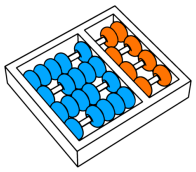
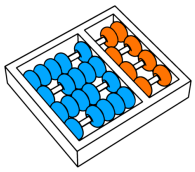


Marcelo Guedes Silva

"Uma abordagem em ArchC para caracterização e desenvolvimento de processadores em nível de arquitetura"

CAMPINAS
2012



Universidade Estadual de Campinas
Instituto de Computação

Marcelo Guedes Silva

"Uma abordagem em ArchC para caracterização e desenvolvimento de processadores em nível de arquitetura"

Orientador(a): **Prof. Dr. Rodolfo Jardim de Azevedo**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.

Este exemplar corresponde à versão final da Dissertação defendida por Marcelo Guedes Silva, sob orientação de Prof. Dr. Rodolfo Jardim de Azevedo.

Assinatura do Orientador(a)

CAMPINAS
2012

FICHA CATALOGRÁFICA ELABORADA POR
ANA REGINA MACHADO - CRB8/5467
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E
COMPUTAÇÃO CIENTÍFICA - UNICAMP

Guedes, Marcelo, 1985-
G934a Uma abordagem em ArchC para caracterização e desenvolvimento de processadores em nível de arquitetura / Marcelo Guedes Silva. – Campinas, SP : [s.n.], 2012.

Orientador: Rodolfo Jardim de Azevedo.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Simulação (Computadores digitais). 2. Arquitetura de computador. 3. Energia - Consumo. 4. Sistemas embutidos de computador. I. Azevedo, Rodolfo Jardim de,1974-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

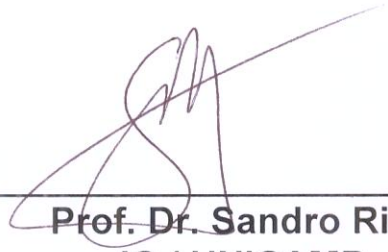
Informações para Biblioteca Digital

Título em inglês: An ArchC approach for characterization and development of processors in architecture level
Palavras-chave em inglês:
Digital computer simulation
Computer architecture
Energy consumption
Embedded computer systems
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Rodolfo Jardim de Azevedo [Orientador]
Fabiano Passuelo Hessel
Sandro Rigo
Data de defesa: 17-12-2012
Programa de Pós-Graduação: Ciência da Computação

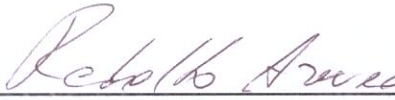
TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 17 de Dezembro de 2012, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Fabiano Passuelo Hessel
PUC-RS



Prof. Dr. Sandro Rigo
IC / UNICAMP



Prof. Dr. Rodolfo Jardim de Azevedo
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Uma abordagem em ArchC para caracterização e desenvolvimento de processadores em nível de arquitetura

Marcelo Guedes Silva

17 de Dezembro de 2012

Banca Examinadora:

- Prof. Dr. Rodolfo Jardim de Azevedo (Supervisor/*Orientador*)
- Prof. Dr. Fabiano Passuelo Hessel
  Faculdade de Informática - PUC-RS
- Prof. Dr. Sandro Rigo
  Instituto de Computação - UNICAMP
- Prof. Dr. Paulo Centoducatte - Suplente
  Instituto de Computação - UNICAMP
- Prof. Dr. Alexandro Baldassin - Suplente
  Departamento de Estatística, Matemática Aplicada e Computação - UNESP

Abstract

This work presents acSynth, an integrated framework for development and synthesis based on ArchC ADL descriptions. In its first application, acSynth includes caracterization tools to allow power consumption analysis for supported processor architectures, through Tiwari's method. The power analysis and characterization tools were achieved by integrating PowerSC, acPower and acPowerGen, allowing acSynth to gather, process and store power consumption data in order to create power reports. This data could then be used in acSim simulations, generating ADL level power analysis reports automatically. We show characterization results for MIPS-I Plasma processor and SPARCv8 Leon3 processor using two different synthesis tools and workflows, Altera and Xilinx. The processors were tested with acStone, Mibench and Mediabench benchmarks, generating power reports and energy consumption profile graphs with energy per time data. We analysed the error comparing to RTL simulations. The analysis with MIPS-I and Xilinx tool set presented effective error between 0.02% and 61.05%, with 91% of the total number of analised cases presenting errors with less than or equal to 30%. Adopting Altera tool set, the effective error was between 0.01% and 17.49% with 96% of the total number of analised cases showing error with less than or equal to 15%. For SPARCv8 architecture, using Xilinx tool set, the effective error ranged between 0.14% and 40.66% with 95% of the total number of analised cases presenting errors with less than or equal to 20%. Furthermore, a MIPS-I pipelined processor was developed using the acRTL workflow. The complete development is detailed in this dissertation, highlighting the method advantages and disadvantages. The new processor power consumption data was collected and an acSynth power database generated. Finally, power, area and performance was investigated and compared to the stable processor Plasma. The main contributions of the present dissertation are: ArchC tool set integration showing the benefits in high level analysis; introduction of a new power characterization method in architecture level, expanding ArchC environment; design of a practical method to expand the acSim analysis and behavior, covering new high level simulation aspects; the practical use of acRTL.

Resumo

A dissertação apresenta acSynth, um conjunto de ferramentas integradas que tem por objetivo fornecer uma plataforma aberta de desenvolvimento e síntese de projetos a partir de descrições em ADL ArchC. Como primeiro trabalho, acSynth foi equipado com ferramentas para caracterização de consumo de energia de processadores através do método Tiwari. Isto foi concretizado através da composição das ferramentas PowerSC, acPower e acPowerGen, capacitando acSynth a obter e armazenar informações de consumo de energia. Estes dados podem, então, ser utilizados em simulações em acSim, com geração automática de relatórios em nível ADL. Após a caracterização, é possível distribuir as informações coletadas para evitar reexecutar o fluxo para as mesmas ferramentas e processadores. O trabalho apresenta resultados de caracterização dos processadores MIPS-I Plasma e SPARCv8 Leon3, bem como integração com as ferramentas de síntese da Altera e da Xilinx. Os processadores foram submetidos a testes com os *benchmarks* acStone, Mibench e Mediabench, com elaboração de relatórios de consumo de energia e gráficos de perfil energético no tempo. Um estudo do erro da caracterização foi apresentado. Para testes com MIPS-I o erro efetivo sobre plataforma Xilinx variou de 0,02% a 61,05%, com 91% dos casos com erro menor ou igual a 30%. Em plataforma Altera o erro efetivo variou de 0,01% a 17,49% com 96% dos casos com erro menor ou igual a 15%. Para testes com SPARCv8 em plataforma Xilinx o erro efetivo variou de 0,14% a 40,66% com 95% dos casos com erro menor ou igual a 20%. Adicionalmente, desenvolveu-se um processador MIPS-I *pipelined* através do fluxo da ferramenta acRTL. Um histórico do processo com detalhes dos prós e contras é apresentado. Um arquivo com dados de consumo de energia das instruções suportadas foi elaborado. Por fim, energia, área e desempenho foram estudados e comparados ao processador Plasma. As principais contribuições deste trabalho são: interconexão de ferramentas e mostra dos benefícios obtidos com isto; apresentação de uma abordagem de caracterização de consumo de energia de processadores no nível de arquitetura; demonstração de um método funcional para expansão de acSim para abarcar novos aspectos de simulação em alto nível; aplicação prática de acRTL.

Agradecimentos

Acima de tudo, agradeço a Deus, Força Maior que tudo rege.

Agradeço aos meus pais, Mayra do Couto da Silva e Manoel Guedes da Silva, que tornam tudo na minha vida possível pelo apoio constante e incondicional. De um sonho aqui cheguei.

Agradeço a toda equipe do LSC. Rodolfo Azevedo, professores e funcionários, que cedo me ensinaram os primeiros passos no campo da microeletrônica. Sou, por assim dizer, um dos muitos filhos do Brazil-IP.

Agradeço também a todos que direta ou indiretamente contribuíram para este trabalho com palavras de suporte, ideias, atenção, paciência e compreensão. Os nomes são muitos, de amigos de toda a parte que me acompanharam nestes anos.

Gostaria de agradecer especialmente àqueles que forneceram as bases teóricas e o auxílio direto para que esta dissertação se tornasse realidade: Sandro Rigo, Felipe Klein, Liana Duenha, Ricardo Borin, Rafael Auler, Samuel Goto, Josué Ma, Marcelo Matsumoto.

Sem a ajuda e o conhecimento compartilhado de tantos a mim, nem mesmo uma linha poderia escrever.

Acrônimos e Abreviações

ADL Architecture Description Language.

ALU Arithmetic Logic Unit.

ASIC Application-Specific Integrated Circuit.

ASIP Application Specific Instruction Set Processor.

CAD Computer Aided Design.

CDFG Control Data Flow Graph.

CSV Comma Separated Value.

DSP Digital Signal Processor.

DUV Design Under Verification.

EPI Energy Per Instruction.

FPGA Field-Programmable Gate Array.

HDL Hardware Description Language.

IOB Input/Output Buffer.

IPC Instruction Per Cycle.

ISA Instruction Set Architecture.

JVM Java Virtual Machine.

LUT Look-Up Tables.

MMU Memory Management Unit.

MPSoC Multi Processor System-On-Chip.

RTL Register-transfer level.

RTOS Real Time Operating System.

SAIF Switching Activity Interchange Format.

SDL System Description Language.

TCL Tool Command Language.

VCD Value Change Dump.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Códigos

Capítulo 1

Introdução

A elevação do nível de abstração para desenvolvimento de circuitos integrados é uma tendência global. A complexidade cada vez maior exige técnicas que automatizem e facilitem tarefas padrões dos desenvolvedores, permitindo acompanhar o rápido crescimento e a alta demanda do setor.

Em alinhamento com tal tendência, esta dissertação apresenta a plataforma de desenvolvimento **acSynth**. Trata-se de um conjunto de ferramentas de alto nível integradas que objetiva fornecer uma plataforma aberta de desenvolvimento e síntese de projetos a partir de descrições de processadores em ADL ArchC. Um esquemático geral do funcionamento de acSynth pode ser visto na figura 1.1.

A plataforma acSynth permitiu a caracterização, geração de relatórios de consumo de energia e estudo de processadores elaborados por terceiros em duas diferentes arquiteturas suportadas por modelos ArchC: MIPS-I e SPARCv8.

Adicionalmente, foi possível analisar o perfil de consumo de energia de um processador elaborado através do fluxo de acRTL. Os dados obtidos puderam então ser comparados com outro processador de mesma arquitetura. Com isto, pode-se analisar a eficiência energética de um processador gerado por acRTL.

Os experimentos demonstraram um ganho significativo no tempo de simulação de consumo de energia, permitindo a elaboração de relatórios através de ArchC que seriam virtualmente impossíveis com o uso apenas de ferramentas de análise em RTL. O ganho de tempo foi representativo em todos os experimentos.

Testes com o *benchmark* acStone apresentaram erros comparativos entre o método utilizando a plataforma acSynth e o método tradicional em RTL.

Para testes com MIPS-I, foi caracterizado o processador Plasma. O erro efetivo em relação a análise com ferramentas Xilinx variou de 0,02% a 61,05%, com 91% dos casos com erro menor ou igual a 30%. O erro efetivo em relação a análise com ferramenta Altera variou de 0,01% a 17,49% com 96% dos casos com erro menor ou igual a 15%.

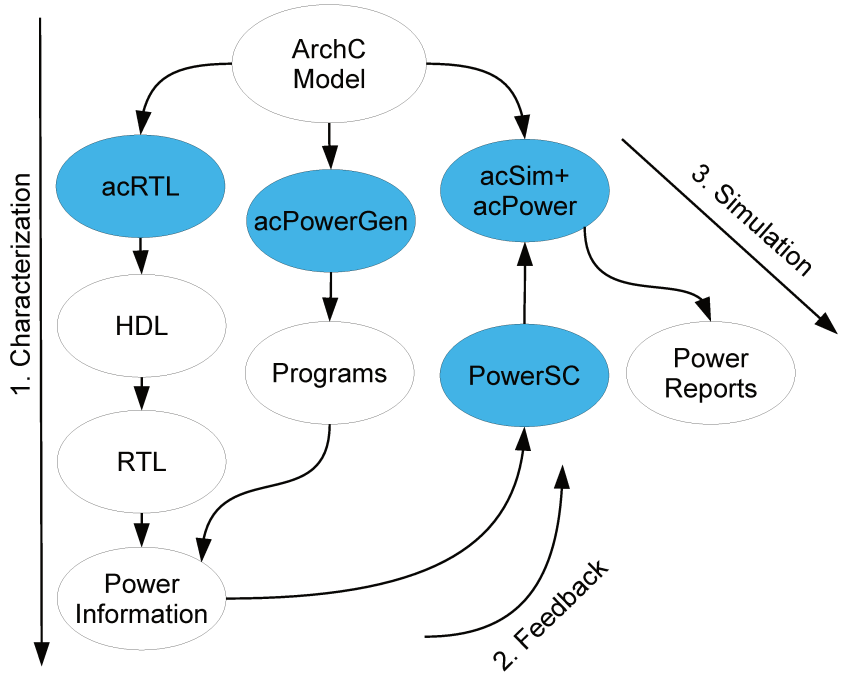


Figura 1.1: Esquemático da plataforma de desenvolvimento acSynth

Para testes com SPARCv8, foi caracterizado o processador Leon3. O erro efetivo em relação a análise com ferramenta Xilinx variou de 0,14% a 40,66%, com 95% dos casos com erro menor ou igual a 20%.

