINT_H
#define TIMING_POINT_H

#include "timing_net.h"
#include "timing_arc.h"
#include "timing_analysis.h"

namespace Timing_Analysis {

    enum Timing_Point_Type
    {
        INPUT, OUTPUT, PI_INPUT, REGISTER_INPUT, PI, PO
    };

    class Timing_Arc;
    class Timing_Net;
    class Timing_Point
    {
//        friend class Timing_Analysis;
        friend class Timing_Net;
        string _name;
        Timing_Net * _net;
        Timing_Arc * _arc;
        Transitions<double> _slack;
        Transitions<double> _slew;
        Transitions<double> _arrival_time;
        size_t _gate_number;
        Timing_Point_Type _type;

        Transitions<double> _ceff;
        int _logic_level;




    public:
        Timing_Point(string name, const size_t gate_number,
            Timing_Point_Type type);
        virtual ~Timing_Point(){}

        // GETTERS
        double load() const;
        Transitions<double> ceff() const;
        const string name() const { return _name; }
        int gate_number() const  { return _gate_number; }

        const Transitions<double> slack() const { return
            _slack; }
```

```
47          const Transitions<double> slew() const { return
                _slew; }
48          const Transitions<double> arrival_time() const {
                return _arrival_time; }
49          const Transitions<double> required_time() const {
                return _slack + _arrival_time;}
50          Timing_Net & net() const { return *_net; }
51          Timing_Arc & arc () const { return *_arc; }
52          int logic_level() const { return _logic_level; }
53
54          void ceff(const Transitions<double> & ceff) { _ceff
                = ceff; }
55          void slack(const Transitions<double> & slack ) {
                _slack = slack; }
56          void slew(const Transitions<double> & slew ) { _slew
                 = slew; }
57          void arrival_time(const Transitions<double> &
                arrival_time ) { _arrival_time = arrival_time; }
58          void net(Timing_Net * net) { _net = net; }
59          void arc(Timing_Arc * arc) { _arc = arc; }
60          void logic_level(int level) { _logic_level = level;
                }
61
62
63          // TIMING ANALYSIS
64          const Transitions<double> update_slack(const
                Transitions<double> required_time);
65          void clear_timing_info();
66
67          bool is_PO() const { return _type == PO; }
68          bool is_PI() const { return _type == PI; }
69          bool is_input_pin() const { return _type == INPUT; }
70          bool is_output_pin() const { return _type == OUTPUT;
                 }
71          bool is_PI_input() const { return _type == PI_INPUT;
                 }
72          bool is_reg_input() const { return _type ==
                REGISTER_INPUT; }
73
74          friend ostream & operator<<(ostream & out, const
                Timing_Point & tp);
75      };
76
77  }
78
79  #endif // TIMING_POINT_H
```

Listing B.17 – timing_point.h

```
1  #ifndef TRANSITIONS_H_
2  #define TRANSITIONS_H_
```

```
3
4  #include <algorithm>
5  using std::swap;
6  using std::max;
7  using std::min;
8
9  #include <ostream>
10 using std::ostream;
11
12 #include <cmath>
13
14 #include <limits>
15 using std::numeric_limits;
16
17 #define MAKE_SELF_OPERATOR( OP ) \
18 friend void operator OP ( Transitions<T> &v0, const
       Transitions<T> v1 ) { v0[RISE] OP v1[RISE], v0[FALL] OP
       v1[FALL]; } \
19 friend void operator OP ( Transitions<T> &v0, const T
                  v1 ) { v0[RISE] OP v1; v0[FALL] OP v1; }
20
21 #define MAKE_OPERATOR( OP ) \
22 friend Transitions<T> operator OP ( const Transitions<T> v0,
        const Transitions<T> v1 ) { return Transitions<T>(v0[
       RISE] OP v1[RISE], v0[FALL] OP v1[FALL]); } \
23 friend Transitions<T> operator OP ( const T             v0,
       const Transitions<T> v1 ) { return Transitions<T>(v0
             OP v1[RISE], v0       OP v1[FALL]); } \
24 friend Transitions<T> operator OP ( const Transitions<T> v0,
        const T             v1 ) { return Transitions<T>(v0[RISE
       ] OP v1       , v0[FALL] OP v1       ); }
25
26 enum EdgeType {
27   RISE = 0, FALL = 1
28 };
29
30 template<typename T>
31 class Transitions {
32
33   friend ostream &operator<<(ostream &out, const Transitions
         <T> array ) {
34     return out << "(" << array[RISE] << ", " << array[FALL]
           << ")";
35   } // end operator
36
37   MAKE_OPERATOR(+);
38   MAKE_OPERATOR(-);
39   MAKE_OPERATOR(*);
40   MAKE_OPERATOR(/);
41
42   MAKE_SELF_OPERATOR(+=);
43   MAKE_SELF_OPERATOR(-=);
```

```
44    MAKE_SELF_OPERATOR(*=);
45    MAKE_SELF_OPERATOR(/=);
46
47    friend Transitions<T> operator-( const Transitions<T> &v0
          ) { return Transitions<T>(-v0[RISE], -v0[FALL]); }
48
49    friend Transitions<T> max( const Transitions<T> v0, const
          Transitions<T> v1 ) { return Transitions<T>(max(v0[
          RISE],v1[RISE]),max(v0[FALL],v1[FALL])); }
50    friend Transitions<T> min( const Transitions<T> v0, const
          Transitions<T> v1 ) { return Transitions<T>(min(v0[
          RISE],v1[RISE]),min(v0[FALL],v1[FALL])); }
51
52    friend Transitions<T> abs( const Transitions<T> v ) {
          return Transitions<T>(fabs(v[RISE]),fabs(v[FALL])); }
53    friend Transitions<T> pow( const Transitions<T> v, const
          double exp ) { return Transitions<T>(pow(v[RISE], exp)
          ,pow(v[FALL], exp)); }
54    friend Transitions<T> sqrt( const Transitions<T> v) {
          return Transitions<T>(sqrt(v[RISE]),sqrt(v[FALL])); }
55    friend Transitions<T> exp( const Transitions<T> v) {
          return Transitions<T>(exp(v[RISE]),exp(v[FALL])); }
56
57  private:
58    T clsValue[2];
59  public:
60    T &operator[]( const EdgeType edgeType ) {return clsValue[
          edgeType];}
61    T  operator[]( const EdgeType edgeType ) const {return
          clsValue[edgeType];}
62
63      Transitions &operator=(const Transitions & array) {
64          clsValue[RISE] = array[RISE];
65          clsValue[FALL] = array[FALL];
66      return *this;
67    } // end operator
68
69    Transitions(const T rise, const T fall ) {
70      clsValue[RISE] = rise;
71      clsValue[FALL] = fall;
72    } // end constructor
73
74    Transitions(){};
75
76    void set( const T rise, const T fall ) {
77      clsValue[RISE] = rise;
78      clsValue[FALL] = fall;
79    } // end method
80
81    T getMax() const { return max(clsValue[RISE], clsValue[
          FALL]); }
82    T getMin() const { return min(clsValue[RISE], clsValue[
```

```
        FALL]); }
83    T getRise() const { return clsValue[RISE]; }
84    T getFall() const { return clsValue[FALL]; }
85
86    Transitions<T> getReversed() const { return Transitions(
          getFall(), getRise()); }
87
88    void reverse() { return swap(clsValue[RISE], clsValue[FALL
          ]); }
89
90    T aggregate() const { return clsValue[RISE] + clsValue[
          FALL]; }
91  }; // end class
92
93
94  namespace std {
95      template<>
96      class numeric_limits<Transitions<double> > {
97      public:
98          static Transitions<double> min(){return Transitions<
                double>(numeric_limits<double>::min(),
                numeric_limits<double>::min());}
99          static Transitions<double> max(){return Transitions<
                double>(numeric_limits<double>::max(),
                numeric_limits<double>::max());}
100         static Transitions<double> zero(){return Transitions
                <double>(0.0f, 0.0f);}
101     };
102 }
103
104 #endif
```

Listing B.18 – transitions.h

```
1  #ifndef WIREDELAYMODEL_H_
2  #define WIREDELAYMODEL_H_
3
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "spef_net.h"
9  #include "liberty_library.h"
10 #include "configuration.h"
11
12 #include <cassert>
13
14 class WireDelayModel
15 {
16 protected:
17     double _lumped_capacitance;
```

```
18      double _total_resistance;
19    static LinearLibertyLookupTableInterpolator interpolator;
20
21  public:
22      WireDelayModel(const double & lumped_capacitance, const
            double & total_resistance) : _lumped_capacitance(
            lumped_capacitance), _total_resistance(
            total_resistance){}
23      virtual ~WireDelayModel(){}
24      virtual const Transitions<double> simulate(const
            LibertyCellInfo & cellInfo, const int input, const
            Transitions<double> slew, bool is_input_driver) = 0;
25      virtual const Transitions<double> delay_at_fanout_node(
            const string fanout_node_name) const = 0;
26      virtual const Transitions<double> slew_at_fanout_node(
            const string fanout_node_name) const = 0;
27    virtual void setFanoutPinCapacitance(const string
          fanoutNameAndPin, const double pinCapacitance) = 0;
28
29
30      virtual Transitions<double> root_delay(int arc_number) =
            0;
31      virtual Transitions<double> root_slew(int arc_number) =
            0;
32      virtual void clear() = 0;
33
34
35      double lumped_capacitance() const;
36      double total_resistance() const;
37  };
38
39  class LumpedCapacitanceWireDelayModel : public
        WireDelayModel
40  {
41      Transitions<double> _delay;
42      Transitions<double> _slew;
43
44      Transitions<double> _max_slew;
45  public:
46      LumpedCapacitanceWireDelayModel(const SpefNet &
            descriptor, const string root_node, const bool
            dummy_edge = false) : WireDelayModel(descriptor.
            netLumpedCap, descriptor.total_resistance){  }
47      const Transitions<double> simulate(const LibertyCellInfo
             & cellInfo, const int input, const Transitions<
            double> _slew, bool is_input_driver);
48      const Transitions<double> delay_at_fanout_node(const
            string fanout_node_name) const;
49      const Transitions<double> slew_at_fanout_node(const
            string fanout_node_name) const;
50      void setFanoutPinCapacitance(const string
            fanout_name_and_pin, const double pinCapacitance) {
```

```
        _lumped_capacitance += pinCapacitance; }
51
52
53      Transitions<double> root_delay(int arc_number);
54      Transitions<double> root_slew(int arc_number);
55      void clear();
56
57
58  };
59
60  class RC_Tree_Wire_Delay_Model : public WireDelayModel
61  {
62  protected:
63
64    struct Node
65    {
66      int parent;
67          Transitions<double> nodeCapacitance;
68          Transitions<double> totalCapacitance;
69      Transitions<double> effectiveCapacitance;
70          Transitions<double> resistance;
71      Transitions<double> slew;
72      Transitions<double> delay;
73      bool sink;
74      static vector<string> nodesNames;
75      Node() :
76              parent(-1), nodeCapacitance(numeric_limits<
                    Transitions<double> >::zero()),
                    totalCapacitance(numeric_limits<Transitions<
                    double> >::zero()), effectiveCapacitance(
                    numeric_limits<Transitions<double> >::zero()
                    ), resistance(numeric_limits<Transitions<
                    double> >::zero()), slew(numeric_limits<
                    Transitions<double> >::zero()), delay(
                    numeric_limits<Transitions<double> >::zero()
                    ), sink(false)
77      {
78      }
79      ;
80    };
81
82      vector<Node> _nodes;
83
84    struct NodeAndResistor
85    {
86      int nodeIndex;
87      int resistorIndex;
88      NodeAndResistor(const int & node, const int & resistor)
            :
89          nodeIndex(node), resistorIndex(resistor)
90      {
91      }
```

```
92      ;
93    };
94
95      vector<string> _nodes_names;
96      vector<vector<Transitions<double> > >_slews;
97      vector<vector<Transitions<double> > >_delays;
98
99      map<std::string, int> _node_name_to_node_number;
100
101
102     void IBM_update_downstream_capacitances();
103     void IBM_initialize_effective_capacitances();
104     void IBM_update_slews(const LibertyCellInfo & cellInfo,
            const int input, const Transitions<double> slew,
            bool is_input_driver);
105
106
107     void IBM_update_effective_capacitances();
108
109 protected:
110     const Transitions<double> run_IBM_algorithm(const
            LibertyCellInfo & cellInfo, const int input, const
            Transitions<double> slew, bool is_input_driver);
111 public:
112     RC_Tree_Wire_Delay_Model(const SpefNetISPD2013 &
            descriptor, const string rootNode, const size_t
            arcs_size, const bool dummyEdge = false);
113     virtual const Transitions<double> simulate(const
            LibertyCellInfo & cellInfo, const int input, const
            Transitions<double> slew, bool is_input_driver) = 0;
114     const Transitions<double> delay_at_fanout_node(const
            string fanout_node_name) const;
115     const Transitions<double> slew_at_fanout_node(const
            string fanout_node_name) const;
116   void setFanoutPinCapacitance(const string fanoutNameAndPin
        , const double pinCapacitance);
117
118
119     Transitions<double> root_delay(int arc_number);
120     Transitions<double> root_slew(int arc_number);
121     void clear();
122 };
123
124
125 /*
126
127   DRIVER: CEFF
128   INTERCONNECT: ELMORE + CEFF
129   SLEW: PURI02 (IBM)
130
131 */
132 class Ceff_Elmore_Slew_Degradation_PURI: public
```

```
    RC_Tree_Wire_Delay_Model
{


public:
    Ceff_Elmore_Slew_Degradation_PURI(const SpefNetISPD2013
        & descriptor, const string rootNode, const size_t
        arcs_size, const bool dummyEdge = false)
        : RC_Tree_Wire_Delay_Model(descriptor, rootNode,
            arcs_size, dummyEdge)
    {

    }

    const Transitions<double> simulate(const LibertyCellInfo
        & cellInfo, const int input, const Transitions<
        double> slew, bool is_input_driver);
};

/*

  DRIVER: CEFF
  INTERCONNECT: ELMORE + CEFF
  SLEW: sqrt(driver_slew ^ 2 + slew_degradation^2)

*/
class Ceff_Elmore_Slew_Degradation: public
    RC_Tree_Wire_Delay_Model
{


public:
    Ceff_Elmore_Slew_Degradation(const SpefNetISPD2013 &
        descriptor, const string rootNode, const size_t
        arcs_size, const bool dummyEdge = false)
        : RC_Tree_Wire_Delay_Model(descriptor, rootNode,
            arcs_size, dummyEdge)
    {

    }

    const Transitions<double> simulate(const LibertyCellInfo
        & cellInfo, const int input, const Transitions<
        double> slew, bool is_input_driver);
};

/*

  DRIVER: LUMPED
  INTERCONNECT: ELMORE
  SLEW: PURI02 (IBM)

```

```
173 */
174 class Lumped_Elmore_Slew_Degradation: public
        RC_Tree_Wire_Delay_Model
175 {
176
177
178 public:
179     Lumped_Elmore_Slew_Degradation(const SpefNetISPD2013 &
            descriptor, const string rootNode, const size_t
            arcs_size, const bool dummyEdge = false)
180         : RC_Tree_Wire_Delay_Model(descriptor, rootNode,
                arcs_size, dummyEdge)
181     {
182
183     }
184
185     const Transitions<double> simulate(const LibertyCellInfo
            & cellInfo, const int input, const Transitions<
            double> slew, bool is_input_driver);
186 };
187
188 /*
189
190 DRIVER: LUMPED
191 INTERCONNECT: ELMORE
192 SLEW: NO!
193
194 */
195 class Lumped_Elmore_No_Slew_Degradation: public
        RC_Tree_Wire_Delay_Model
196 {
197
198
199 public:
200     Lumped_Elmore_No_Slew_Degradation(const SpefNetISPD2013
            & descriptor, const string rootNode, const size_t
            arcs_size, const bool dummyEdge = false)
201         : RC_Tree_Wire_Delay_Model(descriptor, rootNode,
                arcs_size, dummyEdge)
202     {
203
204     }
205
206     const Transitions<double> simulate(const LibertyCellInfo
            & cellInfo, const int input, const Transitions<
            double> slew, bool is_input_driver);
207 };
208
209 /*
210
211 DRIVER: CEFF
212 INTERCONNECT: ELMORE
```

```
213 SLEW: NO!
214
215 */
216 class Ceff_Elmore_No_Slew_Degradation: public
        RC_Tree_Wire_Delay_Model
217 {
218
219
220 public:
221     Ceff_Elmore_No_Slew_Degradation(const SpefNetISPD2013 &
            descriptor, const string rootNode, const size_t
            arcs_size, const bool dummyEdge = false)
222         : RC_Tree_Wire_Delay_Model(descriptor, rootNode,
                arcs_size, dummyEdge)
223     {
224
225     }
226
227     const Transitions<double> simulate(const LibertyCellInfo
            & cellInfo, const int input, const Transitions<
            double> slew, bool is_input_driver);
228 };
229
230 /*
231
232 DRIVER: CEFF
233 INTERCONNECT: NO!
234 SLEW: NO!
235
236 */
237 class Ceff_Without_Wire_Delay_And_Slew_Degradation: public
        RC_Tree_Wire_Delay_Model
238 {
239
240
241 public:
242     Ceff_Without_Wire_Delay_And_Slew_Degradation(const
            SpefNetISPD2013 & descriptor, const string rootNode,
             const size_t arcs_size, const bool dummyEdge =
            false)
243         : RC_Tree_Wire_Delay_Model(descriptor, rootNode,
                arcs_size, dummyEdge)
244     {
245
246     }
247
248     const Transitions<double> simulate(const LibertyCellInfo
            & cellInfo, const int input, const Transitions<
            double> slew, bool is_input_driver);
249 };
250
251 class Reduced_Pi : public RC_Tree_Wire_Delay_Model
```

```
252 {
253
254
255     double _c1;
256     double _r;
257     double _c2;
258
259
260     Transitions<double> _slew_fanout;
261     Transitions<double> _delay_fanout;
262
263     void reduce_to_pi_model(double & c_near, double & r,
            double & c_far);
264
265 public:
266     Reduced_Pi(const SpefNetISPD2013 & descriptor, const
            string rootNode, const size_t arcs_size, const bool
            dummyEdge = false)
267         : RC_Tree_Wire_Delay_Model(descriptor, rootNode,
                arcs_size, dummyEdge)
268     {
269 //         cout << rootNode << endl;
270     }
271
272     const Transitions<double> simulate(const LibertyCellInfo
            & cellInfo, const int input, const Transitions<
            double> slew, bool is_input_driver);
273     const Transitions<double> delay_at_fanout_node(const
            string fanout_node_name) const;
274     const Transitions<double> slew_at_fanout_node(const
            string fanout_node_name) const;
275 };
276
277
278 #endif
```

Listing B.19 – wire_delay_model.h

```
1 #include "include/ceff_ratio_experiment.h"
2
3
4 void Ceff_Ratio_Experiment::run_sorted_by_wire_size(
      Timing_Analysis::Timing_Analysis &ta)
5 {
6     priority_queue<ratio_t, std::vector<ratio_t>,
          resistance_comparator > pq;
7     priority_queue<ratio_t, std::vector<ratio_t>, std::
          greater<ratio_t> > pq1;
8     priority_queue<ratio_t, std::vector<ratio_t>, std::
          greater<ratio_t> > pq2;
9     priority_queue<ratio_t, std::vector<ratio_t>, std::
          greater<ratio_t> > pq3;
```

```
10
11      double min_resistance = numeric_limits<double>::max();
12      double max_resistance = numeric_limits<double>::min();
13      double average_resistance = 0.0f;
14      int num_items = 0;
15      for(int i = 0; i < ta.timing_points_size(); i++)
16      {
17          const Timing_Analysis::Timing_Point & tp = ta.
                timing_point(i);
18          if(tp.is_PI() || tp.is_output_pin())
19          {
20              num_items++;
21              const double total_resistance = tp.net().
                    wire_delay_model()->total_resistance();
22              average_resistance += total_resistance;
23              max_resistance = max(max_resistance,
                    total_resistance);
24              min_resistance = min(min_resistance,
                    total_resistance);
25              pq.push(ratio_t { tp.ceff().getMax() / tp.net().
                    wire_delay_model()->lumped_capacitance(),
                    total_resistance});
26          }
27      }
28      average_resistance /= num_items;
29
30      int count = 0;
31      while(!pq.empty())
32      {
33          ratio_t item = pq.top();
34          pq.pop();
35          if(count <= num_items * 1/3)
36              pq1.push(item);
37          else if(count <= num_items * 2/3)
38              pq2.push(item);
39          else if(count <= num_items)
40              pq3.push(item);
41          count++;
42      }
43
44      cout << "####" << Traits::ispd_contest_benchmark << endl
            ;
45      cout << "max " << max_resistance << " min " <<
            min_resistance << " avg " << average_resistance <<
            endl;
46      cout << "ratio\tresistance" << endl;
47      while(!pq1.empty())
48      {
49          ratio_t item = pq1.top();
50          pq1.pop();
51          cout << item << endl;
52      }
```

```
53      cout << "#" << endl;
54      while(!pq2.empty())
55      {
56          ratio_t item = pq2.top();
57          pq2.pop();
58          cout << item << endl;
59      }
60      cout << "#" << endl;
61      while(!pq3.empty())
62      {
63          ratio_t item = pq3.top();
64          pq3.pop();
65          cout << item << endl;
66      }
67
68  }
69
70  void Ceff_Ratio_Experiment::run_sorted_by_slew(
        Timing_Analysis::Timing_Analysis &ta)
71  {
72      priority_queue<ratio_t, std::vector<ratio_t>,
            slew_comparator > pq;
73      priority_queue<ratio_t, std::vector<ratio_t>, std::
            greater<ratio_t> > pq1;
74      priority_queue<ratio_t, std::vector<ratio_t>, std::
            greater<ratio_t> > pq2;
75      priority_queue<ratio_t, std::vector<ratio_t>, std::
            greater<ratio_t> > pq3;
76
77      double min_resistance = numeric_limits<double>::max();
78      double max_resistance = numeric_limits<double>::min();
79      double average_resistance = 0.0f;
80      int num_items = 0;
81      for(int i = 0; i < ta.timing_points_size(); i++)
82      {
83          const Timing_Analysis::Timing_Point & tp = ta.
                timing_point(i);
84          if(tp.is_PI() || tp.is_output_pin())
85          {
86              num_items++;
87              const double total_resistance = tp.net().
                    wire_delay_model()->total_resistance();
88              average_resistance += total_resistance;
89              max_resistance = max(max_resistance,
                    total_resistance);
90              min_resistance = min(min_resistance,
                    total_resistance);
91
92              int num_of_timing_arcs = 0;
93              int tp_index = i-1;
94              while(tp_index >= 0 && ta.timing_point(tp_index)
                    .gate_number() == tp.gate_number())
```

```
95              {
96                  tp_index--;
97                  num_of_timing_arcs++;
98              }
99
100             Transitions<double> slew = numeric_limits<
                    Transitions<double> >::min();
101             for(int j = i - num_of_timing_arcs; j < i; j++)
102             {
103                 const Timing_Analysis::Timing_Point &
                        input_pin = ta.timing_point(j);
104                 slew = max(slew, input_pin.arc().slew());
105             }
106
107
108             pq.push(ratio_t { tp.ceff().getMax() / tp.net().
                    wire_delay_model()->lumped_capacitance(),
                    total_resistance, slew.getMax() });
109         }
110     }
111     average_resistance /= num_items;
112
113     int count = 0;
114     while(!pq.empty())
115     {
116         ratio_t item = pq.top();
117         pq.pop();
118         if(count <= num_items * 1/3)
119             pq1.push(item);
120         else if(count <= num_items * 2/3)
121             pq2.push(item);
122         else if(count <= num_items)
123             pq3.push(item);
124         count++;
125     }
126
127     cout << "####" << Traits::ispd_contest_benchmark << endl
            ;
128     cout << "max " << max_resistance << " min " <<
            min_resistance << " avg " << average_resistance <<
            endl;
129     cout << "ratio\tresistance" << endl;
130     while(!pq1.empty())
131     {
132         ratio_t item = pq1.top();
133         pq1.pop();
134         cout << item << endl;
135     }
136     cout << "#" << endl;
137     while(!pq2.empty())
138     {
139         ratio_t item = pq2.top();
```

```
140             pq2.pop();
141             cout << item << endl;
142         }
143         cout << "#" << endl;
144         while(!pq3.empty())
145         {
146             ratio_t item = pq3.top();
147             pq3.pop();
148             cout << item << endl;
149         }
150     }
151
152     void Ceff_Ratio_Experiment::run_average_calculation(
        Timing_Analysis::Timing_Analysis &ta)
153     {
154         int num_items = 0;
155         Transitions<double> average(0.0f, 0.0f);
156         Transitions<double> slew_average(0.0f, 0.0f);
157         for(int i = 0; i < ta.timing_points_size(); i++)
158         {
159             const Timing_Analysis::Timing_Point & tp = ta.
                timing_point(i);
160             if(tp.is_PI() || tp.is_output_pin())
161             {
162                 num_items++;
163                 average += (tp.ceff().getMax() / tp.net().
                    wire_delay_model()->lumped_capacitance());
164                 slew_average += tp.slew();
165             }
166         }
167         slew_average /= num_items;
168         average /= num_items;
169
170
171
172         cout << average << " slew " << slew_average;
173     }
```

Listing B.20 – ceff_ratio_experiment.cpp

```
1   #include "include/circuit_netlist.h"
2
3   // CIRCUIT NETLIST
4   void Circuit_Netlist::updateTopology()
5   {
6       vector<size_t> num_of_visited_inputs(gates.size(), 0);
7       vector<bool> inserted_gate(gates.size(), false);
8       vector<bool> inserted_net(nets.size(), false);
9
10      queue<int> gates_queue;
11      queue<int> primary_output_queue;
```

