UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA - CI
CURSO ENGENHARIA DE COMPUTAÇÃO

EPITÁCIO PESSOA DE BRITO NETO
11506856

RELATÓRIO DE MICROELETRÔNICA – PARTE 3

JOÃO PESSOA
2020

UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA - CI
CURSO ENGENHARIA DE COMPUTAÇÃO

EPITÁCIO PESSOA DE BRITO NETO
11506856

RELATÓRIO DE MICROELETRÔNICA – PARTE 3

Relatório referente à disciplina de Introdução à Microeletrônica do Ensino Superior da Universidade Federal da Paraíba (UFPB) como requisito parcial da avaliação semestral.

Professor: Antonio Carlos Cavalcanti

Hugo Leonardo Davi de Souza Cavalcante

JOÃO PESSOA
2020

RELATÓRIO DE MICROELETRÔNICA – PARTE 3

O objetivo principal deste projeto é a aplicação teórica dos estudos feitos na disciplina de Introdução à Microeletrônica ministrada pelos professores Antonio Carlos e Hugo Leonardo, ofertada pela Universidade Federal da Paraíba (UFPB), para fins didáticos e avaliação do aprendizado durante o percurso da disciplina. Especificamente, este relatório se tratará de aplicações práticas, simulações, desenhos e análises na construção de tecnologias.

Buscamos, com este relatório, um entendimento sucinto das atividades realizadas em sala, provendo informações do que utilizamos para realizar nossas simulações e dados tanto da listagem de grandezas e componentes utilizados junto com seus respectivos valores.

1 Introdução

Este relatório se remete à, basicamente, aplicações das teorias e análises de técnicas de construção e interpretação de hardwares ministradas nas aulas dos professores Antonio Carlos e Hugo Leonardo. Neste, criaremos um chip a partir da linguagem procedural criada em C, converteremos, quando necessário, a descrição e extração dos arquivos necessários para construção, de forma que possamos utilizar as diferentes descrições para uso das ferramentas que nos foram apresentadas durante o andamento da disciplina.

Também faremos várias séries de teste para que essas conversões de descrições, de fato, traduza a descrição para a outra sem que haja perda, incoerência lógica ou estrutural. Desta forma, com auxílio de uma série de arquivos disponibilizados na especificação do trabalho, conseguiremos chegar a um modelo de chip com um núcleo e pads inseridos nele.

Este passo a passo será feito da seguinte maneira: serão explicados os procedimentos de construção, como o uso das ferramentas (o que elas fazem, o que alteram, por que a utilizamos), a descrição do comando destas que foram utilizadas durante o processo via terminal e, em seguida, uma série de imagens que também serão analisadas, tanto no que se refere ao que ela mostra, como uma breve explicação do que pode estar implícito nelas.

2 Construção do ADAC sem atraso

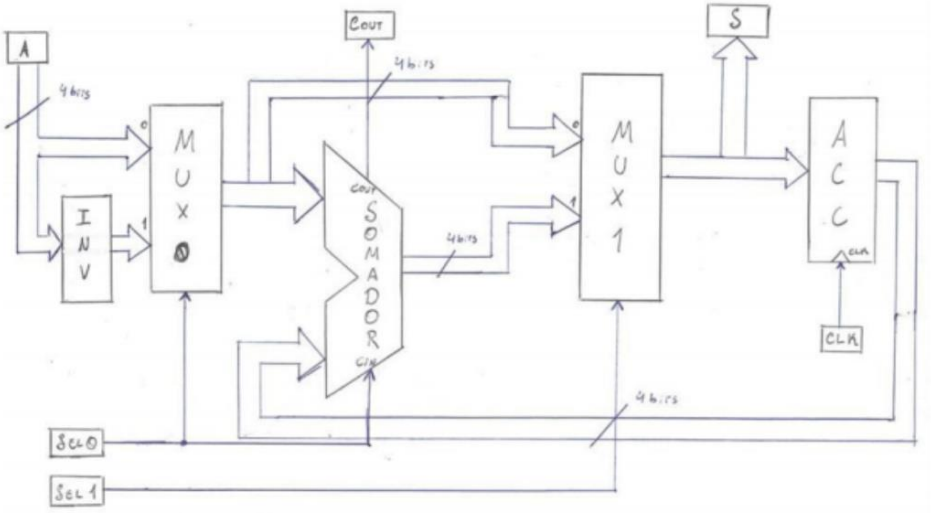

Figura 1: Desenho e respostas do projeto final.

Primeiramente, analisamos a estrutura do código disponibilizado pelo professor, identificando as funcionalidades do mesmo e se corresponde com a descrição, em forma procedural com C, utilizando a biblioteca genpat, ao projeto identificado na figura 1. Após isso, trabalharemos na linha de comando de forma que consigamos gerar o arquivo genpat equivalente à descrição que foi feita no arquivo C anterior.

Em seguida, ainda com arquivos disponibilizados pelo professor, pegamos o arquivo adac.vhd, que seria um tipo de "VHDL de alto nível" e não um RTL. Este não tem tradução imediata para um modelo de silício, portanto, deve ser transformado por meio de uma compilação para que consigamos chegar num modelo RTL e, assim, chegarmos ao silício. No nosso, caso utilizaremos a ferramenta Vasy para transformar este arquivo adac.vhd em um modelo próximo ao de silício.

Alliance-genpat adac_sem_atraso

Vasy –a –I vhd adac adac_vasy

Asimut –b adac_vasy adac_sem_atraso adac_vasy_res

Ao fazermos a conversão, utilizando o Vasy, conseguimos perceber diferenças entre uma descrição e outra. Enquanto o adac.vhd conseguia descrever sinais, entradas e saídas de formas mais sucintas e organizadas, o adac_vasy.vbe é limitado nas entradas e saídas para valores 0, 1 e Z e, também, existe a necessidade de criar sinais intermediários para descrever todo o comportamento do circuito, além do uso de lógica booleana na descrição do arquivo.

Após a análise entre os dois arquivos, utilizamos (e continuaremos com ela durante o relatório) a ferramenta Asimut (a simulation tool). Como queremos analisar o comportamento, utilizamos a instrução "-b" (behavior) na linha do terminal, que pode ser vista acima. Depois, continuando na sintaxe do Asimut, passamos o nome do nosso arquivo.vbe, em seguida, o arquivo.pat feito pelo genpat. Por fim, um arquivo resultante da simulação feita pelo Asimut. A operação de simulação pode ser conferida abaixo:

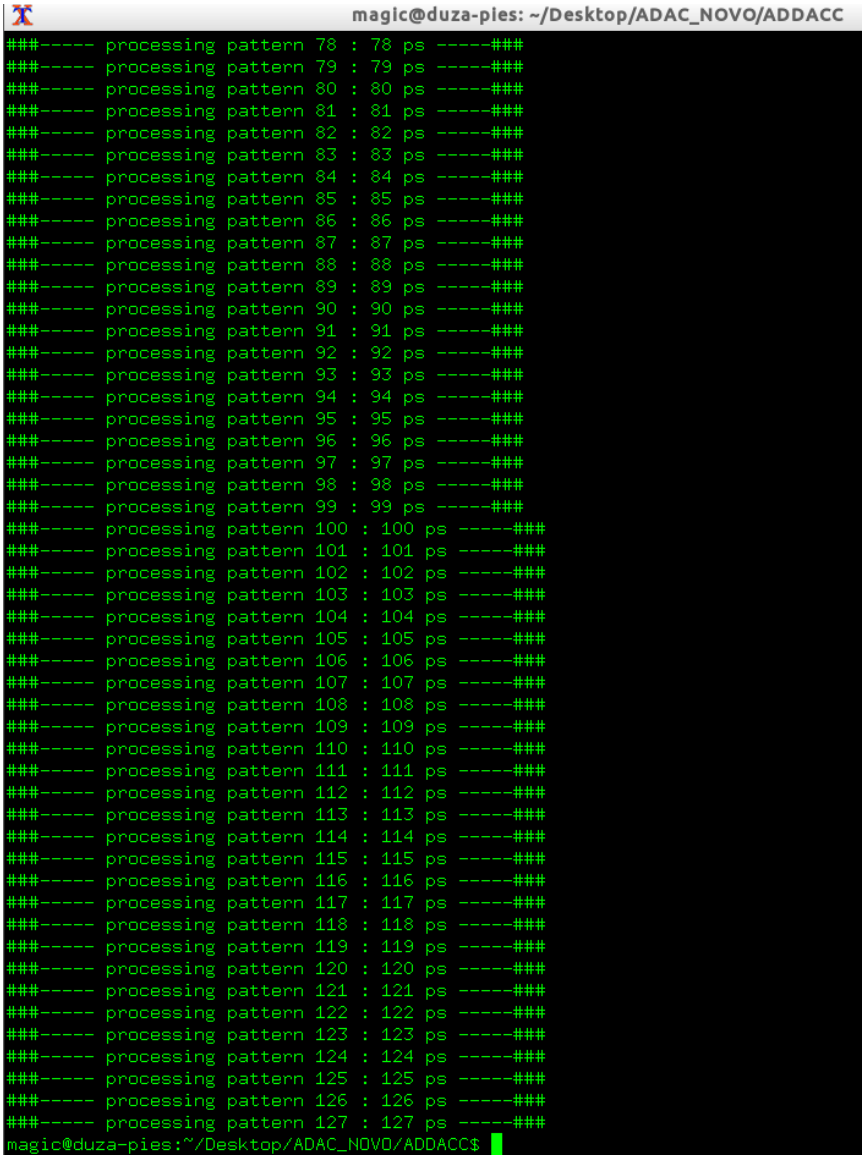

Figura 2: Resultado da simulação do adac_vasy_res.

É importante dizer que, se o Asimut (como aconteceu durante a execução do projeto) não retornar algum erro, significa que a especificação em VHDL é equivalente à implementação feita no arquivo C e, qualquer erro que seja encontrado durante a operação do Asimut deve ser corrigido diretamente do arquivo de mais alto nível, no nosso caso o arquivo C.

Para confirmarmos as operações do circuito ao longo do tempo, utilizaremos a ferramenta Xpat, disponibilizada pelo Alliance. O resultado será o que está exposto na figura 3 e, por fim, nosso papel é analisar e confirmar, por meio de formas de onda, se o comportamento do projeto está condizente à sua especificação.

Xpat –l ‘adac_vasy_res’&

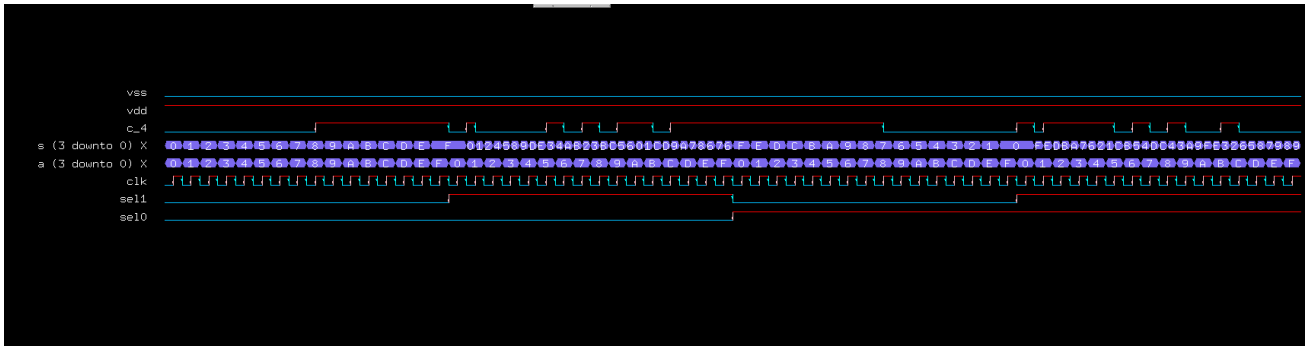

Figura 3: Modelo de visualização das entradas e saídas em formato de tabela.