Os erros relacionados a instruções de multiplicação e divisão não foram incluídos pois recaíram em um subcaso especial que será detalhado nos resultados experimentais. A análise demonstrou que para reduzir os erros de análise é preciso aprimorar o modelo ArchC com dados de consumo de acesso de cache e informações de casos de *stall* do *pipeline*. Apesar desta limitação fora do escopo do presente trabalho, acSynth foi capaz de gerar relatórios com uma distribuição de erro consistente.

1.1 Contribuições

Este trabalho apresenta como principais contribuições:

- **Integração das ferramentas ArchC PowerSC, acPower, acPowerGen e acSim**, permitindo a concatenação de dados até então desconexos, trazendo novas informações relevantes de potência ao estudo de processadores em ArchC;
- **Apresentação de uma metodologia de caracterização de processadores sob plataforma acSynth**, compatível com qualquer arquitetura suportada por

um modelo ArchC e adaptável a dados provenientes de qualquer fonte, não limitado a uso de um fluxo específico de análise de consumo de energia;

- **Exemplificação com aplicações práticas de teste sobre processadores MIPS-I Plasma e SPARCv8 Leon3**, demonstrando a eficácia de acSynth para situações no mundo real, sobre projetos acadêmicos e comerciais de terceiros;

- **Apresentação de métodos de expansão de acSim para abarcar novos aspectos de simulação em alto nível**, que podem ser utilizados como um conceito para a elaboração de outras expansões com mais análises pertinentes em nível ADL;

- **Demonstração do uso de acRTL na elaboração de um MIPS-I *pipelined* funcional**, apresentando resultados práticos do uso da ferramenta com exposição das vantagens e dificuldades atuais do processo;

- **Caracterização de um processador acRTL novo através do fluxo sobre plataforma acSynth**, corroborando a compatibilidade do fluxo apresentado na plataforma acSynth com diferentes fluxos de projeto, desenvolvimento e ferramentas de análise de energia.

1.2 Publicações

A execução das tarefas deste trabalho permitiram a submissão de dois artigos científicos internacionais, a citar:

1. *"An ArchC approach for automatic energy consumption characterization of processors"* [2], submetido a *Rapid Speed Prototyping Symposium*, IEEE, em colaboração com Rafael Auler, Ricardo Borin e Rodolfo Azevedo. Este trabalho apresentou os primeiros resultados concretos do uso do fluxo completo da plataforma acSynth com caracterização e análise de dados do processador MIPS-I Plasma. Foram apresentados estudos do erro e comparativos de tempo de simulação entre análise tradicional em RTL e o fluxo sobre acSynth.

2. *"ESLBench: A benchmark suite for evaluating electronic system level tools and methodologies"* [3], submetido a *Embedded Systems Letters*, IEEE, tendo como autora principal Liana Duenha, em colaboração com Matheus Boy e Rodolfo Azevedo. Neste trabalho, acSynth contribuiu com os dados de consumo de energia de núcleos de processamento SPARCv8, permitindo a elaboração de relatórios e gráficos de perfil energético de sistemas *multicore* em *benchmarks* Mibench e Mediabench. Este

trabalho foi adicionalmente representativo pois demonstrou a facilidade de integração de acSynth com outras ferramentas ArchC previamente não suportadas, caso de ESLBench.

1.3 Estrutura do texto

A dissertação é dividida em quatro capítulos principais. O capítulo 2 apresenta uma revisão bibliográfica dos trabalhos pertinentes ao estudo para elaboração de acSynth, especialmente ferramentas de análise e simulação em alto nível. O capítulo 3 apresentar a fundo o desenvolvimento da plataforma acSynth, seu uso, funcionamento e detalhes técnicos e práticos. O capítulo 4 apresenta os resultados da aplicação da caracterização de consumo de energia implementada em acSynth sobre dois processadores de arquiteturas diferentes, Plasma e Leon3. Neste capítulo ainda, são apresentadas as tabelas e gráficos com os dados gerados por acSynth na execução dos *benchmarks* acStone, Mibench e Mediabench. Por fim, o capítulo 5 encerra o trabalho destacando os principais resultados da dissertação e as considerações finais.

Capítulo 2

Trabalhos Relacionados

Há um esforço, tanto da indústria quanto da academia, para elevação do nível de abstração no fluxo de desenvolvimento *System on Chip*. É uma reação ao crescente *gap* de produtividade, ocasionado pela demanda exponencial de complexidade de projetos, em contraste com a habilidade limitada dos desenvolvedores em lidar com ainda mais informação [1,4]. A figura 2.1 apresenta graficamente as implicações deste fenômeno. A análise toma como base a Lei de Moore para estimativas.

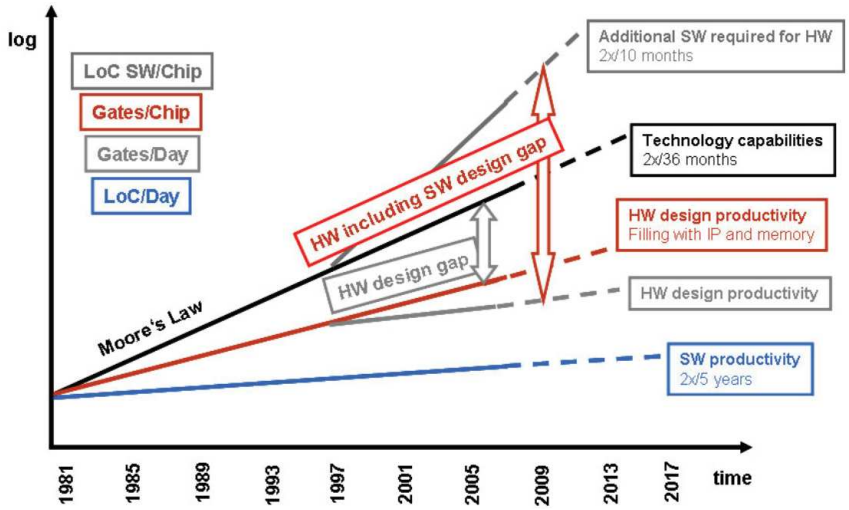


Figura 2.1: Explicação gráfica sobre o *gap* de produtividade e as suas implicações [1]

Paralelamente, há uma preocupação também crescente em reduzir a potência de operação de SoCs e, por esta razão, este tema encontra-se frequentemente em foco nos trabalhos recentes em arquitetura de computadores.

A presente dissertação encontra-se neste grupo de trabalhos com foco em consumo de energia. Do ponto de vista prático, este trabalho é o início do desenvolvimento da plata-

forma acSynth. Seu objetivo final é fornecer uma plataforma aberta para desenvolvimento e síntese de projetos HDL com base em descrições ArchC de processadores e descrições SystemC de módulos. Neste primeiro momento, acSynth realiza apenas a caracterização de processadores com base em informações de sua arquitetura. As informações são obtidas através da descrição em ADL ArchC.

Tendo em face este avanço inicial na implemetação de acSynth, o presente trabalho contribui ao definir uma metodologia para desenvolver ferramentas de análise, síntese e estudo através de uma interpretação direta da descrição ADL ArchC. O foco das ferramentas deve ser reduzir o tempo de desenvolvimento em comparação aos fluxos tradicionais baseados em descrições menos abstratas. Em outras palavras, uma ferramenta de análise de consumo é um primeiro passo, mas toda uma plataforma pode ser construída com base nos mesmos princípios e ideias fundamentais.

Este capítulo apresenta uma revisão bibliográfica dos trabalhos acadêmicos que contribuíram no desenvolvimento desta dissertação e na criação de acSynth. Uma seção discorrerá sobre métodos de estimativa de consumo de energia. Será dada especial atenção aos artigos e teses de ferramentas voltadas para automatização da análise de consumo em alto nível de abstração. Uma subseção é dedicada a trabalhos que acrescentaram ferramentas e novas funcionalidades ao ArchC, expandindo-o para além de sua funcionalidade primária como ADL [5].

2.1 Métodos para estimativa de consumo de energia para processadores

Os métodos de estimativa tradicionais baseados em circuitos, lógica ou blocos funcionais são bastante precisos em suas análises. Isto porque consideram aspectos físicos e, nos mais baixos níveis de abstração, equações matemáticas altamente correlacionadas ao comportamento real. No entanto, estes métodos apresentam dois problemas principais.

- Elevado tempo de simulação: as execuções de simulações em baixo nível são computacionalmente custosas e demoradas.
- Pouco aproveitamento de históricos: execuções de simulação em baixo nível supõem cada iteração como algo totalmente novo, sem reaproveitar características inerentes, dados sistêmicos e de arquitetura ou histórico e, desta forma, executam repetidas vezes sub-testes idênticos.

Estimativas de consumo de energia em maior nível de abstração buscam obter vantagem nestas duas áreas. Através de premissas de arquitetura, o programa de simulação acumula históricos de execução que são armazenados em bancos de dados para futuro

reuso. Isto é feito com o suporte de caracterização por ferramentas em níveis mais baixos, porém, este processo é realizado apenas uma vez. A partir de então, simulações semelhantes aproveitarão os dados do banco de dados. Isso gera uma redução considerável no tempo de execução de simulações.

O sucesso dessa abordagem se dá na definição de políticas e regras representativas. Para atingir este objetivo, é necessário encontrar regularidades no módulo em análise. Dessa forma, as operações regulares podem ser caracterizadas e, deste ponto em diante, diretamente utilizadas em simulações de níveis mais altos de abstração.

As subseções a seguir compõem uma revisão bibliográfica de técnicas de estimativa em alto nível de abstração, suas aplicações, vantagens e desvantagens.

2.1.1 Estimativa em nível de instrução

Métodos de estimativa de energia em nível de instrução aproveitam-se exatamente do processo de caracterização para gerar subsequentes estimativas rápidas para processadores. Apesar de serem circuitos complexos, processadores possuem *pipelines* muito regulares. O consumo médio da execução de uma instrução é muito representativo.

O método desenvolvido por Tiwari et al. [6,7] definiu as bases desta metodologia. Em seus trabalhos, analisou o consumo de processadores utilizando um modelo simples de teste com a execução de um conjunto repetido de instruções e alimentação controlada. Com a medição direta da informação de corrente, a taxa de consumo pôde ser obtida. Ele utilizou esta técnica para medir o consumo sobre um processador 486DX2 e variações da mesma técnica foram aplicadas sobre ARM7TDMI e Motorola DSP 56100. Tiwari ainda analisou qual o impacto da relação entre operações, ao estudar as variações do consumo de uma instrução quando precedidas por instruções diferentes. Uma vez caracterizado, o consumo de energia de um processador pode ser estimado com base nas informações coletadas, sem a necessidade do dispêndio de tempo sobre a análise completa.

Tiwari apresentou seus trabalhos há algum tempo, porém, sua metodologia possui um bom balanço entre eficiência e facilidade de aplicação. Além disso, há uma forte correlação entre o método e a linguagem ArchC, que tem sua estrutura descritiva também baseada em instruções.

Com a presença de ferramentas capazes de fornecer dados simulados em níveis mais baixos, a metodologia ganha uma vantagem ao basear-se em estimativas físicas mais precisas previamente coletadas. O presente trabalho de análise de energia sobre acSynth utilizou esta abordagem ao caracterizar *netlists* de processadores em ferramentas de análise de energia em RTL.

2.1.2 Estimativa baseada em caracterização de *bytecode*

Em [8] Hao et al. desenvolveu a ferramenta eCalc. Trata-se de um ferramenta de estimativa de consumo para softwares Android com base em *bytecodes* Dalvik. A técnica apresenta grande similaridade com a apresentada por Tiwari, apenas elevando o nível de atuação da medição. Ao invés de criar rotinas com instruções repetidas, Hao elaborou rotinas com *bytecodes* repetidos.

A técnica foi eficiente nos testes apresentados, com desvio da ordem de 9,5%. No entanto, eCalc é evidentemente restrito a máquina virtual Dalvik. Além disso, eCalc nada pode dizer a respeito do hardware final executando a máquina virtual e, desta forma, fornece poucos dados para atuação no nível de hardware. Sua contribuição principal está na possibilidade de explorar eficiência energética nos algoritmos elaborados para executar na máquina Dalvik alvo.

Podemos conjecturar que da mesma forma que uma JVM foi descrita em ArchC [9], a máquina virtual Dalvik para Android poderia ser igualmente elaborada em ADL. Isto permitiria o uso da mesma técnica apresentada em [8] utilizando-se a plataforma de trabalho ArchC que é mais ampla. A vantagem seria trazer mais recursos e a possibilidade de análises mais elaboradas.

2.1.3 Estimativa de consumo de sistemas completos

Métodos semelhantes baseados em caracterização e reuso dos dados em simulações futuras podem ser vistos em estudos com maior amplitude, mostrando a eficiência da abordagem [10]. Zhang et al. desenvolveram duas ferramentas, a PowerBooter responsável pela caracterização e a PowerTutor [11] que, com base nos dados obtidos de PowerBooter, fornece estimativas de energia *online*. O objetivo dos trabalhos de Zhang foi fornecer dados rápidos de consumo de energia em simulações de programas para a plataforma Android.

O trabalho apresenta uma análise do perfil energético de um celular, com uso do equipamento Monsoon FTA22D [12] que fornece tensão controlada e registra informações de consumo de bateria. Desta forma, o processo é automatizado, permitindo rápida exploração. A técnica é baseada em sensores de voltagem associados a bateria e conhecimento prévio a respeito do seu comportamento ao fornecer energia. Ao controlar o gerenciamento de memória do celular, bem como a atividade de estado de componentes específicos do sistema, é possível caracterizar de forma bastante precisa.