```
12    for(size_t i = 0; i < gates.size(); i++)
13    {
14      if(gates[i].inputDriver)
15      {
16              gates_queue.push(i);
17              inserted_gate[i] = true;
18      }
19    }
20
21    topology.clear();
22    netTopology.clear();
23
24    inverseTopology.resize(gates.size(), -1);
25    inverseNetTopology.resize(nets.size(), -1);
26
27      while(!gates_queue.empty())
28    {
29          const int current_index = gates_queue.front();
30          gates_queue.pop();
31
32          const Logic_Gate & gate = gates.at(current_index);
33          const Net & fanout_net = nets.at(gate.fanoutNetIndex
                );
34
35          // insert input nets to the nets topology
36          for(size_t i = 0; i < gate.inNets.size(); i++)
37      {
38              if(!inserted_net.at(gate.inNets.at(i)))
39        {
40                  netTopology.push_back(gate.inNets.at(i));
41                  inverseNetTopology[gate.inNets.at(i)] =
                        netTopology.size() - 1;
42                  inserted_net[gate.inNets.at(i)] = true;
43        }
44      }
45          // insert output net to the nets topology
46          if(!inserted_net.at(gate.fanoutNetIndex))
47      {
48        netTopology.push_back(gate.fanoutNetIndex);
49              inverseNetTopology[gate.fanoutNetIndex] =
                    netTopology.size() - 1;
50              inserted_net[gate.fanoutNetIndex] = true;
51      }
52
53          // insert gate to the topolgy
54          topology.push_back(current_index);
55          inverseTopology[current_index] = topology.size() -
                1;
56
57
58  //          cout << gate.name << " " << (gate.sequential?"
        SEQUENTIAL":"") << endl;
```

```
59
60
61              // push fanout gates to the process queue
62              for(size_t i = 0; i < fanout_net.sinks.size(); i++)
63          {
64                  const size_t fanoutIndex = fanout_net.sinks.at(i
                        ).gate;
65                  const Logic_Gate & fanout = gates[fanoutIndex];
66
67                  if(!inserted_gate.at(fanoutIndex))
68            {
69                      if(++num_of_visited_inputs[fanoutIndex] ==
                            fanout.inNets.size())
70              {
71                          if(fanout.primary_output || fanout.
                                sequential && !fanout.inputDriver)
72                          {
73                              primary_output_queue.push(
                                    fanoutIndex);
74                          }
75                          else
76                              gates_queue.push(fanoutIndex);
77                          inserted_gate[fanoutIndex] = true;
78              }
79            }
80          }
81      }
82
83          while(!primary_output_queue.empty())
84          {
85
86
87              const int PO_index = primary_output_queue.front();
88              primary_output_queue.pop();
89              Logic_Gate & gate = gates.at(PO_index);
90
91              // Insert Input Nets
92              for(size_t i = 0; i < gate.inNets.size(); i++)
93              {
94                  if(!inserted_net.at(gate.inNets.at(i)))
95                  {
96                      netTopology.push_back(gate.inNets.at(i));
97                      inverseNetTopology[gate.inNets.at(i)] =
                            netTopology.size() - 1;
98                      inserted_net[gate.inNets.at(i)] = true;
99                  }
100             }
101             // insert output nets
102             if(!inserted_net.at(gate.fanoutNetIndex))
103             {
104                 netTopology.push_back(gate.fanoutNetIndex);
105                 inverseNetTopology[gate.fanoutNetIndex] =
```

```
                netTopology.size() - 1;
106             inserted_net[gate.fanoutNetIndex] = true;
107         }
108
109         topology.push_back(PO_index);
110         inverseTopology[PO_index] = topology.size() - 1;
111
112
113     }
114
115     assert(netTopology.size() == inverseNetTopology.size());
116     assert(topology.size() == inverseTopology.size());
117
118 }
119
120 int Circuit_Netlist::addGate(const string name, const string
        cellType, const int inputs, const bool isInputDriver,
    const bool primary_output)
121 {
122   if(gateNameToGateIndex.find(name) != gateNameToGateIndex.
        end())
123     return gateNameToGateIndex[name];
124
125   //const string name, const string cellType, const unsigned
        inputs, int fanoutNetIndex
126   gates.push_back(Logic_Gate(name, cellType, inputs, -1,
        isInputDriver, primary_output));
127
128 //     _timing_points++;
129 //     if(cellType != "__PO__" || (gates.back().sequential &&
      !gates.back().inputDriver))
130 //     {
131 //         _timing_points += gates.back().inNets.size();
132 //         _timing_arcs += inputs;
133 //     }
134
135   gateNameToGateIndex[name] = gates.size() - 1;
136
137   return gateNameToGateIndex[name];
138 }
139
140 int Circuit_Netlist::addNet(const string name)
141 {
142   if(netNameToNetIndex.find(name) != netNameToNetIndex.end()
        )
143     return netNameToNetIndex[name];
144
145   nets.push_back(Net(name, -1, "o"));
146   netNameToNetIndex[name] = nets.size() - 1;
147   return netNameToNetIndex[name];
148 }
149
```

```
150 int Circuit_Netlist::addNet(const string name, const int
        sourceNode, const string sourcePin)
151 {
152   if(netNameToNetIndex.find(name) != netNameToNetIndex.end()
        )
153   {
154     if(nets.at(netNameToNetIndex.at(name)).sourceNode == -1)
155     {
156       // cout << "solved source" << endl;
157       nets.at(netNameToNetIndex.at(name)).sourceNode =
            sourceNode;
158     }
159
160     assert(nets.at(netNameToNetIndex.at(name)).sourceNode ==
          sourceNode);
161
162     return netNameToNetIndex.at(name);
163   }
164
165   nets.push_back(Net(name, sourceNode, sourcePin));
166   netNameToNetIndex[name] = nets.size() - 1;
167   return netNameToNetIndex[name];
168 }
169
170 void Circuit_Netlist::addCellInst(const string name, const
        string cellType, vector<pair<string, string> >
        inputPinPairs, const bool isSequential, const bool
        isInputDriver, const bool primary_output)
171 {
172   const string outputPin = inputPinPairs.back().first;
173   const string fanoutNetName = inputPinPairs.back().second;
174
175
176 //        if(isSequential)
177 //        {
178 //            _timing_points += 3;
179 //            _timing_arcs++;
180 //        }
181 //        else if (isInputDriver)
182 //        {
183 //            _timing_points += 2;
184 //            _timing_arcs++;
185 //        }
186 //        else if (!primary_output)
187 //        {
188 //            _timing_points++;
189 //        }
190 //        else
191 //        {
192 //            _timing_points += inputPinPairs.size();
193 //            _timing_arcs += inputPinPairs.size() - 1;
194 //        }
```

```
195
196
197     if(isSequential)
198     {
199       const int gateIndex = addGate(name, cellType,
              inputPinPairs.size() - 1);
200       const int netIndex = addNet(name + "_PO", gateIndex,
              outputPin);
201       Logic_Gate & gate = gates[gateIndex];
202       gate.fanoutNetIndex = netIndex;
203       gate.sequential = isSequential;
204       for(size_t i = 0; i < inputPinPairs.size() - 1 ; i++)
205       {
206         const int faninNetIndex = addNet(inputPinPairs[i].
                second);
207         Net & faninNet = nets[faninNetIndex];
208         faninNet.addSink(Sink(gateIndex, inputPinPairs[i].
                first));
209         gate.inNets[i] = faninNetIndex;
210       }
211
212
213       // Creates a INPUT DRIVER to the flip flop
214       inputPinPairs.front().second += "_PI";
215       const int sequentialCellGateIndex = addGate(name + "_PI"
              , cellType, inputPinPairs.size() - 1, true);
216       const int sequentialCellGatePINetIndex = addNet(
              fanoutNetName, sequentialCellGateIndex, outputPin);
217       Logic_Gate & sequentialGate = gates[
              sequentialCellGateIndex];
218       sequentialGate.fanoutNetIndex =
              sequentialCellGatePINetIndex;
219       sequentialGate.sequential = isSequential;
220       for(size_t i = 0; i < inputPinPairs.size() - 1 ; i++)
221       {
222         const int faninNetIndex = addNet(inputPinPairs[i].
                second);
223         Net & faninNet = nets[faninNetIndex];
224         faninNet.addSink(Sink(sequentialCellGateIndex,
                inputPinPairs[i].first));
225         sequentialGate.inNets[i] = faninNetIndex;
226       }
227
228     }
229     else
230     {
231       const int gateIndex = addGate(name, cellType,
              inputPinPairs.size() - 1, isInputDriver,
              primary_output);
232       const int netIndex = addNet(fanoutNetName, gateIndex,
              outputPin);
233       Logic_Gate & gate = gates[gateIndex];
```

```
234       Net & net = nets[netIndex];
235       if(net.sourceNode == -1)
236         net.sourceNode = gateIndex;
237       gate.fanoutNetIndex = netIndex;
238       for(size_t i = 0; i < inputPinPairs.size() - 1 ; i++)
239       {
240         const int faninNetIndex = addNet(inputPinPairs[i].
              second);
241         Net & faninNet = nets[faninNetIndex];
242         faninNet.addSink(Sink(gateIndex, inputPinPairs[i].
              first));
243         gate.inNets[i] = faninNetIndex;
244       }
245       gate.sequential = isSequential;
246
247     }
248     if( !primary_output && !isInputDriver )
249       _numberOfGates++;
250
251
252
253   }
254
255   const vector<pair<int, string> > Circuit_Netlist::verilog()
          const
256   {
257     vector<pair<int, string> > verilogVector(_numberOfGates);
258     int j = 0;
259       for(size_t i = 0; i < gates.size(); i++)
260     {
261       if(gates.at(i).inputDriver || gates.at(i).primary_output
              )
262         continue;
263       const pair<int, string> indexAndName = make_pair(
              inverseTopology.at(i), gates.at(i).name);
264       verilogVector[j++] = indexAndName;
265     }
266       return verilogVector;
267   }
268
269   int Circuit_Netlist::timing_arcs() const
270   {
271       return _timing_arcs;
272   }
273
274   int Circuit_Netlist::timing_points() const
275   {
276       return _timing_points;
277   }
```

Listing B.21 – circuit_netlist.cpp

```
#include "include/configuration.h"

string Traits::ispd_contest_root;
string Traits::ispd_contest_benchmark;
string Traits::arrival_time_file_name;
```

Listing B.22 – configuration.cpp

```
#include "include/design_constraints.h"

// private
void Design_Constraints::clock(const double clock)
{
    _clock = clock;
}
bool Design_Constraints::input_delay(const string input_name
    , const Transitions<double> delay)
{
    if(_input_delays.find(input_name) != _input_delays.end()
        )
    return false;
    _input_delays.insert(make_pair(input_name, delay));
  return true;
}
bool Design_Constraints::output_delay(const string
    output_name, const Transitions<double> delay)
{
    if(_output_delays.find(output_name) != _output_delays.
        end())
    return false;
    _output_delays.insert(make_pair(output_name, delay));
  return true;
}
bool Design_Constraints::output_load(const string
    output_name, const double output_load)
{
    if(_output_loads.find(output_name) != _output_loads.end
        ())
    return false;
    _output_loads.insert(make_pair(output_name, output_load)
        );
  return true;
}
bool Design_Constraints::driving_cell(const string
    input_name, const string driving_cell)
{
    if(_driving_cells.find(input_name) != _driving_cells.end
        ())
    return false;
    _driving_cells.insert(make_pair(input_name, driving_cell
        ));
```

```
34    return true;
35  }
36
37  bool Design_Constraints::input_transition(const string
        input_name, const Transitions<double> transition)
38  {
39      if(_input_transitions.find(input_name) !=
            _input_transitions.end())
40      return false;
41      _input_transitions.insert(make_pair(input_name,
            transition));
42    return true;
43  }
44
45
46  // public
47  double Design_Constraints::clock() const
48  {
49      return _clock;
50  }
51  const Transitions<double> Design_Constraints::input_delay(
        const string input_name) const
52  {
53      return _input_delays.at(input_name);
54  }
55  const Transitions<double> Design_Constraints::output_delay(
        const string output_name) const
56  {
57      return _output_delays.at(output_name);
58  }
59  double Design_Constraints::output_load(const string
        output_name) const
60  {
61      return _output_loads.at(output_name);
62  }
63
64  size_t Design_Constraints::output_loads_size() const
65  {
66      return _output_loads.size();
67  }
68  const string Design_Constraints::driving_cell(const string
        input_name) const
69  {
70      return _driving_cells.at(input_name);
71  }
72
73  const Transitions<double> Design_Constraints::
        input_transition(const string input_name) const
74  {
75      return _input_transitions.at(input_name);
76  }
```

Listing B.23 – design_constraints.cpp

```
#include "include/liberty_library.h"


LibertyLibrary::LibertyLibrary(const double maxTransition) :
    maxTransition(maxTransition)
{
  library.push_back(vector<LibertyCellInfo>());

// DUMMY CELL TYPE TO PRIMARY OUTPUT
  LibertyCellInfo po;
  po.name = "__PO__";
  po.footprint = "__PO__";
  po.pins.resize(1);
  po.timingArcs.resize(1);
  po.primaryOutput = true;
  library[0].push_back(po);
  footPrintToIndex[po.footprint] = 0;
  cellOptionNumber[po.name] = 0;
  cellToFootprintIndex[po.name] = footPrintToIndex[po.
      footprint];

}
LibertyLibrary::~LibertyLibrary()
{

}

const pair<int, int> LibertyLibrary::addCellInfo(const
    LibertyCellInfo & cellInfo)
{

  if(footPrintToIndex.find(cellInfo.footprint) ==
      footPrintToIndex.end())
  {
    library.push_back(vector<LibertyCellInfo>());
    footPrintToIndex[cellInfo.footprint] = library.size() -
        1;
  }

  const int footPrintIndex = footPrintToIndex[cellInfo.
      footprint];
  const int optionIndex = library[footPrintIndex].size();

  library[footPrintIndex].push_back(cellInfo);
  cellOptionNumber[cellInfo.name] = optionIndex;
  cellToFootprintIndex[cellInfo.name] = footPrintIndex;

  return make_pair(footPrintIndex, optionIndex);
```

```
43 }
44
45 const LibertyCellInfo & LibertyLibrary::getCellInfo(const
       string & footPrint, const int & i) const
46 {
47   const int footPrintIndex = footPrintToIndex.at(footPrint);
48   return library.at(footPrintIndex).at(i);
49 }
50
51 const LibertyCellInfo & LibertyLibrary::getCellInfo(const
       string & cellName) const
52 {
53   const int footPrintIndex = cellToFootprintIndex.at(
         cellName);
54   const int optionNumber = cellOptionNumber.at(cellName);
55   return library.at(footPrintIndex).at(optionNumber);
56 }
57
58 const LibertyCellInfo & LibertyLibrary::getCellInfo(const
       int & footPrintIndex, const int & optionIndex) const
59 {
60     return library.at(footPrintIndex).at(optionIndex);
61 }
62
63 size_t LibertyLibrary::number_of_options(const int
       footprint_index) const
64 {
65     return library.at(footprint_index).size();
66 }
67
68
69 const pair<int, int> LibertyLibrary::getCellIndex(const
       string &cellName) const
70 {
71   const int footPrintIndex = cellToFootprintIndex.at(
         cellName);
72   const int optionNumber = cellOptionNumber.at(cellName);
73   return make_pair(footPrintIndex, optionNumber);
74 }
75
76
77
78 double LinearLibertyLookupTableInterpolator::interpolate(
       const LibertyLookupTable & lut, const double load, const
        double transition)
79 {
80   double wTransition, wLoad, y1, y2, x1, x2;
81   double t[2][2];
82   int row1, row2, column1, column2;
83   wTransition = 0.0f;
84   wLoad = 0.0f;
85
```

```
86    assert(load >= 0 && transition >= 0);
87
88    row1 = lut.loadIndices.size() - 2;
89    row2 = lut.loadIndices.size() - 1;
90
91    y1 = lut.loadIndices[row1];
92    y2 = lut.loadIndices[row2];
93
94    // loads -- rows
95    for(size_t i = 0; i < lut.loadIndices.size() - 1; i++)
96    {
97      if(load >= lut.loadIndices[i] && load <= lut.loadIndices
          [i + 1])
98      {
99        row1 = i;
100       row2 = i + 1;
101       y1 = lut.loadIndices[row1];
102       y2 = lut.loadIndices[row2];
103     }
104   }
105
106   // transitions -- columns
107   if(transition < lut.transitionIndices[0])
108   {
109     column1 = 0;
110     column2 = 1;
111     x1 = lut.transitionIndices[column1];
112     x2 = lut.transitionIndices[column2];
113   }
114   else if (transition > lut.transitionIndices[lut.
        transitionIndices.size()-1])
115   {
116     column1 = lut.transitionIndices.size() - 2;
117     column2 = lut.transitionIndices.size() - 1;
118     x1 = lut.transitionIndices[column1];
119     x2 = lut.transitionIndices[column2];
120   }
121   else
122   {
123     for(size_t i = 0; i < lut.transitionIndices.size() - 1;
          i++)
124     {
125       if(transition >= lut.transitionIndices[i] &&
            transition <= lut.transitionIndices[i + 1])
126       {
127         column1 = i;
128         column2 = i + 1;
129         x1 = lut.transitionIndices[column1];
130         x2 = lut.transitionIndices[column2];
131       }
132     }
133   }
```

```
134
135     //equation for interpolation (Ref - ISPD Contest: http://
            www.ispd.cc/contests/12/ISPD_2012_Contest_Details.pdf)
            , slide 17
136     wTransition = (transition - x1) * (1.0f / (x2 - x1));
137     wLoad = (load - y1) * (1.0f / (y2 - y1));
138
139     t[0][0] = lut.tableVals[row1][column1];
140     t[0][1] = lut.tableVals[row1][column2];
141     t[1][0] = lut.tableVals[row2][column1];
142     t[1][1] = lut.tableVals[row2][column2];
143
144       return ((1 - wTransition) * (1 - wLoad) * t[0][0]) + (
              wTransition * (1 - wLoad) * t[0][1]) + ((1 -
              wTransition) * wLoad * t[1][0]) + (wTransition *
              wLoad * t[1][1]);
145   }
146
147
148 const int LibertyLookupTableInterpolator::
        DEFAULT_DECIMAL_PLACES = 2;
149
150   const Transitions<double>
          LinearLibertyLookupTableInterpolator::interpolate(const
           LibertyLookupTable & rise_lut, const
          LibertyLookupTable & fall_lut, const Transitions<double
          > load, const Transitions<double> transition, Unateness
           unateness, bool is_input_driver)
151   {
152       Transitions<double> result;
153       double rise_delay, fall_delay;
154       switch(unateness)
155       {
156       case NEGATIVE_UNATE:
157
158           rise_delay = interpolate(rise_lut, load.getRise(),
                  transition.getFall());
159           fall_delay = interpolate(fall_lut, load.getFall(),
                  transition.getRise());
160
161           if(is_input_driver)
162           {
163               rise_delay -= interpolate(rise_lut, 0.0f,
                      transition.getFall());
164               fall_delay -= interpolate(rise_lut, 0.0f,
                      transition.getRise());
165           }
166           break;
167       case POSITIVE_UNATE:
168           rise_delay = interpolate(rise_lut, load.getRise(),
                  transition.getRise());
169           fall_delay = interpolate(fall_lut, load.getFall(),
```

```
                transition.getFall());
170             break;
171         case NON_UNATE:
172             rise_delay = max(interpolate(rise_lut, load.getRise
                    (), transition.getFall()), interpolate(rise_lut,
                     load.getRise(), transition.getRise()));
173             fall_delay = max(interpolate(fall_lut, load.getFall
                    (), transition.getRise()), interpolate(fall_lut,
                     load.getFall(), transition.getFall()));
174 //            rise_delay = 16.0f + 0.0390625f * load.getRise();
175 //            fall_delay = rise_delay;
176             break;
177         }
178
179         result = Transitions<double>(rise_delay, fall_delay);
180         round(result, DEFAULT_DECIMAL_PLACES);
181         return result;
182     }
183
184 double LibertyLibrary::getMaxTransition() const
185 {
186         return maxTransition;
187 }
188
189 void LibertyLookupTableInterpolator::round(Transitions<
        double> &transitions, const int decimal_places)
190 {
191         return;
192         const Transitions<int> truncated = Transitions<int>(int(
                transitions.getRise() * pow(10, decimal_places)),
                int(transitions.getFall() * pow(10, decimal_places))
                );
193         transitions = Transitions<double>(truncated.getRise(),
                truncated.getFall());
194         transitions /= pow(10, decimal_places);
195 }
```

Listing B.24 – liberty_library.cpp

```
1  #include <iostream>
2  using std::cout;
3  using std::cerr;
4  using std::endl;
5
6  #include "include/timing_analysis.h"
7  #include "include/parser.h"
8  #include "include/circuit_netlist.h"
9  #include "include/spef_net.h"
10
11 #include "include/configuration.h"
12 #include "include/timer.h"
```

```
13 #include "include/ceff_ratio_experiment.h"
14 #include "include/slew_degradation_experiment.h"
15
16 #include <cstdio>
17 #include <queue>
18 using std::priority_queue;
19
20 #include <ostream>
21 using std::ostream;
22
23 using std::make_pair;
24
25
26
27 struct PassingArgs {
28     string _contest_root;
29     string _contest_benchmark;
30     PassingArgs(string contestRoot, string contestBenchmark)
          : _contest_root(contestRoot), _contest_benchmark(
          contestBenchmark){}
31 };
32
33 struct ISPDContestFiles
34 {
35     string verilog;
36     string spef;
37     string liberty;
38     string designConstraints;
39     ISPDContestFiles(string contestRoot, string
          contestBenchmark) {
40         verilog = contestRoot + "/" + contestBenchmark + "/"
               + contestBenchmark + ".v";
41         spef = contestRoot + "/" + contestBenchmark + "/" +
              contestBenchmark + ".spef";
42         designConstraints = contestRoot + "/" +
              contestBenchmark + "/" + contestBenchmark + ".
              sdc";
43         liberty = contestRoot + "/lib/contest.lib";
44     }
45 };
46
47 int main(int argc, char const *argv[])
48 {
49     if(argc != 3)
50     {
51         cerr << "Using: " << argv[0] << " <CONTEST_ROOT> <
              CONTEST_BENCHMARK>" << endl;
52         return -1;
53     }
54
55     const PassingArgs args(argv[1], argv[2]);
56     Traits::ispd_contest_root = argv[1];
```

```
57      Traits::ispd_contest_benchmark = argv[2];
58
59      VerilogParser vp;
60      LibertyParser lp;
61      SpefParser sp;
62      SDCParser dcp;
63
64      ISPDContestFiles files(argv[1], argv[2]);
65
66      const Circuit_Netlist netlist = vp.readFile(files.
            verilog);
67      const LibertyLibrary library = lp.readFile(files.liberty
            );
68      const Parasitics parasitics = sp.readFile(files.spef);
69      const Design_Constraints constraints = dcp.readFile(
            files.designConstraints);
70
71      Timing_Analysis::Timing_Analysis ta(netlist, &library, &
            parasitics, &constraints);
72
73      ta.set_all_gates_to_max_size();
74      ta.full_timing_analysis();
75      ta.write_timing_file("min_power.timing");
76
77      ta.incremental_timing_analysis(ta.number_of_gates()/2,
            15);
78      ta.write_timing_file("after_change_inc.timing");
79      ta.full_timing_analysis();
80      ta.write_timing_file("after_change.timing");
81
82      bool ok = ta.check_timing_file("after_change_inc.timing"
            );
83      assert(ok);
84      cout << "OK!" << endl;
85
86  //    cout << "number of gates " << ta.number_of_gates() <<
        endl;
87  //    cout << "timing points   " << ta.timing_points_size()
        << endl;
88  //    cout << "timing nets     " << ta.timing_nets_size() <<
         endl;
89  //    cout << "timing arcs     " << ta.timing_arcs_size() <<
         endl;
90
91
92  //    cout << "gates " << endl;
93  //    for (int i = 0; i < ta.number_of_gates(); ++i) {
94  //        cout << " #" << i << endl;
95  //        cout << "  footprint index " << ta.option(i).
        footprint_index() << endl;
96  //        cout << "  option index    " << ta.option(i).
        option_index() << endl;
```

```
97  //         cout << "  is dont touch?  " << (ta.option(i).
        is_dont_touch() ? "yes" : "no") << endl;
98  //     }
99
100 //     cout << "timing points " << endl;
101 //     for (int i = 0; i < ta.timing_points_size(); ++i) {
102
103 //         string type;
104 //         if(ta.timing_point(i).is_input_pin())
105 //             type = "input_pin";
106 //         else if(ta.timing_point(i).is_output_pin())
107 //             type = "output_pin";
108 //         else if(ta.timing_point(i).is_PI())
109 //             type = "PI";
110 //         else if(ta.timing_point(i).is_PI_input())
111 //             type = "PI_input";
112 //         else if(ta.timing_point(i).is_PO())
113 //             type = "PO";
114 //         else if(ta.timing_point(i).is_reg_input())
115 //             type = "reg_input";
116 //         else
117 //             type = "__UNKNOWN__";
118 //         cout << " #" << i << endl;
119 //         cout << "  name        " << ta.timing_point(i).
        name() << endl;
120 //         cout << "  gate number " << ta.timing_point(i).
        gate_number() << endl;
121 //         cout << "  type        " << type << endl;
122 //     }
123
124     return 0;
125 }
```

Listing B.25 – main.cpp