Após a análise utilizando o Xpat, utilizaremos a funcionalidade Boom, para que possamos otimizar o circuito quanto à lógica booleana aplicada em nível RTL, que fizemos com o Vasy. É descrito, durante o comando no terminal, a porcentagem em relação à área e a velocidade (no nosso caso, escolhemos 50%) e o número de iterações que serão feitas (no nosso caso, serão 100 iterações).

Após isso, para continuarmos verificando a equivalência deste modelo modificado com a descrição do arquivo em C, voltamos a utilizar o Asimut, e esse resultado pode ser conferido na figura 4. Conseguido uma simulação sem erros, prosseguimos.

Boom –l 3 –d 50% -i 100 adac_vasy adac_vasy_boom_3_50_100

Asimut –b adac_vasy_boom_3_50_100 adac_sem_atraso adac_vasy_boom_3_50_100_res

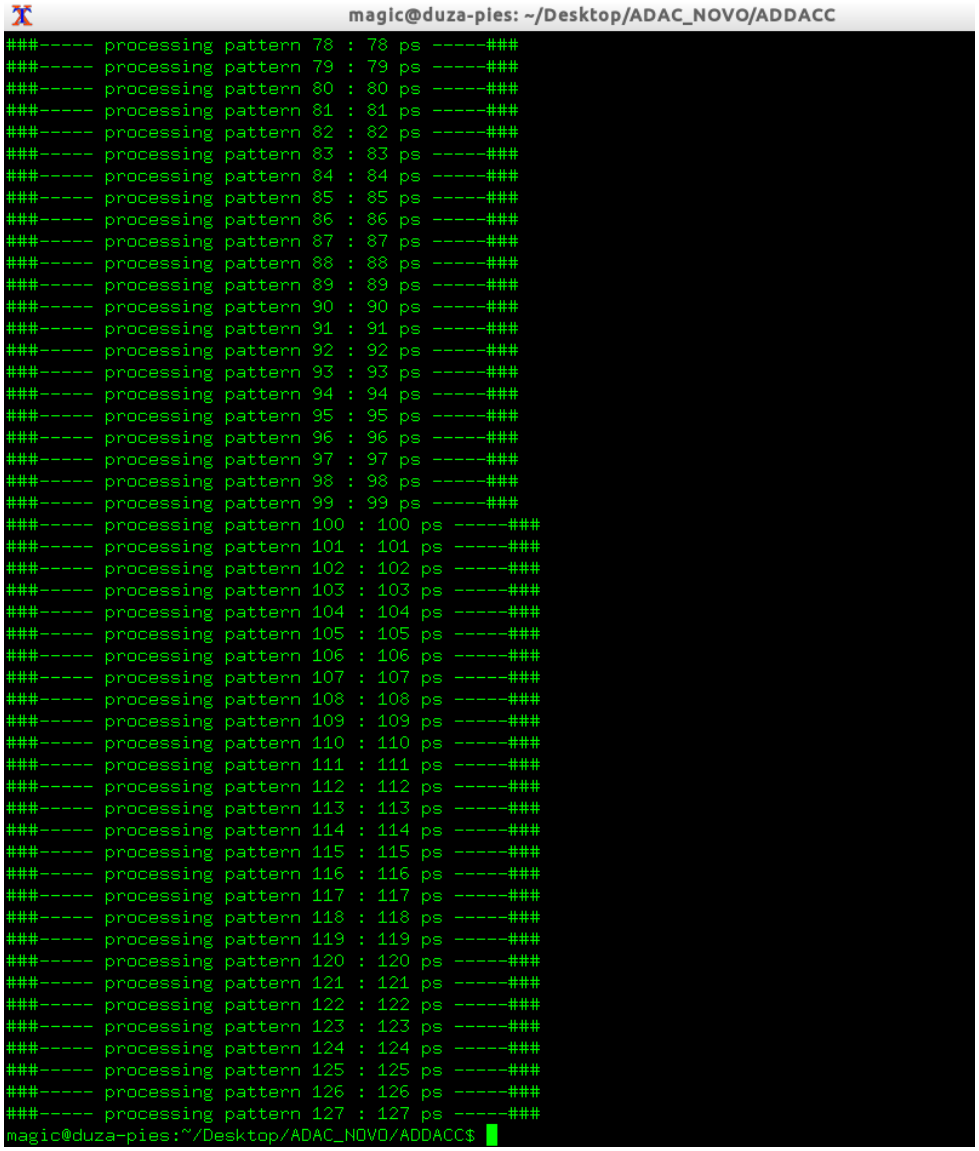

Figura 4: Resultado da simulação do adac_vasy_boom_3_50_100_res.

Em seguida, utilizaremos o Proof, que é utilizado para provar, formalmente, se duas descrições comportamentais VHDL são equivalentes, utilizando as regras matemáticas com relação à equivalência no conceito da lógica booleana. Dentro do Proof, teremos duas representações dos códigos em formato de árvore binária, portanto, se chegamos à mesma árvore (ou árvores equivalentes), teremos duas descrições equivalentes.

Proof –a –d adac_vasy adac_vasy_boom_3_50_100

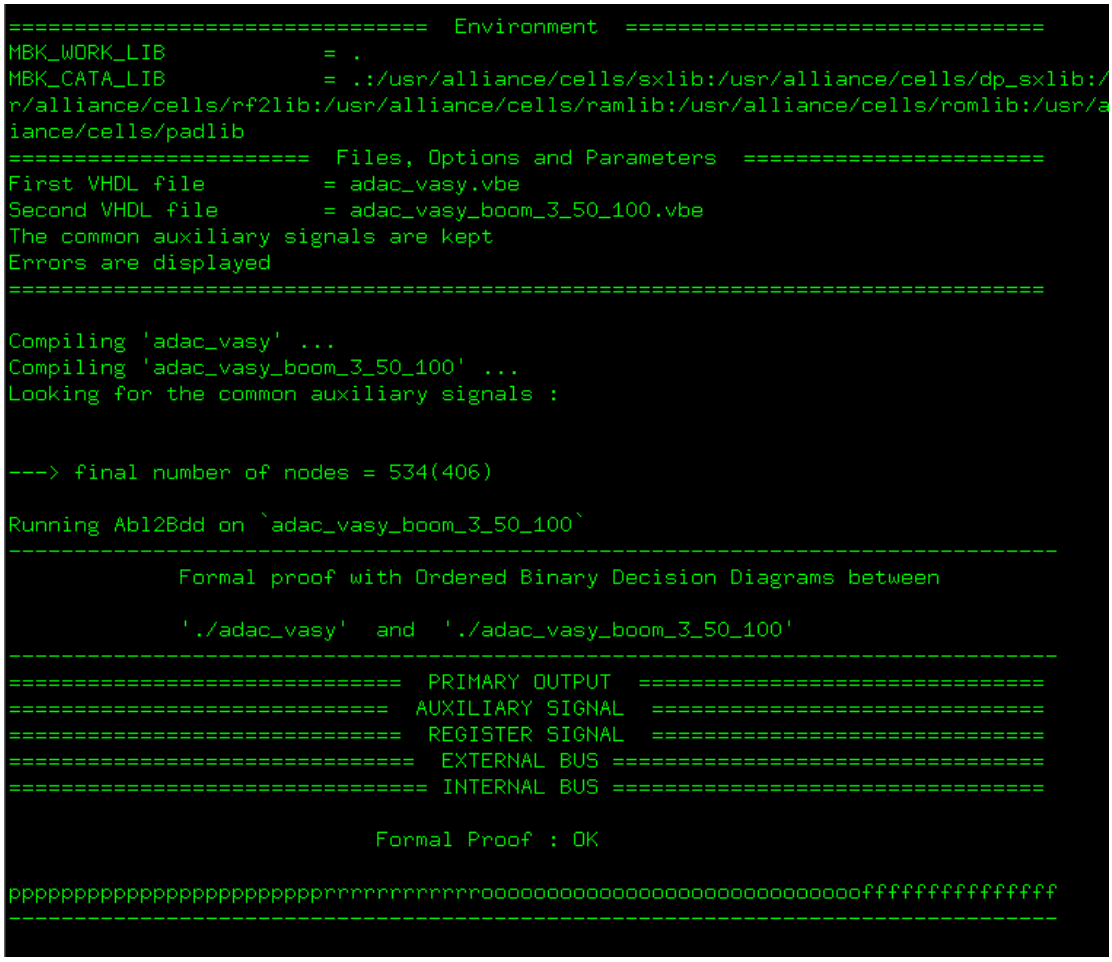

Figura 5: Resultado da avaliação do adac_vasy_boom_3_50_100 utilizando a ferramenta Proof.

Agora, utilizaremos a ferramenta Boog, que mapeia a descrição comportamental na biblioteca sxlib, de forma que consigamos chegar, a partir de um arquivo RTL, em um modelo sintetizável.

Export MBK_OUT_LO=vst

Boog adac_vasy_boom_3_50_100 adac_vasy_boom_3_50_100_boog_2

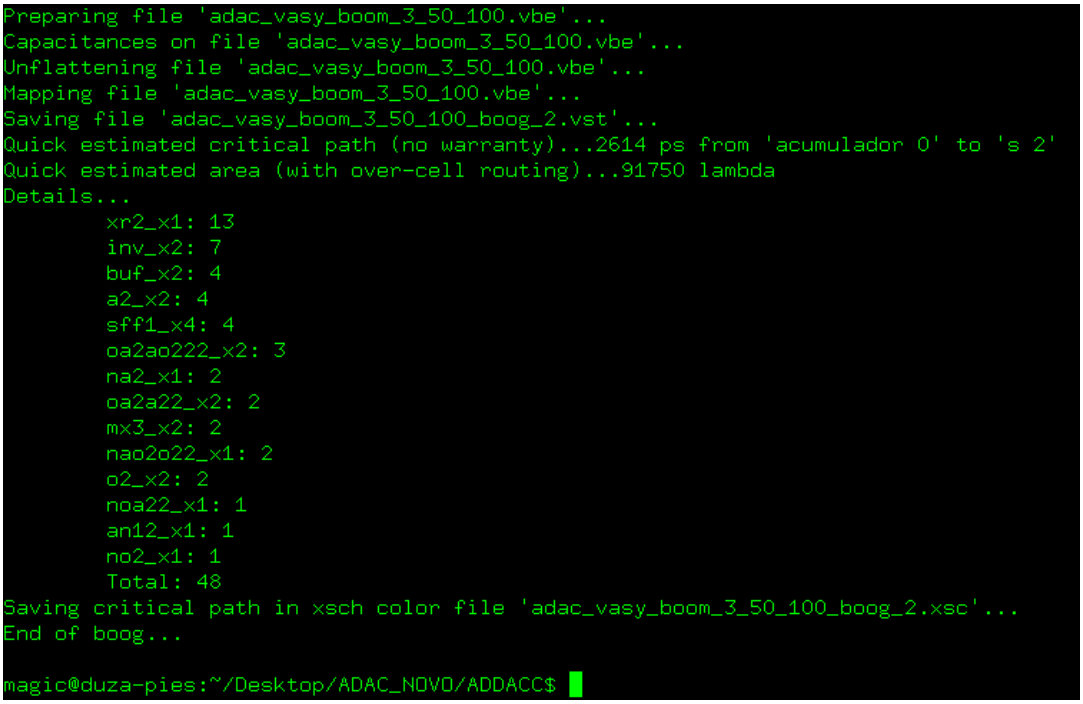

Figura 6: Utilizando a ferramenta Boog.