O artigo informa que o método baseia-se na suposição de que há uma relação linear entre a atividade dos componentes do sistema e o consumo mensurado. Esta premissa mostrou-se bastante coerente com a realidade. Assim, as informações coletadas pelo PowerBooter são utilizadas pelo PowerTutor com um baixo desvio em relação as medições

físicas. A informação caracterizada resume-se no fim a um fator de consumo para cada componente do hardware. Estes fatores podem ser rapidamente consultados e utilizados nos cálculos das equações lineares.

PowerTutor apresenta uma interface amigável e a possibilidade de acesso *online* a dados caracterizados, o que é de grande valor, permitindo alto reuso dos dados dentro da comunidade de desenvolvimento Android. O aplicativo executa no celular alvo e coleta dados em tempo real, como por exemplo, nível de brilho da tela, nível de uso do GPS e percentagem de uso do núcleo de processamento. Depois, multiplica estes dados com os fatores caracterizados. A soma destas operações fornece a energia total.

Apesar do método ser bastante abrangente, a caracterização do núcleo de processamento não traz uma correlação clara entre o consumo e o algoritmo em execução. Isto porque optou-se por uma análise menos fina, penalizando precisão por mais desempenho nos testes e redução do tempo de caracterização.

São utilizados apenas 2 fatores de consumo como resultado da caracterização, um para alto nível de execução, outro para baixo nível. Este fator é então multiplicado por um fator de utilização do processador num dado instante de tempo.

ArchC cresce hoje como uma poderosa plataforma de análise e desenvolvimento e poderia contribuir com um passo adiante sobre o trabalho de Zhang et al. Já existem abordagens sistêmicas com base em ArchC que poderiam agregar as vantagens de análise apresentadas por PowerTutor em simulações na ADL. A integração através de ARP [13] e uso de plataformas de teste em nível de sistema como o ESLBench [3] poderiam rapidamente fornecer uma base de análise sobre ArchC com uso dos dados obtidos de PowerBooter.

É ainda importante notar que a abordagem de análise de PowerBooter/PowerTutor em nada conflita com o método de Tiwari e a plataforma acSynth. Ambos se baseiam em princípios fundamentais semelhantes, caracterização e rápida simulação e, portanto, são correlatas.

PowerBooter fornece ferramentas suficientes para aplicar o método Tiwari sobre processadores de celulares, bastando a criação de programas de caracterização compilados para Android com rotinas de instruções repetidas.

Desta forma, seria possível ter uma análise mais fina do consumo do processador e mais detalhes sobre o impacto dos algoritmos no consumo geral do sistema. Isto abriria espaço para a exploração sistêmica, o que poderia contribuir na decisão de uso de certos componentes em detrimento a outros. Se por um lado PowerTutor se concentra em fornecer referência rápida a comunidade para o que já existe no mercado, uma abordagem com uso de ArchC poderia fornecer informações ao projetistas para as interações seguintes de produtos com foco em redução do consumo.

2.2 Ferramentas de análise de consumo em alto nível de abstração

A necessidade de elevação do nível de abstração é uma realidade no desenvolvimento de projetos *System on Chip*. Esta busca ascendente deve-se ao fato de que um maior nível de abstração traz ganhos reais de produtividade, permitindo simulações mais rápidas e maior foco em aspectos globais. Desta forma, aproveita-se o *know-how* dos níveis inferiores, sem necessariamente manipulá-los.

O diagrama apresentado na figura 2.2 explicita como os níveis superiores englobam os inferiores [14] em um projeto de SoC. Uma implicação direta da realidade simplificada por este modelo é que decisões errôneas de arquitetura e sistema provocarão elaboração de algoritmos incorretos ou imprecisos. O mesmo replica-se abaixo, até que o circuito seja comprometido. Por isso é que aspectos arquiteturais e sistêmicos devem ser analisados profundamente nas etapas iniciais de projeto. Uma equipe de desenvolvimento deve buscar restrições físicas, de desempenho e de recursos para atender os requisitos estabelecidos.

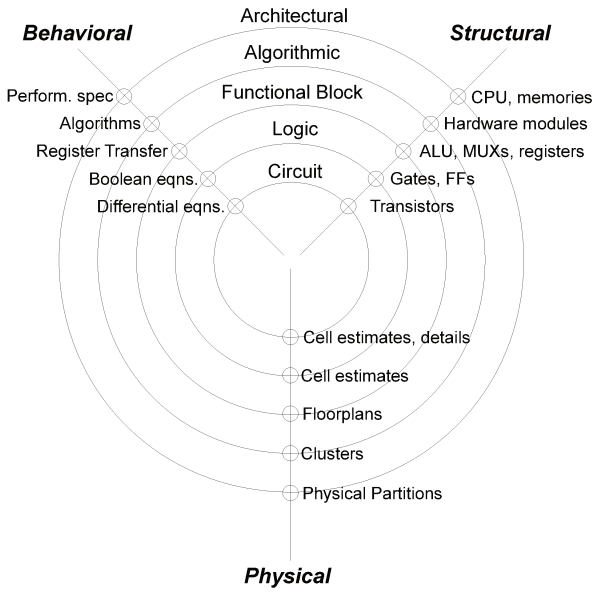


Figura 2.2: Diagrama de Nível de Abstração versus Domínio de descrição: note que as abstrações superiores englobam as inferiores

Assim, dado que o consumo de energia é um dos principais problemas que engenheiros de arquitetura de computadores precisam enfrentar, somada a necessidade de uma análise rápida logo nos estágios iniciais de um projeto, ferramentas que automaticamente geram modelos de análise de consumo são muito importantes para redução do tempo de desenvolvimento de soluções. Diversas ferramentas tem sido criadas para estimar o consumo

de energia para processadores e sistemas embarcados.

Degalahal et al. [15] aponta ainda o aumento da relevância do mercado de FPGAs no setor de semicondutores. Características particulares as FPGAs como fluxo de desenvolvimento mais fácil que projetos *full-custom*, maior possibilidade de reuso de código e maior testabilidade de soluções colocam os chips de FPGA numa posição ideal para soluções de baixo-médio volume de produção. No entanto, em comparação com sistemas ASIC, FPGAs usam muito mais transistores para hardwares de funcionalidade equivalente. Este fator eleva ainda mais a importância de uma análise bem apurada do consumo de energia.

A implicação direta disto é uma alta preocupação com análise de potência de circuitos integrados. Estudar o consumo energético de uma arquitetura em alto nível, direcionando o desenvolvimento da solução para baixo consumo logo nos primeiros estágios de desenvolvimento é hoje um paradigma dos mais importantes na indústria.

2.2.1 A ferramenta ORINOCO

Dentre as ferramentas de análise em alto nível de abstração, encontramos a ferramenta ORINOCO (***O****FFIS* ***R****esearch* ***IN****stitute p****O****wer* ***C****haracterizer, estimator and* ***O****ptimizer*) [16]. O fluxo de desenvolvimento ORINOCO pode ser visto na figura 2.3.

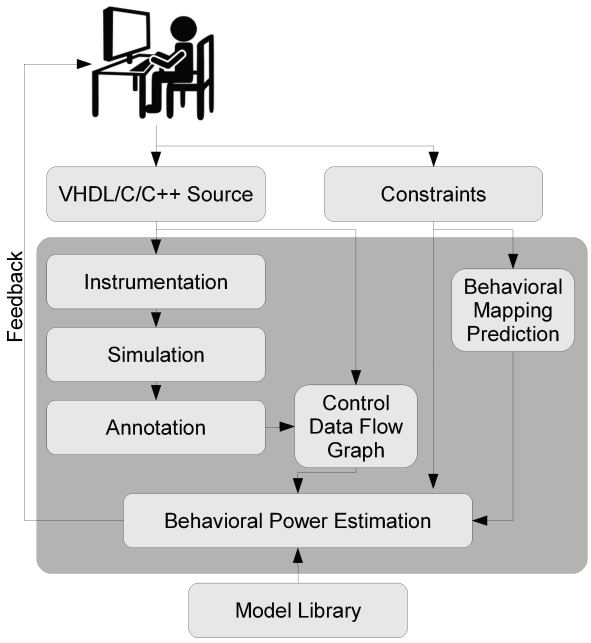


Figura 2.3: Fluxo de desenvolvimento ORINOCO

Trata-se de uma ferramenta para análise de dissipação de potência que busca oferecer meios de otimização em nível de algoritmo para sistemas descritos em C/C++, SystemC

e VHDL. Stammermann et al. afirmam que as análises são tão mais eficientes quanto mais elevado é o nível de abstração. Isto porque a influência das decisões de projeto é mais impactante sobre a demanda de energia e mais evidenciada quanto maior o nível de abstração adotado.

ORINOCO aceita como entrada uma SDL como SystemC. A ferramenta traduz a descrição em alto nível e a executa com o objetivo de obter as sequências de entrada de dados. Um arquivo de estrutura é gerado deste processo, o que permite, num passo seguinte, gerar um CDFG. Um outro arquivo também é gerado, armazenando dados de atividade de transição do sistema. Os dados de transição são então alimentados no grafo e uma análise determina uma ordem ótima do ponto de vista energético para as operações do sistema.

O principal objetivo da ferramenta ORINOCO é apontar os pontos críticos do circuito além de definir limites para o consumo de energia. O foco é direcionar o projetista a realizar modificações no nível de algoritmos em prol de um consumo de energia mais eficiente. Segundo os autores, a influência desta análise em alto nível de abstração é estimada em uma variância de até 400% na potência dissipada de um projeto.

ORINOCO encontra-se no grupo de ferramentas de análise completa de circuitos, que busca considerar todos os componentes de hardware presentes para a caracterização, simulação e geração de relatórios.

2.2.2 Análise de potência sobre ADL LISA

LISA, ***L****anguage for* ***I****nstruction-****S****et* ***A****rchitectures* [17], trata-se de uma ADL para descrição de processadores assim como ArchC. LISA se utiliza de três níveis de descrição de arquitetura, hardware explícito, hardware implícito e hardware não-formalizado. O fluxo de exploração e implementação baseado em LISA pode ser visto na figura 2.4.

Uma arquitetura descrita em LISA deve passar pela ferramenta de compilação LISA, que por sua vez gera automaticamente compilador, montador, *linker* e simulador na arquitetura descrita. Além disso, a ferramenta ainda pode gerar uma HDL a partir das descrições explícitas e implícitas do modelo LISA.

Em [18] é descrito como estas ferramentas podem ser utilizadas para exploração de *timing*, área e consumo de energia a partir de uma descrição de alto nível. No aspecto de análise de potência de um processador, a plataforma de trabalho LISA realiza uma caracterização completa levando em conta todos os componentes de hardware relevantes no sistema. Neste aspecto, LISA assemelha-se ao método de análise de ORINOCO.

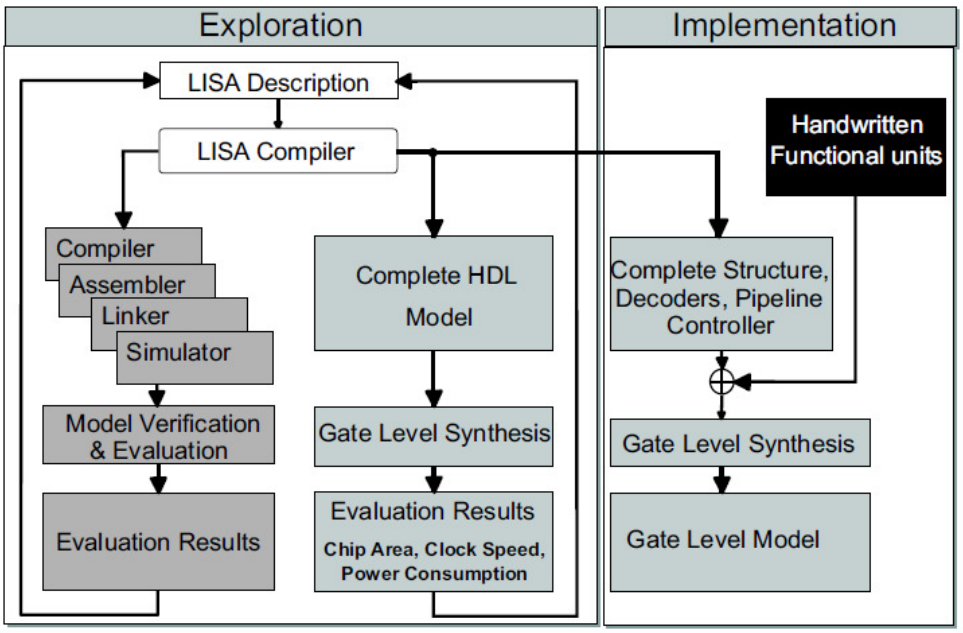


Figura 2.4: Fluxo de exploração e implementação com base em ADL LISA

2.2.3 A plataforma MPARM

Outras técnicas de estudo de consumo em processadores, por outro lado, restringiram a análise dos perfis de consumo de energia para apenas alguns poucos estados do processador [19, 20].

Loghi et al. utilizou um modelo baseado em apenas dois estados de execução, *RUNNING* e *IDLE*. Informações de consumo foram caracterizadas e dois valores apenas foram utilizados, na escala de mW/MHz. A caracterização ocorreu sobre um ARM7 com 0.13μm, obtendo-se 0.055 mW/MHz em *RUNNING* e 0.036mW/MHz em *IDLE*. Nota-se que se trata de um modelo muito simplificado, porém, de rápida elaboração e uso.