```
1  #include "include/parser.h"
2
3  Parser::Parser()
4  {
5
6  }
7
8  Parser::~Parser()
9  {
10
11 }
12
13 bool Parser::isSpecialChar(const char & c)
14 {
15   static const char specialChars[] =
16   { '(', ')', ',', ':', ';', '/', '#', '[', ']', '{', '}', '
         *', '\"', '\\' };
```

```
17
18    for (unsigned i = 0; i < sizeof(specialChars); ++i)
19    {
20      if (c == specialChars[i])
21        return true;
22    }
23
24    return false;
25  }
26
27
28  bool Parser::readLineAsTokens(istream& is, vector<string>&
        tokens, bool includeSpecialChars)
29  {
30    tokens.clear();
31
32    string line;
33    std::getline (is, line);
34
35    while (is && tokens.empty()) {
36
37      string token = "";
38
39          for (size_t i=0; i < line.size(); ++i) {
40        char currChar = line[i];
41        bool _isSpecialChar = isSpecialChar(currChar);
42
43        if (std::isspace (currChar) || _isSpecialChar) {
44
45        if (!token.empty()) {
46          // Add the current token to the list of tokens
47          tokens.push_back(token);
48          token.clear();
49        }
50
51        if (includeSpecialChars && _isSpecialChar) {
52          tokens.push_back(string(1, currChar));
53        }
54
55        } else {
56          // Add the char to the current token
57          token.push_back(currChar);
58        }
59
60      }
61
62      if (!token.empty())
63        tokens.push_back(token);
64
65
66      if (tokens.empty())
67        // Previous line read was empty. Read the next one.
```

```
68          std::getline (is, line) ;
69      }
70
71        //for (size_t i=0; i < tokens.size(); ++i)
72      //  cout << tokens[i] << " " ;
73      //cout << endl ;
74
75      return !tokens.empty() ;
76  }
77
78
79  // VERILOG PARSER
80  const string VerilogParser::SEQUENTIAL_CELL = "ms00f80";
81  const string VerilogParser::INPUT_DRIVER_CELL = "in01f80";
82  const string VerilogParser::PRIMARY_OUTPUT_CELL = "__PO__";
83  const string VerilogParser::CLOCK_NET = "ispd_clk";
84
85  const Circuit_Netlist VerilogParser::readFile(const string
        filename)
86  {
87      is.open(filename.c_str(), fstream::in);
88      string moduleName;
89      bool valid = read_module(moduleName);
90      assert(valid);
91
92      Circuit_Netlist netlist;
93
94      // cout << "Module " << moduleName << endl << endl;
95
96      do
97      {
98        string primaryInput;
99        valid = read_primary_input(primaryInput);
100
101       if (valid)
102       {
103         // cout << "Primary input: " << primaryInput << endl;
104         if(primaryInput != CLOCK_NET)
105         {
106           vector<std::pair<string, string> > piPins;
107           piPins.push_back(make_pair("a", primaryInput + "_PI"
                ));
108           piPins.push_back(make_pair("o", primaryInput));
109           netlist.addCellInst(primaryInput, INPUT_DRIVER_CELL,
                  piPins, false, true);
110         }
111       }
112
113     }
114     while (valid);
115
116     // cout << endl;
```

```
117
118     do
119     {
120       string primaryOutput;
121       valid = read_primary_output(primaryOutput);
122
123       if (valid)
124       {
125         // cout << "Primary output: " << primaryOutput << endl
                ;
126         // netlist.addNet(primaryOutput);
127         vector<std::pair<string, string> > poPins;
128         poPins.push_back(make_pair("i", primaryOutput));
129         poPins.push_back(make_pair("o", primaryOutput + "_PO")
                );
130         netlist.addCellInst(primaryOutput, PRIMARY_OUTPUT_CELL
                , poPins, false, false, true);
131       }
132
133     }
134     while (valid);
135
136     // cout << endl;
137
138     do
139     {
140       string net;
141       valid = read_wire(net);
142
143       if (valid)
144       {
145         // cout << "Net: " << net << endl;
146         // netlist.addNet(net);
147       }
148
149     }
150     while (valid);
151
152     // cout << endl;
153     // cout << "Cell insts: " << std::endl;
154
155     do
156     {
157       string cellType, cellInst;
158       vector<std::pair<string, string> > pinNetPairs;
159
160       valid = read_cell_inst(cellType, cellInst, pinNetPairs);
161
162       if (valid)
163       {
164         // cout << cellType << " " << cellInst << " ";
165               // for (size_t i = 0; i < pinNetPairs.size(); ++
```

```
                    i)
166         // {
167         //  cout << "(" << pinNetPairs[i].first << " " <<
                pinNetPairs[i].second << ") ";
168         // }
169
170         // cout << endl;
171         const bool isSequential = ( cellType ==
                SEQUENTIAL_CELL );
172         if(isSequential)
173         {
174           for(vector<pair<string, string> >::iterator it =
                  pinNetPairs.begin(); it != pinNetPairs.end(); it
                  ++)
175           {
176             if((*it).second == CLOCK_NET )
177             {
178               pinNetPairs.erase(it);
179               break;
180             }
181           }
182         }
183
184
185         netlist.addCellInst(cellInst, cellType, pinNetPairs,
                isSequential);
186       }
187
188
189     }
190     while (valid);
191     is.close();
192
193     for(size_t i = 0; i < netlist.getNetsSize(); i++)
194     {
195       const int sourceNodeIndex = netlist.getNet(i).sourceNode
              ;
196       const int sinkNodeIndex = (netlist.getNet(i).sinks.empty
              () ? -1 : netlist.getNet(i).sinks.front().gate);
197       if( sourceNodeIndex == -1 || sinkNodeIndex == -1 )
198         netlist.getNet(i).dummyNet = true;
199     }
200
201     netlist.updateTopology();
202
203     return netlist;
204 }
205
206
207 bool VerilogParser::read_module(string& moduleName)
208 {
209
```

```
210     vector<string> tokens;
211     bool valid = readLineAsTokens(is, tokens);
212
213     while (valid)
214     {
215
216       if (tokens.size() == 2 && tokens[0] == "module")
217       {
218         moduleName = tokens[1];
219
220         break;
221       }
222
223       valid = readLineAsTokens(is, tokens);
224     }
225
226     // Read and skip the port names in the module definition
227     // until we encounter the tokens {"Start", "PIs"}
228     while (valid && !(tokens.size() == 2 && tokens[0] == "
            Start" && tokens[1] == "PIs"))
229     {
230
231       valid = readLineAsTokens(is, tokens);
232       assert(valid);
233     }
234
235     return valid;
236   }
237
238   bool VerilogParser::read_primary_input(string& primaryInput)
239   {
240
241     primaryInput = "";
242
243     vector<string> tokens;
244     bool valid = readLineAsTokens(is, tokens);
245
246     assert(valid);
247     assert(tokens.size() == 2);
248
249     if (valid && tokens[0] == "input")
250     {
251       primaryInput = tokens[1];
252
253     }
254     else
255     {
256       assert(tokens[0] == "Start" && tokens[1] == "POs");
257       return false;
258     }
259
260     return valid;
```

```
261 }
262
263 bool VerilogParser::read_primary_output(string&
        primaryOutput)
264 {
265
266   primaryOutput = "";
267
268   vector<string> tokens;
269   bool valid = readLineAsTokens(is, tokens);
270
271   assert(valid);
272   assert(tokens.size() == 2);
273
274   if (valid && tokens[0] == "output")
275   {
276     primaryOutput = tokens[1];
277
278   }
279   else
280   {
281     assert(tokens[0] == "Start" && tokens[1] == "wires");
282     return false;
283   }
284
285   return valid;
286 }
287
288 bool VerilogParser::read_wire(string& wire)
289 {
290
291   wire = "";
292
293   vector<string> tokens;
294   bool valid = readLineAsTokens(is, tokens);
295
296   assert(valid);
297   assert(tokens.size() == 2);
298
299   if (valid && tokens[0] == "wire")
300   {
301     wire = tokens[1];
302
303   }
304   else
305   {
306     assert(tokens[0] == "Start" && (tokens[1] == "cells" ||
          tokens[1] == "assigns"));
307     return false;
308   }
309
310   return valid;
```

```
311 }
312
313 bool VerilogParser::read_assign(pair<string, string> &
        assignment)
314 {
315   vector<string> tokens;
316   bool valid = readLineAsTokens(is, tokens);
317
318   assert(valid);
319   assert(tokens.size() == 4 || tokens.size() == 2);
320
321   if (valid && tokens[0] == "assign")
322   {
323     assignment.first = tokens[1];
324     assignment.second = tokens[3];
325
326   }
327   else
328   {
329     assert(tokens[0] == "Start" && tokens[1] == "cells");
330     return false;
331   }
332
333   return valid;
334 }
335 bool VerilogParser::read_cell_inst(string& cellType, string&
        cellInstName, vector<std::pair<string, string>>&
        pinNetPairs)
336 {
337
338   cellType = "";
339   cellInstName = "";
340   pinNetPairs.clear();
341
342   vector<string> tokens;
343   bool valid = readLineAsTokens(is, tokens);
344
345   assert(valid);
346
347   if (tokens.size() == 1)
348   {
349     assert(tokens[0] == "endmodule");
350     return false;
351   }
352
353   assert(tokens.size() >= 4);
354   // We should have cellType, instName, and at least one pin
        -net pair
355
356   cellType = tokens[0];
357   cellInstName = tokens[1];
358
```

```
359       for (size_t i = 2; i < tokens.size() - 1; i += 2)
360     {
361
362       assert(tokens[i][0] == '.');
363       // pin names start with '.'
364       string pinName = tokens[i].substr(1); // skip the first
              character of tokens[i]
365
366       pinNetPairs.push_back(std::make_pair(pinName, tokens[i +
              1]));
367     }
368
369
370
371     return valid;
372   }
373
374
375
376   // LIBERTY PARSER
377   // No need to parse the 3D LUTs, because they will be
        ignored
378   void LibertyParser::_skip_lut_3D () {
379
380     std::vector<string> tokens ;
381
382     bool valid = readLineAsTokens (is, tokens) ;
383     assert (valid) ;
384     assert (tokens[0] == "index_1") ;
385     assert (tokens.size() >= 2) ;
386     int size1 = tokens.size() - 1 ;
387
388     valid = readLineAsTokens (is, tokens) ;
389     assert (valid) ;
390     assert (tokens[0] == "index_2") ;
391     assert (tokens.size() >= 2) ;
392     int size2 = tokens.size() - 1 ;
393
394     valid = readLineAsTokens (is, tokens) ;
395     assert (valid) ;
396     assert (tokens[0] == "index_3") ;
397     assert (tokens.size() >= 2) ;
398     int size3 = tokens.size() - 1 ;
399
400     valid = readLineAsTokens (is, tokens) ;
401     assert (valid) ;
402     assert (tokens.size() == 1 && tokens[0] == "values") ;
403
404     for (size_t i=0; i < size1; ++i) {
405       for (size_t j=0; j < size2; ++j) {
406
407         valid = readLineAsTokens (is, tokens) ;
```

```
408       assert (valid) ;
409       assert (tokens.size() == size3) ;
410     }
411   }
412
413 }
414
415 void LibertyParser::_begin_read_lut (LibertyLookupTable& lut
        ) {
416
417   std::vector<string> tokens ;
418   bool valid = readLineAsTokens (is, tokens) ;
419
420   assert (valid) ;
421   assert (tokens[0] == "index_1") ;
422   assert (tokens.size() >= 2) ;
423
424   int size1 = tokens.size()-1 ;
425   lut.loadIndices.resize(size1) ;
426   for (size_t i=0; i < tokens.size()-1; ++i) {
427
428     lut.loadIndices[i] = std::atof(tokens[i+1].c_str()) ;
429   }
430
431   valid = readLineAsTokens (is, tokens) ;
432
433   assert (valid) ;
434   assert (tokens[0] == "index_2") ;
435   assert (tokens.size() >= 2) ;
436
437   int size2 = tokens.size()-1 ;
438   lut.transitionIndices.resize(size2) ;
439   for (size_t i=0; i < tokens.size()-1; ++i) {
440
441     lut.transitionIndices[i] = std::atof(tokens[i+1].c_str()
          ) ;
442   }
443
444   valid = readLineAsTokens (is, tokens) ;
445   assert (valid) ;
446   assert (tokens.size() == 1 && tokens[0] == "values") ;
447
448   lut.tableVals.resize(size1) ;
449   for (size_t i=0 ; i < lut.loadIndices.size(); ++i) {
450     valid = readLineAsTokens (is, tokens) ;
451     assert (valid) ;
452     assert (tokens.size() == lut.transitionIndices.size()) ;
453
454     lut.tableVals[i].resize(size2) ;
455     for (size_t j=0; j < lut.transitionIndices.size(); ++j)
          {
456       lut.tableVals[i][j] = std::atof(tokens[j].c_str()) ;
```

```
457
458       }
459     }
460
461
462   }
463
464   void LibertyParser::_begin_read_timing_info (string toPin,
          LibertyTimingInfo& timing) {
465
466     timing.toPin = toPin ;
467
468     bool finishedReading = false ;
469
470     std::vector<string> tokens ;
471     while (!finishedReading) {
472
473       bool valid = readLineAsTokens (is, tokens) ;
474       assert (valid) ;
475       assert (tokens.size() >= 1) ;
476
477       if (tokens[0] == "cell_fall") {
478         _begin_read_lut (timing.fallDelay) ;
479
480       } else if (tokens[0] == "cell_rise") {
481         _begin_read_lut (timing.riseDelay) ;
482
483       } else if (tokens[0] == "fall_transition") {
484         _begin_read_lut (timing.fallTransition) ;
485
486       } else if (tokens[0] == "rise_transition") {
487         _begin_read_lut (timing.riseTransition) ;
488
489       } else if (tokens[0] == "fall_constraint") {
490
491         _skip_lut_3D() ; // will ignore fall constraints
492
493       } else if (tokens[0] == "rise_constraint") {
494
495         _skip_lut_3D() ; // will ignore rise constraints
496
497       } else if (tokens[0] == "timing_sense") {
498         timing.timingSense = tokens[1] ;
499
500       } else if (tokens[0] == "related_pin") {
501
502         assert (tokens.size() == 2) ;
503         timing.fromPin = tokens[1] ;
504
505       } else if (tokens[0] == "End") {
506
507         assert (tokens.size() == 2) ;
```

```
508       assert (tokens[1] == "timing") ;
509       finishedReading = true ;
510
511     } else if (tokens[0] == "double") {
512       // ignore data
513
514     } else if (tokens[0] == "related_output_pin") {
515       // ignore data
516
517     } else if (tokens[0] == "timing_type") {
518       // ignore
519     } else {
520
521       cout << "Error: Unknown keyword: " << tokens[0] <<
              endl ;
522       assert (false) ; // unknown keyword
523     }
524
525   }
526
527
528 }
529
530
531 void LibertyParser::_begin_read_pin_info (string pinName,
      LibertyCellInfo& cell, LibertyPinInfo& pin) {
532
533   pin.name = pinName ;
534   pin.isClock = false ;
535   pin.maxCapacitance = std::numeric_limits<double>::max() ;
536
537   bool finishedReading = false ;
538
539   std::vector<string> tokens ;
540   while (!finishedReading) {
541
542     bool valid = readLineAsTokens (is, tokens) ;
543     assert (valid) ;
544     assert (tokens.size() >= 1) ;
545
546     if (tokens[0] == "direction") {
547
548       assert (tokens.size() == 2) ;
549       if (tokens[1] == "input")
550         pin.isInput = true ;
551       else if (tokens[1] == "output")
552         pin.isInput = false ;
553       else
554         assert (false) ; // undefined direction
555
556     } else if (tokens[0] == "capacitance") {
557
```

```
558         assert (tokens.size() == 2) ;
559         pin.capacitance = std::atof(tokens[1].c_str()) ;
560 
561       } else if (tokens[0] == "max_capacitance") {
562 
563         assert (tokens.size() == 2) ;
564         pin.maxCapacitance = std::atof(tokens[1].c_str()) ;
565 
566 
567       } else if (tokens[0] == "timing") {
568 
569         cell.timingArcs.push_back(LibertyTimingInfo()) ; //
              add an empty TimingInfo object
570         _begin_read_timing_info (pinName, cell.timingArcs.back
              ()) ; // pass the empty object to the function to
              be filled
571 
572       } else if (tokens[0] == "clock") {
573 
574         pin.isClock = true ;
575 
576       } else if (tokens[0] == "End") {
577 
578         assert (tokens.size() == 2) ;
579         assert (tokens[1] == "pin") ;
580         finishedReading = true ;
581 
582       } else if (tokens[0] == "function") {
583 
584         // ignore data
585 
586       } else if (tokens[0] == "min_capacitance") {
587 
588         // ignore data
589 
590       } else if (tokens[0] == "nextstate_type") {
591 
592         // ignore data
593 
594       } else {
595         cout << "Error: Unknown keyword: " << tokens[0] <<
              endl ;
596         assert (false) ; // unknown keyword
597 
598       }
599 
600     }
601 
602 
603 }
604 
605 void LibertyParser::_begin_read_cell_info (string cellName,
```

```
    LibertyCellInfo& cell) {

  cell.name = cellName ;
  cell.isSequential = false ;
  cell.dontTouch = false ;

  bool finishedReading = false ;

  std::vector<string> tokens ;
  while (!finishedReading) {

    bool valid = readLineAsTokens (is, tokens) ;
    assert (valid) ;
    assert (tokens.size() >= 1) ;

    if (tokens[0] == "cell_leakage_power") {

      assert (tokens.size() == 2) ;
      cell.leakagePower = std::atof(tokens[1].c_str()) ;

    } else if (tokens[0] == "cell_footprint") {

      assert (tokens.size() == 2) ;
      cell.footprint = tokens[1] ;

    } else if (tokens[0] == "area") {

      assert (tokens.size() == 2) ;
      cell.area = std::atof(tokens[1].c_str()) ;

    } else if (tokens[0] == "clocked_on") {

      cell.isSequential = true ;

    } else if (tokens[0] == "dont_touch") {

      cell.dontTouch = true ;

    } else if (tokens[0] == "pin") {

      assert (tokens.size() == 2) ;

      cell.pins.push_back(LibertyPinInfo()) ; // add empty
          PinInfo object
      _begin_read_pin_info (tokens[1], cell, cell.pins.back
          ()) ; // pass the new PinInfo object to be filled

    } else if (tokens[0] == "End") {

      assert (tokens.size() == 3) ;
      assert (tokens[1] == "cell") ;
      assert (tokens[2] == cellName) ;
```

```
655         finishedReading = true ;
656
657       } else if (tokens[0] == "cell_footprint") {
658
659         // ignore data
660
661       } else if (tokens[0] == "ff") {
662
663         // ignore data
664
665       } else if (tokens[0] == "next_state") {
666
667         // ignore data
668
669       } else if (tokens[0] == "dont_use") {
670
671         // ignore data
672
673       } else {
674
675         cout << "Error: Unknown keyword: " << tokens[0] <<
              endl ;
676         assert (false) ; // unknown keyword
677       }
678     }
679
680 }
681
682
683 // Read the default max_transition defined for the library.
684 // Return value indicates if the last read was successful or
       not.
685 // This function must be called in the beginning before any
      read_cell_info function call.
686 bool LibertyParser::read_default_max_transition (double&
      maxTransition) {
687
688   maxTransition = 0.0 ;
689   vector<string> tokens ;
690
691   bool valid = readLineAsTokens (is, tokens) ;
692
693   while (valid) {
694
695     if (tokens.size() == 2 && tokens[0] == "
          default_max_transition") {
696       maxTransition = std::atof(tokens[1].c_str()) ;
697       return true ;
698     }
699
700     valid = readLineAsTokens (is, tokens) ;
701   }
```

```
702
703     return false ;
704   }
705
706
707
708   // Read the next standard cell definition.
709   // Return value indicates if the last read was successful or
          not.
710   bool LibertyParser::read_cell_info (LibertyCellInfo& cell) {
711
712     vector<string> tokens ;
713     bool valid = readLineAsTokens (is, tokens) ;
714
715
716     while (valid) {
717
718       if (tokens.size() == 2 && tokens[0] == "cell") {
719         _begin_read_cell_info (tokens[1], cell) ;
720
721         return true ;
722       }
723
724       valid = readLineAsTokens (is, tokens) ;
725     }
726
727     return false ;
728   }
729
730   ostream& operator<< (ostream& os, LibertyLookupTable& lut) {
731
732     if (lut.loadIndices.empty() && lut.transitionIndices.empty
            () && lut.tableVals.empty())
733       return os ;
734
735       // We should have either all empty or none empty.
736     assert (!lut.loadIndices.empty() && !lut.transitionIndices
          .empty() && !lut.tableVals.empty()) ;
737
738     assert (lut.tableVals.size() == lut.loadIndices.size()) ;
739     assert (lut.tableVals[0].size() == lut.transitionIndices.
          size()) ;
740
741     os << "\t" ;
742     for (size_t i=0; i < lut.transitionIndices.size(); ++i) {
743       os << lut.transitionIndices[i] << "\t" ;
744     }
745     os << endl ;
746
747
748     for (size_t i=0; i < lut.loadIndices.size(); ++i) {
749       os << lut.loadIndices[i] << "\t" ;
```

```
750
751     for (size_t j=0; j < lut.transitionIndices.size(); ++j)
752       os << lut.tableVals[i][j] << "\t" ;
753
754     os << endl ;
755
756   }
757
758   return os ;
759 }
760
761
762 ostream& operator<< (ostream& os, LibertyTimingInfo& timing)
        {
763
764   os << "Timing info from " << timing.fromPin << " to " <<
        timing.toPin << ": " << endl ;
765   os << "Timing sense: " << timing.timingSense << endl ;
766
767   os << "Fall delay LUT: " << endl ;
768   os << timing.fallDelay ;
769
770   os << "Rise delay LUT: " << endl ;
771   os << timing.riseDelay ;
772
773   os << "Fall transition LUT: " << endl ;
774   os << timing.fallTransition ;
775
776   os << "Rise transition LUT: " << endl ;
777   os << timing.riseTransition ;
778
779   return os ;
780 }
781
782
783 ostream& operator<< (ostream& os, LibertyPinInfo& pin) {
784
785   os << "Pin " << pin.name << ":" << endl ;
786   os << "capacitance: " << pin.capacitance << endl ;
787   os << "maxCapacitance: " << pin.maxCapacitance << endl ;
788   os << "isInput? " << (pin.isInput ? "true" : "false") <<
        endl ;
789   os << "isClock? " << (pin.isClock ? "true" : "false") <<
        endl ;
790   os << "End pin" << endl ;
791
792   return os ;
793 }
794
795
796 ostream& operator<< (ostream& os, LibertyCellInfo& cell) {
797
```

```
798     os << "Library cell " << cell.name << ": " << endl ;
799
800     os << "Footprint: " << cell.footprint << endl ;
801     os << "Leakage power: " << cell.leakagePower << endl ;
802     os << "Area: " << cell.area << endl ;
803     os << "Sequential? " << (cell.isSequential ? "yes" : "no")
            << endl ;
804     os << "Dont-touch? " << (cell.dontTouch ? "yes" : "no") <<
            endl ;
805
806     os << "Cell has " << cell.pins.size() << " pins: " << endl
            ;
807     for (size_t i=0; i < cell.pins.size(); ++i) {
808       os << cell.pins[i] << endl ;
809     }
810
811     os << "Cell has " << cell.timingArcs.size() << " timing
            arcs: " << endl ;
812     for (size_t i=0; i < cell.timingArcs.size(); ++i) {
813       os << cell.timingArcs[i] << endl ;
814     }
815
816     os << "End of cell " << cell.name << endl << endl ;
817
818     return os ;
819   }
820
821   const LibertyLibrary LibertyParser::readFile(const string
          filename)
822   {
823     is.open(filename.c_str(), fstream::in);
824
825     double maxTransition = 0.0f;
826     bool valid = read_default_max_transition(maxTransition) ;
827
828     LibertyLibrary lib(maxTransition);
829
830     assert (valid) ;
831   //    cout << "The default max transition defined is " <<
          maxTransition << endl ;
832
833     int readCnt = 0 ;
834     do {
835       LibertyCellInfo cell ;
836       valid = read_cell_info (cell) ;
837
838       if (valid) {
839         ++readCnt ;
840
841         // cout << cell << endl ;
842         lib.addCellInfo(cell);
843       }
```

```
844
845     } while (valid) ;
846
847     // cout << "Read " << readCnt << " number of library
            cells" << endl ;
848
849     is.close();
850     return lib;
851   }
852
853
854   // SPEF ISPD 2013
855   // The return value indicates whether the *CONN section has
        been read or not
856   bool SpefParserISPD2013::read_connections(SpefNetISPD2013 &
        net)
857   {
858     bool terminateEarly = false;
859
860     vector<string> tokens;
861     bool valid = readLineAsTokens(is, tokens, true /*include
          special chars*/);
862
863     // Skip the lines that are not "*CONN"
864     while (valid && !(tokens.size() == 2 && tokens[0] == "*"
          && tokens[1] == "CONN"))
865     {
866
867       // The following if condition checks for nets without
            any connections
868       // This is needed for clock nets.
869       if (tokens.size() == 2 && tokens[0] == "*" && tokens[1]
            == "END")
870       {
871         terminateEarly = true;
872         break;
873       }
874
875       valid = readLineAsTokens(is, tokens, true /*include
            special chars*/);
876     }
877
878     assert(valid); // end of file not expected here
879
880     if (terminateEarly)
881       return false;
882
883     while (valid)
884     {
885       valid = readLineAsTokens(is, tokens, true /*include
            special chars*/);
886
```