Ao finalizarmos o mapeamento, podemos utilizar a ferramenta Xsch para enxergarmos o circuito resultante. Esta visualização pode ser acompanhada abaixo, na figura 7:

Xsch –l ‘adac_vasy_boom_3_50_100_boog_2’&

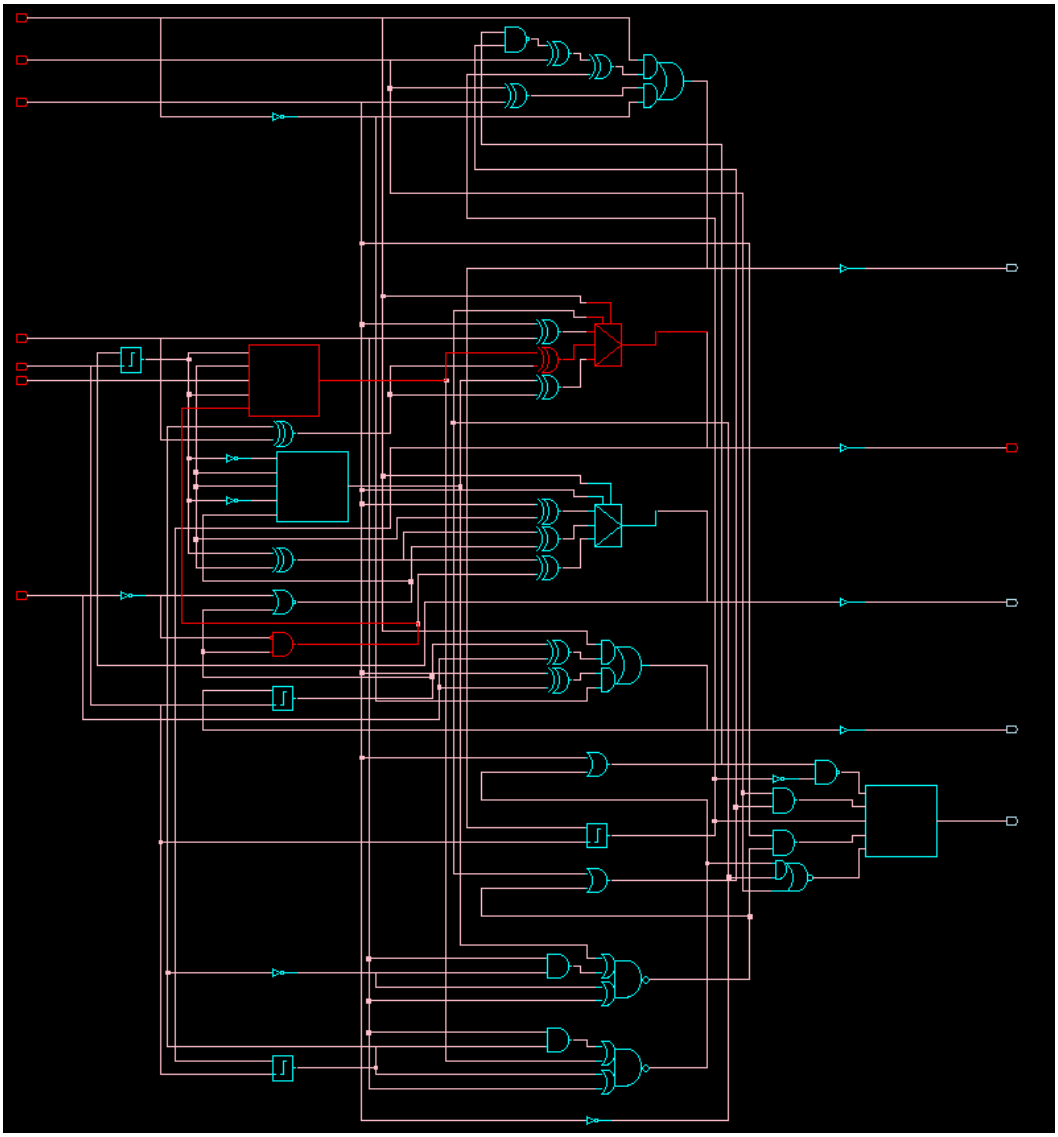

Figura 7: Modelo esquemático do adac_vasy_boom_3_50_100_boog_2.

Vale salientar que as representações em vermelho acima significam que o caminho percorrido nestas é mais longo entre a entrada e a saída do circuito e, como foi mencionado anteriormente, como utilizamos a biblioteca sxlib para a construção do modelo, as componentes podem ser vistas com a função identify, dentro do Xsch.

Prosseguindo, utilizaremos o Asimut, novamente, para verificar se continuamos dentro da proposta inicial:

Asimut –zd adac_vasy_boom_3_50_100_boog_2 adac_sem_atraso adac_vasy_boom_3_50_100_boog_2_res

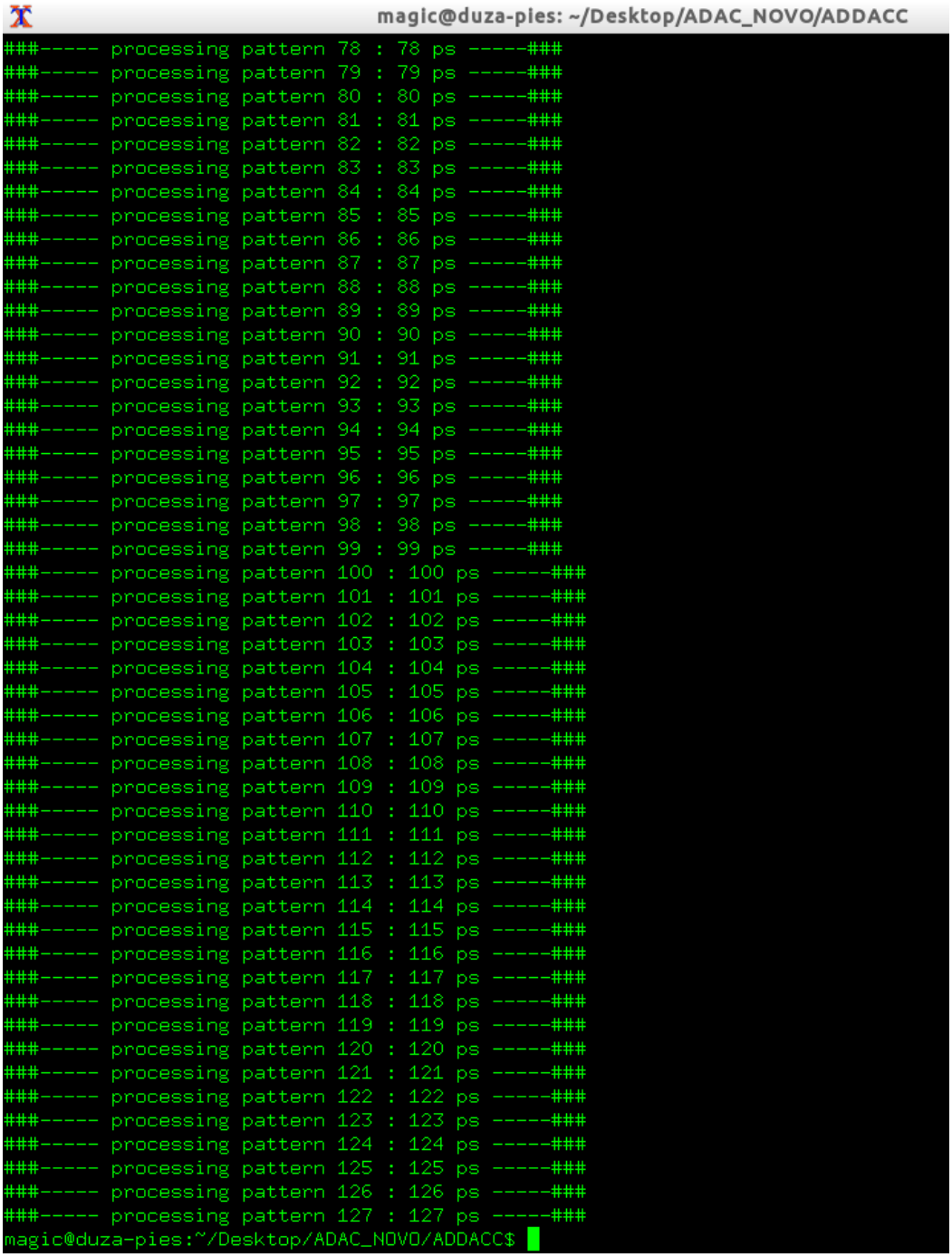

Figura 8: Resultado da simulação do adac_vasy_boom_3_50_100_boog_2_res com o comando –zd.

Como podemos ver na figura 8, o comando Asimut foi executado com sucesso. Foi necessário utilizar a extensão -zd para que a simulação da netlist não houvesse delay. Para analisarmos a simulação com delay, retiramos esta extensão e observamos a execução do Asimut abaixo:

Asimut adac_vasy_boom_3_50_100_boog_2 adac_sem_atraso adac_vasy_boom_3_50_100_boog_2_res

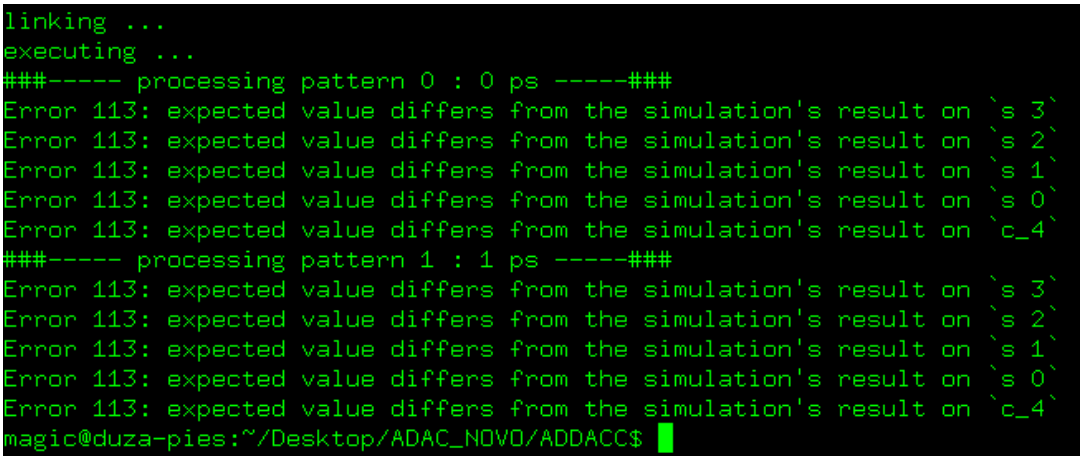

Figura 9: Resultado da simulação do adac_vasy_boom_3_50_100_boog_2_res sem o comando –zd.