Apesar das simplificações no modelo de análise de processadores, os trabalhos sobre MPARM se destacam pela amplitude de análise, uma vez que se trata de uma plataforma MPSoC completa onde estudam-se, ainda: memórias, barramento, cache, recursos de hardware dedicado e ainda as implicações de consumo de energia em um ambiente de execução de programas com múltiplos processadores. Veja a implementação da plataforma em SDL na figura 2.5.

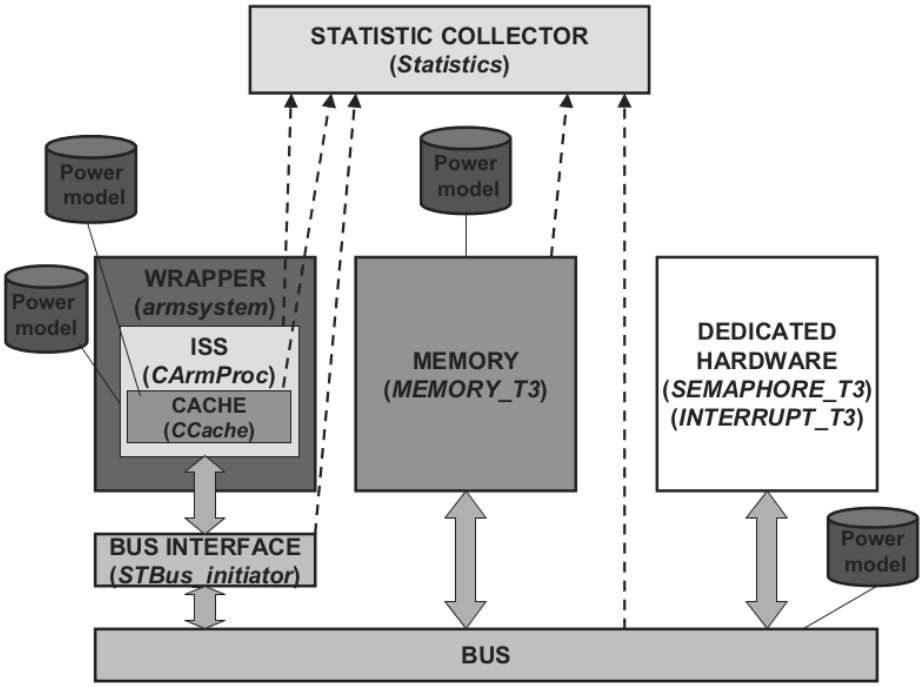


Figura 2.5: Implementação em software da plataforma MPARM

2.3 Análise em nível de arquitetura sobre infraestrutura ArchC

Hoje, ArchC pode ser considerada uma **plataforma** de análise, estudo, simulação e desenvolvimento de arquiteturas e processadores em alto nível de abstração. Isto se deve aos diversos trabalhos envolvendo ArchC, que o expandiram para além de suas premissas iniciais.

Trata-se de um processo natural no desenvolvimento de ADLs a criação de um meio-ambiente de desenvolvimento objetivando acelerar o fluxo de desenvolvimento de soluções. Veja na tabela 2.1 um comparativo entre diversas ADLs existentes. Esta figura é uma versão expandida da tabela 1 do artigo de Mishra e Dutt [21], incluindo ArchC a comparação.

Em suma, ArchC hoje possui:

- Geração automática de compiladores (acAsm);
- Geração automática de simuladores (acSim);
- Geração automática de hardware (parcialmente através de acRTL);
- Geração automática de código de testes (acPowerGen e acStone);

Tabela 2.1: Comparativo entre ADLs

	MIMOLA	UDL/I	nML	ISDL	HMDES	EXPRESSION	LISA	RADL	AIDL	ArchC
Gerador de compilador	✓	✓	✓	✓	✓	✓	✓			✓
Gerador de simulação	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Precisão de ciclo	✓	Δ		✓	✓	✓	✓	✓	Δ	
Verificação formal						Δ			Δ	
Geração de hardware	✓	✓		Δ		✓	✓		Δ	Δ
Geração de testes	✓		✓			✓				✓
Geração de JTAG I/O							✓			
Informação de instruções			✓	✓	✓	✓	✓	✓		✓
Informação estrutural	✓	✓			✓	✓	✓	✓		✓
Informação de memória					Δ	✓	✓			✓

Legenda: ✓= suportado; Δ = suportado com restrições.

- Descrição de informação de conjunto de instruções (arquivo ISA ArchC);
- Descrição de informação estrutural (arquivo ARCH ArchC);
- Descrição de memória (função ac_mem em ARCH ArchC).

Os trabalhos desenvolvidos para acSynth contribuíram para a plataforma ArchC nos seguintes aspectos:

- Criação de novas diretrizes para acSim, permitindo gerar simulações com relatórios de consumo de energia;
- Aplicação prática e testes sobre o fluxo de desenvolvimento acRTL;
- Desenvolvimento do acPowerGen e execução de testes sobre o *testbench* acStone;

A presente seção irá discorrer a respeito do desenvolvimento sobre a arquitetura ArchC necessário para atender estes recursos.

2.3.1 O acSim

O acSim é o gerador de simuladores da plataforma de desenvolvimento ArchC [22–24]. O gerador acSim elabora um modelo comportamental da arquitetura com base na descrição ADL. A implementação utiliza dois arquivos, uma descrição do conjunto de instruções (arquivo ISA ArchC) e uma descrição da arquitetura dos recursos (arquivo ARCH ArchC).

O programa acSim faz uso de outras duas ferramentas para atingir seus objetivos, o preprocessador ArchC acPP e um gerador de decodificador, ambos não visíveis para o usuário final da plataforma de simulação gerada. O resultado final é uma descrição em SystemC da arquitetura em análise.

O acSim foi um dos primeiros componentes a compor o conjunto de ferramentas ArchC. É hoje largamente utilizado e estável, servindo de base para a aplicação e uso de outras ferramentas. A plataforma acSynth recai neste grupo de recursos dependentes do simulador acSim.

2.3.2 O acAsm

O acAsm trata-se de uma ferramenta baseada em redirecionamento de montadores GNU. A partir de um modelo ArchC com uso de extensões que englobam os padrões de instruções *assembly*, acAsm gera arquivos relacionados a arquitetura descrita pela ADL, no formado exigido pela ferramenta GNU Gas. Estes arquivos podem então ser compilados e instalados através do fluxo padrão de montadores Gas, resultando em um montador para a arquitetura descrita em ArchC [25, 26].

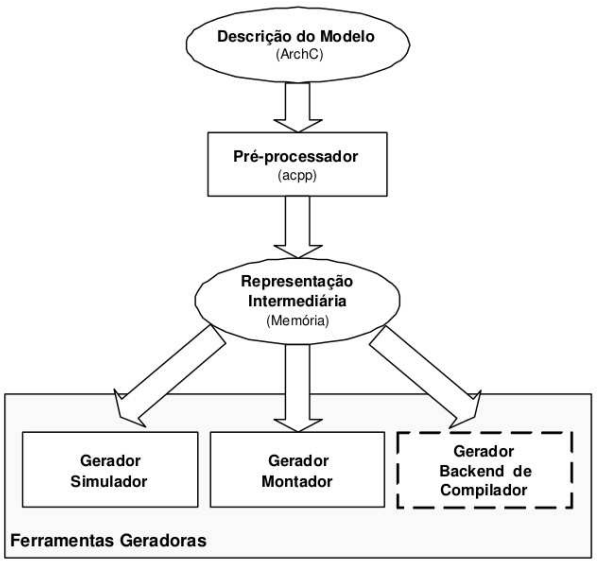


Figura 2.6: Estrutura genérica para a geração de ferramentas a partir de uma descrição ArchC. O acAsm é responsável pela geração de montadores

O acAsm baseia-se em novas chamadas de suporte em ArchC que fornecem a descrição de registradores (através de ac_asm_map) e de sintaxe de instruções (através de set_asm). A extensão ainda prevê sobrecarga de sintaxe, criação de formatadores e mnemônicos e pseudo-instruções (através de pseudo_instr).

A partir da descrição, o preprocessador ArchC possui a capacidade de fornecer informações suficientes para a geração de montadores. A ferramenta acAsm é a responsável por esta tarefa. O processo genérico de geração de ferramentas a partir de ArchC é exibido na figura 2.6.

A ferramenta acAsm é utilizada como um recurso no processo de geração de programas pela ferramenta acPowerGen.

2.3.3 O acPowerGen

No acSynth, o desenvolvimento baseou-se em informações de caracterização de programas gerados automaticamente. Este processo foi realizado com uso da ferramenta desenvolvida por Auler et al. [27], aqui chamada acPowerGen. Esta ferramenta cria programas em *assembly* para uma dada arquitetura, com base nas informações encontradas em um modelo ArchC correspondente e com uso da ferramenta de geração automática de *back-end* de compiladores [28].

O desenvolvimento de acPowerGen baseia-se em estudos de Abbaspour et al. [29], que demonstra em seus trabalhos como redirecionar os utilitários de compilação GNU com base em um modelo ADL abstrato. Ele ainda exibe resultados experimentais para a arquitetura SPARC.

No entanto, acPowerGen vai além e utiliza técnicas apresentadas por Baldassin et al. [30] que mostraram maior versatilidade, gerando quatro possíveis montadores. Além disso, Baldassin desenvolveu um método que se baseia na estrutura de modelos de processadores ArchC, o que facilitou o desenvolvimento e a utilização dos seus recursos e resultados.

O acPowerGen pode ser adaptado a diferentes arquiteturas através de arquivos de configuração dos padrões de código a serem gerados, além dos arquivos ArchC de descrição de arquitetura e instruções.

Ao longo dos trabalhos com acSynth, foi desenvolvida uma versão diferenciada de acPowerGen com base nos mesmos princípios da ferramenta original. O objetivo foi reduzir o *overhead* da descrição de informações que poderiam ser extraídas diretamente da ADL ArchC ou que não eram diretamente necessárias no contexto da geração de códigos aleatórios para teste. Além disso, o novo código foi desenvolvido em Python, que trouxe eficiência na implementação de novos recursos ou na atualização de algoritmos e processos.

Tanto a versão atual em Python quanto a versão anterior original desenvolvida por Auler encontram-se no pacote de ferramentas da plataforma acSynth e são passíveis de uso.

2.3.4 O acRTL

O desenvolvimento de acRTL foi iniciado por Goto et al. [9] e teve por objetivo criar ferramentas para geração automática de código em RTL a partir de descrições de arquitetura em ADL ArchC. Vale destacar que o foco era tornar o fluxo possível, sem ocupar-se da obrigação de encontrar uma solução ótima de número de unidades lógicas, área, desempenho ou baixo consumo de energia.

O trabalho focou na criação da ferramenta acRTL que estabeleceu um fluxo equivalente

em HDL ao da geração de código SystemC pelo simulador acSim. Esta comparação contextual pode ser vista na figura 2.7

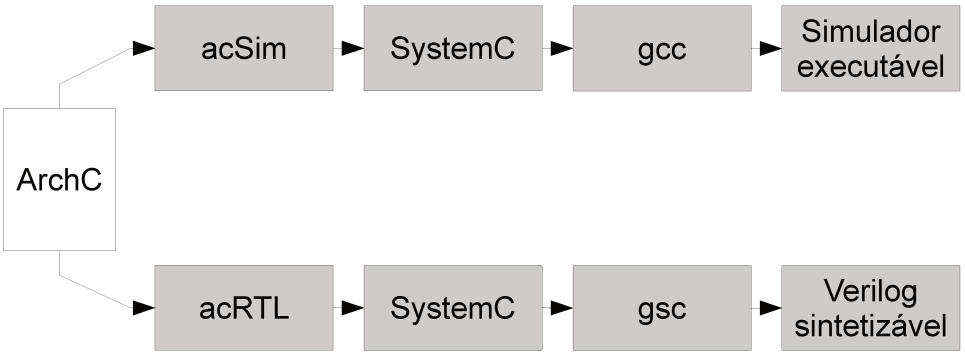


Figura 2.7: Comparação entre acSim e acRTL

O fluxo funciona com base em três componentes:

1. Um gerador de *template* SystemC a partir da descrição ArchC;
2. Um arquivo com descrições comportamentais das instruções;
3. Um tradutor SystemC/Verilog adaptado para as especificidades de acRTL, chamado gsc.

Assim, é importante dizer que a descrição habitual de ArchC não é suficiente para a geração completa do processador, requerendo uma atividade adicional do desenvolvedor em elaborar as descrições comportamentais.

No entanto, a geração do *template* retira uma carga considerável de trabalho uma vez que o controle de fluxo, controle de PC, além dos estágios de *fetch* e *decode* encontram-se pré-prontos, o que permite a um desenvolvedor focar especificamente nas especializações de cada instrução da arquitetura.

Este tipo de fluxo de projeto é muito importante em desenvolvimento de sistemas ASIP (*Application-Specific Instruction-Set Processor*) porque acelera o desenvolvimento de novos processadores com um foco específico em uma aplicação. Trabalhos interessantes de aplicação de ADLs na geração automática de HDL encontram-se em [17] onde são descritas aplicações comerciais do uso da ADL LISA para geração de módulos para um DSP. Com refinos, em teoria é possível realizar o mesmo fluxo de desenvolvimento sobre ArchC.

Na dissertação, Goto demonstra que é possível realizar o processo de geração automática a partir de descrições ArchC de arquiteturas monociclo, multiciclo e *pipeline*. No presente trabalho, foi gerado uma versão de um processador MIPS-I *pipeline* usando como *golden model* o processador Plasma. Uma infraestrutura de teste comprovou a corretude

das instruções desenvolvidas e maiores informações do processo encontram-se nos capítulo 4, de resultados experimentais.