```
887       if (tokens.size() == 2 && tokens[0] == "*" && tokens[1]
              == "CAP")
888         break; // the beginning of the next section
889
890       // Line format: "*nodeType nodeName direction"
891       // Note that nodeName can be either a single token or 3
              tokens
892
893       assert(tokens.size() == 4 || tokens.size() == 6);
894       assert(tokens[0] == "*");
895
896       int tokenIndex = 1;
897
898       assert(tokens[tokenIndex].size() == 1); // should be a
              single character
899       const char nodeType = tokens[tokenIndex++][0];
900       assert(nodeType == 'P' || nodeType == 'I');
901       const std::string nodeNameN1 = tokens[tokenIndex++];
902       if (tokens[tokenIndex] == ":")
903       {
904         ++tokenIndex; // skip the current token
905         const std::string nodeNameN2 = tokens[tokenIndex++];
906       }
907
908       assert(tokens[tokenIndex].size() == 1); // should be a
              single character
909       const char direction = tokens[tokenIndex++][0];
910       assert(direction == 'I' || direction == 'O');
911     }
912
913     return true;
914   }
915
916   void SpefParserISPD2013::read_capacitances(SpefNetISPD2013 &
          net)
917   {
918
919     vector<string> tokens;
920     bool valid = true;
921     while (valid)
922     {
923
924       valid = readLineAsTokens(is, tokens, true /*include
              special chars*/);
925
926       if (tokens.size() == 2 && tokens[0] == "*" && tokens[1]
              == "RES")
927         break; // the beginning of the next section
928
929       // Line format: "index nodeName cap"
930       // Note that nodeName can be either a single token or 3
              tokens
```

```
931
932       assert(tokens.size() == 3 || tokens.size() == 5);
933
934       int tokenIndex = 1;
935
936       std::string nodeName = tokens[tokenIndex++];
937       if (tokens[tokenIndex] == ":")
938       {
939         ++tokenIndex; // skip the current token
940         nodeName += ":" + tokens[tokenIndex++];
941       }
942
943       const double value = std::atof(tokens[tokenIndex++].
              c_str());
944       net.addCapacitor(nodeName, value);
945       assert(value >= 0);
946     }
947   }
948
949   void SpefParserISPD2013::read_resistances(SpefNetISPD2013 &
          net)
950   {
951     vector<string> tokens;
952     bool valid = true;
953       double total_resistance = 0.0f;
954     while (valid)
955     {
956       valid = readLineAsTokens(is, tokens, true /*include
              special chars*/);
957       if (tokens.size() == 2 && tokens[0] == "*" && tokens[1]
              == "END")
958         break; // end for this net
959       // Line format: "index fromNodeName toNodeName res"
960       // Note that each nodeName can be either a single token
              or 3 tokens
961       assert(tokens.size() >= 4 && tokens.size() <= 8);
962       int tokenIndex = 1;
963       std::string fromNodeName = tokens[tokenIndex++];
964       if (tokens[tokenIndex] == ":")
965       {
966         ++tokenIndex; // skip the current token
967         fromNodeName += ":" + tokens[tokenIndex++];
968       }
969       std::string toNodeName = tokens[tokenIndex++];
970       if (tokens[tokenIndex] == ":")
971       {
972         ++tokenIndex; // skip the current token
973         toNodeName += ":" + tokens[tokenIndex++];
974       }
975       const double value = std::atof(tokens[tokenIndex++].
              c_str());
976       assert(value >= 0);
```

```
977       net.addResistor(fromNodeName, toNodeName, value);
978           total_resistance += value;
979     }
980       net.total_resistance = total_resistance;
981 
982 }
983 
984 // Read the spef data for the next net.
985 // Return value indicates if the last read was successful or
        not.
986 bool SpefParserISPD2013::read_net_data(SpefNetISPD2013&
        spefNet)
987 {
988   vector<string> tokens;
989 
990   bool valid = readLineAsTokens(is, tokens, true /*include
        special chars*/);
991 
992   // Read until a valid D_NET line is found
993   while (valid)
994   {
995     if (tokens.size() == 4 && tokens[0] == "*" && tokens[1]
          == "D_NET")
996     {
997           // for(size_t i = 0; i < tokens.size(); i++)
998       //  cout << tokens[i] << " ";
999       // cout << endl;
1000      spefNet.netName = tokens[2];
1001      spefNet.netLumpedCap = std::atof(tokens[3].c_str());
1002 
1003 
1004      bool readConns = read_connections(spefNet);
1005      if (readConns)
1006      {
1007        read_capacitances(spefNet);
1008        read_resistances(spefNet);
1009      }
1010 
1011      return true;
1012    }
1013 
1014    valid = readLineAsTokens(is, tokens, true /*include
          special chars*/);
1015  }
1016 
1017  return false; // a valid net was not read
1018 }
1019 
1020 const Parasitics2013 SpefParserISPD2013::readFile(const
        string filename)
1021 {
1022 //  cout << "SPEF model is ISPD2013" << endl;
```

```
1023    is.open(filename.c_str(), fstream::in);
1024    Parasitics2013 parasitics;
1025    SpefNetISPD2013 spefNet;
1026    bool valid = read_net_data(spefNet);
1027
1028    int readCnt = 0;
1029    while (valid)
1030    {
1031      ++readCnt;
1032      parasitics[spefNet.netName] = spefNet;
1033      spefNet = SpefNetISPD2013();
1034      valid = read_net_data(spefNet);
1035    }
1036
1037  //  cout << "Read " << readCnt << " nets in the spef file."
          << endl;
1038    is.close();
1039    return parasitics;
1040  }
1041
1042  bool SpefParserISPD2012::read_net_cap(string & net, double &
          cap)
1043  {
1044     net = "" ;
1045     cap = 0.0 ;
1046
1047     vector<string> tokens ;
1048     bool valid = readLineAsTokens (is, tokens) ;
1049
1050      // Read until a valid D_NET line is found
1051     while (valid) {
1052       if (tokens.size() == 3 && tokens[0] == "D_NET") {
1053
1054         net = tokens[1] ;
1055         cap = std::atof(tokens[2].c_str()) ;
1056         return true ;
1057       }
1058
1059       valid = readLineAsTokens (is, tokens) ;
1060     }
1061
1062     return false ;
1063  }
1064
1065  const Parasitics2012 SpefParserISPD2012::readFile(const
          string filename)
1066  {
1067  //  cout << "SPEF model is ISPD2012" << endl;
1068    is.open(filename.c_str(), fstream::in);
1069    Parasitics2012 parasitics;
1070
1071    string net;
```

```
1072    double cap;
1073
1074    bool valid = read_net_cap(net, cap);
1075
1076    while (valid)
1077    {
1078      SpefNetISPD2012 spefNet;
1079      spefNet.netName = net;
1080      spefNet.netLumpedCap = cap;
1081      parasitics[net] = spefNet;
1082      //     cout << "Lumped cap of net " << net << " is " <<
              cap << endl;
1083      valid = read_net_cap(net, cap);
1084    }
1085
1086    is.close();
1087    return parasitics;
1088  }
1089
1090
1091  // SDC PARSER
1092
1093  // Read clock definition
1094  // Return value indicates if the last read was successful or
       not.
1095  bool SDCParser::read_clock(string& clockName, string&
       clockPort, double& period)
1096  {
1097
1098    clockName = "";
1099    clockPort = "";
1100    period = 0.0;
1101
1102    vector<string> tokens;
1103    bool valid = readLineAsTokens(is, tokens);
1104
1105    while (valid)
1106    {
1107
1108      if (tokens.size() == 7 && tokens[0] == "create_clock" &&
            tokens[1] == "-name")
1109      {
1110
1111        clockName = tokens[2];
1112
1113        assert(tokens[3] == "-period");
1114        period = std::atof(tokens[4].c_str());
1115
1116        assert(tokens[5] == "get_ports");
1117        clockPort = tokens[6];
1118        break;
1119      }
```

```
1120
1121      valid = readLineAsTokens(is, tokens);
1122    }
1123
1124    // Skip the next comment line to prepare for the next
          stage
1125    bool valid2 = readLineAsTokens(is, tokens);
1126    assert(valid2);
1127    assert(tokens.size() == 2);
1128    assert(tokens[0] == "input" && tokens[1] == "delays");
1129
1130    return valid;
1131  }
1132
1133  // Read input delay
1134  // Return value indicates if the last read was successful or
       not.
1135  bool SDCParser::read_input_delay(string& portName, double&
       delay)
1136  {
1137
1138    portName = "";
1139    delay = 0.0;
1140
1141    vector<string> tokens;
1142    bool valid = readLineAsTokens(is, tokens);
1143
1144    assert(valid);
1145    assert(tokens.size() >= 2);
1146
1147    if (valid && tokens[0] == "set_input_delay")
1148    {
1149      assert(tokens.size() == 6);
1150
1151      delay = std::atof(tokens[1].c_str());
1152
1153      assert(tokens[2] == "get_ports");
1154
1155      portName = tokens[3];
1156
1157      assert(tokens[4] == "-clock");
1158
1159    }
1160    else
1161    {
1162
1163      assert(tokens.size() == 2);
1164      assert(tokens[0] == "input" && tokens[1] == "drivers");
1165
1166      return false;
1167
1168    }
```

```
1169
1170    return valid;
1171  }
1172
1173  // Read output delay
1174  // Return value indicates if the last read was successful or
          not.
1175  bool SDCParser::read_output_delay(string& portName, double&
          delay)
1176  {
1177
1178    portName = "";
1179    delay = 0.0;
1180
1181    vector<string> tokens;
1182    bool valid = readLineAsTokens(is, tokens);
1183
1184    assert(valid);
1185    assert(tokens.size() >= 2);
1186
1187    if (valid && tokens[0] == "set_output_delay")
1188    {
1189      assert(tokens.size() == 6);
1190
1191      delay = std::atof(tokens[1].c_str());
1192
1193      assert(tokens[2] == "get_ports");
1194
1195      portName = tokens[3];
1196
1197      assert(tokens[4] == "-clock");
1198
1199    }
1200    else
1201    {
1202
1203      assert(tokens.size() == 2);
1204      assert(tokens[0] == "output" && tokens[1] == "loads");
1205
1206      return false;
1207
1208    }
1209
1210    return valid;
1211  }
1212
1213  // Read driver info for the input port
1214  // Return value indicates if the last read was successful or
          not.
1215  bool SDCParser::read_driver_info(string& inPortName, string&
          driverSize, string& driverPin, double&
          inputTransitionFall, double& inputTransitionRise)
```

```
1216 {
1217
1218   inPortName = "";
1219   driverSize = "";
1220   driverPin = "";
1221   inputTransitionFall = 0.0;
1222   inputTransitionRise = 0.0;
1223
1224   vector<string> tokens;
1225   bool valid = readLineAsTokens(is, tokens);
1226
1227   assert(valid);
1228   assert(tokens.size() >= 2);
1229
1230   if (valid && tokens[0] == "set_driving_cell")
1231   {
1232     assert(tokens.size() == 11);
1233     assert(tokens[1] == "-lib_cell");
1234
1235     driverSize = tokens[2];
1236
1237     assert(tokens[3] == "-pin");
1238     driverPin = tokens[4];
1239
1240     assert(tokens[5] == "get_ports");
1241     inPortName = tokens[6];
1242
1243     assert(tokens[7] == "-input_transition_fall");
1244     inputTransitionFall = std::atof(tokens[8].c_str());
1245
1246     assert(tokens[9] == "-input_transition_rise");
1247     inputTransitionRise = std::atof(tokens[10].c_str());
1248
1249   }
1250   else
1251   {
1252
1253     assert(tokens.size() == 2);
1254     assert(tokens[0] == "output" && tokens[1] == "delays");
1255
1256     return false;
1257   }
1258
1259   return valid;
1260 }
1261
1262 // Read output load
1263 // Return value indicates if the last read was successful or
       not.
1264 bool SDCParser::read_output_load(string& outPortName, double
      & load)
1265 {
```

```
1266
1267    outPortName = "";
1268    load = 0.0;
1269
1270    vector<string> tokens;
1271    bool valid = readLineAsTokens(is, tokens);
1272
1273    if (valid && tokens[0] == "set_load")
1274    {
1275      assert(tokens.size() == 5);
1276
1277      assert(tokens[1] == "-pin_load");
1278      load = std::atof(tokens[2].c_str());
1279
1280      assert(tokens[3] == "get_ports");
1281      outPortName = tokens[4];
1282
1283    }
1284    else
1285    {
1286
1287      assert(!valid);
1288      return false;
1289    }
1290
1291    return valid;
1292  }
1293
1294  const Design_Constraints SDCParser::readFile(const string
       filename)
1295  {
1296    is.open(filename.c_str(), fstream::in);
1297    string clockName;
1298    string clockPort;
1299    double period;
1300    bool valid = read_clock(clockName, clockPort, period);
1301
1302    assert(valid);
1303    // cout << "Clock " << clockName << " connected to port "
         << clockPort << " has period " << period << endl;
1304
1305    Design_Constraints constraints;
1306    constraints.clock(period);
1307
1308    do
1309    {
1310      string portName;
1311      double delay;
1312
1313      valid = read_input_delay(portName, delay);
1314
1315      if (valid)
```

```
1316      {
1317              // cout << "Input port " << portName << " has
                      delay " << delay << endl;
1318        constraints.input_delay(portName, Transitions<double>(
              delay,delay));
1319      }
1320
1321    }
1322    while (valid);
1323
1324    do
1325    {
1326      string portName;
1327      string driverSize;
1328      string driverPin;
1329      double inputTransitionFall;
1330      double inputTransitionRise;
1331
1332      valid = read_driver_info(portName, driverSize, driverPin
            , inputTransitionFall, inputTransitionRise);
1333
1334      if (valid)
1335      {
1336              // cout << "Input port " << portName << " is
                      assumed to be connected to the " <<
                      driverPin << " pin of lib cell " <<
                      driverSize << endl;
1337              // cout << "This virtual driver is assumed to
                      have input transitions: " <<
                      inputTransitionFall << " (fall) and " <<
                      inputTransitionRise << " (rise)" << endl;
1338
1339        constraints.driving_cell(portName, driverSize);
1340        constraints.input_transition(portName, Transitions<
              double>(inputTransitionRise, inputTransitionFall))
              ;
1341      }
1342
1343    }
1344    while (valid);
1345
1346    do
1347    {
1348      string portName;
1349      double delay;
1350
1351      valid = read_output_delay(portName, delay);
1352
1353      if (valid)
1354      {
1355              // cout << "Output port " << portName << " has
                      delay " << delay << endl;
```

```
1356                constraints.output_delay(portName, Transitions<
                        double>(delay,delay));
1357        }
1358
1359      }
1360      while (valid);
1361
1362      do
1363      {
1364        string portName;
1365        double load;
1366
1367        valid = read_output_load(portName, load);
1368
1369        if (valid)
1370        {
1371                // cout << "Output port " << portName << " has
                        load " << load << endl;
1372          constraints.output_load(portName, load);
1373        }
1374
1375      }
1376      while (valid);
1377      is.close();
1378      return constraints;
1379  }
1380
1381
1382  const Prime_Time_Output_Parser::Prime_Time_Output
        Prime_Time_Output_Parser::parse_prime_time_output_file(
        const string filename)
1383  {
1384    Prime_Time_Output output;
1385    vector<string> tokens;
1386    is.open(filename.c_str(), istream::in);
1387    bool valid = readLineAsTokens(is, tokens, true);
1388    while( valid )
1389    {
1390      if(tokens.front() != "#" && !tokens.empty())
1391      {
1392        if(tokens.size() == 9)
1393        {
1394          output._pins.push_back(Pin_Timing());
1395          output._pins.back().pin_name = tokens.at(0) + ":" +
                tokens.at(2);
1396          output._pins.back().slack = Transitions<double>(atof
                (tokens.at(3).c_str()), atof(tokens.at(4).c_str
                ()));
1397          output._pins.back().slew = Transitions<double>(atof(
                tokens.at(5).c_str()), atof(tokens.at(6).c_str()
                ));
1398          output._pins.back().arrival_time = Transitions<
```

```
                double>(atof(tokens.at(7).c_str()), atof(tokens.
                at(8).c_str()));
1399            } else if(tokens.size() == 5 || tokens.size() ==
                    7)
1400        {
1401          output._ports.push_back(Port_Timing());
1402          output._ports.back().port_name = tokens.at(0);
1403          output._ports.back().slack = Transitions<double>(
                atof(tokens.at(1).c_str()), atof(tokens.at(2).
                c_str()));
1404          output._ports.back().slew = Transitions<double>(atof
                (tokens.at(3).c_str()), atof(tokens.at(4).c_str
                ()));
1405                if(tokens.size() == 7)
1406                    output._ports.back().arrival_window =
                        Transitions<double>(atof(tokens.at
                        (5).c_str()), atof(tokens.at(6).
                        c_str()));
1407        }
1408            else
1409          assert(false);
1410      }
1411      valid = readLineAsTokens(is, tokens, true);
1412    }
1413
1414    is.close();
1415    return output;
1416  }
```

Listing B.26 – parser.cpp

```
1  #include "include/slew_degradation_experiment.h"
2
3  const double Slew_Degradation_Experiment::EPSILON = 0.01;
4
5  bool Slew_Degradation_Experiment::nearly_equals(const
       Transitions<double> a, const Transitions<double> b)
6  {
7      return abs(a - b).getMax() <=
           Slew_Degradation_Experiment::EPSILON;
8  }
9
10 void Slew_Degradation_Experiment::run(Timing_Analysis::
       Timing_Analysis &ta)
11 {
12 //     ta.print_info();
13
14     queue<pair<int, int>> degradation;
15     queue<pair<int, int>> nDegradation;
16
17     const double degradation_threshold = 0.2f;
```

```
18
19      int yes = 0;
20      int no  = 0;
21      for(int i = 0; i < ta.timing_points_size(); i++)
22      {
23          const Timing_Analysis::Timing_Point & tp = ta.
                timing_point(i);
24          if(tp.is_output_pin())
25          {
26              const Timing_Analysis::Timing_Net & net = tp.net
                    ();
27              for(int j = 0; j < net.fanouts_size(); j++)
28              {
29                  const Timing_Analysis::Timing_Point & fanout
                        = net.to(j);
30                  if(!nearly_equals(tp.slew(), fanout.slew()))
31                  {
32                      degradation.push(make_pair(i, j));
33                      yes++;
34                  }
35                  else
36                  {
37                      nDegradation.push(make_pair(i, j));
38                      no++;
39                  }
40              }
41
42          }
43      }
44
45      cout << yes << " degradations; " << no << " non-
            degradations" << endl;
46
47      cout << "slew degradation: " << endl;
48      int i = 0;
49      while(!degradation.empty())
50      {
51          pair<int, int> pts = degradation.front();
52          degradation.pop();
53
54          const Timing_Analysis::Timing_Point & tp1 = ta.
                timing_point(pts.first);
55          const Timing_Analysis::Timing_Point & tp2 = tp1.net
                ().to(pts.second);
56
57          cout << "degradation[" << i++ << "] = " << tp1.name
                () << " -> " << tp2.name() << endl;
58          cout << " slew degradation = " << tp2.slew() - tp1.
                slew() << endl;
59          cout << "        wire delay = " << tp2.arrival_time()
                 - tp1.arrival_time() << endl;
60          cout << "       driver slew = " << tp1.slew() << endl
```

```
            ;
61          cout << "       fanout slew = " << tp2.slew() << endl
                ;
62
63
64          if((tp2.arrival_time() - tp1.arrival_time()).getMax
                () > tp1.slew().getMax() * degradation_threshold
                )
65          {
66              cout << "       wire delay (" << tp2.arrival_time
                    () - tp1.arrival_time() << ") is MORE than
                    20% of max driver slew (" << tp1.slew() << "
                    ) (20% = " << tp1.slew() * 0.2f << ")" <<
                    endl;
67          }
68          else
69          {
70              cout << "       wire delay (" << tp2.arrival_time
                    () - tp1.arrival_time() << ") is LESS than
                    20% of max driver slew (" << tp1.slew() << "
                    ) (20% = " << tp1.slew() * 0.2f << ")" <<
                    endl;
71          }
72          cout << endl;
73      }
74
75
76      cout << "####" << endl;
77      i=0;
78      while(!nDegradation.empty())
79      {
80          pair<int, int> pts = nDegradation.front();
81          nDegradation.pop();
82
83          const Timing_Analysis::Timing_Point & tp1 = ta.
                timing_point(pts.first);
84          const Timing_Analysis::Timing_Point & tp2 = tp1.net
                ().to(pts.second);
85
86          cout << "non-degradation[" << i++ << "] = " << tp1.
                name() << " -> " << tp2.name() << endl;
87          cout << " slew degradation = " << tp2.slew() - tp1.
                slew() << endl;
88          cout << "       wire delay = " << tp2.arrival_time()
                 - tp1.arrival_time() << endl;
89          cout << "       driver slew = " << tp1.slew() << endl
                ;
90          cout << "       fanout slew = " << tp2.slew() << endl
                ;
91
92          if((tp2.arrival_time() - tp1.arrival_time()).getMax
                () > tp1.slew().getMax() * degradation_threshold
```

```
                )
93          {
94              cout << "       wire delay (" << tp2.arrival_time
                    () - tp1.arrival_time() << ") is MORE than
                    20% of max driver slew (" << tp1.slew() << "
                    ) (20% = " << tp1.slew() * 0.2f << ")" <<
                    endl;
95          }
96          else
97          {
98              cout << "       wire delay (" << tp2.arrival_time
                    () - tp1.arrival_time() << ") is LESS than
                    20% of max driver slew (" << tp1.slew() << "
                    ) (20% = " << tp1.slew() * 0.2f << ")" <<
                    endl;
99          }
100         cout << endl;
101     }
102 }
```

Listing B.27 – slew_degradation_experiment.cpp

```
1  #include "include/spef_net.h"
2
3  void SpefNetISPD2013::set(string name, double
       lumpedCapacitance, double total_resistance)
4  {
5    this->netName = name;
6      this->netLumpedCap = lumpedCapacitance;
7      this->total_resistance = total_resistance;
8  }
9  int SpefNetISPD2013::addNode(const string & name)
10 {
11   if (nodeMap.find(name) != nodeMap.end())
12     return nodeMap.at(name);
13   const int nodeIndex = nodes.size();
14   nodes.push_back(Node(nodeIndex, name));
15   nodeMap[name] = nodeIndex;
16   return nodeIndex;
17 }
18
19 void SpefNetISPD2013::addResistor(const string & node1,
       const string & node2, const double & value)
20 {
21   const int node1Index = addNode(node1);
22   const int node2Index = addNode(node2);
23   const int newResistorIndex = resistors.size();
24   resistors.push_back(Resistor(node1Index, node2Index, value
         ));
25 //     if(nodes.at(node1Index).capacitance == 0.0f)
26 //     {
```

```
27 //         nodes.at(node1Index).capacitance = nodes.at(
       node2Index).capacitance / 2;
28 //         nodes.at(node2Index).capacitance = nodes.at(
       node2Index).capacitance / 2;
29 //     } else if(nodes.at(node2Index).capacitance == 0.0f)
30 //     {
31 //         nodes.at(node2Index).capacitance = nodes.at(
       node1Index).capacitance / 2;
32 //         nodes.at(node1Index).capacitance = nodes.at(
       node1Index).capacitance / 2;
33 //     }
34 //
35 //     if(nodes.at(node1Index).capacitance == 0.0f)
36 //     {
37 //         nodes.at(node1Index).capacitance = nodes.at(
       node2Index).capacitance * 2;
38 ////         nodes.at(node2Index).capacitance = nodes.at(
       node2Index).capacitance / 2;
39 //         this->netLumpedCap += nodes.at(node1Index).
       capacitance;
40 //     } else if(nodes.at(node2Index).capacitance == 0.0f)
41 //     {
42 //         nodes.at(node2Index).capacitance = nodes.at(
       node1Index).capacitance * 2;
43 //         this->netLumpedCap += nodes.at(node2Index).
       capacitance;
44 //     }
45 //
46 //     if(nodes.at(node1Index).capacitance == 0.0f)
47 //     {
48 //         nodes.at(node1Index).capacitance = nodes.at(
       node2Index).capacitance;
49 //         this->netLumpedCap += nodes.at(node1Index).
       capacitance;
50 //     } else if(nodes.at(node2Index).capacitance == 0.0f)
51 //     {
52 //         nodes.at(node2Index).capacitance = nodes.at(
       node1Index).capacitance;
53 //         this->netLumpedCap += nodes.at(node1Index).
       capacitance;
54 //     }
55   nodes.at(node1Index).resistors.push_back(newResistorIndex)
       ;
56   nodes.at(node2Index).resistors.push_back(newResistorIndex)
       ;
57 }
58
59 void SpefNetISPD2013::addCapacitor(const string & node,
       const double & value)
60 {
61   const int nodeIndex = addNode(node);
62   capacitors.push_back(Capacitor(nodeIndex, value));
```

```
63    nodes.at(nodeIndex).capacitance += value;
64  }
65
66  ostream& operator<<(ostream & out, const SpefNetISPD2013 &
        descriptor)
67  {
68      for (size_t i = 0; i < descriptor.nodes.size(); i++)
69    {
70      const SpefNetISPD2013::Node & node = descriptor.nodes.at
            (i);
71      out << "node " << node.nodeIndex << " " << node.name <<
            " {" << endl;
72      out << "  capacitance " << node.capacitance << endl;
73      out << "  resistors {" << endl;
74          for (size_t j = 0; j < node.resistors.size(); j++)
75      {
76        out << "    " << j << " " << descriptor.resistors[node
              .resistors[j]].value;
77        out << endl;
78      }
79      out << "  }" << endl;
80      out << "}" << endl;
81      out << endl;
82    }
83    return out;
84  }
85
86  int SpefNetISPD2013::getNodeIndex(const string & name) const
87  {
88    for (unsigned i = 0; i < nodes.size(); i++)
89    {
90      if (nodes.at(i).name == name)
91        return i;
92    }
93    return -1;
94  }
```

Listing B.28 – spef_net.cpp