Ao analisar a figura 9, percebemos que a simulação com delay gerou erros, que faz sentido, já que a descrição inicial em C não indica atrasos na sua sintaxe, portanto, é necessário que o Asimut retornasse erros nesta execução, validando o código fonte original.

Para fazermos a simulação com atraso, esta deve estar explicitada no código C, que será a próxima construção deste relatório.

3 Construção do ADAC com atraso

Continuando a construção de nosso projeto, utilizamos o arquivo adac_com_atraso.c, onde, pelo que o nome diz, é a versão com atraso do circuito que fizemos anteriormente. Este atraso foi feito dentro do arquivo C por meio de uma variável auxiliar descrita no código como “atraso” e o seu valor foi definida, para este código, um atraso de 13 mil.

Este valor deve ser verificado como um valor plausível durante a posterior execução do Asimut, após o uso do genpat, como fizemos na construção do ADAC sem atraso, anteriormente. Se retornar algum erro durante a execução do Asimut, pode ser que o valor de atraso escolhido não seja maior (ou menor) suficiente para a descrição feita no arquivo C (como aconteceu durante a aula do Asimut não funcionar para um atraso de 10 mil ou 15 mil, chegando, então, no atraso de 13 mil, onde não houveram erros na execução).

Alliance-genpat adac_com_atraso

Asimut adac_vasy_boom_3_50_100_boog_2 adac_com_atraso_13000 adac_vasy_boom_3_50_100_boog_2_13000_res

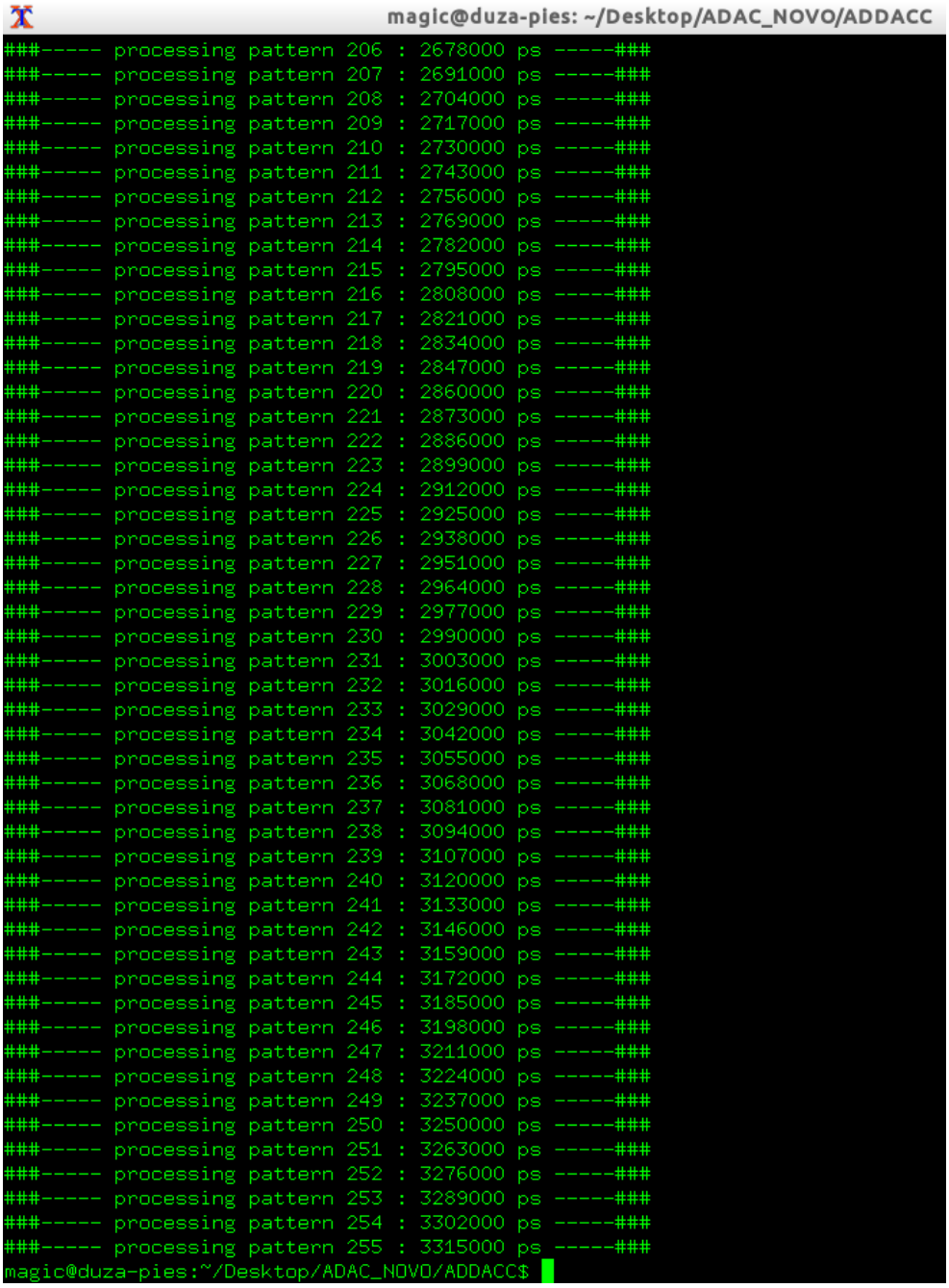

Figura 10: Resultado da simulação do adac_vasy_boom_3_50_100_boog_2_13000_res.

Após a verificação da integridade, utilizaremos o Loon. Esta ferramenta serve para otimizar os caminhos críticos mencionados quando comentamos sobre a figura 7, de forma que estes caminhos não estejam ou extrapolem o limite de tempo necessário. Vale dizer que, pelo nosso circuito ser relativamente pequeno, o uso do Loon pode piorar o desempenho do mesmo.

Loon –m 4 adac_vasy_boom_3_50_100_boog_2
adac_vasy_boom_3_50_100_boog_2_loon_4

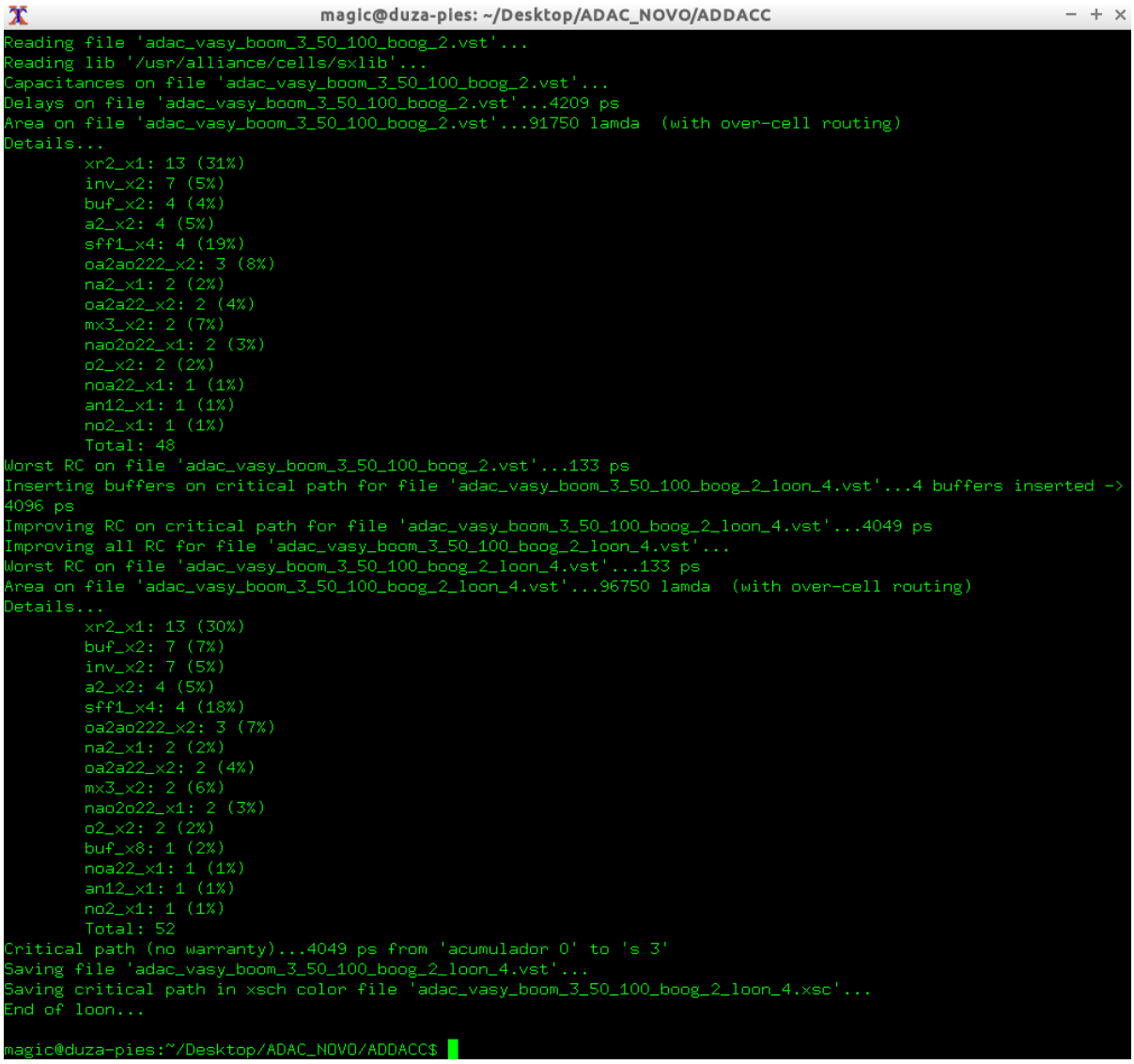

Figura 11: Resultado da otimização de componentes utilizando o Loon.