2.3.5 Análise de consumo de energia em ArchC

Em contraste com outros trabalhos em análise de consumo de energia em alto nível, a análise de consumo sobre plataforma ArchC encontra-se em uma região intermediária entre os dois extremos relatados: de um lado, análise sobre LISA e outros estudos similares, onde há consideração completa dos componentes do sistema; do outro, análise sobre MPARM e outros estudos similares onde existe uma grande simplificação do consumo de um processador para apenas alguns poucos estágios de execução.

O processo de caracterização e o subsequente fluxo de simulação com relatórios de energia em acSynth faz uso de diversos estudos e recursos desenvolvidos em trabalhos anteriores sobre a plataforma ArchC, incluindo todo o desenvolvimento de acPower. As próximas subseções irão discorrer a respeito dos detalhes que concernem a estes recursos.

O acPower

Josué Ma e Rodolfo Azevedo desenvolveram o programa acPower [31, 32] que aufere a potência de um processador com base na metodologia de Tiwari et al. [7]. Neste trabalho, Tiwari desenvolveu um método de caracterização de consumo de processadores onde obtém-se informação de energia por instrução executada pelo circuito. Para um dado processador, geram-se programas de caracterização em que uma mesma instrução é executada diversas vezes. Com um número suficientemente grande de execuções de uma mesma operação, atenuam-se os impactos das instruções acessórias de *setup* do programa e das instruções de controle de laço.

Executam-se estes programas de caracterização em nível de registradores, anotando-se os valores obtidos de consumo. Ao fim, obtendo-se um valor de potência do processador para cada programa, que corresponde a cada operando. Com aritmética básica e informações de frequência de execução e IPC (*Instruction Per Cycle*), é possível dividir o valor de potência obtido pelo número de instruções de mesmo operando executadas pelo software de caracterização. Tem-se, assim, uma estimativa da energia consumida por cada instrução do processador.

Este dado pode ser então utilizado em futuras simulações com baixo esforço computacional e em alto nível de abstração, em SDLs ou ADLs.

Podemos dizer que acPower é uma implementação da metodologia Tiwari sobre processadores descritos em ADL ArchC. Ma relata o processo de caracterização onde as instruções da arquitetura SPARCv8 são divididas em classes distintas. Foram criados programas de caracterização que executaram 100.000 vezes a instrução alvo. Compilando-se

estes programas e executando-os em um simulador como Modelsim, obtém-se arquivos do tipo SAIF ou VCD de atividade de chaveamento. Estes dados podem então ser fornecidos a uma ferramenta de análise de consumo de energia, como XPower ou PowerCompiler e assim obter o valor de potência de cada instrução alvo. Este processo de caracterização pode ser visto graficamente na figura 2.8.

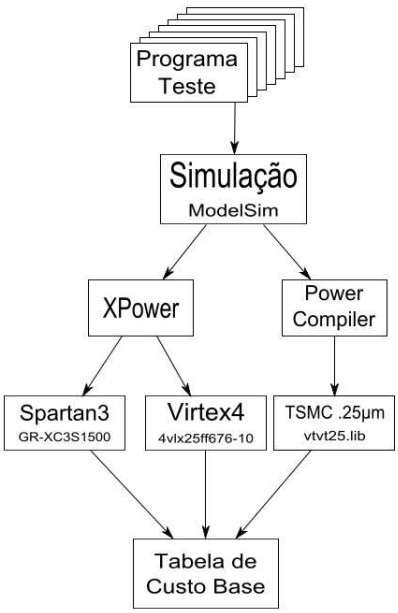


Figura 2.8: Metodologia de caracterização utilizada para captação de dados para acPower

Com estes valores, executaram-se programas sobre o simulador acSim da mesma arquitetura em estudo e relatórios de estatísticas do número de instruções foram gerados. Tendo-se o número de instruções executadas e os valores de consumo por instrução obtidos pela caracterização é possível então calcular o consumo de energia do programa simulado.

Este processo de cruzamento dos dados caracterizados com os dados da simulação foram realizados com acPower. É importante apontar que, no momento da apresentação da ferramenta, acPower era um recurso independente, não integrado a ArchC, apenas fazendo uso dos relatórios de estatística gerados pela execução de programas. Maiores detalhes são encontrados na dissertação em [31].

O acSynth integrou os algoritmos e métodos de acPower ao fluxo de geração de simuladores acSim com suporte da biblioteca PowerSC, trazendo maior abrangência para os relatórios de consumo gerados.

A biblioteca PowerSC

A ferramenta PowerSC foi desenvolvida por Klein et al. [33–36]. Trata-se de uma sofisticada biblioteca de extensão em SystemC [37] que tem como objetivo mensurar em alto nível o consumo de potência de um circuito. A figura 2.9 mostra como esta extensão foi realizada.

Figura 2.9: Estrutura da biblioteca de expansão PowerSC sobre a SDL SystemC

Inicialmente, seu alicerce foi a coleta de estatísticas de atividade de transição [38]. Esta informação é chave para calcular o consumo de potência dinâmica que é comumente modelada pela equação:

$$P_{dyn} = \sum C_L A_i {V_{dd}}^2 f$$

Onde, P_{dyn} é a potência dinâmica total de um circuito, soma do consumo de todos os *gates* em estudo. C_L é a capacitância de carga, A_i é a atividade de transição do i-ésimo *gate*, V_{dd} é a tensão de alimentação e f a frequência de operação. Note que todas as variáveis, exceto A_i são dependentes de tecnologia e podem ser englobadas em um único termo K_t, que chamaremos constante de tecnologia. Tem-se, portanto:

$$P_{dyn} = \sum K_t A_i$$

Logo, tendo-se as estatísticas de atividade de transição e dados de tecnologia, temos o consumo de um módulo. A grande contribuição de PowerSC foi permitir acesso a estes dados em níveis elevados de abstração através da SDL SystemC.

A continuidade do trabalho, desenvolvido em [39] pelo mesmo autor, foi aplicar a técnica de análise de potência a módulos genéricos, com uso de bibliotecas de tecnologia adaptadas que forneceram base para o cálculo da constante de tecnologia K_t. O resultado foi uma plataforma de análise e um método sistemático para cálculo de consumo de potência de IPs genéricos através de caracterização de macromodelos.

Dado que ArchC tem como base SystemC, seria, em tese, possível aplicar as várias ferramentas da biblioteca sobre um processador ArchC. Isto porque a infraestrutura de simulação de ArchC é gerada em SystemC. A ferramenta de análise de energia em acSynth fez uso direto de PowerSC, integrando a biblioteca ao fluxo de simulação ArchC. Isto permitiu elaborar plataformas de simulação em acSim com geração automática de relatórios de energia.

Desta forma, acSynth expandiu o escopo de PowerSC permitindo-o atuar em dois níveis diferentes de abstração, em nível de sistema com SystemC e em nível de arquitetura com ArchC.

O sistema Power-ArchC

Gupta et al. [40] desenvolveu uma metodologia chamada Power-ArchC sobre a estrutura das ferramentas ArchC onde valores de potência eram obtidos da caracterização de circuitos em nível de registradores. Power-ArchC retorna um modelo de consumo de energia que produz uma média dos valores de consumo por instrução. Eles definiram rotinas de caracterização para avaliar um pequeno subconjunto de diferentes operandos, além de levantar dados da inter-relação de instruções executadas. Cada rotina era uma abordagem diferenciada para analisar o consumo.

Criaram três rotinas de caracterização de consumo em nível de instrução (ILPC, *"Instruction Level Power Characterization"*) com base em programas de referência. O fluxo é graficamente representado na figura 2.10. O primeiro ILPC utilizou o *benchmark Motion* [41]. O segundo, fez uso de uma versão modificada do *Qsort* proveniente da suíte de *benchmark* Mibench [42].

O terceiro ILPC foi uma combinação dos dados da primeira campanha de caracterização com os dados da segunda campanha de caracterização. Eles então compararam a energia em nível de instrução do modelo baseado em dados caracterizados com a energia em simulações no nível de registradores. Os trabalhos foram desenvolvidos sobre a arquitetura MIPS. O resultado da comparação apresentou um desvio máximo de 21% em programas de teste analisados.

Como um resultado secundário dos trabalhos desenvolvidos, Power-ArchC incluiu os valores de potência total para cada instrução por estágio do *pipeline*, apontando não só o consumo, mas também, em que estágio uma dada instrução está consumindo mais ou menos energia. Isto trás uma vantagem bastante interessante para desenvolvedores

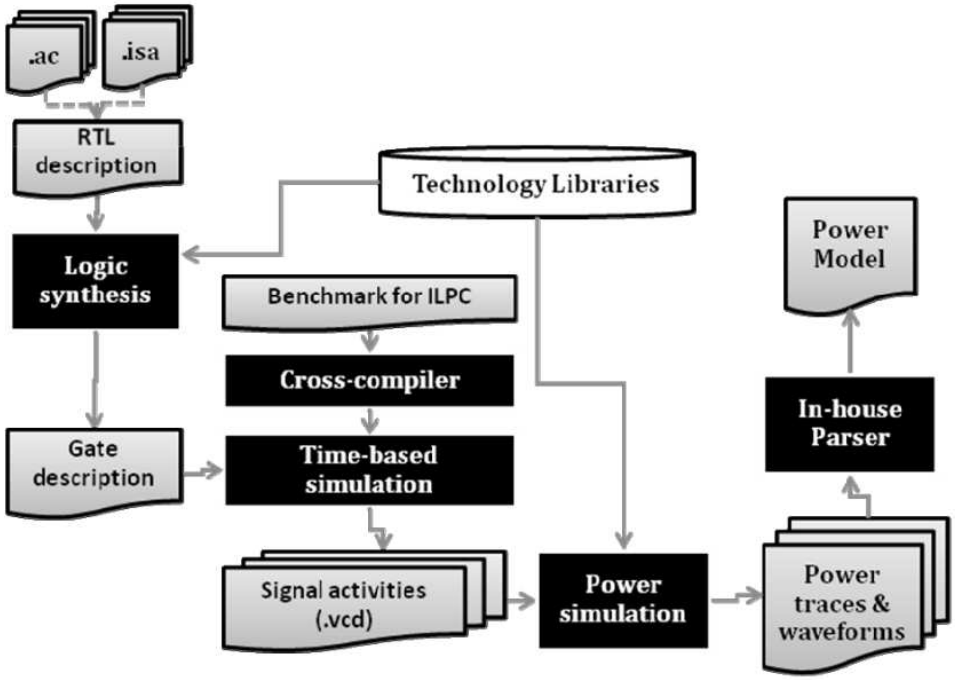


Figura 2.10: O fluxo de ILPC sobre a plataforma ArchC

uma vez que aponta onde devem ser direcionados os esforços para otimização em um processador analisado.

2.3.6 A integração através de ARP

ARP é um sistema para desenvolvimento de SoCs com suporte a processadores modelados em ArchC [13]. O principal objetivo da ferramenta é facilitar o gerenciamento de projetos em sistemas embarcados através de maior organização dos componentes do sistema ArchC.

Uma vez instalada, a plataforma ARP oferece ao usuário uma estrutura de diretórios, separando módulos em categorias de tipos de componentes:

***processors*:** modelos de processadores em ArchC;

***is*:** elementos de interconexão, barramentos, roteadores e NoCs;

***ip*:** IPs de propósito geral;

***sw*:** componentes de software compatíveis com a plataforma;

***wrappers*:** conversores de protocolos;

***platforms*:** plataformas para integração dos componentes.

ARP possui uma ferramenta de empacotamento e desempacotamento de plataformas, o que permite fácil trânsito de sistemas inteiros de uma área de trabalho para outra. Além disso, ARP permite um gerenciamento simplificado do processo de compilação dos diversos componentes de um sistema.

Os trabalhos iniciais já apresentam aplicações práticas no estudo de memórias transacionais com uso de ARP. Além disto, a plataforma serve como mecanismo de ensino sobre arquitetura de computadores.

Dado que a ferramenta acSim encontra-se integrada à plataforma ARP, é possível integrar o desenvolvimento do presente trabalho sobre acSynth, trazendo análise de potência a SoCs complexos, indo além da análise, apenas, dos núcleos de processamento.

2.3.7 Um ambiente de análise e estudo

Como dito, ArchC evoluiu de uma ADL para todo um ambiente de análise e estudo de arquiteturas. Isto resultou na criação da variedade de ferramentas aqui citadas. No entanto, para que haja um impacto representativo nas análises em nível arquitetural é preciso integrar e cruzar os dados provenientes dos diferentes programas de suporte a ArchC, buscando máximo reaproveitamento de recursos, relatórios e poder computacional.

A plataforma acSynth é apresentada justamente como uma abordagem para iniciar esta desejada integração. É esperado que outros trabalhos acadêmicos sobre ArchC sigam tendência semelhante no futuro.

Capítulo 3

O acSynth

Este trabalho apresenta uma nova ferramenta ArchC chamada acSynth. Trata-se de uma plataforma voltada para integração automática de outras ferramentas ArchC com o objetivo de facilitar a análise e desenvolvimento de arquiteturas em alto nível. Neste primeiro trabalho de apresentação, acSynth será utilizado para caracterizar processadores e gerar relatórios de consumo de energia. No entanto, é importante salientar que o conceito de *framework* ou de plataforma de desenvolvimento podem ser utilizado em outras aplicações e contextos.