```
1   #include "include/timer.h"
2
3   const double Timer::MICRO = 1000000.0f;
4   const string Timer::micro = "us";
5   const double Timer::MILI = 1000.0f;
6   const string Timer::mili = "ms";
7   const double Timer::SECOND = 1.0f;
8   const string Timer::second = "s";
9
10  Timer::Timer()
11  {
12
```

```
13 }
14
15 Timer::~Timer()
16 {
17
18 }
19
20 void Timer::start()
21 {
22     gettimeofday(&_start_time, NULL);
23 }
24
25 void Timer::end()
26 {
27     gettimeofday(&_stop_time, NULL);
28     float time = (float) (_stop_time.tv_sec - _start_time.
           tv_sec);
29     time += (float) (_stop_time.tv_usec - _start_time.
           tv_usec) / MICRO;
30     _execution_time.set(time, string("s"));
31 }
32
33 const Timer::Result & Timer::value(const double
       time_definition)
34 {
35     assert(time_definition == MICRO || time_definition ==
           MILI || time_definition == SECOND);
36
37     string def;
38     if(time_definition == MICRO)
39         def = Timer::micro;
40     else if (time_definition == MILI)
41         def = Timer::mili;
42     else if (time_definition == SECOND)
43         def = Timer::second;
44
45     _execution_time.set(_execution_time._time *
           time_definition, def);
46     return _execution_time;
47 }
```

Listing B.29 – timer.cpp

```
1 #include "include/timer_interface.h"
2
3
4 // Function Definitions
5 // Get timer status
6 TimerInterface::Status TimerInterface::getTimerStatus(const
```

```
    std::string &contest_root, const std::string &benchmark)
     {
7   const std::string dir = contest_root + "/" + benchmark;
8
9   // Get files from directory to see if there are any timer
        status
10  std::vector<std::string> files;
11  if (!getFiles(files,dir)) {
12    return TIMER_INTERFACEERROR;
13  }
14
15  // Get command and perform action
16  std::string cmd = getTimerStatusString(files);
17  if ("__SIZERCMD_TIMERERROR_" == cmd) {
18    return TIMER_FINISHED_ERROR;
19  } else if ("__SIZERCMD_TIMERDONE_" == cmd) {
20    return TIMER_FINISHED_SUCCESS;
21  } else if ("__TCMD_RUNTIMER_" == cmd) {
22    return TIMER_BUSY;
23  } else {
24    return TIMER_NOT_STARTED;
25  }
26 }
27
28 // Write sizes and run timing analysis in blocking mode
29 TimerInterface::Status TimerInterface::
    runTimingAnalysisBlocking(const std::vector<std::pair<
    std::string, std::string> > &sizes, const std::string &
    contest_root, const std::string &benchmark, const
    unsigned pollingTime) {
30 //cout << "runTimingAnalysisBlocking: " << contest_root <<
    "/" << benchmark << endl;
31   if (!writeSizesForTimer(sizes, contest_root, benchmark)) {
32     std::cout << "-E- runTimingAnalysisBlocking: problem
         writing sizes" << std::endl;
33     return TIMER_INTERFACEERROR;
34   }
35   return runTimingAnalysisBlocking(contest_root,benchmark,
       pollingTime);
36 }
37
38 // Start timing analysis in non-blocking mode
39 TimerInterface::Status TimerInterface::
    startTimingAnalysisNonBlocking(const std::vector<std::
    pair<std::string, std::string> > &sizes, const std::
    string &contest_root, const std::string &benchmark) {
40   if (!writeSizesForTimer(sizes, contest_root, benchmark)) {
41     std::cout << "-E- startTimingAnalysisNonBlocking:
         problem writing sizes" << std::endl;
42     return TIMER_INTERFACEERROR;
43   }
44   return startTimingAnalysisNonBlocking(contest_root,
```

```
        benchmark);
}

// Wait for given number of seconds (useful function if you
    want to wait before checking timer status after calling
    startTimingAnalysisNonBlocking)
void TimerInterface::wait(int seconds) {
  std::ostringstream ostr;
  ostr << seconds;
  system(("sleep " + ostr.str()).c_str());
}

// PRIVATE SECTION
// ------------------------------------------------------------

// Get timer status (helper function for isTimerDone)
std::string TimerInterface::getTimerStatusString(const std::
    vector<std::string> &files) {
  std::string cmd = "";
  for (unsigned i=0; i<files.size(); ++i) {
    if ("__SIZERCMD_TIMERERROR_" == files[i] ||
        "__SIZERCMD_TIMERDONE_" == files[i] ||
        "__TCMD_RUNTIMER_" == files[i]) {
      if (cmd != "") {
        std::cout << "-Error- getTimerStatusString: multiple
            status found" << std::endl;
        for (unsigned j=0; j<files.size(); ++j) {
          if ("__SIZERCMD_TIMERERROR_" == files[j] ||
              "__SIZERCMD_TIMERDONE_" == files[j] ||
              "__TCMD_RUNTIMER_" == files[j]) {
            std::cout << "   Status File: " << files[j] <<
                std::endl;
          }
        }
        assert(false);
      }
      cmd = files[i];
    }
  }
  return cmd;
}

// Checks if a file exists (returns true if it does, false
    otherwise)
bool TimerInterface::doesFileExist(const std::string &file)
    {
  std::ifstream infile(file.c_str());
  if (!infile) {
    return false;
  }
  infile.close();
  return true;
```

```
87  }
88
89  // Get a list of files from given directory (used by
        getTimerStatus to check if timer is done)
90  bool TimerInterface::getFiles(std::vector<std::string> &
        files, const std::string &dir) {
91    files.clear();
92    DIR *d = opendir(dir.c_str());
93    if (NULL == d) {
94      std::cout << "-E- getFiles: could not list files in
            directory '" << dir << "' to get timer status" <<
            std::endl;
95      return false;
96    }
97    files.clear();
98    dirent *f = readdir(d);
99    while (NULL != f) {
100     files.push_back(f->d_name);
101     f = readdir(d);
102   }
103   closedir(d);
104   //for (unsigned i=0; i<files.size(); ++i) {
105   //  std::cout << files[i] << std::endl;
106   //}
107   return true;
108 }
109
110 // Remove a file from the given directory (helper function
        used by startTimingAnalysis)
111 bool TimerInterface::removeFile(const std::string &dir,
        const std::string &file) {
112   if (file != "__SIZERCMD_TIMERERROR_" &&
113       file != "__SIZERCMD_TIMERDONE_") {
114     std::cout << "-E-: removeFile: You can't use this to
            remove any files other than __SIZERCMD_TIMERERROR_
            and __SIZERCMD_TIMERDONE_" << std::endl;
115     assert(false);
116   }
117   std::string filename = dir + "/" + file;
118   if (doesFileExist(filename) && remove(filename.c_str())) {
119     perror(("-E- removeFile: could not remove '" + filename
            + "'").c_str());
120     return false;
121   }
122   return true;
123 }
124
125 // Write sizes to a file for timing analysis call
126 bool TimerInterface::writeSizesForTimer(const std::vector<
        std::pair<std::string, std::string> > &sizes, const std
        ::string &contest_root, const std::string &benchmark) {
127   const std::string filename = contest_root + "/" +
```

```
      benchmark + "/" + benchmark + ".int.sizes";
128   std::ofstream ofile(filename.c_str());
129   if (!ofile) {
130     std::cout << "-E- could not open file '" << filename <<
          "' for output" << std::endl;
131     return false;
132   }
133   for (unsigned i=0; i<sizes.size(); ++i)
134     ofile << sizes[i].first << " " << sizes[i].second << std
          ::endl;
135   ofile.close();
136   return true;
137 }
138
139 // Start timing analysis (does not wait for it to finish)
140 bool TimerInterface::startTimingAnalysis(const std::string &
      contest_root, const std::string &benchmark) {
141   const std::string filename = contest_root + "/" +
        benchmark + "/__TCMD_RUNTIMER_";
142   if (doesFileExist(filename)) {
143     return false;
144   }
145
146   // Delete previous status files
147   removeFile(contest_root+"/"+benchmark,"
        __SIZERCMD_TIMERERROR_");
148   removeFile(contest_root+"/"+benchmark,"
        __SIZERCMD_TIMERDONE_");
149
150   // Instruct the timer to start timing analysis
151   std::ofstream ofile(filename.c_str());
152   if (!ofile) {
153     std::cout << "-E- startTimingAnalysis: problem
          instructing timer to run timing, could not write out
          '" << filename << "'" << std::endl;
154     assert(false);
155   }
156   ofile.close();
157
158   return true;
159 }
160
161 // Run timing analysis in blocking mode
162 TimerInterface::Status TimerInterface::
      runTimingAnalysisBlocking(const std::string &
      contest_root, const std::string &benchmark, const
      unsigned pollingTime) {
163   // Write a file out to instruct timer loop to run timing
        analysis
164   if (!startTimingAnalysis(contest_root,benchmark))
165     return TIMER_INTERFACEERROR;
166
```

```
167     // Wait till timer is done
168     Status status = getTimerStatus(contest_root,benchmark);
169     while (status == TIMER_BUSY) {
170       status = getTimerStatus(contest_root, benchmark);
171       wait(pollingTime);
172     }
173     return status;
174   }
175
176   // Start timing analysis in non-blocking mode
177   TimerInterface::Status TimerInterface::
          startTimingAnalysisNonBlocking(const std::string &
          contest_root, const std::string &benchmark) {
178     if (!startTimingAnalysis(contest_root,benchmark))
179       return TIMER_INTERFACEERROR;
180     return TIMER_BUSY;
181   }
182
183   // END PRIVATE SECTION
          ------------------------------------------------------------
184
185   // Function to pretty-print Status
186   std::ostream& operator<<(std::ostream &o, const
          TimerInterface::Status &s) {
187     switch (s) {
188     case TimerInterface::TIMER_NOT_STARTED:
189       o << "TIMER_NOT_STARTED"; break;
190     case TimerInterface::TIMER_BUSY:
191       o << "TIMER_BUSY"; break;
192     case TimerInterface::TIMER_FINISHED_SUCCESS:
193       o << "TIMER_FINSIHED_SUCCESS"; break;
194     case TimerInterface::TIMER_FINISHED_ERROR:
195       o << "TIMER_FINISHED_ERROR"; break;
196     case TimerInterface::TIMER_INTERFACEERROR:
197       o << "TIMER_INTERFACEERROR"; break;
198     default:
199       break;
200     }
201     return o;
202   }
```

Listing B.30 – timer_interface.cpp

```
1   #include "include/timing_analysis.h"
2
3   namespace Timing_Analysis
4   {
5   /*
6
7       TIMING ANALYSIS
```

```
8
9   */
10
11
12  Timing_Analysis::Timing_Analysis(const Circuit_Netlist &
        netlist, const LibertyLibrary * lib, const Parasitics *
        parasitics, const Design_Constraints * sdc) :
        _gate_index_to_timing_point_index(netlist.getGatesSize()
        ), _verilog(netlist.verilog()), _sizes(_verilog.size()),
         _library(lib), _parasitics(parasitics), _first_PO_index
        (-1), _total_violating_POs(0)
13  {
14
15      _target_delay = Transitions<double>(sdc->clock(), sdc->
            clock());
16      _max_transition = Transitions<double>(lib->
            getMaxTransition(), lib->getMaxTransition());
17
18      int timing_points, timing_arcs;
19
20      // FORNECER PELA NETLIST (Mais eficiente)
21      number_of_timing_points_and_timing_arcs(timing_points,
            timing_arcs, netlist, lib);
22
23      _points.reserve(timing_points);
24      _arcs.reserve(timing_arcs);
25      _nets.reserve(netlist.getNetsSize());
26      _options.reserve(netlist.getGatesSize());
27
28      // Creating Timing Points & Timing Arcs
29      for(size_t i = 0; i < netlist.getGatesSize(); i++)
30      {
31          const Circuit_Netlist::Logic_Gate & gate = netlist.
                getGateT(i);
32          // cout << "Gate " << gate.name << " (" << i << ")
                created!" << endl;
33          const pair<int, int> cell_index = lib->getCellIndex(
                gate.cellType);
34          const LibertyCellInfo & cell_info = lib->getCellInfo
                (cell_index.first, cell_index.second);
35          const bool is_PI = gate.inputDriver;
36          const bool is_PO = !cell_index.first || ( gate.
                sequential && !is_PI );
37
38          assert(i < _gate_index_to_timing_point_index.size())
                ;
39          _gate_index_to_timing_point_index[i] =
                create_timing_points(i, gate, cell_index,
                cell_info);
40          create_timing_arcs(_gate_index_to_timing_point_index
                .at(i), is_PI, is_PO);
41
```

```
42          _options.push_back(Option(cell_index.first ,
                cell_index.second));
43          if(gate.sequential || gate.inputDriver || cell_index
                .first == 0 /* PRIMARY OUTPUT*/)
44              _options.back()._dont_touch = true;
45          // cout << "   gate option: (" << cellIndex.first <<
                ", " << cellIndex.second << ")" << endl;
46          if(_options.size() != i+1)
47              cout << "error! options.size() ("<< _options.
                    size() << ") != " << i+1 << endl;
48          assert(_options.size() == i + 1);
49      }
50
51      _dirty.resize(_options.size());
52
53      // SETTING DESIGN CONSTRAINTS
54      // INPUT DELAY && INPUT SLEW
55      for(size_t i = 0; i < _points.size(); i++)
56      {
57          const Timing_Point & tp = _points.at(i);
58
59          if(tp.is_PI())
60          {
61              const LibertyCellInfo & opt = liberty_cell_info(
                    tp.gate_number());
62              if(opt.isSequential)
63                  continue;
64              const string PI_name = tp.name();
65              Timing_Point & inPin = _points.at(i - 1);
66              inPin.arrival_time(sdc->input_delay(PI_name));
67              inPin.slew(sdc->input_transition(PI_name));
68
69              // cout << "setting input delay " << inPin.
                    arrivalTime << " to pin " << inPin.name <<
                    endl;
70              // cout << "setting input slew " << inPin.slew
                    << " to pin " << inPin.name << endl;
71          }
72
73
74          if(tp.is_PO())
75          {
76              if(_first_PO_index == -1)
77                  _first_PO_index = i;
78              const LibertyCellInfo & opt = liberty_cell_info(
                    tp.gate_number());
79              if(opt.isSequential)
80                  continue;
81
82              _PO_loads.insert(make_pair(i, sdc->output_load(
                    tp.name())));
83              // cout << "setting poLoads["<<i<<"] = " <<
```

```
                    poLoads.at(i) << endl;
84          }
85      }
86
87      // Create Timing_Nets
88      for(size_t i = 0; i < netlist.getNetsSize(); i++)
89      {
90          const Circuit_Netlist::Net & net = netlist.getNetT(i
                );
91          const int driverTopologicIndex = netlist.
                getTopologicIndex(net.sourceNode); // -1 if
                driver is the 'source'
92
93          if(driverTopologicIndex == -1)
94          {
95              const string dummy_net_name = net.name;
96              _nets.push_back(Timing_Net(dummy_net_name, 0, 0)
                    );
97
98          }
99          else
100         {
101             const int timing_point_index =
                    _gate_index_to_timing_point_index.at(
                    driverTopologicIndex).second;
102             Timing_Point & driver_timing_point = _points.at(
                    timing_point_index);
103
104             // Just a test {
105             WireDelayModel * delay_model = 0;
106             const LibertyCellInfo & opt = liberty_cell_info(
                    driver_timing_point.gate_number());
107
108             if(parasitics->find(net.name) != parasitics->end
                    ())
109             {
110                 delay_model = new
                        Ceff_Elmore_Slew_Degradation_PURI(
                        parasitics->at(net.name),
                        driver_timing_point.name(), opt.
                        timingArcs.size());
111
112                 //                  delay_model = new
                        Lumped_Elmore_Slew_Degradation(
                        parasitics->at(net.name),
                        driver_timing_point.name(), opt.
                        timingArcs.size());
113                 //                  delay_model = new
                        Lumped_Elmore_No_Slew_Degradation(
                        parasitics->at(net.name),
                        driver_timing_point.name(), opt.
                        timingArcs.size());
```

```
114                 //                  delay_model = new
                        Ceff_Elmore_No_Slew_Degradation(
                        parasitics->at(net.name),
                        driver_timing_point.name(), opt.
                        timingArcs.size());
115
116                 //                  delay_model = new
                        LumpedCapacitanceWireDelayModel(
                        parasitics->at(net.name),
                        driver_timing_point.name());
117
118                 //                  delay_model = new
                        Ceff_Elmore_Slew_Degradation(parasitics
                        ->at(net.name), driver_timing_point.name
                        (), opt.timingArcs.size());
119                 //                  delay_model = new
                        Ceff_Without_Wire_Delay_And_Slew_Degradation
                        (parasitics->at(net.name),
                        driver_timing_point.name(), opt.
                        timingArcs.size());
120                 //                  delay_model = new
                        Reduced_Pi(parasitics->at(net.name),
                        driver_timing_point.name(), opt.
                        timingArcs.size());
121             }
122             // }
123
124             _nets.push_back(Timing_Net(net.name, &
                    driver_timing_point, delay_model));
125
126             // if(delayModel)
127             //  cout << "created net " << nets.back().
                    netName << " with lumped capacitance " <<
                    driverTp->load() << endl;
128             driver_timing_point.net(&(_nets.back()));
129         }
130
131     }
132
133     // Now, setting the fanin edges of nodes
134     for(size_t i = 0; i < netlist.getGatesSize(); i++)
135     {
136         const Circuit_Netlist::Logic_Gate & gate = netlist.
                getGateT(i);
137
138         // cout << "gate " << gate.name << endl;
139         const pair<size_t, size_t> timing_point_index =
                _gate_index_to_timing_point_index.at(i);
140         for(size_t j = 0; j < gate.inNets.size(); j++)
141         {
142             const int in_net_topologic_index = netlist.
                    get_net_topologic_index(gate.inNets.at(j));
```

```
143                 Timing_Net * in_net = &_nets.at(
                        in_net_topologic_index);
144
145                 const int in_timing_point_index =
                        timing_point_index.first + j;
146                 Timing_Point & fanout_timing_point = _points.at(
                        in_timing_point_index);
147
148                 in_net->add_fanout(&fanout_timing_point);
149                 if(in_net->_wire_delay_model)
150                 {
151
152                     const double pin_cap = pin_capacitance(
                            in_timing_point_index);
153                     in_net->_wire_delay_model->
                            setFanoutPinCapacitance(
                            fanout_timing_point.name(), pin_cap);
154                 }
155                 fanout_timing_point.net(in_net);
156             }
157         }
158         _interpolator = new LinearLibertyLookupTableInterpolator
                ();
159
160
161         for(int i = 0; i < _points.size(); i++)
162         {
163             Timing_Point & tp = _points.at(i);
164             if(tp.is_PI_input() || tp.is_reg_input())
165             {
166                 tp.logic_level(0);
167             }
168             else if(tp.is_input_pin() || tp.is_PO())
169             {
170                 tp.logic_level(tp.net().from()->logic_level() +
                        1);
171             } else if(tp.is_output_pin() || tp.is_PI())
172             {
173
174                 int current_index = i-1;
175                 int current_gate = _points.at(current_index).
                        gate_number();
176                 int max_logic_level = 0;
177                 while(current_gate == tp.gate_number())
178                 {
179                     Timing_Point & tp_in = _points.at(
                            current_index);
180                     max_logic_level = max(max_logic_level, tp_in
                            .logic_level());
181                     current_index--;
182                     if(current_index == -1)
183                         break;
```

```
184                     current_gate = _points.at(current_index).
                            gate_number();
185                 }
186                 tp.logic_level(max_logic_level + 1);
187             }
188
189         }
190
191
192         // CHECKING FIRST PO INDEX
193         for(int i = 0; i < _points.size(); i++)
194         {
195             if(i >= _first_PO_index)
196             {
197                 //                  cout << _points.at(i).name()
                        << endl;
198                 assert(_points.at(i).is_PO());
199             }
200             else
201                 assert(!_points.at(i).is_PO());
202         }
203
204         //         cout << "OK " << endl;
205
206 }
207
208 const pair<size_t, size_t> Timing_Analysis::
        create_timing_points(const int i,const Circuit_Netlist::
        Logic_Gate & gate,const pair<int, int> cellIndex, const
        LibertyCellInfo & cellInfo)
209 {
210
211
212         const bool is_primary_input = gate.inputDriver;
213         const bool is_sequential = gate.sequential;
214         const bool is_primary_output = !cellIndex.first || (
            is_sequential && !is_primary_input );
215
216         const size_t firstTimingPointIndex = _points.size();
217         string gateName = gate.name;
218         if( is_sequential && is_primary_input )
219             gateName = gateName.substr(0, gateName.size() -
                string("_PI").size());
220
221         Timing_Point_Type type;
222         string timingPointName;
223
224         // INPUT PINS
225         if(!is_sequential || (is_sequential && is_primary_input
            ) )
226         {
227             for(size_t j = 1; j < cellInfo.pins.size(); j++)
```

```
228             {
229                 if(cellInfo.pins.at(j).name == "d" &&
                        is_sequential)
230                     continue;
231                 timingPointName = gateName + ":" + cellInfo.pins
                        .at(j).name;
232                 if(is_sequential)
233                     type = REGISTER_INPUT;
234                 else if (is_primary_input)
235                     type = PI_INPUT;
236                 else
237                     type = INPUT;
238 
239                 _pin_name_to_timing_point_index.insert(make_pair
                        (timingPointName, _points.size()));
240                 _points.push_back(Timing_Point(timingPointName,
                        i, type));
241             }
242         }
243 
244 
245         // OUTPUT PIN
246         if( is_primary_input )
247         {
248             type = PI;
249             if( is_sequential )
250                 timingPointName = gateName + ":" + cellInfo.pins
                        .front().name;
251             else
252                 timingPointName = gateName;
253         }
254         else if( is_primary_output )
255         {
256             type = PO;
257             if( is_sequential )
258                 timingPointName = gateName + ":" + cellInfo.pins
                        .at(2).name;
259             else
260                 timingPointName = gateName;
261         }
262         else
263         {
264             type = OUTPUT;
265             timingPointName = gateName + ":" + cellInfo.pins.
                    front().name;
266         }
267 
268         _pin_name_to_timing_point_index.insert(make_pair(
                timingPointName, _points.size()));
269         _points.push_back(Timing_Point(timingPointName, i, type)
                );
270         return make_pair(firstTimingPointIndex, _points.size() -
```

```
            1);
271 }
272
273 void Timing_Analysis::
        number_of_timing_points_and_timing_arcs(int &
        numberOfTimingPoints, int & numberOfTiming_Arcs, const
        Circuit_Netlist & netlist,const  LibertyLibrary * lib)
274 {
275     numberOfTimingPoints = 0;
276     numberOfTiming_Arcs = 0;
277     for(size_t i = 0; i < netlist.getGatesSize(); i++)
278     {
279         const Circuit_Netlist::Logic_Gate & gate = netlist.
              getGateT(i);
280         const pair<int, int> cellIndex = lib->getCellIndex(
              gate.cellType);
281         const bool is_PI = gate.inputDriver;
282         const bool is_sequential = gate.sequential;
283         const bool is_PO = !cellIndex.first || (
              is_sequential && !is_PI);
284         numberOfTimingPoints += 1;
285         if( !is_PO )
286         {
287             numberOfTimingPoints += gate.inNets.size();
288             numberOfTiming_Arcs += (is_sequential ? 1 : gate
                  .inNets.size());
289         }
290     }
291     //    cout << "number of timing points " <<
          numberOfTimingPoints << " netlist number " <<
          netlist.timing_points() << endl;
292     //    assert(numberOfTimingPoints == netlist.
          timing_points());
293     //    cout << "number of timing arcs " <<
          numberOfTiming_Arcs << " netlist number " << netlist
          .timing_arcs() << endl;
294     //    assert(numberOfTiming_Arcs == netlist.timing_arcs
          ());
295
296 }
297
298 void Timing_Analysis::create_timing_arcs(const pair<size_t,
      size_t> tpIndexes, const bool is_pi, const bool is_po )
299 {
300     if( is_pi )
301     {
302         // cout << "  PI timing arc " << points.at(tpIndexes
              .first).name << " -> " << points.at(tpIndexes.
              second).name << endl;
303         _arcs.push_back(Timing_Arc(&_points.at(tpIndexes.
              first), &_points.at(tpIndexes.second), 0,
              _points.at(tpIndexes.first).gate_number()));
```

```
304             _points.at(tpIndexes.first).arc(&_arcs.back());
305             assert(_arcs.back().from() == &_points.at(tpIndexes.
                    first) && &_arcs.back().to() == &_points.at(
                    tpIndexes.second));
306             // cout << "  PI timing arc OK" << endl;
307         }
308         else
309         {
310             if( !is_po )
311             {
312                 for(size_t j = tpIndexes.first; j < tpIndexes.
                        second; j++)
313                 {
314                     // cout << "  timing arc " << points.at(j).
                            name << " -> " << points.at(tpIndexes.
                            second).name << endl;
315                     _arcs.push_back(Timing_Arc(&_points.at(j), &
                            _points.at(tpIndexes.second), j-
                            tpIndexes.first, _points.at(j).
                            gate_number()));
316                     _points.at(j).arc(&_arcs.back());
317                     assert(_arcs.back().from() == &_points.at(j)
                             && &_arcs.back().to() == &_points.at(
                            tpIndexes.second));
318                     // cout << "  timing arc OK" << endl;
319                 }
320             }
321         }
322     }
323
324     Timing_Analysis::~Timing_Analysis()
325     {
326
327     }
328
329     void Timing_Analysis::initialize_timing_data()
330     {
331         std::fill(_dirty.begin(), _dirty.end(), false);
332         _slew_violations = numeric_limits<Transitions<double>
                >::zero();
333         _capacitance_violations = numeric_limits<Transitions<
                double> >::zero();
334         _total_negative_slack = numeric_limits<Transitions<
                double> >::zero();
335         _critical_path = numeric_limits<Transitions<double> >::
                zero();
336         _total_violating_POs = 0;
337     }
338
339     void Timing_Analysis::full_timing_analysis()
340     {
341         initialize_timing_data();
```