Pela figura 11, conseguimos perceber o aumento na quantidade de componentes do circuito com relação ao que era anteriormente, para tentar resolver esses problemas de caminho destacados. Essa alteração no circuito pode ser vista, a seguir, na figura 12:

Xsch –l ‘adac_vasy_boom_3_50_100_boog_2_loon_4’&

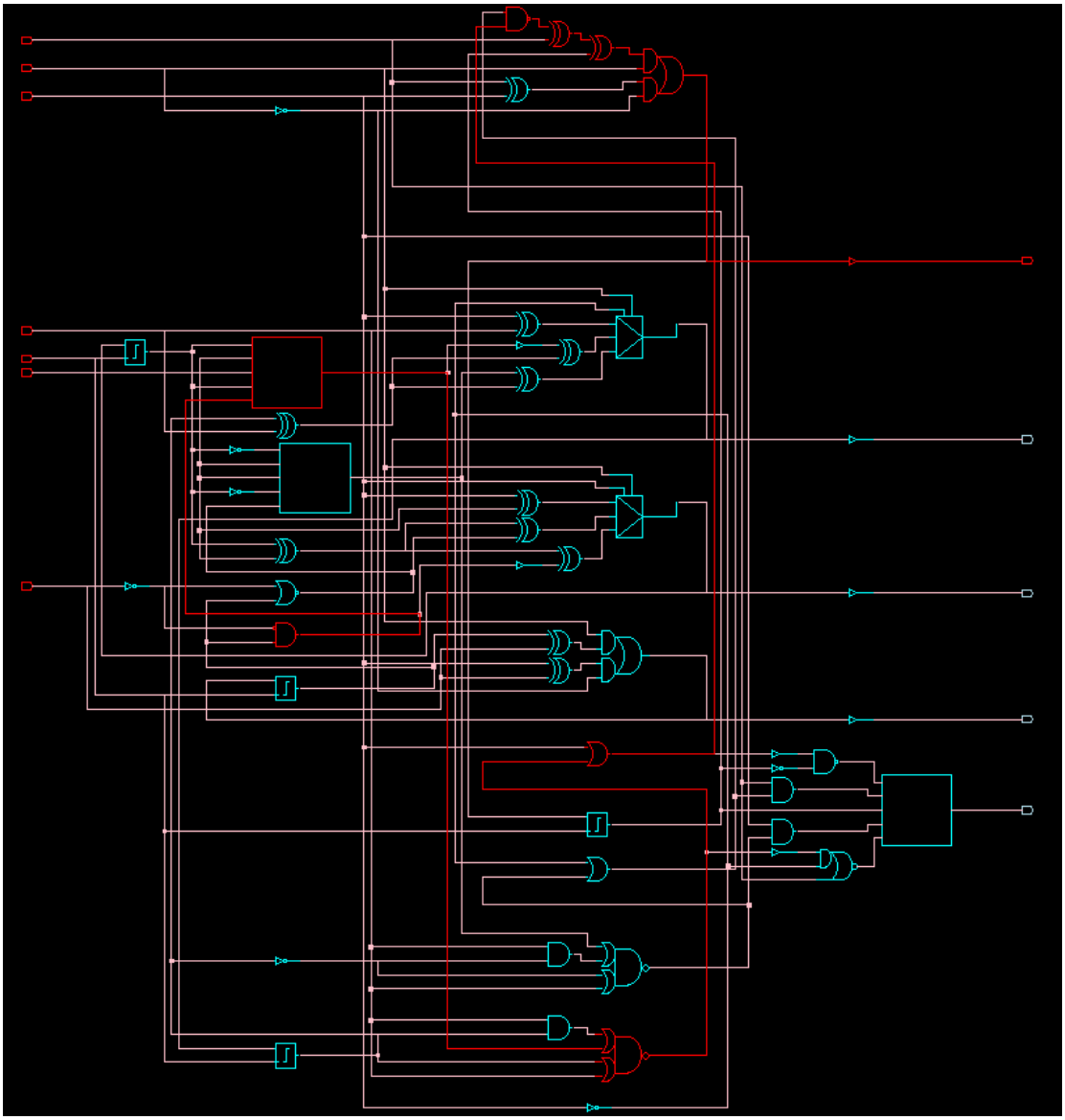

Figura 12: Modelo esquemático do adac_vasy_boom_3_50_100_boog_2_loon_4.

É possível ver o posicionamento dos buffers colocados nas saídas, além do aumento dos componentes. E, para mantermos a prática, utilizaremos, novamente, a ferramenta Asimut para que possamos confirmar se a mudança na estrutura feita pelo Loon afetou a integridade do projeto:

Asimut adac_vasy_boom_3_50_100_boog_2_loon_4 adac_com_atraso_13000 adac_vasy_boom_3_50_100_boog_2_loon_4_13000_res

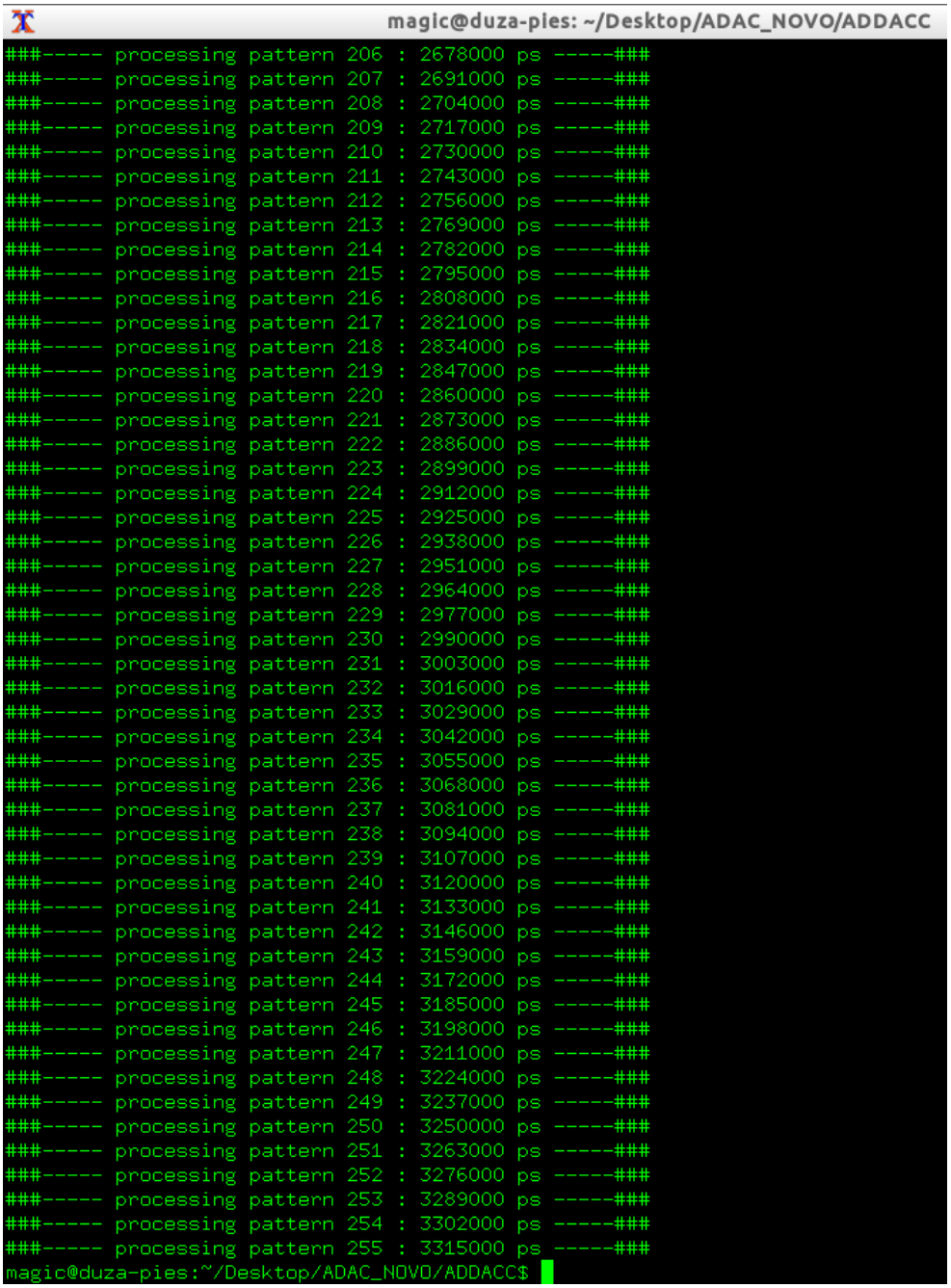

Figura 13: Resultado da simulação do adac_vasy_boom_3_50_100_boog_2_loon_4_13000_res.

Para que possamos ajudar a otimizar o roteamento e o posicionamento dos pads, no produto final, utilizaremos o arquivo adac.ioc, formato industrial que descreve os pinos de entrada e saída. Antes de fazer o ocp, ferramenta utilizada para posicionamento de standard cells utilizado em relatórios anteriores, fazemos o comando cp, para transferir o arquivo resultante de várias aplicações para um arquivo com nome menos extenso.

Diferentemente das utilizações anteriores, o comando ocp, neste caso, precisou de alguns parâmetros a mais para a construção do adac core. Nestes parâmetros, temos o número de rows (linhas) que foi definida com 6 (número testado por um monitor antigo da disciplina,

já que números diferentes geravam erros e, por algum motivo, 6 linhas fazia com que o comando funcionasse).

Além disso, como foi mencionado a pouco, passamos o arquivo adac.ioc como parâmetro do ocp para que o mesmo identifique as posições descritas nele, com intuito de melhorar a construção dos pads posteriormente.

Cp adac_vasy_boom_3_50_100_boog_2_loon_4.vst adac_core.vst

Alliance-ocp –rows 6 –ring –ioc adac adac_core adac_core_posicionado

Graal –l ‘adac_core_posicionado’&

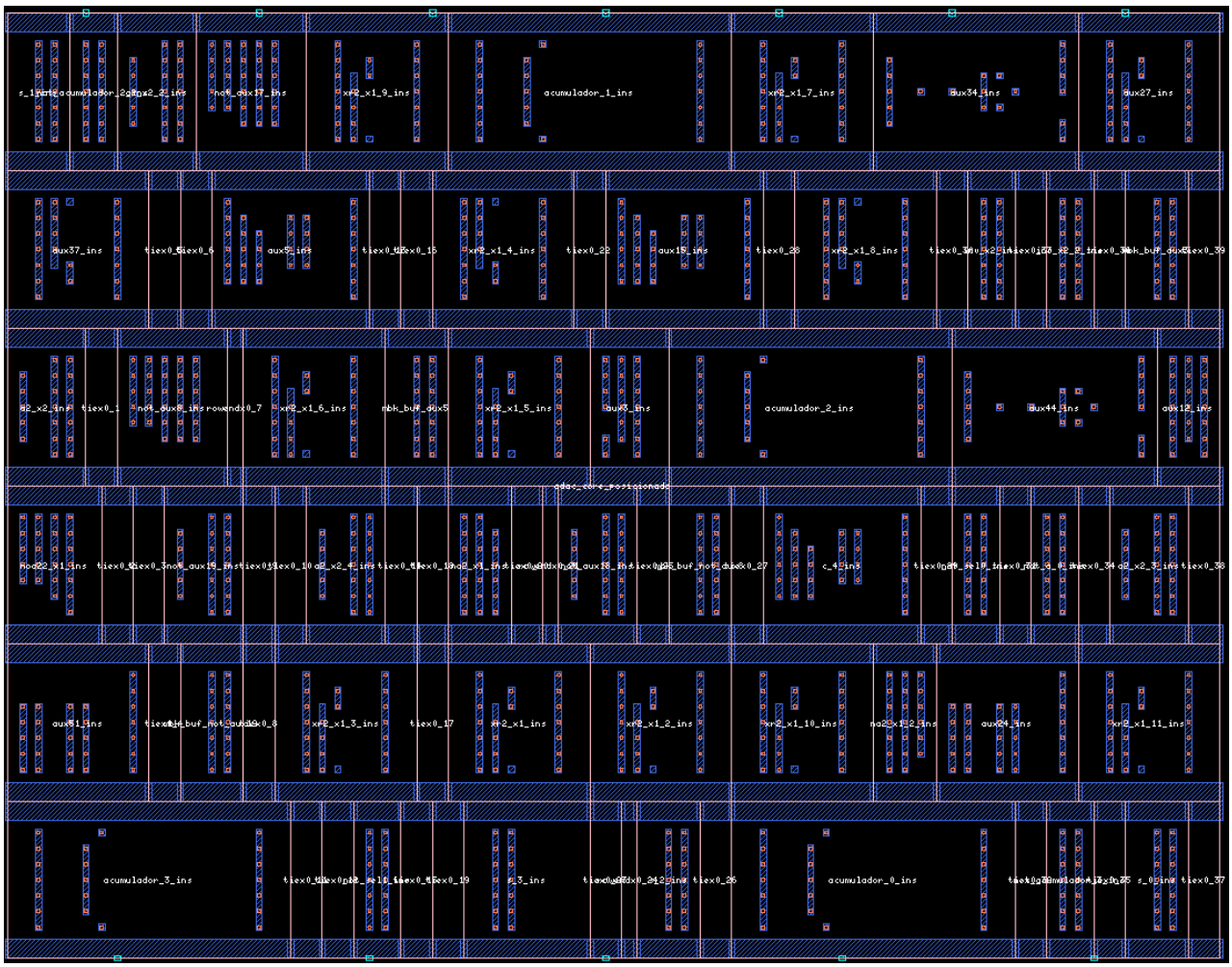

Figura 14: Modelo em standard cells do adac_core_posicionado.