Espera-se que com o aumento da integração das ferramentas ArchC, um número maior de dados sejam adquiridos e a visualização de problemas torne-se uma tarefa mais clara para um projetista. Podemos dizer que esta primeira etapa de acSynth é uma prova deste conceito, uma vez que a integração de ferramentas até então desconexas apresentaram novos e interessantes resultados simplesmente pela justaposição de suas tarefas e características. A aplicação em análise de consumo de energia exemplifica a utilidade desta abordagem.

3.1 Usos e recursos da plataforma acSynth

A plataforma acSynth em sua versão atual encontra-se descrita em linguagem Python e possui a lista de funções apresentada no trecho de retorno *bash* 3.1.

```
$ ./acSynth -h
usage: acSynth [-h] [-n] [-g] [-b] [-c] [-w] [-a] [-s] [-p] [-x SUFFIX]
               [--clear] [--done]
               arch

acSynth is a framework for ArchC Tools. Version 0.1.0

positional arguments:
  arch                  Target architecture
```

```
optional arguments:
  -h, --help            show this help message and exit
  -n, --new             Create a new project workspace for characterization,
                        simulation or RTL development
  -g, --gen             Generate a new set of characterization programs using
                        acPowerGen
  -b, --build           Synthesize project using TCL files
  -c, --char            Characterize char_target project in order to obtain
                        VCD Switch Activity Files
  -w, --rtlpower        Analyze char_target project in order to obtain RTL
                        Power Reports
  -a, --autoreport      Try to translate power report package into an acPower
                        database
  -s, --sim             Generate acSim framework
  -p, --powersim        Generate acSim framework with acPower support
  -x SUFFIX, --suffix SUFFIX
                        Add a suffix in the project name. It can be use to
                        generate several projects from the same architecture
  --clear               Clear target project path. It will delete all files in
                        target project
  --done                Mark input files as 'DONE' so they won't be execute
                        next time (valid to 'char' and 'rtlpower' processes)
```

Código 3.1: Retorno de -h na chamada de acSynth em sua versão 0.1.0

A ferramenta funciona com base em projetos que são automaticamente gerados, tendo como origem pacotes de descrições de processadores em ArchC. A ferramenta faz uso de um banco de dados destas descrições de processadores. A origem dos arquivos na base de dados é o SVN oficial do projeto ArchC. É preciso fornecer o HDL de um processador como referência para a caracterização. A figura 3.1 exibe a relação dos comandos de acSynth com o fluxo de caracterização. Ao fim, pode-se simular o projeto com acSim tradicional, pelo comando `sim` ou com suporte a geração de relatórios de consumo de energia, através do comando `simpower`.

É importante dizer que esta sequência é um fluxo sugerido e o projetista não está preso a este método específico. Diferentes formas de uso são possíveis e há abertura para intercalar processos. Pode-se, por exemplo, fornecer os programas externamente, sem gerá-los automaticamente. Isto é importante caso deseje-se realizar operações com *benchmarks* externos ou na aplicação de outros métodos de caracterização. O comando `build` pode ser ignorado caso o desenvolvedor prefira seguir um fluxo personalizado. O mesmo vale para todas as demais etapas.

Vamos detalhar agora qual o fluxo padrão sugerido, com base na sequência descrita no diagrama em 3.2. Antes de iniciar o uso, acSynth exige o *setup* de ambiente. Isto é feito através do *script* Bash set_env.sh . Este *script* define variáveis de ambiente importantes para a execução de acSynth. Feito isto, pode-se começar pelo `new`.

new: A primeira etapa do processo é a criação automática da infraestrutura de trabalho através do comando `new`, que deve ser executado antes de qualquer outro comando

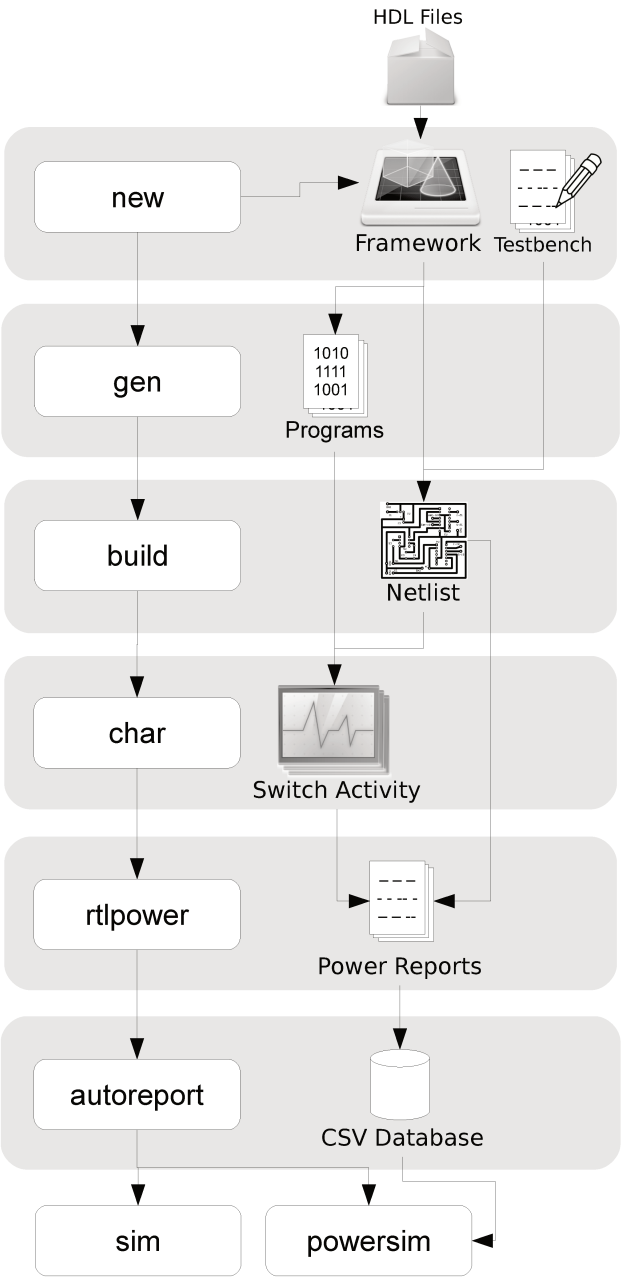


Figura 3.1: Processo simplificado do fluxo de caracterização de acSynth com indicação do comando de cada etapa. Ao fim, o arquivo CSV com as informações de energia é fornecido para o simulador acSim gerado pelo comando `acpowersim`

de acSynth. O único parâmetro importante aqui é o nome da arquitetura que deseja ser utilizada. Atualmente, acSynth suporta as arquiteturas MIPS-I e SPARCv8, que podem ser referenciadas através dos comandos: `acSynth mips -n` e `acSynth sparc -n`. Uma vez que acSynth cria um novo projeto, é criada a pasta *workspace* e uma subpasta com o nome da arquitetura escolhida. Assim, supondo que estamos realizando um projeto para MIPS, é criada uma subpasta de nome `mips`. É também gerado um arquivo de configuração acSynth.conf dentro da pasta de projeto. Este arquivo permite que o usuário defina parâmetros importantes ao longo do processo. Ele é dividido em subseções, cada uma com os parâmetros para aquela etapa específica do projeto. O acSynth.conf permite reconfigurações entre uma etapa e outra do fluxo sobre acSynth. É previsto que o projetista itere sobre o arquivo alterando seus parâmetros ao longo da evolução do projeto.

gen: Nesta etapa, acSynth utiliza os recursos de acPowerGen para criar programas de caracterização automaticamente. O comando `gen` cria um conjunto de instruções válidas em *assembly* em um arquivo .s e gera um outro conjunto de instruções de inicialização de registradores. Os dois grupos podem ser unidos em um único programa autocontido e com um controle de laço, ou podem ser utilizados separadamente delegando o controle de fluxo ao *testbench*. A execução do processo `gen` não exige ação do desenvolvedor para arquiteturas suportadas.

build: É a etapa onde o projeto HDL é sintetizado e gera-se uma *netlist* passível de simulação por ferramentas como Xpower e PowerPlay. Esta etapa é muito dependente de ferramentas externas no momento, então necessita de tarefas manuais. Manualmente o desenvolvedor precisa fornecer um projeto HDL na pasta `char_target`. A conexão entre acSynth e o projeto RTL é feita com uso de *scripts* TCL. O endereço do *script* pode ser fornecido no arquivo de configuração acSynth.conf . O *script* deve conter todo o fluxo para obter uma *netlist* válida do processador alvo a ser caracterizado. O fluxo foi testado em acSynth juntamente com as ferramentas ISE da Xilinx e Quartus da Altera. Além do TCL, o desenvolvedor precisa elaborar um *testbench* com os seguintes requisitos: aceitar como entrada um programa *assembly* em formato texto de nome `rom.txt` e armazenar os dados em uma ROM ou RAM interna; se for realizar o controle de fluxo pelo *testbench*, aceitar também como entrada o arquivo com o bloco de código de inicialização de nome `init.txt` e elaborar a lógica de controle de laço; configurar o *clock* para uma frequência alvo e informar o sistema para gerar um arquivo de atividade de transição com todos os sinais do projeto sob o nome `tb.vcd`. Em Verilog, isto pode ser feito diretamente no código. Em VHDL, será preciso um controle pelo TCL para informar ao simulador para que gere arquivos de atividade de transição. O *testbench* precisa ser montado para um

só caso de tríade de arquivos entrada `rom.txt`, entrada `init.txt` e saída `tb.vcd`, sendo tarefa de acSynth iterar sobre todos os programas gerados por acPowerGen. É boa prática também definir um tempo máximo de simulação.

char: A etapa de caracterização é responsável por gerar arquivos de atividade de transição, um por instrução analisada. Esta etapa recebe os TCLs de simulação, bem como o *testbench* elaborado e, então, itera cada programa gerado por `gen` sobre o *testbench* desenvolvido. O controle de acSynth encarrega-se de fazer o *loop* de chamadas sobre todos os programas gerados por `gen`. O resultado final é uma lista de arquivos de atividade de transição. Neste projeto, foi utilizada a ferramenta Modelsim e o padrão de arquivo VCD de atividade de transição.

rtlpower: Nesta etapa, acSynth itera todos os arquivos VCD gerados em char e os executa em um programa de simulação em RTL juntamente com o *Netlist* gerado na etapa de `build`. Esta etapa é automática, bastando informar em acSynth.conf o nome da suíte de projeto utilizada. Este processo foi testado com as ferramentas Xilinx Xpower e Altera PowerPlay. Ao final, tem-se um conjunto de relatórios de consumo de energia, um por instrução analisada.

autoreport: Com os relatórios prontos, basta apenas agrupá-los em um formato legível a simulação acSim. O formato escolhido foi um banco de dados em CSV por ser flexível e portável. Esta etapa do processo é totalmente automática, bastando o desenvolvedor fornecer dados como frequência de simulação e a relação EPI (*Energy Per Instruction*) desejada. A plataforma acSynth analisa diretamente a descrição ArchC da arquitetura e elabora um arquivo com os dados de energia, um por linha. O mapeamento de instruções feito por acPower dentro da simulação acSim é feito com base em um identificador único por instrução. A etapa `autoreport` faz este mapeamento automaticamente. Ao fim do fluxo, temos o banco de dados e termina o fluxo de caracterização de acSynth.

sim e powersim: Neste instante, temos toda a bagagem necessária para executar simulações. Tanto `sim` quanto `powersim` utilizam os dados de projeto e geram simulações válidas com acSim. O primeiro caso, gera a simulação padrão. A segunda chamada aplica o modificador sobre acSim que o habilita a ler o arquivo CSV e gerar relatórios de consumo de energia juntamente com a execução de programas em acSim.

Por fim, temos acesso a uma plataforma de simulação capaz de executar qualquer programa na arquitetura alvo, em alto nível de abstração, fornecendo relatórios com um baixo consumo de tempo. Toda e qualquer simulação elaborada para acSim permanece suportada nas simulações com relatório de consumo de energia, bastando fornecer o banco de dados.

Explanados os aspectos técnicos do uso da ferramenta, a seção 3.2 detalhará a faceta sistêmica de acSynth. Em especial, a correlação entre as diversas ferramentas ArchC utilizadas. A integração de mais outra ferramenta ArchC ao fluxo é explicada na seção 3.3, com detalhes a respeito do uso de acRTL em acSynth.

3.2 Análise de consumo de energia de processadores com acSynth

A análise de consumo de energia de processadores descritos em ArchC foi realizada através da composição de duas ferramentas ArchC, além de duas bibliotecas de funções SystemC.

Como base, **acSim** forneceu a infraestrutura de simulação. A tarefa de acSim é interpretar a linguagem ADL de ArchC e traduzir sua descrição para um sistema SystemC. Através da extensão de classes e criação de variáveis de sondagem para adquirir o número de instruções executadas, foi possível criar a plataforma necessária para gerar relatórios de consumo de energia juntamente com a simulação de programas sobre a descrição de arquitetura.

Os relatórios foram obtidos com base na contagem de cada instrução, cruzando-se tal contagem com os dados caracterizados a partir de simulações em RTL. A caracterização foi realizada de forma semi-automatizada, muito por conta de **acPowerGen** que forneceu os programas de caracterização. Atualmente, é complexo garantir um fluxo isento de atuação do projetista porque há dependência de ferramentas externas e ArchC não possui um gerador automático de *testbench*. Apesar disto, acSynth oferece uma redução substancial no tempo de projetos que requerem análise de consumo de energia através de um fluxo completo, o que é de grande importância. Espera-se que este fluxo seja incrementado e aprimorado em projetos futuros.

O cruzamento dos dados se deu com uso da **biblioteca acPower**. Esta biblioteca é a responsável por trazer as ferramentas da **biblioteca PowerSC** para dentro da plataforma de simulação gerada por acSim, carregar os dados caracterizados e fazer o cruzamento para obter consumo em nível de instruções.