```
342
343         for(size_t i = 0; i < _points.size(); i++)
344             update_timing(i);
345
346         for(size_t i = 0; i < _points.size(); i++)
347         {
348             size_t n = _points.size() - i - 1;
349             update_slacks(n);
350         }
351
352 }
353
354 typedef priority_queue<Timing_Point*, vector<Timing_Point*>,
        ita_comparator> ita_priority_queue;
355
356 /*
357 Timing_Analysis::incremental_timing_analysis
358
359 Enfilera o timing point de saÃda da porta que estÃ¡
        trocando de opÃ§Ã£o;
360 Enfilera os timing points de saÃda das portas que sÃ£o
        fanins da porta que estÃ¡ trocando de opÃ§Ã£o;
361
362 Processamento da fila (atÃ© esvaziar):
363     Atualiza informaÃ§Ã£o de timing;
364     Se alterou arrival time ou slew:
365         Enfilera os timing points de saÃda das portas
                fanouts da porta que estÃ¡ sendo processada.
366
367 Atualiza Slack nas saÃdas primÃ¡rias
368
369 */
370
371 void Timing_Analysis::incremental_timing_analysis(int
        gate_number, int new_option)
372 {
373     int tp_index = _gate_index_to_timing_point_index.at(
            gate_number).second;
374
375     Timing_Point * timing_point = &_points.at(tp_index);
376     assert(timing_point->is_output_pin());
377     assert(timing_point->gate_number() == gate_number);
378     ita_priority_queue pq;
379
380     _total_negative_slack = numeric_limits<Transitions<
            double> >::zero();
381     _critical_path = numeric_limits<Transitions<double> >::
            zero();
382
383     set<Timing_Point*> inserted;
384     if(inserted.insert(timing_point).second)
385     {
```

```
386             pq.push(timing_point);
387             assert(option(timing_point->gate_number(),
                    new_option));
388         }
389
390         //input nodes must be inserted only when the size
                changes, i.e., when the input capacitance of the
                gate is changed
391         int input_pin_index = tp_index - 1;
392         while(input_pin_index >= 0 && _points.at(input_pin_index
                ).gate_number() == timing_point->gate_number())
393         {
394             Timing_Point * fanin = _points.at(input_pin_index).
                    net().from();
395             if(inserted.insert(fanin).second)
396                 pq.push(fanin);
397             input_pin_index--;
398         }
399
400         while(!pq.empty())
401         {
402             Timing_Point * tp = pq.top();
403
404             assert(tp->is_output_pin() || tp->is_PI());
405
406             pq.pop();
407
408             Transitions<double> slew0 = tp->slew();
409             Transitions<double> arrival_time0 = tp->arrival_time
                    ();
410
411             update_timing_points(tp);
412
413             Transitions<double> slewF = tp->slew();
414             Transitions<double> arrival_timeF = tp->arrival_time
                    ();
415
416             const bool changed = abs(slewF - slew0).getMax() >
                    Traits::STD_THRESHOLD || abs(arrival_timeF -
                    arrival_time0).getMax() > Traits::STD_THRESHOLD;
417             if(changed)
418             {
419                 for(int i = 0; i < tp->net().fanouts_size(); i
                        ++)
420                 {
421                     Timing_Point * input_tp_of_fanout = &tp->net
                            ().to(i);
422                     Timing_Point * output_tp_of_fanout = &tp->
                            net().to(i).arc().to();
423                     if(input_tp_of_fanout->is_input_pin() &&
                            inserted.insert(output_tp_of_fanout).
                            second)
```

```
424                         pq.push(output_tp_of_fanout);
425                 }
426             }
427         }
428
429         for (int i = _points.size() - 1 ; i >= _first_PO_index;
                --i) {
430             update_slacks(i);
431         }
432
433     }
434
435     void Timing_Analysis::update_timing_points(const
            Timing_Point *output_timing_point)
436     {
437
438         unsigned output_timing_point_index = output_timing_point
                - &_points.front();
439         unsigned input_timing_point_index =
                output_timing_point_index - 1;
440
441         _dirty.at(_points.at(output_timing_point_index).
                gate_number()) = false;
442
443
444         while (input_timing_point_index >= 0  &&
                output_timing_point->gate_number() == _points.at(
                input_timing_point_index).gate_number())
445         {
446             this->update_timing(input_timing_point_index);
447             input_timing_point_index--;
448         }
449
450         this->update_timing(output_timing_point_index);
451     }
452
453     bool Timing_Analysis::option(const int gate_index, const int
             option_number)
454     {
455         const LibertyCellInfo & old_cell_info =
                liberty_cell_info(gate_index);
456         const Option & gate_option = _options.at(gate_index);
457         if( !gate_option._dont_touch )
458         {
459             _options.at(gate_index)._option_index =
                    option_number;
460             const LibertyCellInfo & new_cell_info =
                    liberty_cell_info(gate_index);
461
462             // UPDATE FANINS OUTPUT LOADS
463             const pair<size_t, size_t> timing_points =
                    _gate_index_to_timing_point_index.at(gate_index)
```

```
            ;
464         for(size_t timing_point_index = timing_points.first;
                timing_point_index <= timing_points.second;
                timing_point_index++)
465         {
466             Timing_Point & timing_point = _points.at(
                    timing_point_index);
467
468             if(timing_point.is_input_pin())
469             {
470                 // IF SEQUENTIAL, INPUT PIN NUMBER = 2
471                 const int pin_number = timing_point.arc().
                        arc_number() + 1; /* +1 porque o
                        primeiro Ã© o pino de saÃda */
472                 const double old_pin_capacitance =
                        old_cell_info.pins.at(pin_number).
                        capacitance;
473                 const double new_pin_capacitance =
                        new_cell_info.pins.at(pin_number).
                        capacitance;
474                 Timing_Net & in_net = timing_point.net();
475                 if(in_net.wire_delay_model())
476                     in_net.wire_delay_model()->
                            setFanoutPinCapacitance(timing_point
                            .name(), new_pin_capacitance -
                            old_pin_capacitance);
477             }
478         }
479
480         return true;
481     }
482     return false;
483 }
484
485 void Timing_Analysis::set_all_gates_to_max_size()
486 {
487     for(int i = 0; i < _points.size(); i++)
488         option(_points.at(i).gate_number(), 0);
489 }
490
491 void Timing_Analysis::set_all_gates_to_min_size()
492 {
493     for(int i = 0; i < _points.size(); i++)
494     {
495         if(_points.at(i).is_PO() || _points.at(i).is_PI())
496             continue;
497         const int gate_number = _points.at(i).gate_number();
498         const int number_of_options = _library->
                number_of_options(_options.at(gate_number).
                _footprint_index);
499         option(gate_number, number_of_options - 1);
500     }
```

```
501 }
502
503 void Timing_Analysis::update_slacks(const int
        timing_point_index)
504 {
505     Timing_Point & timing_point = _points.at(
            timing_point_index);
506     Timing_Net & net = timing_point.net();
507
508     Transitions<double> required_time = numeric_limits<
            Transitions<double> >::max();
509
510     if( timing_point.is_PI_input() || timing_point.
            is_reg_input() ) // doesn't matter
511         return;
512     if( timing_point.is_input_pin() )
513         required_time = (timing_point.arc().to().
                required_time() - timing_point.arc().delay()).
                getReversed();
514     else if( timing_point.is_PO() )
515         required_time = _target_delay;
516     else if( timing_point.is_output_pin() || timing_point.
            is_PI() )
517     {
518         for(size_t i = 0; i < net.fanouts_size(); i++)
519         {
520             const Transitions<double> interconnect_delay =
                    net.to(i).arrival_time() - timing_point.
                    arrival_time();
521             required_time = min(required_time, net.to(i).
                    required_time() - interconnect_delay);
522         }
523         _slew_violations += max(numeric_limits<Transitions<
                double> >::zero(), timing_point.slew() -
                _max_transition);
524     }
525
526     if(timing_point.update_slack(required_time).getMin() < 0
             && timing_point.is_PO())
527     {
528         _total_violating_POs++;
529         _total_negative_slack -= timing_point.slack();
530     }
531 }
532
533
534 void Timing_Analysis::update_timing(const int
        timing_point_index)
535 {
536     Timing_Point & timing_point = _points.at(
            timing_point_index);
537
```

```
538     if(timing_point.is_input_pin() || timing_point.
            is_PI_input() || timing_point.is_reg_input())
539     {
540         Timing_Point & output_pin = timing_point.arc().to();
541         assert(output_pin.gate_number() == timing_point.
                gate_number());
542         Timing_Arc & timing_arc = timing_point.arc();
543         assert(timing_arc.gate_number() == timing_point.
                gate_number());
544         Timing_Net & output_net = output_pin.net();
545         const bool is_the_first_input_pin_of_a_gate = !
                _dirty.at(output_pin.gate_number());
546         const LibertyCellInfo & cell_info =
                liberty_cell_info(timing_point.gate_number());
547
548         if( is_the_first_input_pin_of_a_gate )
549         {
550             assert(output_pin.gate_number() < _dirty.size())
                    ;
551             _dirty[output_pin.gate_number()] = true;
552             output_pin.clear_timing_info();
553             output_net.wire_delay_model()->clear();
554         }
555
556         const Transitions<double> ceff_by_this_timing_arc =
                output_net.wire_delay_model()->simulate(
                cell_info, timing_arc.arc_number(), timing_point
                .slew(), timing_point.is_PI_input());
557         output_pin.ceff(max(output_pin.ceff(),
                ceff_by_this_timing_arc));
558
559         //              if(_max_ceff.find(output_pin.name())
                == _max_ceff.end())
560         //                  _max_ceff[output_pin.name()] =
                ceff_by_this_timing_arc;
561         //              else
562         //                  _max_ceff[output_pin.name()] = max
                (_max_ceff.at(output_pin.name()),
                ceff_by_this_timing_arc);
563         //              if(_min_ceff.find(output_pin.name())
                == _min_ceff.end())
564         //                  _min_ceff[output_pin.name()] =
                ceff_by_this_timing_arc;
565         //              else
566         //                  _min_ceff[output_pin.name()] = max
                (_min_ceff.at(output_pin.name()),
                ceff_by_this_timing_arc);
567
568         const Transitions<double>
                current_arc_delay_at_output_pin =
                calculate_timing_arc_delay(timing_arc,
                timing_point.slew(), ceff_by_this_timing_arc);
```

```
569             const Transitions<double>
                    current_arc_slew_at_output_pin = output_net.
                    wire_delay_model()->root_slew(timing_arc.
                    arc_number());
570
571             timing_arc.delay(current_arc_delay_at_output_pin);
572             timing_arc.slew(current_arc_slew_at_output_pin);
573
574             // SETTING OUTPUT SLEW AND ARRIVAL TIME
575             const Transitions<double> max_arrival_time = max(
                    output_pin.arrival_time(), timing_point.
                    arrival_time().getReversed() + timing_arc.delay
                    ());
576             const Transitions<double> max_slew = max(output_pin.
                    slew(), timing_arc.slew());
577
578             output_pin.arrival_time(max_arrival_time); //
                    NEGATIVE UNATE
579             output_pin.slew(max_slew);
580
581             //              assert(output_pin.arrival_time().
                    getRise() >= timing_point.arrival_time().getFall
                    () + timing_arc.delay().getRise());
582             //              assert(output_pin.arrival_time().
                    getFall() >= timing_point.arrival_time().getRise
                    () + timing_arc.delay().getFall());
583
584             //              assert(output_pin.slew().getRise() >=
                    timing_arc.slew().getRise());
585             //              assert(output_pin.slew().getFall() >=
                    timing_arc.slew().getFall());
586
587             // SETTING OUTPUT PIN VIOLATIONS
588             if( is_the_first_input_pin_of_a_gate )
589             {
590                 const LibertyCellInfo & cell_info =
                        liberty_cell_info(output_pin.gate_number());
591                 _capacitance_violations += max(numeric_limits<
                        Transitions<double> >::zero(), (
                        ceff_by_this_timing_arc - cell_info.pins.
                        front().maxCapacitance));
592             }
593
594         }
595         else if(timing_point.is_output_pin() || timing_point.
                is_PI())
596         {
597             //              timing_point.ceff(_max_ceff.at(
                    timing_point.name()));
598             Timing_Net & output_net = timing_point.net();
599
600             for(size_t i = 0; i < output_net.fanouts_size(); i
```

```
                ++)
601         {
602             Timing_Point & fanout_timing_point = output_net.
                    to(i);
603
604             int fanout_number = &fanout_timing_point - &
                    _points.front();
605
606             fanout_timing_point.arrival_time(timing_point.
                    arrival_time() + output_net.wire_delay_model
                    ()->delay_at_fanout_node(fanout_timing_point
                    .name()));
607             fanout_timing_point.slew(output_net.
                    wire_delay_model()->slew_at_fanout_node(
                    fanout_timing_point.name()));
608
609
610             //                  assert(fanout_timing_point.
                    arrival_time().getRise() >= timing_point.
                    arrival_time().getRise());
611             //                  assert(fanout_timing_point.
                    arrival_time().getFall() >= timing_point.
                    arrival_time().getFall());
612             //                  assert(fanout_timing_point.
                    slew().getRise() >= timing_point.slew().
                    getRise());
613             //                  assert(fanout_timing_point.
                    slew().getFall() >= timing_point.slew().
                    getFall());
614         }
615     }
616     else if(timing_point.is_PO())
617     {
618         _critical_path = max(_critical_path, timing_point.
                arrival_time());
619     }
620
621 }
622
623 const Transitions<double> Timing_Analysis::
        calculate_timing_arc_delay(const Timing_Arc & timing_arc
        , const Transitions<double> transition, const
        Transitions<double> ceff)
624 {
625     const LibertyCellInfo & cellInfo = liberty_cell_info(
            timing_arc.gate_number());
626     const LibertyLookupTable & fallLUT = cellInfo.timingArcs
            .at(timing_arc.arc_number()).fallDelay;
627     const LibertyLookupTable & riseLUT = cellInfo.timingArcs
            .at(timing_arc.arc_number()).riseDelay;
628     return _interpolator->interpolate(riseLUT, fallLUT, ceff
            , transition, cellInfo.isSequential ? NON_UNATE :
```

```
            NEGATIVE_UNATE, timing_arc.from()->is_PI_input());
629 }
630
631 const LibertyCellInfo &Timing_Analysis::liberty_cell_info(
        const int gate_index) const
632 {
633     return _library->getCellInfo(_options.at(gate_index).
            _footprint_index, _options.at(gate_index).
            _option_index);
634 }
635
636 void Timing_Analysis::print_info()
637 {
638     // for(size_t i = 0; i < nodes.size(); i++)
639     // {
640     //  Node & node = nodes.at(i);
641     //  const bool primaryOutput = !(nodesOptions[i].
            footprintIndex);
642     //  const LibertyCellInfo & cellInfo = library->
            getCellInfo(nodesOptions[i].footprintIndex,
            nodesOptions[i].optionIndex);
643     //  if(primaryOutput || (node.sequential && !node.
            inputDriver)) // PRIMARY OUTPUT
644     //  {
645     //    TimingPoint & o = node.timingPoints.front();
646     //    cout << "PO "<< o.name << " " << o.slack << " " <<
             o.slew << " " << o.arrivalTime << endl;
647     //  }
648     //  else
649     //  {
650     //    TimingPoint & o = node.timingPoints.back();
651     //    if(!node.sequential || (node.sequential && node.
            inputDriver))
652     //    {
653     //      cout << node.name << endl;
654     //      cout << "-- "<< o.name << " net " << o.net->
            netName << " " << o.slack << " " << o.slew << " " <<
             o.arrivalTime << endl;
655     //    }
656     //    if(!node.inputDriver)
657     //    {
658     //      for(size_t j = 0; j < nodes.at(i).timingPoints.
            size() -1 ; j++)
659     //      {
660     //        TimingPoint & tp = nodes.at(i).timingPoints.at
            (j);
661     //        cout <<"-- "<< tp.name << " net " << tp.net->
            netName << tp.slack << " " << tp.slew << " " << tp.
            arrivalTime << endl;
662     //      }
663     //    }
664     //  }
```

```
665     // }
666
667     // printCircuitInfo();
668
669     printf("########################################## CIRCUIT
            GRAPH INFO
            ##############################################\n");
670     printf("| %u Timing Points\n", unsigned(_points.size()))
            ;
671     printf("| %u Timing Arcs\n", unsigned(_arcs.size()));
672     printf("| %u Timing Nets\n", unsigned(_nets.size()));
673     printf("
            ------------------------------------------------------------------------------
            n\n\n");
674
675     printf("########################################## CIRCUIT
            TIMING INFO
            ##############################################\n");
676     printf(">>>> Timing Points Infos (pins)\n");
677     queue<int> ports;
678     queue<int> sequentials;
679
680
681     //         cout << "Ceff" << endl;
682     //         for(size_t i = 0; i < _points.size(); i++)
683     //         {
684     //             if(_points.at(i).is_PI() || _points.at(i).
            is_output_pin())
685     //                 cout << "ceff " << _points.at(i).name
            () << " = " << _points.at(i).ceff() << endl;
686     //         }
687     //         cout << "##" << endl;
688
689     queue<int> pins;
690     for(size_t i = 0 ; i < _points.size(); i++)
691     {
692         const Timing_Point & tp = _points.at(i);
693         const LibertyCellInfo & cellInfo = liberty_cell_info
                (tp.gate_number());
694
695         if( tp.is_PI() && cellInfo.isSequential )
696         {
697             // cout << tp.name << " PI and sequential" <<
                    endl;
698             sequentials.push(i);
699             continue;
700         }
701
702         if (tp.is_input_pin() )
703         {
704
705             pins.push(i);
```

```
706             continue;
707         }
708
709         if( tp.is_output_pin() || (tp.is_PO() && cellInfo.
                isSequential))
710         {
711             printf("%s %f %f %f %f %f %f\n", tp.name().c_str
                    (), tp.slack().getRise(), tp.slack().getFall
                    (), tp.slew().getRise(), tp.slew().getFall()
                    , tp.arrival_time().getRise(), tp.
                    arrival_time().getFall());
712             while( !pins.empty() )
713             {
714                 const Timing_Point & iPin = _points.at(pins.
                        front());
715                 pins.pop();
716                 printf("%s %f %f %f %f %f %f\n", iPin.name()
                        .c_str(), iPin.slack().getRise(), iPin.
                        slack().getFall(), iPin.slew().getRise()
                        , iPin.slew().getFall(), iPin.
                        arrival_time().getRise(), iPin.
                        arrival_time().getFall());
717             }
718         }
719
720
721         if( !cellInfo.isSequential && (tp.is_PI() || tp.
                is_PO()) )
722             ports.push(i);
723
724         if( tp.is_PO() && cellInfo.isSequential )
725         {
726             const int reg = sequentials.front();
727             sequentials.pop();
728             const Timing_Point & regTp = _points.at(reg);
729             printf("%s %f %f %f %f %f %f\n", regTp.name().
                    c_str(), regTp.slack().getRise(), regTp.
                    slack().getFall(), regTp.slew().getRise(),
                    regTp.slew().getFall(), regTp.arrival_time()
                    .getRise(), regTp.arrival_time().getFall());
730         }
731     }
732
733     printf("\n>>>> Timing Points Infos (ports)\n");
734     while(!ports.empty())
735     {
736         const int tp_index = ports.front();
737         ports.pop();
738         const Timing_Point & tp = _points.at(tp_index);
739         printf("%s %f %f %f %f\n", tp.name().c_str(), tp.
                slack().getRise(), tp.slack().getFall(), tp.slew
                ().getRise(), tp.slew().getFall());
```

```
740     }
741
742     printf("
            ------------------------------------------------------------------------
            n\n\n");
743
744 }
745
746 void Timing_Analysis::print_circuit_info()
747 {
748     printf("##############################################
            TIMING INFO
            ##############################################\n");
749     cout << "| Critical Path Values = " << _critical_path <<
            " / " << _target_delay << endl;
750     cout << "| Slew Violations = " << _slew_violations <<
            endl;
751     cout << "| Capacitance Violations = " <<
            _capacitance_violations << endl;
752     cout << "| Total Negative Slack = " <<
            _total_negative_slack << endl;
753     printf("
            ------------------------------------------------------------------------
            n\n\n");
754 }
755
756 void Timing_Analysis::report_timing()
757 {
758     stack<int> current_gate;
759
760
761     bool done = false;
762     int previous_gate = -1;
763
764     int i = 0;
765     int k = 0;
766     do {
767         if(!_points.at(i).is_input_pin())
768         {
769             i++;
770             continue;
771         }
772
773         current_gate.push(_points.at(i).gate_number());
774         if(previous_gate != current_gate.top())
775         {
776             cout << endl;
777             cout << "timing info for cell '"<< current_gate.
                    top() <<"'" << endl;
778             cout << "
                    ------------------------------------------------------------------------
                    " << endl;
```

```
779             }
780             cout << endl;
781             cout << "from_pin\tto_pin\tarc_rise\tarc_fall\tsense
                    " << endl;
782             previous_gate = current_gate.top();
783
784             k = 1;
785             while(_points.at(i+k).gate_number() == previous_gate
                    )
786             {
787                 Timing_Point & timing_point = _points.at(i+k-1);
788                 Timing_Arc & arc = timing_point.arc();
789                 Timing_Point & to_pin = arc.to();
790                 cout << timing_point.name() << "\t" << to_pin.
                        name() << "\t"<<arc.delay().getRise()<<"\t"
                        << arc.delay().getFall() << endl;
791                 k++;
792             }
793             cout << endl;
794
795             k = 1;
796             while(_points.at(i+k).gate_number() == previous_gate
                    )
797             {
798                 cout << "from_pin\tslack_r\tslack_f\tarrival_r\
                        tarrival_f\ttrans_r\ttrans_f" << endl;
799                 Timing_Point & timing_point = _points.at(i+k-1);
800                 cout << timing_point.name() << "\t" <<
                        timing_point.slack().getRise() << "\t"<<
                        timing_point.slack().getFall()<<"\t" <<
                        timing_point.arrival_time().getRise() << "\t
                        " << timing_point.arrival_time().getFall()
                        << "\t" << timing_point.slew().getRise() <<
                        "\t" << timing_point.slew().getFall() <<
                        endl;
801                 cout << endl;
802                 k++;
803             }
804
805
806             cout << "to_pin\tslack_r\tslack_f" << endl;
807             Timing_Point & timing_point = _points.at(i+k);
808             cout << timing_point.name() << "\t" << timing_point.
                    slack().getRise() << "\t"<< timing_point.slack()
                    .getFall()<<"\t"<< endl;
809
810             i += k;
811
812
813         } while(i < _points.size());
814
815 }
```

```
816
817 void Timing_Analysis::print_effective_capacitances()
818 {
819     //        cout << "-- Effective Capacitances" << endl;
820     //        for(vector<Timing_Point>::iterator it =
            _points.begin(); it != _points.end(); it++)
821     //        {
822     //            const Timing_Point & tp = (*it);
823     //            if(_max_ceff.find(tp.name()) != _max_ceff.
            end())
824     //                cout << tp.name() << " " << tp.ceff().
            getRise() << " " << tp.ceff().getFall() << endl;
825     //        }
826     //        cout << "--" << endl;
827 }
828
829 set<int> Timing_Analysis::timing_points_in_longest_path()
830 {
831     int max_path_PO = _first_PO_index;
832
833     for(int i = _first_PO_index + 1; i < _points.size(); i
            ++)
834     {
835         const Timing_Point & tp = _points.at(i);
836         if(tp.logic_level() > _points.at(max_path_PO).
                logic_level())
837             max_path_PO = i;
838     }
839     //    cout << "longest path size = " << _points.at(
            max_path_PO).logic_level() << endl;
840
841     int current = max_path_PO;
842     stack<int> path;
843     set<int> longest_path;
844     while(!_points.at(current).is_PI())
845     {
846         path.push(current);
847         longest_path.insert(current);
848         const Timing_Point & tp = _points.at(current);
849         if(tp.is_PO() || tp.is_input_pin())
850         {
851             const int input_index = tp.net().from() - &
                    _points[0];
852             current = input_index;
853         } else if(tp.is_output_pin())
854         {
855             int input_index = current-1;
856             int max_size_input = input_index;
857             input_index--;
858             while(_points.at(input_index).gate_number() ==
                    tp.gate_number())
859             {
```