Novamente com uma ferramenta já utilizada na disciplina, faremos o uso do Nero para criação do roteamento do circuito. Vejam que, na figura 15, o Nero se permitiu o uso de alumínio 2 na vertical e horizontal nos extremos do circuito, por saber que está passando por apenas alumínio 1 em vdd e vss, apesar de diferir das regras de desenho, a aplicação faz sentido, nesta situação específica.

Nero –p adac_core_posicionado adac_core adac_core_roteado

Graal –l ‘adac_core_roteado’&

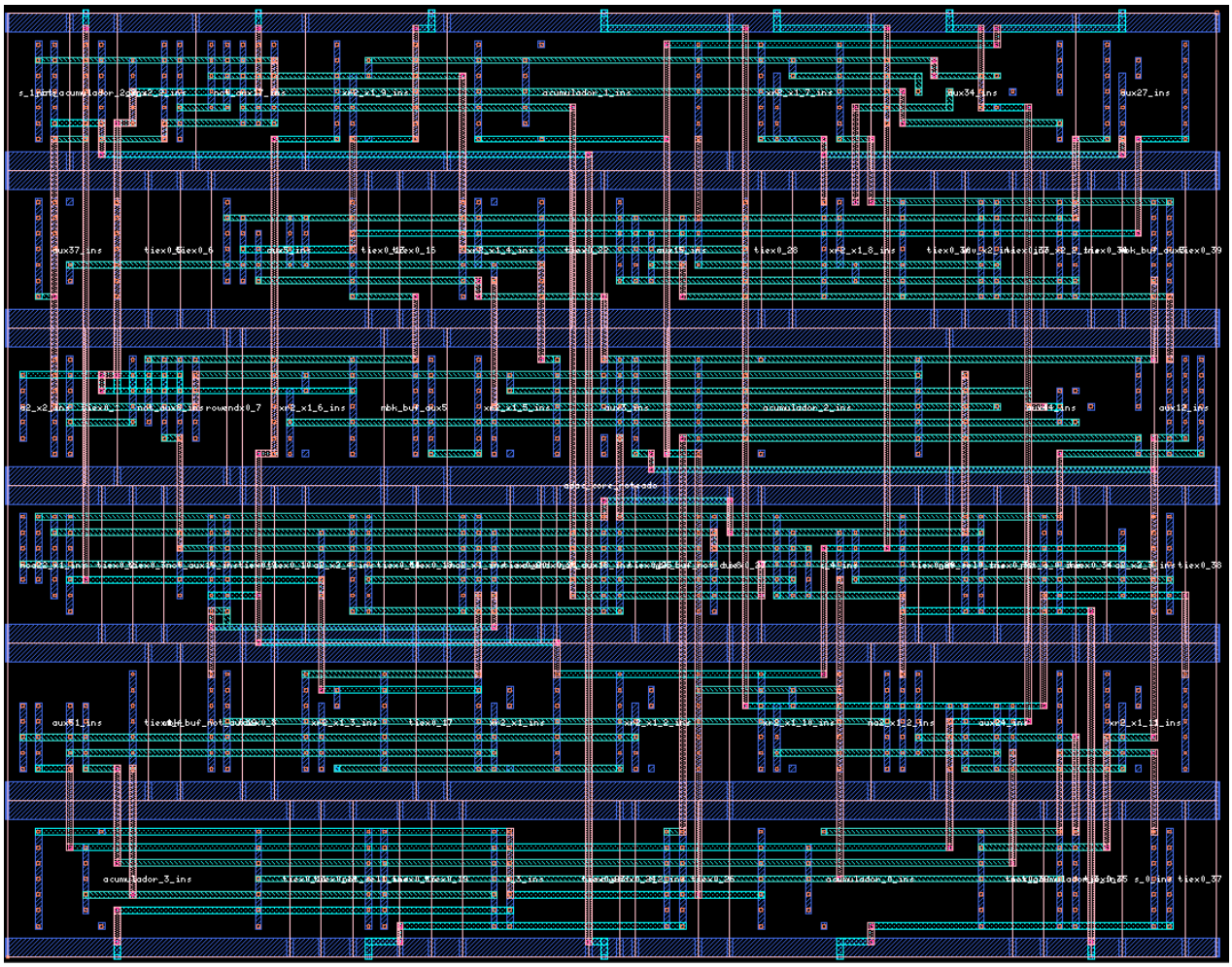

Figura 15: Modelo em standard cells do adac_core_roteado.

Em seguida, utilizamos o Cougar, também utilizado no passado. Neste caso, não utilizamos as extensões -t, -ar ou –ac, já que o uso destes causaria uma extração de muita capacitância ou estouraria as resistências, que não é de nosso interesse.

Após a extração, utilizaremos o LVX (layout versus extracted), para comparar as netlists e ver se o modelo de silício bateu com a netlist de entrada. Este procedimento pode ser visto na figura 16, logo abaixo:

Export MBK_OUT_LO=al

Cougar adac_core_roteado adac_core_roteado_extra

Lvx al vst adac_core_roteado_extra adac_core

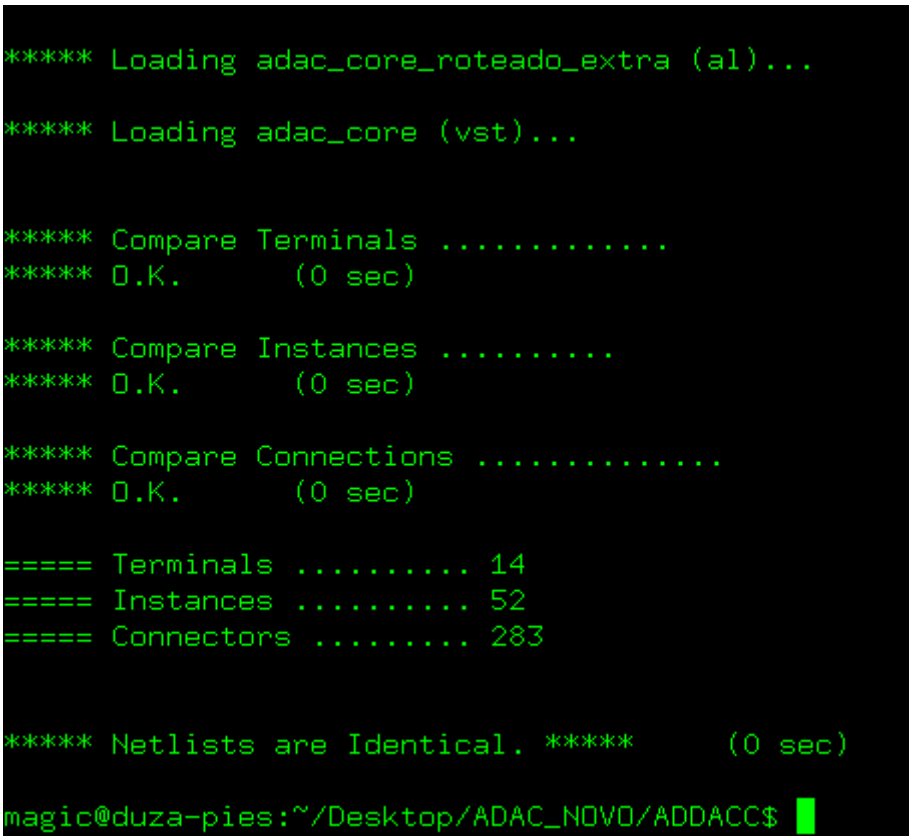

Figura 16: Resultado da comparação entre as netlists utilizando o LVX.

Novamente, utilizaremos o Asimut para simular o modelo roteado:

Export MBK_IN_LO=al

Asimut adac_core_roteado_extra adac_com_atraso_13000 adac_core_roteado_extra_13000_res

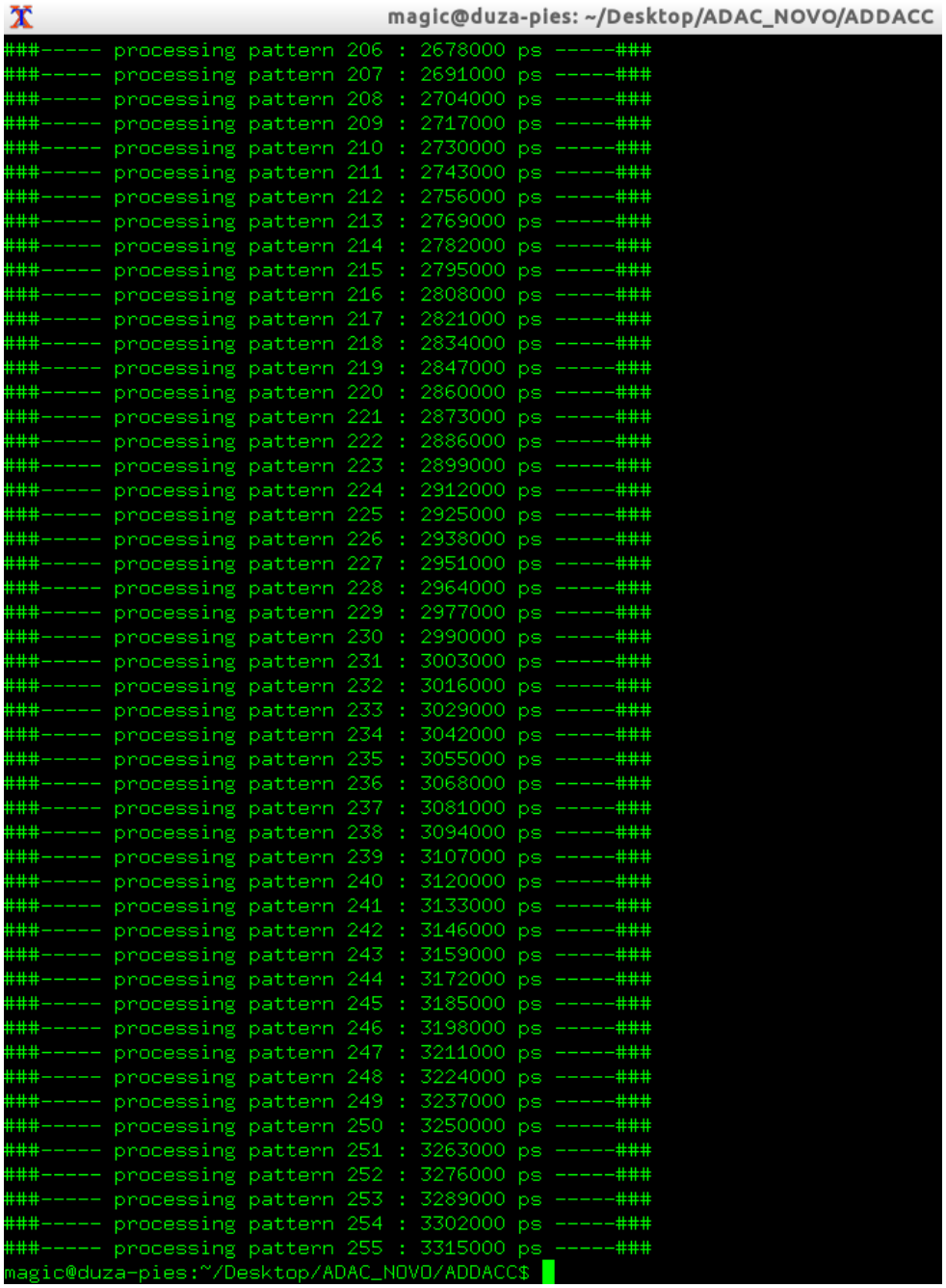

Figura 17: Simulação do adac_core_roteado_extra_13000_res.