O fluxo completo pode ser visto na figura 3.2. Podemos dividi-lo em duas etapas:

- Etapa de caracterização;
- Etapa de simulação.

As subseções seguintes fornecerão maiores detalhes sobre cada uma destas etapas.

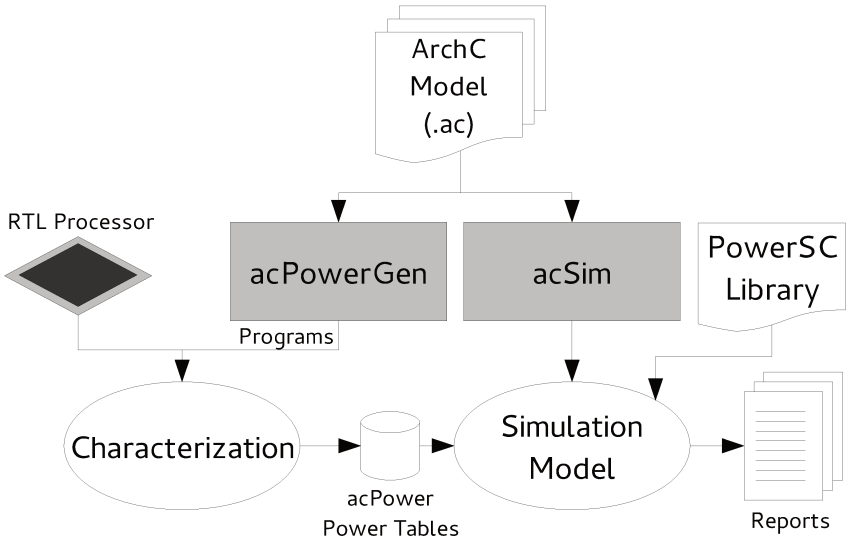


Figura 3.2: Fluxo de caracterização e elaboração de relatórios de simulação em acSynth

3.2.1 A etapa de caracterização

A primeira etapa do fluxo de operação da plataforma acSynth para consumo de energia, a caracterização, é responsável por levantar os dados em níveis mais baixos de abstração para que estes sejam utilizados nas simulações em nível de arquitetura. A maior parte do aspecto técnico está descrita na seção 3.1. Esta subseção se dedicará a uma análise mais detalhada da relação entre as diversas ferramentas ArchC atuando dentro de acSynth.

O procedimento de caracterização é usual em processos de simulação em alto-nível, variando apenas no método adotado para obtenção destes dados. A principal inovação apresentada por acSynth neste quesito é a geração automática de programas de caracterização com ajuda de acPowerGen, relatado na subseção 2.3.3. O acPowerGen será detalhado a frente na presente subseção. A figura 3.3 resume a etapa de caracterização de acSynth.

Neste fluxo, a primeira tarefa é definir qual a arquitetura será estudada e se esta encontra-se disponível em linguagem ArchC. Então, escolhe-se um processador descrito em linguagem HDL que será alvo da caracterização. Definem-se também os parâmetros em que o processador trabalhará. Informações como frequência de operação, restrições de área e tecnologia e intervalo de temperaturas de funcionamento. O foco no presente trabalho foi centrado nas informações de frequência, considerando-se os demais componentes em valores padrões.

Esta etapa é importante porque influi diretamente nos dados a serem gerados. Analogamente, pode-se dizer que esta fase é um retrato do processador alvo e, portanto, dependendo do ajuste do foco podemos ter uma imagem completamente diferente do que

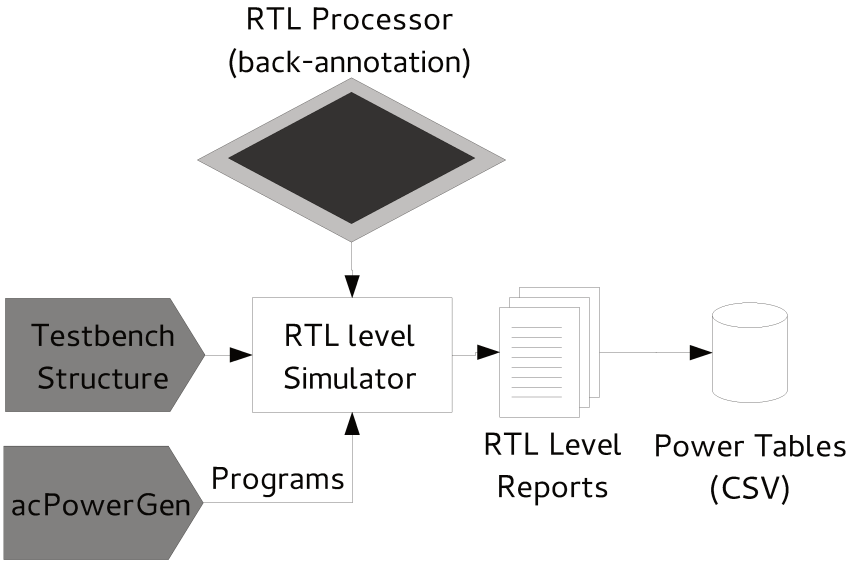


Figura 3.3: Processo de caracterização para extração de informação de consumo de energia. O processador RTL referenciado na imagem é sintetizado e então simulado com códigos de programa gerados automaticamente

está em observação.

Definidos os fatores básicos, iniciam-se os trabalhos para obter três componentes requeridos a caracterização:

1. *Back-annotation* do processador RTL;
2. Programas de caracterização;
3. Estrutura de *testbench* para execução dos programas sobre o processador.

A respeito do primeiro item, é importante dizer que foi escolhido utilizar o *back-annotation* do processador porque as ferramentas de análise de consumo em RTL apresentaram resultados mais apurados nestes casos. Como esperado, os relatórios apresentavam estimativas de erro muito maiores quando lidava-se com simulações de atividade de transição diretas do nível HDL.

Utilizando-se o *back-annotation* que descreve o processador para a geração dos perfis de atividade de transição, foi possível gerar simulações com bom nível de precisão.

O segundo item, como descrito na seção 2.3.3, é obtido automaticamente. Para tanto, Python acPowerGen faz uso de duas entradas:

1. Descrição ArchC da arquitetura em análise, tanto a descrição arquitetural quanto a descrição do conjunto de instruções;

2. Parâmetros de configuração que informam o formato dos registradores e o tamanho dos campos referenciados na descrição ArchC.

O acPowerGen é totalmente independente e se encarrega de todo o esforço computacional para gerar os programas automaticamente. Como relatado em 2.3.3, existem duas versões do programa. A vantagem da versão em Python é uma redução substancial nas tarefas necessárias para a adição de suporte a uma nova arquitetura. Isto se deve ao fato de que acPowerGen obtém a sintaxe de código diretamente da descrição ISA de ArchC. Um pequeno trecho de configuração é necessário para que acPowerGen saiba como criar chamadas a registradores de forma aleatória e como referenciar imediatos na linguagem *assembly* alvo.

A ferramenta acPowerGen em sua versão atual possui a lista de funções apresentada em 3.2.

```
usage: acPowerGen.py [-h] [-r REG] [-v VAR] [-c CASES] [-b BLACKLIST]
                     [-s SPECIAL] [-i INSTR]
                     isa_file

acPowerGen.py . Version 0.1.0

positional arguments:
  isa_file              Target ArchC ISA file

optional arguments:
  -h, --help            show this help message and exit
  -r REG, --reg REG     Register format with the range of valid random values,
                        e. g."$[1-31]"
  -v VAR, --var VAR     CSV List of variable size, e. g. "imm=16s,pcrel=30".
                        The letter 's' is for signed
  -c CASES, --cases CASES
                        Define the number of cases to analize in ArchC ISA
                        file. Default=0 (ilimited)
  -b BLACKLIST, --blacklist BLACKLIST
                        Ignore listed cases
  -s SPECIAL, --special SPECIAL
                        Special cases where you apply custom rules
  -i INSTR, --instr INSTR
                        Number of instructions per software. Default=1000
```

Código 3.2: Retorno de -h na ferramenta acPowerGen em sua versão Python 0.1.0

O uso básico da ferramenta supõe utilizar as entradas `reg`, ou `-r`, e `var` ou `-v`. O primeiro argumento, define o formato textual da referência a registradores da linguagem *assembly* alvo. Para MIPS-I, por exemplo, este parâmetro é:

```
"\"$\"[1..31]"
```

Isto irá traduzir todas as referências a registradores encontradas pelo interpretador para um registrador aleatório entre `$1` e `$31`, neste formato. Note que para referenciar

texto foram utilizadas aspas e estas, para não infringirem restrições de chamada *shell* precisam de uma barra ao contrário no início. Foi decidido por este formato para permitir a separação entre caracteres puramente textuais e possíveis caracteres de controle. No momento, não há suporte a caracteres de controle.

Para SPARCv8, temos:

```
"\"%r\"[1..31]"
```

Da mesma forma, isto irá mapear qualquer referência a registradores nas instruções para um registrador aleatório entre %r1 e %r31.

O formato permite intervalos menores, mas exige que os números sejam definidos com um menor seguido de outro maior. Idealmente, a informação do formato da assinatura dos registradores deveria ser obtida diretamente do arquivos ARCH ArchC, seção `ac_asm_map reg`, porém, isto exigiria um interpretador mais complexo, fora do escopo da presente dissertação. Assim, esta tarefa fica sugerida como trabalho futuro.

O segundo comando mais importante é o `-v` para variáveis. Ele informa o tamanho em bits de cada trecho de instrução dentro do *opcode* através de uma referência numérica. O formato é bastante direto, com o nome da região seguido pelo número de bits e opcionalmente seguido por dois pontos e uma letra modificadora. Por padrão, assume-se que a região indicada é *unsigned*, caso do campo `imm7` em MIPS-I, referenciado como `imm7=7`. A letra modificadora `s` pode ser utilizada para informar que se trata de um campo *signed*, como no caso do campo `disp22` em MIPS-I, referenciado como `disp22=22:s`. A letra modificadora `a` indica que se trata de um campo alinhado, necessário para referências de endereçamento e saltos, que não permitem o uso de valores ímpares, por exemplo, caso de `addr` em SPARCv8, referenciado como `addr=26:a`. As variáveis são separadas por vírgula. Este campo ainda pode ser utilizado com texto e, neste caso, é realizada a substituição direta do conteúdo. Um exemplo é o campo condicional de MIPS-I que foi forçado para o valor `a` pela referência `cond='a'`.

Assim como na análise dos registradores, há espaço para melhorias na análise de variáveis, através de um interpretador mais complexo. Ele poderia utilizar os dados presentes no arquivo ARCH ArchC, nas seções `ac_format`. Pelas mesmas razões, esta tarefa fica como sugestão para trabalhos futuros.

Os demais comandos são, via de regra, facultativos. O comando `cases` ou `-c` é utilizado para limitar o número de casos analisados, uma vez que existe mais de um formato de escrita *assembly* para uma mesma instrução. Isto é representado pelas múltiplas chamadas `set_asm` para uma mesma instrução presente em ISA ArchC. No entanto, não necessariamente é desejável analisar todos os casos. Por padrão, o arquivo ARCH ArchC é interpretado completamente, gerando um programa de caracterização por subcaso para todas as instruções.

O comando `special` ou `-s` tem como objetivo gerar novos campos com base na interpretação textual das descrições `set_asm` em situações onde não é possível gerar código *assembly* válido diretamente da interpretação ARCH ArchC. Um exemplo é a referência a endereços relativos em MIPS-I. Em geral, estes endereços são referenciados com uso de etiquetas textuais. Quando é necessário utilizar um número, é preciso adicionar os caracteres ".+" antes do valor. Além disso, para branches, o valor não pode exceder 16 bits, apesar de que pela leitura direta de ArchC, valores maiores seriam aceitáveis. Estas informações não existem dentro do arquivo ARCH ArchC e para estes casos existe `-s`:

```
-s "ima=ima,%exp(pcrel)=.+%ima"
```

A primeira parte informa para acPowerGen gerar um comando novo chamado `ima`. Ele é tratado então exatamente como todos os demais nomes de região. Pode-se, então substituir um certo trecho de texto por outro com esta região nova. Isto é feito em seguida, trocando-se `exp(pcrel)` por `.+ima`, sendo `ima` um número aleatório de tamanho 16 definido em `-r`. Pode-se definir quantos comandos novos quanto se queira e realizar quantas substituições quanto se queira.

As chamadas utilizadas para gerar programas para arquiteturas MIPS-I e SPARCv8 são apresentadas no código 3.3. É importante destacar que a única entrada necessária para acPowerGen em Python é o arquivo padrão ISA de ArchC na arquitetura alvo, sem a necessidade de nenhuma modificação.

```
$ ./bin/acPowerGen.py -r "\"%r\"[1..31]" -v
"disp30=30,imm7=7,imm22=22,an='',cond='a',disp22=22:s,simm13=13:s" -c 1
sparc_isa.ac

$ ./bin/acPowerGen.py -r "\"$\"[1..31]" -v
"ima=16:a,imm=16,addr=26:a,shamt=5,rt=5" -s "ima=ima,%exp(pcrel)=.+%ima" -c 2
mips_isa.ac
```

Código 3.3: Chamadas de acPowerGen para arquiteturas MIPS-I e SPARCv8

Com estas entradas, é gerado um grupo de programas de caracterização que cobrem todas as instruções de uma arquitetura alvo. Vale notar que Python acPowerGen encarrega-se somente do corpo do código. Há um trecho de código em acSynth responsável pela geração do trecho de inicialização, gerando o `init.s`. É possível, com dois arquivos separados, delegar o controle de laço para o *testbench* do projeto, como relatado na seção 3.1.