```
860                 if(_points.at(input_index).logic_level() >
                        _points.at(max_size_input).logic_level()
                        )
861                     max_size_input = input_index;
862                 input_index--;
863             }
864             current = max_size_input;
865         }
866         else
867             assert(false);
868     }
869     path.push(current);
870     longest_path.insert(current);
871 
872 
873     //     while(!path.empty())
874     //     {
875     //         const int index = path.top();
876     //         path.pop();
877 
878     //         const Timing_Point & tp = _points.at(index);
879     //         cout << tp.name();
880     //         if(tp.is_PI())
881     //             cout << " PI";
882     //         else if (tp.is_PO())
883     //             cout << " PO";
884 
885     //         if(!path.empty())
886     //             cout << " -> ";
887     //         else
888     //             cout << ";";
889     //     }
890     //     cout << endl;
891     return longest_path;
892 }
893 
894 set<int> Timing_Analysis::timing_points_in_critical_path()
895 {
896     int max_path_PO = _first_PO_index;
897 
898     for(int i = _first_PO_index + 1; i < _points.size(); i
            ++)
899     {
900         const Timing_Point & tp = _points.at(i);
901         if(tp.arrival_time().getMax() > _points.at(
                max_path_PO).arrival_time().getMax())
902             max_path_PO = i;
903     }
904     int current = max_path_PO;
905     set<int> critical_path;
906     stack<int> path;
907     while(!_points.at(current).is_PI())
```

```
908     {
909         critical_path.insert(current);
910         path.push(current);
911         const Timing_Point & tp = _points.at(current);
912         if(tp.is_PO() || tp.is_input_pin())
913         {
914             const int input_index = tp.net().from() - &
                    _points[0];
915             current = input_index;
916         } else if(tp.is_output_pin())
917         {
918             int input_index = current-1;
919             int max_size_input = input_index;
920             input_index--;
921             while(_points.at(input_index).gate_number() ==
                    tp.gate_number())
922             {
923                 if(_points.at(input_index).arrival_time().
                        getMax() > _points.at(max_size_input).
                        arrival_time().getMax())
924                     max_size_input = input_index;
925                 input_index--;
926             }
927             current = max_size_input;
928         }
929         else
930             assert(false);
931     }
932     critical_path.insert(current);
933     path.push(current);
934
935
936     //    cout << "critical path (" << _critical_path << ")
            size = " << critical_path.size() << endl;
937     //    while(!path.empty())
938     //    {
939     //        const int index = path.top();
940     //        path.pop();
941
942     //        const Timing_Point & tp = _points.at(index);
943     //        cout << tp.name();
944     //        if(tp.is_PI())
945     //            cout << " PI";
946     //        else if (tp.is_PO())
947     //            cout << " PO";
948
949     //        if(!path.empty())
950     //            cout << " -> ";
951     //        else
952     //            cout << ";";
953     //    }
954     //    cout << endl;
```

```
955     return critical_path;
956 }
957
958 bool Timing_Analysis::has_timing_violations()
959 {
960     return _total_negative_slack.getFall() != 0.0f ||
            _total_negative_slack.getRise() != 0.0f;
961 }
962
963 pair<pair<int, int>, pair<Transitions<double>, Transitions<
        double> > > Timing_Analysis::check_ceffs(double
        precision)
964 {
965     const string ceff_file = Traits::ispd_contest_root + "/"
            + Traits::ispd_contest_benchmark + "/" + Traits::
            ispd_contest_benchmark + ".ceff";
966     fstream in;
967     in.open(ceff_file.c_str(), fstream::in);
968
969     string pin_name;
970     double rise, fall;
971     Transitions<double> ceff;
972
973     int first_point_index = numeric_limits<int>::max();
974     int first_logic_level = numeric_limits<int>::max();
975     Transitions<double> tool_ceff, pt_ceff;
976
977     while(!in.eof())
978     {
979         in >> pin_name;
980         in >> rise;
981         in >> fall;
982         ceff.set(rise, fall);
983
984         size_t slash_position = pin_name.find_first_of('/');
985         if(slash_position != string::npos)
986         {
987             pin_name.replace(slash_position, 1, ":");
988             assert(_pin_name_to_timing_point_index.find(
                    pin_name) != _pin_name_to_timing_point_index
                    .end());
989             const Timing_Point & tp = _points.at(
                    _pin_name_to_timing_point_index.at(pin_name)
                    );
990
991             Transitions<double> ceff_error;
992             ceff_error = abs(tp.ceff() - ceff) / max(abs(tp.
                    ceff()), abs(ceff));
993             //                  if(tp.name() == "g2412_u1:o")
994             //                      cout << "g2412_u1:o ceff "
                     << tp.ceff() << " pt ceff " << ceff << endl
                    ;
```

```
995
996                 if(ceff_error.getMax() >= precision)
997                 {
998                     cout << "pin " << pin_name << " ceff " << tp
                            .ceff() << " pt ceff " << ceff << " CEFF
                             ERROR > " << precision*100 << "% = " <<
                             ceff_error << endl;
999                     if(tp.logic_level() < first_logic_level)
1000                    {
1001                        first_point_index = &tp - &_points.at(0)
                                ;
1002                        cout << "new first logic level " <<  tp.
                                logic_level()  << "(old = " <<
                                first_logic_level << ")" << endl;
1003                        first_logic_level = tp.logic_level();
1004
1005                        tool_ceff = tp.ceff();
1006                        pt_ceff = ceff;
1007                    }
1008                }
1009
1010            }
1011        }
1012        return make_pair(make_pair(first_point_index,
                first_logic_level), make_pair(tool_ceff, pt_ceff));
1013    }
1014
1015
1016    // PRIMETIME CALLING
1017    bool Timing_Analysis::validate_with_prime_time()
1018    {
1019        const string timing_file = Traits::ispd_contest_root + "
                /" + Traits::ispd_contest_benchmark + "/" + Traits::
                ispd_contest_benchmark + ".timing";
1020
1021        get_sizes_vector();
1022        const unsigned pollingTime = 1;
1023
1024        //        cout << "Running timing analysis" << endl;
1025
1026        TimerInterface::Status s = TimerInterface::
                runTimingAnalysisBlocking(_sizes,  Traits::
                ispd_contest_root, Traits::ispd_contest_benchmark,
                pollingTime);
1027        //    if(s != 2)
1028        //        cout << "Timing analysis finished with status:
                     " << s << endl;
1029
1030        return check_timing_file(timing_file);
1031    }
1032
1033    void Timing_Analysis::call_prime_time()
```

```
1034 {
1035     const string timing_file = Traits::ispd_contest_root + "
         /" + Traits::ispd_contest_benchmark + "/" + Traits::
         ispd_contest_benchmark + ".timing";
1036
1037     get_sizes_vector();
1038
1039     initialize_timing_data();
1040     const unsigned pollingTime = 1;
1041
1042     cout << "Running timing analysis" << endl;
1043
1044     TimerInterface::Status s = TimerInterface::
         runTimingAnalysisBlocking(_sizes, Traits::
         ispd_contest_root, Traits::ispd_contest_benchmark,
         pollingTime);
1045     cout << "Timing analysis finished with status: " << s <<
          endl;
1046
1047     cout << "Reading Timing Information" << endl;
1048     Prime_Time_Output_Parser prime_time_parser;
1049     const Prime_Time_Output_Parser::Prime_Time_Output
         prime_time_output = prime_time_parser.
         parse_prime_time_output_file(timing_file);
1050
1051     cout << "Setting Timing Information" << endl;
1052
1053     for(size_t i = 0; i < prime_time_output.pins_size(); i
         ++)
1054     {
1055         const Prime_Time_Output_Parser::Pin_Timing
             pin_timing = prime_time_output.pin(i);
1056         Timing_Point & timing_point = _points.at(
             _pin_name_to_timing_point_index.at(pin_timing.
             pin_name));
1057         timing_point.slack(pin_timing.slack);
1058         timing_point.slew(pin_timing.slew);
1059         timing_point.arrival_time(pin_timing.arrival_time);
1060
1061         if(timing_point.is_PO() /* pode ser o pino d de um
             registrador */)
1062         {
1063             _critical_path = max(_critical_path,
                 timing_point.arrival_time());
1064             if(timing_point.slack().getMin() < 0)
1065             {
1066                 _total_negative_slack -= timing_point.slack
                     ();
1067                 _total_violating_POs++;
1068             }
1069         }
1070
```

```
1071        }
1072
1073        for(size_t i = 0; i < prime_time_output.ports_size(); i
                ++)
1074        {
1075            const Prime_Time_Output_Parser::Port_Timing
                    port_timing = prime_time_output.port(i);
1076            Timing_Point & timing_point = _points.at(
                    _pin_name_to_timing_point_index.at(port_timing.
                    port_name));
1077
1078            timing_point.slack(port_timing.slack);
1079            timing_point.slew(port_timing.slew);
1080            timing_point.arrival_time(port_timing.arrival_window
                    );
1081
1082            //              cout << "port timing " << timing_point
                     << endl;
1083
1084            if(timing_point.is_PO())
1085            {
1086                timing_point.arrival_time(_target_delay -
                        timing_point.slack());
1087                _critical_path = max(_critical_path,
                        _target_delay - timing_point.slack());
1088                if(timing_point.slack().getMin() < 0)
1089                {
1090                    _total_negative_slack -= timing_point.slack
                            ();
1091                    _total_violating_POs++;
1092                }
1093            }
1094        }
1095
1096    }
1097
1098    void Timing_Analysis::write_sizes_file(const string filename
            )
1099    {
1100        get_sizes_vector();
1101        fstream out;
1102        out.open(filename.c_str(), fstream::out);
1103        for(size_t i = 0; i < _sizes.size(); i++)
1104        {
1105            out << _sizes.at(i).first << "\t" << _sizes.at(i).
                    second;
1106            if(i < _sizes.size() - 1)
1107                out << endl;
1108        }
1109        out.close();
1110    }
1111
```

```
1112 void Timing_Analysis::write_timing_file(const std::string
         filename)
1113 {
1114
1115     queue<int> ports;
1116     queue<int> sequentials;
1117     queue<int> pins;
1118
1119     fstream out;
1120     out.open(filename.c_str(), fstream::out);
1121     out << "# pin timing" << endl;
1122
1123
1124     for(size_t i = 0 ; i < _points.size(); i++)
1125     {
1126         const Timing_Point & tp = _points.at(i);
1127         const LibertyCellInfo & cellInfo = liberty_cell_info
             (tp.gate_number());
1128
1129         if( tp.is_PI() && cellInfo.isSequential )
1130         {
1131             sequentials.push(i);
1132             continue;
1133         }
1134
1135         if (tp.is_input_pin() )
1136         {
1137
1138             pins.push(i);
1139             continue;
1140         }
1141
1142         if( tp.is_output_pin() || (tp.is_PO() && cellInfo.
             isSequential))
1143         {
1144             out << tp.name() << " " <<  tp.slack().getRise()
                  << " " << tp.slack().getFall() << " " << tp
                 .slew().getRise() << " " << tp.slew().
                 getFall() << " " << tp.arrival_time().
                 getRise() << " " << tp.arrival_time().
                 getFall() << endl;
1145             while( !pins.empty() )
1146             {
1147                 const Timing_Point & iPin = _points.at(pins.
                     front());
1148                 pins.pop();
1149                 out << iPin.name() << " " <<  iPin.slack().
                     getRise() << " " << iPin.slack().getFall
                     () << " " << iPin.slew().getRise() << "
                     " << iPin.slew().getFall() << " " <<
                     iPin.arrival_time().getRise() << " " <<
                     iPin.arrival_time().getFall() << endl;
```

```
1150                }
1151            }
1152
1153
1154            if( !cellInfo.isSequential && (tp.is_PI() || tp.
                    is_PO()) )
1155                ports.push(i);
1156
1157            if( tp.is_PO() && cellInfo.isSequential )
1158            {
1159                const int reg = sequentials.front();
1160                sequentials.pop();
1161                const Timing_Point & regTp = _points.at(reg);
1162                out << regTp.name() << " " <<  regTp.slack().
                        getRise() << " " << regTp.slack().getFall()
                        << " " << regTp.slew().getRise() << " " <<
                        regTp.slew().getFall() << " " << regTp.
                        arrival_time().getRise() << " " << regTp.
                        arrival_time().getFall() << endl;
1163            }
1164        }
1165
1166        out << "# port timing" << endl;
1167
1168        while(!ports.empty())
1169        {
1170            const int tp_index = ports.front();
1171            ports.pop();
1172            const Timing_Point & tp = _points.at(tp_index);
1173            out << tp.name() << " " <<  tp.slack().getRise() <<
                    " " << tp.slack().getFall() << " " << tp.slew().
                    getRise() << " " << tp.slew().getFall() << endl;
1174        }
1175
1176        out.close();
1177    }
1178
1179    double Timing_Analysis::pin_capacitance(const int
        timing_point_index) const
1180    {
1181        const LibertyCellInfo & opt = liberty_cell_info(_points.
            at(timing_point_index).gate_number());
1182        const Timing_Point & timing_point = _points.at(
            timing_point_index);
1183        if( timing_point.is_input_pin() || timing_point.
            is_PI_input() || timing_point.is_reg_input() )
1184        {
1185            const int pin_number = _points.at(timing_point_index
                ).arc().arc_number();
1186            return opt.pins.at(pin_number+1).capacitance;
1187        }
1188        else if ( _points.at(timing_point_index).is_PO() )
```

```
1189        {
1190            if(opt.isSequential)
1191                return opt.pins.at(2).capacitance;
1192            return _PO_loads.at(timing_point_index);
1193        }
1194        assert(false);
1195        return -1;
1196    }
1197
1198    const Option & Timing_Analysis::option(const int gate_number
        )
1199    {
1200        return _options.at(gate_number);
1201    }
1202
1203    size_t Timing_Analysis::number_of_options(const int
        gate_index)
1204    {
1205        return _library->number_of_options(_options.at(
            gate_index)._footprint_index);
1206    }
1207
1208    void Timing_Analysis::get_sizes_vector()
1209    {
1210        vector<pair<int, string> >::iterator verilog_iterator =
            _verilog.begin();
1211        vector<pair<string, string> >::iterator sizes_iterator =
             _sizes.begin();
1212        for(; verilog_iterator != _verilog.end() &&
            sizes_iterator != _sizes.end(); verilog_iterator++,
            sizes_iterator++)
1213        {
1214            const pair<int, string> & cell = (*verilog_iterator)
                ;
1215            const LibertyCellInfo & opt = liberty_cell_info(cell
                .first);
1216            const pair<string, string> new_item(cell.second, opt
                .name);
1217            (*sizes_iterator) = new_item;
1218        }
1219    }
1220
1221    bool Timing_Analysis::check_timing_file(const string
        timing_file)
1222    {
1223        Prime_Time_Output_Parser prime_time_parser;
1224        const Prime_Time_Output_Parser::Prime_Time_Output
            prime_time_output = prime_time_parser.
            parse_prime_time_output_file(timing_file);
1225
1226        fstream out;
1227        string file = timing_file + ".err";
```

```
1228        out.open(file.c_str(), fstream::out);
1229
1230        Transitions<double> average_pin_slack_error =
                numeric_limits<Transitions<double> >::zero();
1231        Transitions<double> average_pin_slew_error =
                numeric_limits<Transitions<double> >::zero();
1232        Transitions<double> average_pin_arrival_time_error =
                numeric_limits<Transitions<double> >::zero();
1233
1234        Transitions<double> average_port_slack_error =
                numeric_limits<Transitions<double> >::zero();
1235        Transitions<double> average_port_slew_error =
                numeric_limits<Transitions<double> >::zero();
1236
1237        Transitions<double> max_slack_error = numeric_limits<
                Transitions<double> >::min();
1238        Transitions<double> max_slew_error = numeric_limits<
                Transitions<double> >::min();
1239        Transitions<double> max_arrival_time_error =
                numeric_limits<Transitions<double> >::min();
1240
1241        Transitions<double> min_slack_error = numeric_limits<
                Transitions<double> >::max();
1242        Transitions<double> min_slew_error = numeric_limits<
                Transitions<double> >::max();
1243        Transitions<double> min_arrival_time_error =
                numeric_limits<Transitions<double> >::max();
1244
1245        Transitions<int> worst_pin_index(-1, -1);
1246
1247        Transitions<double> critical_path = numeric_limits<
                Transitions<double> >::min();
1248
1249        for(size_t i = 0; i < prime_time_output.pins_size(); i
                ++)
1250        {
1251            const Prime_Time_Output_Parser::Pin_Timing
                    pin_timing = prime_time_output.pin(i);
1252            const Timing_Point & timing_point = _points.at(
                    _pin_name_to_timing_point_index.at(pin_timing.
                    pin_name));
1253
1254            Transitions<double> pin_slack_error = (timing_point.
                    slack()/pin_timing.slack) -1;
1255            Transitions<double> pin_slew_error = (timing_point.
                    slew()/pin_timing.slew) -1;
1256            Transitions<double> pin_arrival_time_error = (
                    timing_point.arrival_time()/pin_timing.
                    arrival_time) -1;
1257
1258            if(timing_point.is_output_pin() || timing_point.
                    is_PO())
```

```
1259            out << timing_point.logic_level() << "\t" <<
                    pin_arrival_time_error.getMax() << endl;
1260
1261        if(timing_point.is_PO())
1262            critical_path = max(critical_path, pin_timing.
                    arrival_time);
1263
1264        if(pin_arrival_time_error.getRise() >
                max_arrival_time_error.getRise())
1265            worst_pin_index.set(i, worst_pin_index.getFall()
                    );
1266
1267        if(pin_arrival_time_error.getFall() >
                max_arrival_time_error.getFall())
1268            worst_pin_index.set(worst_pin_index.getRise(), i
                    );
1269
1270        max_slack_error = max(max_slack_error,
                pin_slack_error);
1271        max_slew_error = max(max_slew_error, pin_slew_error)
                ;
1272        max_arrival_time_error = max(max_arrival_time_error,
                 pin_arrival_time_error);
1273
1274        min_slack_error = min(min_slack_error,
                pin_slack_error);
1275        min_slew_error = min(min_slew_error, pin_slew_error)
                ;
1276        min_arrival_time_error = min(min_arrival_time_error,
                 pin_arrival_time_error);
1277
1278        average_pin_slack_error += pin_slack_error;
1279        average_pin_slew_error += pin_slew_error;
1280        average_pin_arrival_time_error +=
                pin_arrival_time_error;
1281    }
1282
1283    average_pin_slack_error /= prime_time_output.pins_size()
            ;
1284    average_pin_slew_error /= prime_time_output.pins_size();
1285    average_pin_arrival_time_error /= prime_time_output.
            pins_size();
1286
1287    for(size_t i = 0; i < prime_time_output.ports_size(); i
            ++)
1288    {
1289        const Prime_Time_Output_Parser::Port_Timing
                port_timing = prime_time_output.port(i);
1290        const Timing_Point & timing_point = _points.at(
                _pin_name_to_timing_point_index.at(port_timing.
                port_name));
1291
```

```
1292            Transitions<double> port_slack_error = 1 -
                    timing_point.slack() / port_timing.slack;
1293            Transitions<double> port_slew_error = 1 -
                    timing_point.slew() / port_timing.slew;
1294
1295            if(timing_point.is_PO())
1296            {
1297                Transitions<double> arrival_time = _target_delay
                        - port_timing.slack;
1298                Transitions<double> port_arrival_time_error = (
                    timing_point.arrival_time() / arrival_time)
                    - 1;
1299
1300                critical_path = max(critical_path, arrival_time)
                        ;
1301                out << timing_point.logic_level() << "\t" <<
                        port_arrival_time_error.getMax() << endl;
1302            }
1303
1304            average_port_slack_error += port_slack_error;
1305            average_port_slew_error += port_slew_error;
1306
1307            max_slack_error = max(max_slack_error,
                    port_slack_error);
1308            max_slew_error = max(max_slew_error, port_slew_error
                    );
1309
1310            min_slack_error = min(min_slack_error,
                    port_slack_error);
1311            min_slew_error = min(min_slew_error, port_slew_error
                    );
1312
1313        }
1314
1315        out.close();
1316
1317        average_port_slack_error /= prime_time_output.ports_size
                ();
1318        average_port_slew_error /= prime_time_output.ports_size
                ();
1319
1320        cout << "max slack error " << max_slack_error << endl;
1321        cout << "max slew error " << max_slew_error << endl;
1322        cout << "max arrival error " << max_arrival_time_error
                << endl;
1323
1324        cout << "min slack error " << min_slack_error << endl;
1325        cout << "min slew error " << min_slew_error << endl;
1326        cout << "min arrival error " << min_arrival_time_error
                << endl;
1327
1328
```

```
1329        if(max_arrival_time_error.getMax() > Traits::
                STD_THRESHOLD
1330                || max_slack_error.getMax() > Traits::
                        STD_THRESHOLD
1331                || max_slew_error.getMax() > Traits::
                        STD_THRESHOLD)
1332            return false;
1333
1334        return true;
1335    }
1336
1337    int Option::footprint_index() const
1338    {
1339        return _footprint_index;
1340    }
1341
1342    int Option::option_index() const
1343    {
1344        return _option_index;
1345    }
1346
1347    bool Option::is_dont_touch() const
1348    {
1349        return _dont_touch;
1350    }
1351
1352    bool ita_comparator::operator()(Timing_Point *a,
            Timing_Point *b)
1353    {
1354        return a->logic_level() < b->logic_level();
1355    }
1356
1357    }
```

Listing B.31 – timing_analysis.cpp

```
1   #include "include/timing_arc.h"
2
3   namespace Timing_Analysis
4   {
5
6       std::ostream &operator<<(std::ostream &out, const
            Timing_Arc &ta)
7       {
8           return out << ta.from()->name() << " -> " << ta.to()
                .name();
9       }
10
11      void Timing_Arc::clear()
12      {
13          _delay = numeric_limits<Transitions<double> >::zero
                ();
```

```
14          _slew = numeric_limits<Transitions<double> >::zero()
                ;
15      }
16
17  }
```

Listing B.32 – timing_arc.cpp

```
1   #include "include/timing_net.h"
2
3
4   namespace Timing_Analysis
5   {
6
7       const std::string Timing_Net::name() const
8       {
9           return _name;
10      }
11
12      std::ostream & operator<<(std::ostream &out, const
            Timing_Net &tn)
13      {
14          out << tn._name;
15          return out;
16      }
17
18  }
```

Listing B.33 – timing_net.cpp

```
1   #include "include/timing_point.h"
2
3   namespace Timing_Analysis {
4
5   Timing_Point::Timing_Point(std::string name, const size_t
        gate_number, Timing_Point_Type type): _name(name), _net
        (0), _arc(0), _slack(0.0f, 0.0f), _slew(0.0f, 0.0f),
        _arrival_time(0.0f, 0.0f), _gate_number(gate_number),
        _type(type), _logic_level(0)
6       {
7           if(type == REGISTER_INPUT)
8               _slew = Transitions<double>(80.0f, 80.0f);
9       }
10
11      double Timing_Point::load() const
12      {
13          return _net->_wire_delay_model->lumped_capacitance()
                ;
14      }
15
```

```
16      Transitions<double> Timing_Point::ceff() const
17      {
18          return _ceff;
19      }
20
21      const Transitions<double> Timing_Point::update_slack(
            const Transitions<double> required_time)
22      {
23          _slack = required_time - _arrival_time;
24          return _slack;
25      }
26
27      void Timing_Point::clear_timing_info()
28      {
29          _slack =  numeric_limits<Transitions<double> >::zero
                ();
30          _slew = numeric_limits<Transitions<double> >::zero()
                ;
31          _arrival_time = numeric_limits<Transitions<double>
                >::zero();
32          _ceff = numeric_limits<Transitions<double> >::min();
33      }
34
35      std::ostream & operator<<(std::ostream &out, const
            Timing_Point &tp)
36      {
37          return out << tp._name << " slack " << tp._slack <<
                " slew " << tp._slew << " arrival " << tp.
                _arrival_time;
38      }
39
40
41
42  }
```

Listing B.34 – timing_point.cpp

```
1   #include "include/wire_delay_model.h"
2   LinearLibertyLookupTableInterpolator WireDelayModel::
        interpolator;
3
4   const Transitions<double> LumpedCapacitanceWireDelayModel::
        simulate(const LibertyCellInfo & cellInfo, const int
        input, const Transitions<double> slew, bool
        is_input_driver)
5   {
6       Unateness unateness = NEGATIVE_UNATE;
7       if(cellInfo.isSequential)
8           unateness = NON_UNATE;
9       _slew = WireDelayModel::interpolator.interpolate(
            cellInfo.timingArcs.at(input).riseTransition,
```

```
            cellInfo.timingArcs.at(input).fallTransition,
            Transitions<double>(_lumped_capacitance,
            _lumped_capacitance), slew, unateness);
10      _delay = WireDelayModel::interpolator.interpolate(
            cellInfo.timingArcs.at(input).riseDelay, cellInfo.
            timingArcs.at(input).fallDelay, Transitions<double>(
            _lumped_capacitance, _lumped_capacitance), slew,
            unateness, is_input_driver);
11      _max_slew = max(_max_slew, _slew);
12      return Transitions<double>(_lumped_capacitance,
            _lumped_capacitance);
13  }
14
15  // Any fanout node has the same delay and slew
16  const Transitions<double> LumpedCapacitanceWireDelayModel::
        delay_at_fanout_node(const string fanout_node_name)
        const {
17      return numeric_limits<Transitions<double> >::zero();
18  }
19  const Transitions<double> LumpedCapacitanceWireDelayModel::
        slew_at_fanout_node(const string fanout_node_name) const
         {
20      return _max_slew;
21  }
22
23  Transitions<double> LumpedCapacitanceWireDelayModel::
        root_delay(int arc_number)
24  {
25      return _delay;
26  }
27
28  Transitions<double> LumpedCapacitanceWireDelayModel::
        root_slew(int arc_number)
29  {
30      return _slew;
31  }
32
33  void LumpedCapacitanceWireDelayModel::clear()
34  {
35      // NÃfO FAZ NADA
36      _max_slew = numeric_limits<Transitions<double> >::min();
37  }
38
39
40  RC_Tree_Wire_Delay_Model::RC_Tree_Wire_Delay_Model(const
        SpefNetISPD2013 & descriptor, const string rootNode,
        const size_t arcs_size, const bool dummyEdge) :
        WireDelayModel(descriptor.netLumpedCap, descriptor.
        total_resistance), _nodes(descriptor.nodesSize()),
        _nodes_names(descriptor.nodesSize()), _slews(arcs_size,
        vector<Transitions<double> >(descriptor.nodesSize())),
        _delays(arcs_size, vector<Transitions<double> >(
```