Em seguida, trabalharemos com outro arquivo disponibilizado pelo professor, o adac_chip.c, assim, geraremos a netlist do chip (core + pads) a partir dele, por meio do genlib.

Export MBK_IN_LO=vst

Export MBK_OUT_LO=vst

Genlib adac_chip

Xsch –l ‘adac_chip’&

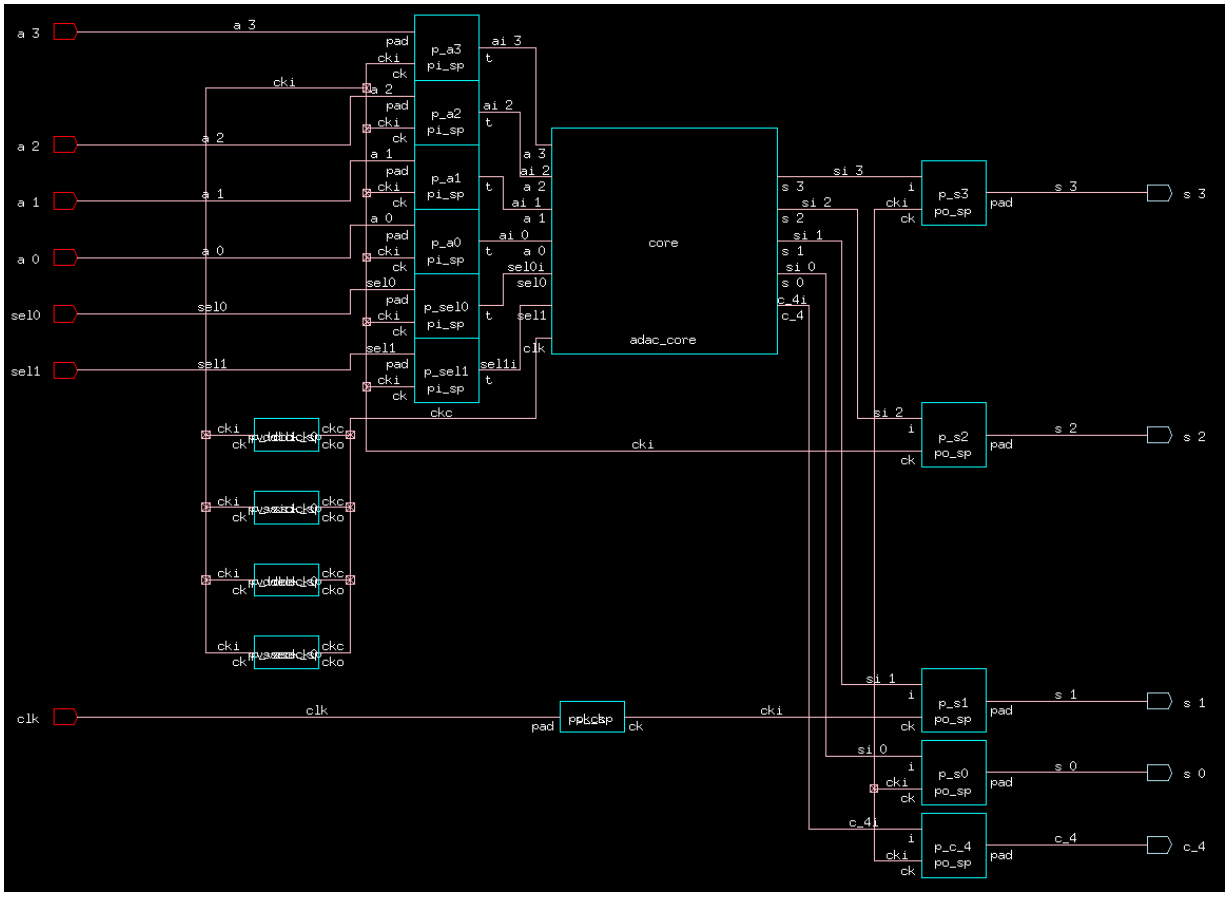

Figura 18: Formato esquemático do chip.

Na figura 18, podemos ver a adição dos pads descritos no adac_chip e, se utilizarmos o "Go down" dentro do Xsch, no core, poderemos enxergar o mesmo circuito da figura 14. Após analisar a representação, voltamos a abrir o adac core roteado, que pode ser visto, novamente, na figura 19, e o salvaremos como adac_core, para podermos passar para a ferramenta Ring, que será explicado posteriormente.

Da mesma forma, se não salvarmos como, dentro do Xsch, teríamos que alterar a primeira linha, editando o arquivo. De qualquer forma, o critério é de quem está construindo, então, existem essas duas formas para prepararmos o uso do Ring.

Graal –l 'adac_core_roteado'&

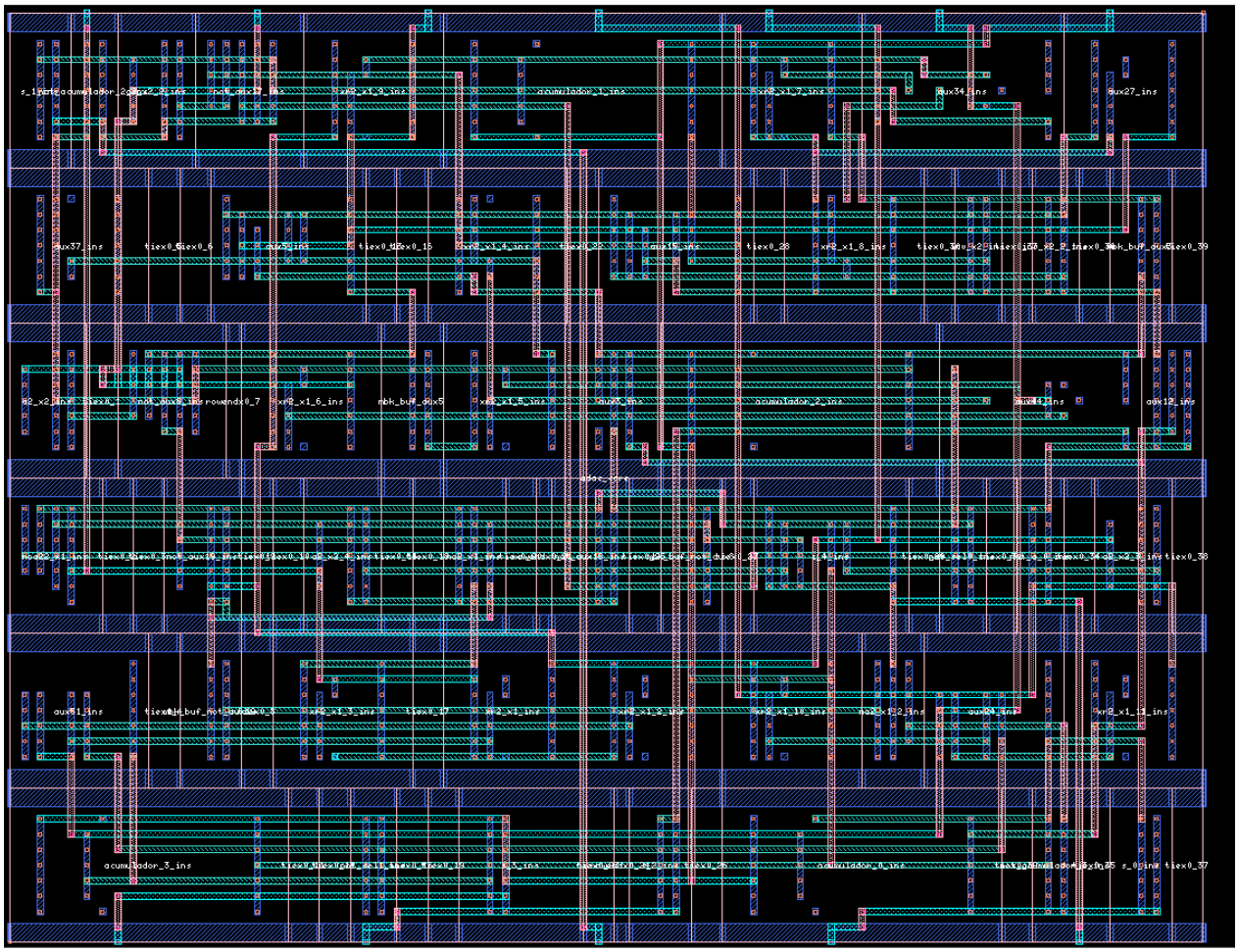

Figura 19: Modelo em standard cells do adac_core_roteado, agora chamado adac_core.

Quanto à ferramenta Ring, ela, basicamente, irá criar o layout final do chip, após o core estar roteado e os pads inseridos. Se o trabalho foi feito de forma correta, o Ring retornará a mensagem “lucky, no error” juntamente do que o mesmo faz durante a execução. Esta situação pode ser verificada na figura 20, logo abaixo:

Ring adac_chip adac_chip

Figura 20: Criação do layout final com core roteado e pads.

Abaixo, podemos ver o modelo, em standard cells, do formato final do nosso chip, com seu core no meio e os pads nas extremidades:

Graal –l ‘adac_chip’&

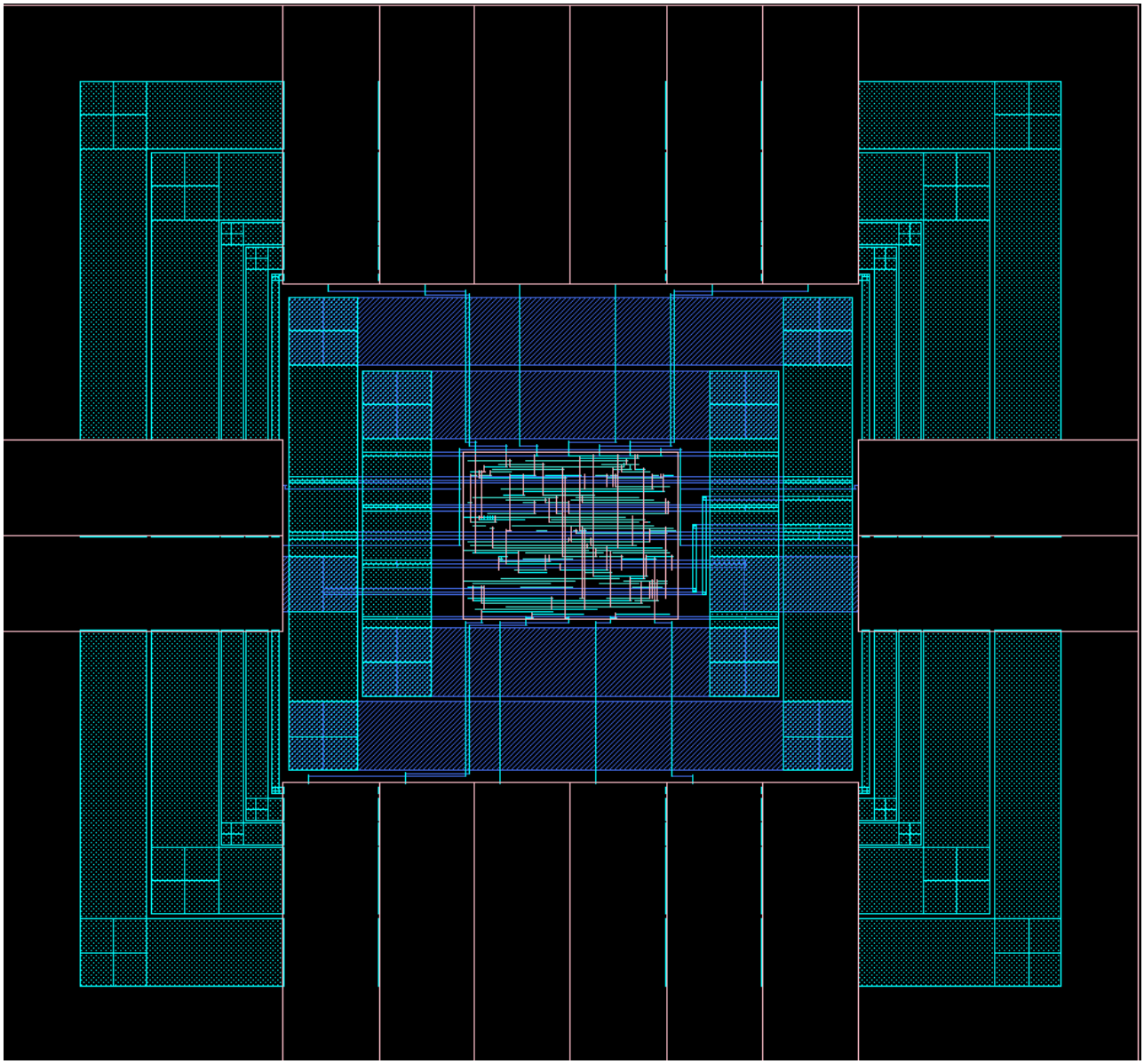

Figura 21: Modelo em standard cells do adac_chip.

Para finalizar a construção do nosso chip, faremos as últimas simulações e comparações com uso das ferramentas, para garantir que está tudo dentro dos conformes. Primeiramente, extraímos o chip utilizando o Cougar e, em seguida, utilizamos, novamente, o LVX, para compararmos as netlists:

Export MBK_OUT_LO=al

Cougar adac_chip adac_chip_ex

Lvx al vst adac_chip_ex adac_chip

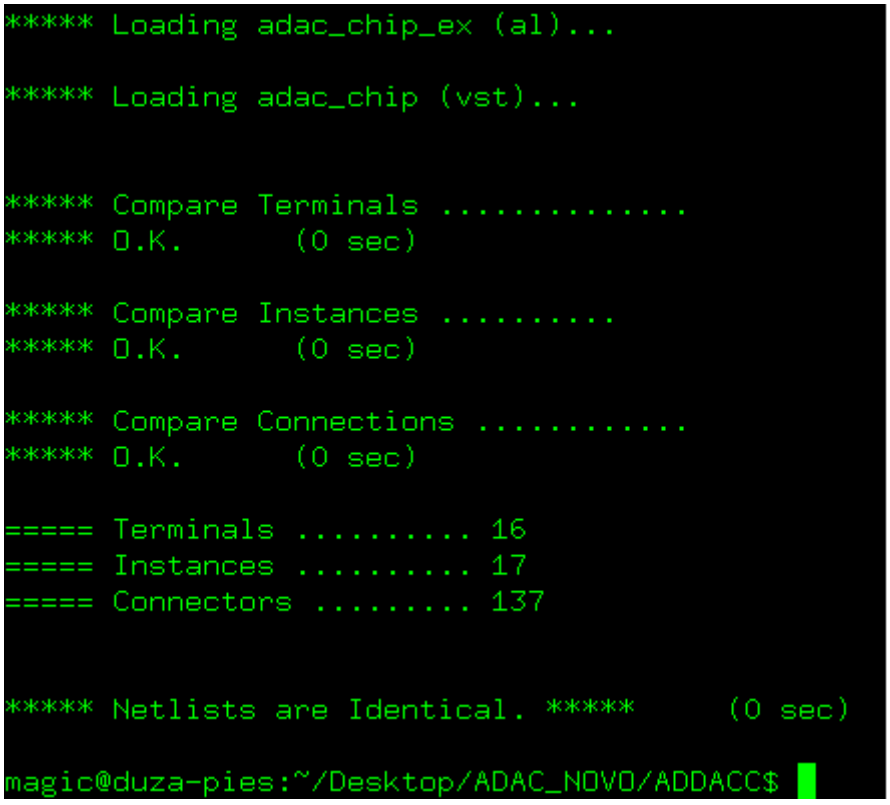

Figura 22: Resultado da compração entre as netlists utilizando o LVX.

Agora, utilizaremos a ferramenta x2y para, basicamente, criar uma cópia chamada adac_core.al a partir do adac_core.vst. O uso desta ferramenta foi necessário para manter todas as netlists, na hora da simulação, em uma extensão compatível. Como começamos no uso de arquivos.al, usamos o x2y para fazer a simulação hierárquica, novamente, utilizando o Asimut, cuja execução pode ser vista abaixo:

Export MBK_IN_LO=al

X2y vst al adac_core adac_core

Asimut adac_chip_ex adac_com_atraso_13000 adac_chip_ex_res_13000

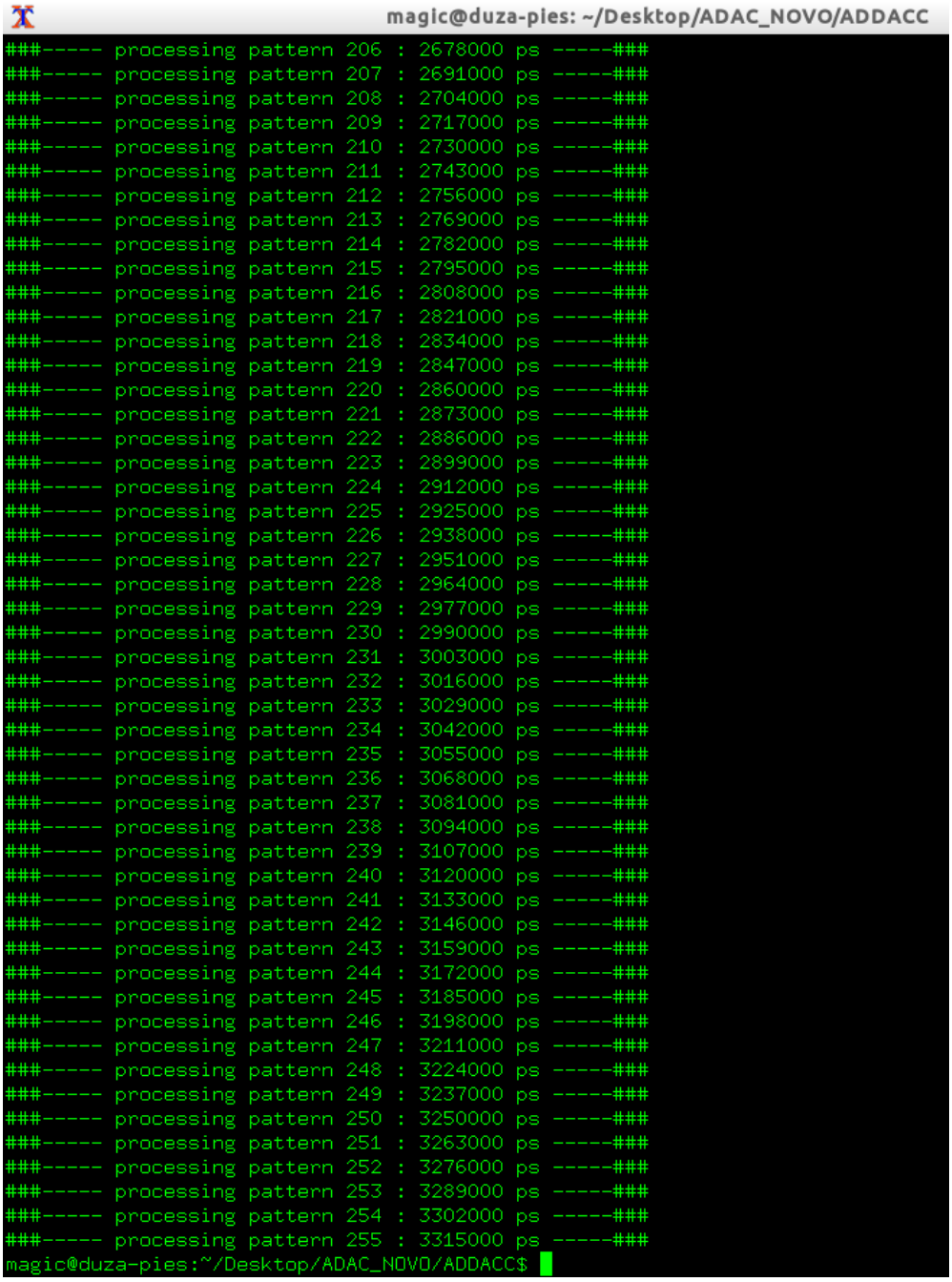

Figura 23: Simulação final do adac_chip_ex_res_13000.

Finalmente, com a última verificação do Asimut feita sem erros, criamos, portanto, o nosso chip adac com sucesso, respeitando o passo a passo descrito pelo professor.

4 Conclusão

Este projeto, além de complementar os assuntos e ferramentas introduzidos durante do andamento da disciplina, serviu para, também, nos introduzir ao conceito de verificação

funcional hierárquica, assunto que será aprofundado de uma forma maior durante a disciplina de Concepção Estruturada de Circuitos Integrados.

Desta forma, a construção deste chip servirá muito bem como porta de entrada para o que veremos futuramente, agora com uma bagagem significativa quanto aos conceitos da microeletrônica que foram utilizados na construção. Conseguimos testemunhar e aplicar, com êxito, as ferramentas disponibilizadas a nós, de forma que estamos bem introduzidos à construção de chips que poderão ser aplicados em ambientes de serviços.