```
    descriptor.nodesSize()))
41 {
42   if (dummyEdge)
43     return;
44
45   // criar um vetor de fanouts com referÃªncia para os
         timing points de seus fanouts
46
47   const int rootIndex = descriptor.getNodeIndex(rootNode);
48   const SpefNetISPD2013::Node & root = descriptor.getNode(
         rootIndex);
49   queue<NodeAndResistor> q;
50   vector<bool> added(descriptor.resistorsSize(), false);
51     vector<bool> nodes_added(_nodes.size(), false);
52
53   for (unsigned i = 0; i < root.resistors.size(); i++)
54   {
55     const int resistorIndex = root.resistors[i];
56     const SpefNetISPD2013::Resistor & resistor = descriptor.
           getResistor(resistorIndex);
57     q.push(NodeAndResistor(resistor.getOtherNode(rootIndex),
            resistorIndex));
58         nodes_added.at(resistor.getOtherNode(rootIndex)) =
               true;
59         added.at(resistorIndex) = true;
60   }
61   int neighbourhood;
62     vector<int> topology(_nodes.size(), -1);
63   vector<int> reverseTopology(topology);
64
65     _nodes[0].nodeCapacitance.set(root.capacitance, root.
           capacitance);
66   topology[0] = rootIndex;
67     _nodes_names[0] = rootNode;
68   reverseTopology[rootIndex] = 0;
69   int counter = 1;
70
71   while (!q.empty())
72   {
73     const int & n = q.front().nodeIndex;
74     const int & r = q.front().resistorIndex;
75     q.pop();
76
77     const SpefNetISPD2013::Node & nDescriptor = descriptor.
           getNode(n);
78     const SpefNetISPD2013::Resistor & rDescriptor =
           descriptor.getResistor(r);
79
80         _nodes_names[counter] = nDescriptor.name;
81         _nodes[counter].parent = reverseTopology[rDescriptor
               .getOtherNode(nDescriptor.nodeIndex)];
82         _nodes[counter].nodeCapacitance.set(nDescriptor.
```

```
                capacitance, nDescriptor.capacitance);
83          _nodes[counter].resistance.set(rDescriptor.value,
                rDescriptor.value);
84
85      topology[counter] = n;
86      reverseTopology[n] = counter;
87
88      neighbourhood = 0;
89      for (unsigned i = 0; i < nDescriptor.resistors.size(); i
            ++)
90      {
91              if (!added.at(nDescriptor.resistors[i]))
92        {
93          const SpefNetISPD2013::Resistor & resistor =
                descriptor.getResistor(nDescriptor.resistors[i])
                ;
94                  if(!nodes_added.at(resistor.getOtherNode(
                        nDescriptor.nodeIndex)))
95                  {
96                      q.push(NodeAndResistor(resistor.
                            getOtherNode(nDescriptor.nodeIndex),
                             nDescriptor.resistors[i]));
97                      nodes_added.at(resistor.getOtherNode(
                            nDescriptor.nodeIndex)) = true;
98                  }
99          added[nDescriptor.resistors[i]] = true;
100         neighbourhood++;
101       }
102     }
103
104         _nodes[counter].sink = (neighbourhood == 0);
105
106         _node_name_to_node_number[nDescriptor.name] =
                counter;
107
108     counter++;
109   }
110
111     IBM_update_downstream_capacitances();
112     IBM_initialize_effective_capacitances();
113 }
114
115
116 const Transitions<double> Ceff_Elmore_Slew_Degradation_PURI
        ::simulate(const LibertyCellInfo & cellInfo, const int
        input, const Transitions<double> slew, bool
        is_input_driver)
117 {
118     Transitions<double> ceff = run_IBM_algorithm(cellInfo,
            input, slew, is_input_driver);
119 //    for(int i = 1; i < _delays.at(input).size(); i++)
120 //    {
```

```
121 //         _delays.at(input).at(i) *= log(2);
122 //     }
123     return ceff;
124 }
125
126
127 void RC_Tree_Wire_Delay_Model::IBM_update_slews(const
        LibertyCellInfo & cellInfo, const int input, const
        Transitions<double> slew, bool is_input_driver)
128 {
129     _nodes[0].delay.set(0.0f, 0.0f);
130
131     _delays.at(input).front() = _nodes[0].delay;
132     _slews.at(input).front() = _nodes[0].slew;
133
134     for (size_t i = 1; i < _nodes.size(); i++)
135   {
136         Transitions<double> & t_0_to_1 = _nodes[i].delay;
137         Transitions<double> & s_0 = _nodes[_nodes[i].parent
                ].slew;
138         Transitions<double> & r_1 = _nodes[i].resistance;
139         Transitions<double> & ceff_1 = _nodes[i].
                effectiveCapacitance;
140         Transitions<double> & s_1 = _nodes[i].slew;
141
142         t_0_to_1 = _nodes[_nodes[i].parent].delay + r_1 *
                ceff_1;
143
144         const Transitions<double> x = r_1 * ceff_1 / s_0;
145         s_1 = s_0 / ( 1 - x * ( 1 - exp( -1/x ) ) );
146
147
148         _delays.at(input).at(i) = t_0_to_1;
149         _slews.at(input).at(i) = s_1;
150   }
151
152 }
153
154 void RC_Tree_Wire_Delay_Model::
        IBM_update_effective_capacitances()
155 {
156     vector<bool> initialized(_nodes.size(), false);
157     for (int j = _nodes.size() - 1; j > 0; j--)
158   {
159         RC_Tree_Wire_Delay_Model::Node & node_j = _nodes.at(
                j);
160         if (node_j.sink)
161             node_j.effectiveCapacitance = node_j.
                    nodeCapacitance;
162
163         const Transitions<double> & c_tot_j = node_j.
                totalCapacitance;
```

```
164             const Transitions<double> & ceff_j = node_j.
                    effectiveCapacitance;
165             const Transitions<double> & r_j = node_j.resistance;
166             const Transitions<double> & s_i = _nodes[node_j.
                    parent].slew;
167
168
169             const Transitions<double> x = 2 * r_j * ceff_j / s_i
                    ;
170             const Transitions<double> y = 1 - exp(-1/x);
171
172
173             // x = 2 * r_j * ceff_j / s_i
174             // y = 1-e^(-1/x)
175             // shielding_factor = 1-x*y
176             const Transitions<double> shielding_factor = 1 - x *
                     y;
177
178             assert(shielding_factor.getRise() > 0.0f &&
                    shielding_factor.getRise() < 1.0f);
179             assert(shielding_factor.getFall() > 0.0f &&
                    shielding_factor.getFall() < 1.0f);
180
181             if (!initialized.at(node_j.parent))
182         {
183                 _nodes.at(node_j.parent).effectiveCapacitance =
                        _nodes.at(node_j.parent).nodeCapacitance;
184                 initialized.at(node_j.parent) = true;
185         }
186
187             _nodes.at(node_j.parent).effectiveCapacitance +=
                    shielding_factor * c_tot_j;
188     }
189 }
190
191 void RC_Tree_Wire_Delay_Model::
        IBM_update_downstream_capacitances()
192 {
193     for (size_t i = 0; i < _nodes.size(); i++)
194     {
195         Node & node = _nodes[i];
196       node.totalCapacitance = node.nodeCapacitance;
197     }
198     for (size_t i = _nodes.size() - 1; i > 0; i--)
199     {
200         Node & node = _nodes[i];
201         _nodes[node.parent].totalCapacitance += node.
                totalCapacitance;
202     }
203 }
204
205 void RC_Tree_Wire_Delay_Model::
```

```
    IBM_initialize_effective_capacitances()
{
    for (size_t i = 0; i < _nodes.size(); i++)
  {
        Node & node = _nodes[i];
        node.effectiveCapacitance = node.totalCapacitance;
    }
}

const Transitions<double> RC_Tree_Wire_Delay_Model::
    run_IBM_algorithm(const LibertyCellInfo &cellInfo, const
     int input, const Transitions<double> slew, bool
    is_input_driver)
{
    IBM_update_downstream_capacitances();
    IBM_initialize_effective_capacitances();
    //forwardIterate();
    Transitions<double> error;

    Transitions<double> current_source_slew, old_source_slew
        ;
    Transitions<double> current_source_ceff, old_source_ceff
        ;
    int i = 0;

    _nodes[0].slew = RC_Tree_Wire_Delay_Model::interpolator.
        interpolate(cellInfo.timingArcs.at(input).
        riseTransition, cellInfo.timingArcs.at(input).
        fallTransition, _nodes[0].totalCapacitance, slew, (
        cellInfo.isSequential?NON_UNATE:NEGATIVE_UNATE));
    current_source_slew = _nodes[0].slew;
    current_source_ceff = _nodes[0].totalCapacitance;

    do
    {


        IBM_update_slews(cellInfo, input, slew,
            is_input_driver);
        IBM_update_effective_capacitances();
        _nodes[0].slew = RC_Tree_Wire_Delay_Model::
            interpolator.interpolate(cellInfo.timingArcs.at(
            input).riseTransition, cellInfo.timingArcs.at(
            input).fallTransition, _nodes[0].
            effectiveCapacitance, slew, (cellInfo.
            isSequential?NON_UNATE:NEGATIVE_UNATE));

        old_source_slew = current_source_slew;
        current_source_slew = _nodes[0].slew;


        old_source_ceff = current_source_ceff;
```

```
242             current_source_ceff = _nodes[0].effectiveCapacitance
                    ;
243
244
245             i++;
246             error = abs(old_source_slew - current_source_slew) /
                     max(abs(old_source_slew), abs(
                    current_source_slew));
247 //            error = abs(old_source_ceff - current_source_ceff)
        / max(abs(old_source_ceff), abs(current_source_ceff));
248
249
250         }
251         while (error.getRise() > Traits::STD_THRESHOLD || error.
                getFall() > Traits::STD_THRESHOLD);
252
253         return _nodes.front().effectiveCapacitance;
254 }
255
256
257
258
259 void RC_Tree_Wire_Delay_Model::setFanoutPinCapacitance(const
        string fanoutNameAndPin, const double pinCapacitance)
260 {
261         _nodes.at(_node_name_to_node_number.at(fanoutNameAndPin)
                ).nodeCapacitance += pinCapacitance;
262         _nodes.at(_node_name_to_node_number.at(fanoutNameAndPin)
                ).sink = true;
263         _lumped_capacitance += pinCapacitance;
264 }
265
266 Transitions<double> RC_Tree_Wire_Delay_Model::root_delay(int
        arc_number)
267 {
268         return _delays.at(arc_number).front();
269 }
270
271 Transitions<double> RC_Tree_Wire_Delay_Model::root_slew(int
        arc_number)
272 {
273         return _slews.at(arc_number).front();
274 }
275
276 void RC_Tree_Wire_Delay_Model::clear()
277 {
278 //     std::fill(_max_delays.begin(), _max_delays.end(),
        numeric_limits<Transitions<double> >::zero());
279 //     std::fill(_max_slews.begin(), _max_slews.end(),
        numeric_limits<Transitions<double> >::zero());
280 }
281
```

```
282 void Reduced_Pi::reduce_to_pi_model(double &c_near, double &
        r, double &c_far)
283 {
284     const int number_of_nodes = _nodes.size();
285 
286     std::vector<double> y1(number_of_nodes, 0.0);
287     std::vector<double> y2(number_of_nodes, 0.0);
288     std::vector<double> y3(number_of_nodes, 0.0);
289 
290     // Compute pi-model of the RC Tree.
291     for ( int n = number_of_nodes - 1; n > 0; n-- ) { // n >
            0 skips root node
292         const Node &node = _nodes[n];
293 
294         const double C = node.nodeCapacitance.getMax();
295         const double R = node.resistance.getMax();
296 
297         const double yD1 = y1[n];
298         const double yD2 = y2[n];
299         const double yD3 = y3[n];
300 
301         const double yU1 = yD1 + C;
302         const double yU2 = yD2 - R * (pow(yD1,2.0) + C*yD1 +
                (1.0/3.0)*pow(C, 2.0));
303         const double yU3 = yD3 - R * (2*yD1*yD2 + C*yD2) +
304             pow(R,2.0)*( pow(yD1,3.0) + (4.0/3.0)*C*pow(yD1
                    ,2.0) + (2.0/3.0)*pow(C,2.0)*yD1 +
                    (2.0/15.0)*pow(C,3.0) );
305 
306         y1[node.parent] += yU1;
307         y2[node.parent] += yU2;
308         y3[node.parent] += yU3;
309 
310         //cout << "Resistor: " << clsNodeNames[node.
                propParent] << " -> " << clsNodeNames[n] << "\n
                ";
311 
312     } // end for
313 
314     c_near = pow(y2[0],2.0) / y3[0];
315     c_far = y1[0] - c_near;
316     r = -pow(y3[0],2.0)/pow(y2[0],3.0);
317 }
318 
319 double WireDelayModel::lumped_capacitance() const
320 {
321     return _lumped_capacitance;
322 }
323 
324 double WireDelayModel::total_resistance() const
325 {
326     return _total_resistance;
```

```
327 }
328
329
330 const Transitions<double> Reduced_Pi::simulate(const
        LibertyCellInfo &cellInfo, const int input, const
        Transitions<double> slew, bool is_input_driver)
331 {
332
333     reduce_to_pi_model(_c1, _r, _c2);
334
335 //    assert(_c1 + _c2 == _lumped_capacitance);
336     std::vector<Node> nodes(2);
337
338     Node & C1 = nodes.front();
339     Node & C2 = nodes.back();
340
341     C1.parent = -1;
342     C1.nodeCapacitance.set(_c1, _c1);
343     C1.totalCapacitance.set(_c1+_c2, _c1+_c2);
344     C1.effectiveCapacitance = C1.totalCapacitance;
345     C1.resistance.set(0.0f, 0.0f);
346
347
348     const Transitions<double> driver_delay =
            RC_Tree_Wire_Delay_Model::interpolator.interpolate(
            cellInfo.timingArcs.at(input).riseTransition,
            cellInfo.timingArcs.at(input).fallTransition, C1.
            totalCapacitance, slew, (cellInfo.isSequential?
            NON_UNATE:NEGATIVE_UNATE));
349     const Transitions<double> driver_resistance =
            driver_delay / C1.totalCapacitance;
350     C1.effectiveCapacitance = _c1 + (driver_resistance / (
            driver_resistance + _r)) * _c2;
351
352     C1.slew = RC_Tree_Wire_Delay_Model::interpolator.
            interpolate(cellInfo.timingArcs.at(input).
            riseTransition, cellInfo.timingArcs.at(input).
            fallTransition, C1.totalCapacitance, slew, (cellInfo
            .isSequential?NON_UNATE:NEGATIVE_UNATE));
353     C1.delay.set(0.0f, 0.0f);
354
355     C2.parent = 0;
356     C2.nodeCapacitance.set(_c2, _c2);
357     C2.totalCapacitance = C2.nodeCapacitance;
358     C2.effectiveCapacitance = C2.totalCapacitance;
359     C2.resistance.set(_r, _r);
360     C2.slew.set(0.0f, 0.0f);
361     C2.delay.set(0.0f, 0.0f);
362
363
364
365     bool converged;
```

```
366     Transitions<double> prev_source_slew,
            current_source_slew;
367     current_source_slew = C1.slew;
368     do
369     {
370         // UPDATE SLEWS
371         Transitions<double> & t_0_to_1 = C2.delay;
372         Transitions<double> & s_0 = C1.slew;
373         Transitions<double> & r_1 = C2.resistance;
374         Transitions<double> & ceff_1 = C2.
                effectiveCapacitance;
375         Transitions<double> & s_1 = C2.slew;
376         t_0_to_1 = C1.delay + r_1 * ceff_1;
377         const Transitions<double> x = r_1 * ceff_1 / s_0;
378         s_1 = s_0 / ( 1 - x * ( 1 - exp( -1/x ) ) );
379
380         // UPDATE CEFF
381         C2.effectiveCapacitance = C2.nodeCapacitance;
382
383         const Transitions<double> & c_tot_j = C2.
                totalCapacitance;
384         const Transitions<double> & ceff_j = C2.
                effectiveCapacitance;
385         const Transitions<double> & r_j = C2.resistance;
386         const Transitions<double> & s_i = C1.slew;
387
388         const Transitions<double> z = 2 * r_j * ceff_j / s_i
                ;
389         const Transitions<double> y = 1 - exp(-1/z);
390
391         const Transitions<double> shielding_factor = 1 - z *
                 y;
392
393         assert(shielding_factor.getRise() > 0.0f &&
                shielding_factor.getRise() < 1.0f);
394         assert(shielding_factor.getFall() > 0.0f &&
                shielding_factor.getFall() < 1.0f);
395
396         C1.effectiveCapacitance = C1.nodeCapacitance;
397         C1.effectiveCapacitance += shielding_factor *
                c_tot_j;
398
399         prev_source_slew = current_source_slew;
400         C1.slew = RC_Tree_Wire_Delay_Model::interpolator.
                interpolate(cellInfo.timingArcs.at(input).
                riseTransition, cellInfo.timingArcs.at(input).
                fallTransition, C1.effectiveCapacitance, slew, (
                cellInfo.isSequential?NON_UNATE:NEGATIVE_UNATE))
                ;
401         current_source_slew = C1.slew;
402
403         const Transitions<double> error =  abs(
```

```
                prev_source_slew - current_source_slew) / max(
                abs(prev_source_slew), abs(current_source_slew))
                ;
404         converged = error.getMin() < 0.01f;
405     } while(!converged);
406
407     _slews.at(input).front() = C1.slew;
408     _slew_fanout = C2.slew;
409     _delay_fanout = C2.delay;
410
411     return C1.effectiveCapacitance;
412
413 }
414
415 const Transitions<double> Reduced_Pi::delay_at_fanout_node(
        const string fanout_node_name)  const
416 {
417     return _delay_fanout;
418 }
419
420 const Transitions<double> Reduced_Pi::slew_at_fanout_node(
        const string fanout_node_name) const
421 {
422     return _slew_fanout;
423 }
424
425
426 const Transitions<double> Lumped_Elmore_Slew_Degradation::
        simulate(const LibertyCellInfo &cellInfo, const int
        input, const Transitions<double> slew, bool
        is_input_driver)
427 {
428     IBM_update_downstream_capacitances();
429     IBM_initialize_effective_capacitances();
430     _nodes[0].slew = RC_Tree_Wire_Delay_Model::interpolator.
            interpolate(cellInfo.timingArcs.at(input).
            riseTransition, cellInfo.timingArcs.at(input).
            fallTransition, _nodes[0].totalCapacitance, slew, (
            cellInfo.isSequential?NON_UNATE:NEGATIVE_UNATE));
431     IBM_update_slews(cellInfo, input, slew, is_input_driver)
            ;
432
433 //    for(int i = 1; i < _delays.at(input).size(); i++)
434 //    {
435 //        _delays.at(input).at(i) *= log(2);
436 //    }
437     return _nodes.front().effectiveCapacitance;
438 }
439
440 const Transitions<double> RC_Tree_Wire_Delay_Model::
        delay_at_fanout_node(const string fanout_node_name)
        const
```

```
441 {
442     int fanout_index = _node_name_to_node_number.at(
            fanout_node_name);
443
444     Transitions<double> max_delay = _delays.front().at(
            fanout_index);
445     for(int i = 1; i < _delays.size(); i++)
446     {
447         max_delay = max(max_delay, _delays.at(i).at(
                fanout_index));
448     }
449     return max_delay;
450 }
451 const Transitions<double> RC_Tree_Wire_Delay_Model::
        slew_at_fanout_node(const string fanout_node_name) const
452 {
453     int fanout_index = _node_name_to_node_number.at(
            fanout_node_name);
454     Transitions<double> max_slew = _slews.front().at(
            fanout_index);
455     for(int i = 1; i < _slews.size(); i++)
456     {
457         max_slew = max(max_slew, _slews.at(i).at(
                fanout_index));
458     }
459     return max_slew;
460 }
461
462
463 const Transitions<double> Lumped_Elmore_No_Slew_Degradation
        ::simulate(const LibertyCellInfo &cellInfo, const int
        input, const Transitions<double> slew, bool
        is_input_driver)
464 {
465     IBM_update_downstream_capacitances();
466     IBM_initialize_effective_capacitances();
467     _nodes[0].slew = RC_Tree_Wire_Delay_Model::interpolator.
            interpolate(cellInfo.timingArcs.at(input).
            riseTransition, cellInfo.timingArcs.at(input).
            fallTransition, _nodes[0].totalCapacitance, slew, (
            cellInfo.isSequential?NON_UNATE:NEGATIVE_UNATE));
468     IBM_update_slews(cellInfo, input, slew, is_input_driver)
            ;
469
470     std::fill(_slews.at(input).begin(), _slews.at(input).end
            (), _slews.at(input).front());
471
472     return _nodes.front().effectiveCapacitance;
473 }
474
475
476 const Transitions<double> Ceff_Elmore_No_Slew_Degradation::
```

```
    simulate(const LibertyCellInfo &cellInfo, const int
    input, const Transitions<double> slew, bool
    is_input_driver)
{
    Transitions<double> ceff = run_IBM_algorithm(cellInfo,
        input, slew, is_input_driver);

    std::fill(_slews.at(input).begin(), _slews.at(input).end
        (), _slews.at(input).front());

    return ceff;
}


const Transitions<double>
    Ceff_Without_Wire_Delay_And_Slew_Degradation::simulate(
    const LibertyCellInfo &cellInfo, const int input, const
    Transitions<double> slew, bool is_input_driver)
{
    IBM_update_downstream_capacitances();
    IBM_initialize_effective_capacitances();
    //forwardIterate();
    Transitions<double> error;

    Transitions<double> current_source_slew, old_source_slew
        ;
    int i = 0;

    _nodes[0].slew = RC_Tree_Wire_Delay_Model::interpolator.
        interpolate(cellInfo.timingArcs.at(input).
        riseTransition, cellInfo.timingArcs.at(input).
        fallTransition, _nodes[0].totalCapacitance, slew, (
        cellInfo.isSequential?NON_UNATE:NEGATIVE_UNATE));
    current_source_slew = _nodes[0].slew;

    do
    {
        IBM_update_slews(cellInfo, input, slew,
            is_input_driver);
        IBM_update_effective_capacitances();
        _nodes[0].slew = RC_Tree_Wire_Delay_Model::
            interpolator.interpolate(cellInfo.timingArcs.at(
            input).riseTransition, cellInfo.timingArcs.at(
            input).fallTransition, _nodes[0].
            effectiveCapacitance, slew, (cellInfo.
            isSequential?NON_UNATE:NEGATIVE_UNATE));

        old_source_slew = current_source_slew;
        current_source_slew = _nodes[0].slew;

        i++;
        error = abs(old_source_slew - current_source_slew) /
```

```
                max(abs(current_source_slew), abs(
                old_source_slew));
510
511         }
512         while (error.getRise() > Traits::STD_THRESHOLD/100 ||
                error.getFall() > Traits::STD_THRESHOLD/100);
513
514         std::fill(_delays.at(input).begin(), _delays.at(input).
                end(), numeric_limits<Transitions<double> >::zero())
                ;
515         std::fill(_slews.at(input).begin(), _slews.at(input).end
                (), _slews.at(input).front());
516         return _nodes.front().effectiveCapacitance;
517     }
518
519
520     const Transitions<double> Ceff_Elmore_Slew_Degradation::
            simulate(const LibertyCellInfo &cellInfo, const int
            input, const Transitions<double> slew, bool
            is_input_driver)
521     {
522         Transitions<double> ceff = run_IBM_algorithm(cellInfo,
                input, slew, is_input_driver);
523         Transitions<double> root_slew = _slews.at(input).front()
                ;
524
525         vector<Transitions<double> > ceff_elmore_delays(_delays.
                at(input).size());
526
527     //     for(int i = 0; i < _slews[input].size(); i++)
528     //     {
529     //         ceff_elmore_delays.at(i) = _delays[input][i];
530     //     }
531
532
533         for(int i = 1; i < _nodes.size(); i++)
534         {
535             Node & node = _nodes.at(i);
536             Node & parent = _nodes.at(node.parent);
537             node.delay = parent.delay + node.resistance * node.
                    totalCapacitance;
538
539             _delays[input][i] = node.delay;
540         }
541
542
543     //     for(int i = 0; i < _slews[input].size(); i++)
544     //     {
545     ////         Transitions<double> degradation = _slews.at(
            input).at(i);
546     ////         Transitions<double> degradation = _slews[input][
            i] - root_slew;
```

```
547 //         Transitions<double> degradation = _delays.at(input
        ).at(i) * log(4);
548 
549 ////         if(_nodes.at(i).sink)
550 ////             cout << "root slew = " << root_slew << ",
        leaf slew = " << _slews.at(input).at(i) << endl;
551 
552 ////         _slews.at(input).at(i) = sqrt(root_slew *
        root_slew + degradation * degradation);
553 ////         _slews.at(input).at(i) = root_slew;
554 
555 ////           _slews[input][i] = (_slews[input][i] +
        root_slew) / 2;
556 ////         _delays[input][i] *= log(2);
557 
558 //     }
559 
560 //     _delays[input] = ceff_elmore_delays;
561 
562 
563 
564     return ceff;
565 }
```

Listing B.35 – wire_delay_model.cpp