A versão original de acPowerGen pode ser utilizada para obter um programa completo autocontido com controle de laço. Estes programas são compostos por quatro seções distintas. Uma região de inicialização a ser executada uma vez, um corpo principal com um mesmo operando repetido n vezes com parâmetros aleatórios; uma região de controle de laço, permitindo um número m de repetições; e, por fim, uma região de conclusão e

saída do programa. Este mesmo suporte está em desenvolvimento em acPowerGen em Python.

Desta forma, um programa gerado irá executar $n \cdot m$ instruções de mesmo operando. Com um valor suficientemente grande, é possível aplicar o método de Tiwari para caracterização. No código 3.4 pode-se ver um trecho de programa gerado automaticamente para os testes com MIPS.

```
globl main
main:
addi $7, $6, -26519
addi $19, $17, 23807
addi $10, $12, 7977
addi $13, $11, 16882
addi $3, $6, 124
addi $2, $20, 10232
addi $8, $7, 17293
addi $22, $3, -15821
(...)
```

Código 3.4: Pequeno excerto de instruções MIPS addi gerados automaticamente por acPowerGen

Nos programas de teste criados para este trabalho, foram escolhidos $n = 1000$ e $m = 10$ para SPARC totalizando 10.000 instruções em cada programa de caracterização. Para MIPS, foram escolhidos $n = 1000$ e $m = 100$ resultando em 100.000 instruções. Isto porque a simplicidade do MIPS permitiu maior geração de dados com um baixo incremento no tempo de simulação. Quantidades maiores de instruções mostraram desvios muito baixos na variável de potência com um aumento substancial do tempo de simulação.

No terceiro e último item requerido para o processo de caracterização, temos a necessidade de um *top level* para *testbench*. O *testbench* criado para o presente trabalho foi simples, alocando uma pequena memória RAM que é ligada diretamente ao processador. A memória é inicialmente alimentada com o programa de caracterização e, então, a estrutura é simulada para geração de atividade de transição do processador, armazenando os resultados em formato VCD.

O *testbench* precisa ser inteiramente criado manualmente, uma vez que não existe suporte a geração automática de testes sobre a plataforma ArchC. Apesar disto, seu desenvolvimento é de fácil execução e, em diversos cenários, já necessário para outros usos dentro de um projeto de desenvolvimento.

A plataforma acSynth cuida para que todos os programas gerados sejam traduzidos em um formato de arquivo legível para o *testbench* e, então, executa todos os programas de caracterização sobre o processador alvo. O resultado deste processo é uma lista de arquivos de atividade de transição, suficiente para estimar todas as instruções.

Em geral, uma instrução necessita de apenas um programa de caracterização para determinar seu perfil de consumo. No entanto, no caso de execução de instruções mais

complexas como `mult` e `div`, ou ainda no caso da caracterização de instruções de salto condicional, existem perfis energéticos completamente distintos entre a tomada de decisão de saltar ser positiva ou negativa. Em casos como este, é necessário realizar execuções em sub casos distintos e então obter resultados mais representativos. Por padrão, o processo de geração automática do banco de dados de instruções utiliza apenas uma referência por instruções, a primeira. Porém, manualmente pode-se atualizar o arquivo final no intuito de aprimorar os resultados gerados. Isso foi feito ao longo dos testes experimentais tanto em MIPS quanto em SPARC e será comentado no capítulo 4.

Ainda sobre saltos condicionais, analisar o consumo destes casos no método de Tiwari apresenta o desafio de não haver uma clareza da posição final do salto caso os parâmetros sejam totalmente aleatórios, transferindo o fluxo de controle para regiões não válidas de memória e quebra do controle de laço gerado por acPowerGen. A solução adotada para este caso foi a transposição do controle de fluxo para o *testbench*, como previamente já relatado. Com isto, o *testbench* sempre garante que a próxima instrução fornecida será uma nova instrução de mesmo tipo e válida, com parâmetros aleatórios, mesmo que PC aponte para um endereço completamente díspare. Isto é feito ignorando-se PC para acesso dos dados na memória de teste. Esta consideração foi feita no desenvolvimento da caracterização apresentada neste trabalho.

Note que, nestes casos, os controles periféricos criados por acPowerGen não são necessários. O controle forçado via *testbench* pode ser aplicado em outros casos com resultados equivalentes, sem perdas das características exigidas pelo método de Tiwari em relação ao sistema com controle dependente do fluxo do programa de caracterização. O resultado final é um conjunto de arquivos de atividade de transição gerados pela execução dos códigos *assembly* com $n \cdot m$ repetições de instrução, independente da opção escolhida. Os diagramas das figuras 3.4 e 3.5 esquematizam simplificadamente o funcionamento dos processos. O primeiro, com controle interno inerente ao programa. O segundo, com controle externo pelo *testbench*. O controle interno segue exatamente o esquema definido pelos trabalhos de Auler [28] e é direto, exigindo esforço baixo por parte do projetista no desenvolvimento do *testbench*. O controle externo é mais complexo de ser elaborado, porém, permite controle total do fluxo e bloqueio contra acesso de memória inválida.

Executados os testes sobre o *testbench* escolhido, são gerados arquivos de transição de atividade com o histórico da simulação. Estes arquivos de atividade são então repassados a uma ferramenta de análise de consumo de energia em RTL, como Xpower ou PowerPlay. A plataforma acSynth se responsabiliza por, automaticamente, executar todos os arquivos de atividade de transição na ferramenta de análise de consumo. O resultado deste processo é uma lista de relatórios de consumo.

No fim, tem-se relatórios de energia que podem então ser organizados em um único arquivo de dados que servirão de entrada ao futuro fluxo de simulação. Isto permite uma

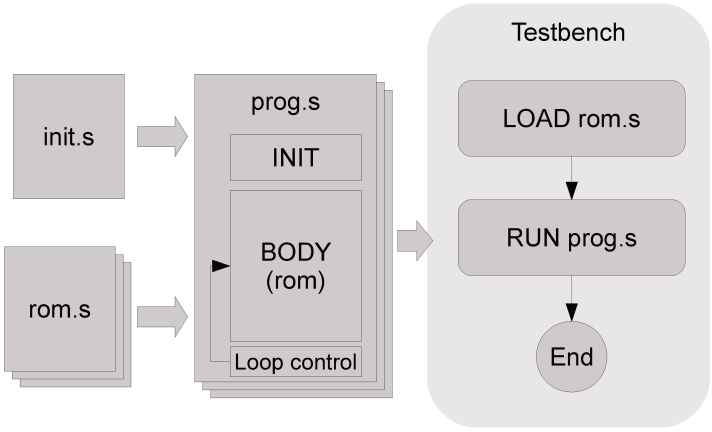


Figura 3.4: Controle interno do laço do código, gerado nos padrões estabelecidos pelos trabalhos de Auler

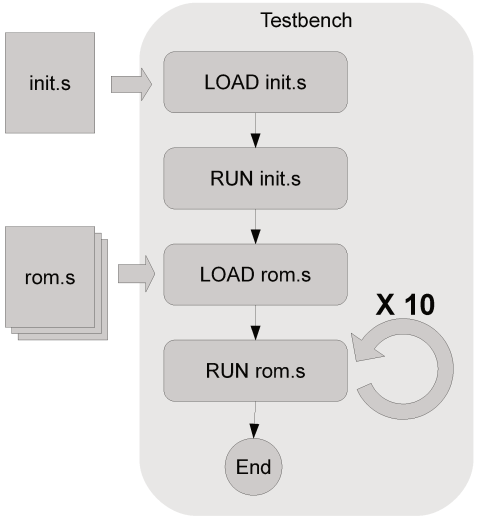


Figura 3.5: Controle externo do laço de código com dependência do *testbench*. Mais complexo, porém, permite maior controle do acesso externo e garantia de região de memória válida

manipulação simplificada por parte das funções que irão utilizar a informação, como o caso das funções presentes em acPower.

Os bancos de dados podem apresentar ainda outros dados relevantes, como informação do consumo em condição de *stall*, a frequência de operação, rótulos que auxiliem na identificação. Um exemplo de arquivo deste tipo pode ser visto no código 3.5. Detalhes completos para MIPS-I e SPARCv8 são encontrados no capítulo 4.

```
# Number of profiles
1
# ProfileID, Frequency, Scale,EPI,Name,Description
0,50,1e6,1e-9,"Leon3 at 50MHz","Using Xilinx tools, Spartan3 at 50MHz"
# Stall consumption
0
1,call,0.10
2,nop,0.15
3,sethi,0.24
4,ba,0.10
5,bn,0.15
6,bne,0.15
7,be,0.10
8,bg,0.15
(...)
```

Código 3.5: Parte do início de um banco de dados de caracterização para SPARCv8 da caracterização de Leon3 a 50 MHz

Vale salientar que é possível, ainda, inserir mais de um grupo de dados de caracterização em um mesmo arquivo de tabelas de consumo. Isto é particularmente útil caso um processador possa trabalhar em diferentes modos de operação. Desta forma, é possível simular esta mudança de modo de operação e observar os impactos do procedimento no consumo de energia do processador.

3.2.2 A etapa de simulação

A segunda etapa do processo é a simulação. O principal foco desta etapa é a realimentação dos dados caracterizados para dentro do ArchC, permitindo que gerem-se relatórios de consumo juntamente com a execução de programas simulados sobre acSim. Este subfluxo encontra-se resumido na figura 3.6

A principal contribuição desta etapa do trabalho vem da integração de acPower e PowerSC ao modelo SystemC gerado a partir de ArchC. Isto porque, como detalhado nas seções 2.3.5, acPower era uma ferramenta externa a ArchC, fazendo uso dos dados retornados pela simulação sem interação direta, limitando-o a valores fechados de consumo obtidos após a simulação ter ocorrido. Além disso, como também relatado na seção 2.3.5, PowerSC não teve como foco ArchC, mas modelos genéricos desenvolvidas em SystemC.

Apesar de PowerSC ser uma ferramenta poderosa, com uma estrutura bem elaborada para lidar com módulos genéricos descritos em SystemC, ele não foi aplicado para uso em

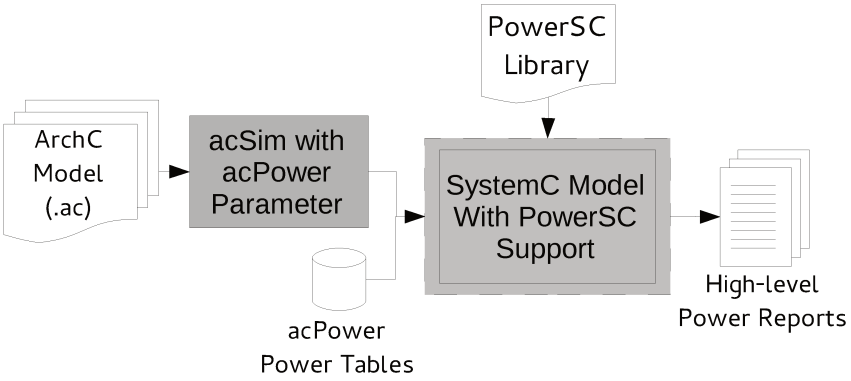


Figura 3.6: Extração de relatórios de consumo de energia através de modelos de simulação SystemC. A plataforma acSynth retorna as informações de energia para dentro do ArchC através da integração de acPower e PowerSC ao modelo SystemC

ArchC. Ao integrar PowerSC ao modelo de ArchC, todo este ferramental se tornou acessível permitindo que os estudos e resultados desenvolvidos em [33] e [39] fossem aplicáveis a processadores em ADL ArchC. Temos em acSynth a possibilidade de simular modelos em 3 níveis diferentes de descrição: arquitetural, sistêmico e adicionalmente RTL, devido ao suporte prévio fornecido por PowerSC.

Só com auxílio de acPower foi possível integração PowerSC a acSim. O mais interessante com respeito a este trabalho foi o fato de PowerSC prever blocos genéricos de informação de consumo de energia, além de capacidades para relatórios de sistemas complexos, prevendo múltiplos modos. Isto abre a possibilidade de utilizar qualquer tipo de algoritmo para aquisição de dados e cálculo de consumo de energia, além de permitir diversos níveis de simulação coexistirem numa mesma plataforma de simulação.

Esta flexibilidade de PowerSC permitiu, com pouco esforço, redirecionar as funções inicialmente focadas em consumo através de coleta de atividade de transição de módulos genéricos para consumo através de coleta de número de instruções executadas de módulos processadores.

Além disso, cada novo módulo é considerado como uma instância completamente independente dentro de PowerSC, com controles, dados de entrada e dados de saída próprios. Isto permite a presença de várias instâncias de um mesmo tipo de módulo na simulação. Com este recurso, é possível simular sistemas *multicore* de descrições de arquitetura ArchC, recurso que foi utilizado com sucesso nos trabalhos de Liana et al. [3].

Feita a integração, basta, deste momento em diante, realizar simulação com execução de programas genéricos e analisar os relatórios gerados. A execução da etapa de simulação de acSynth gera dois relatórios distintos de consumo de energia.

O primeiro, diretamente na saída padrão da execução da simulação, trata-se do relatório padrão de PowerSC com a lista dos módulos, um resumo dos valores obtidos e o

total geral da simulação. Um exemplo deste formato de relatório pode ser visto na figura 3.7.

O processo de integração entre PowerSC e ArchC foi realizado através de uma adaptação direta de um tipo de classe já existente em PowerSC. Trata-se da classe *cell* que tinha por objetivo simular células RTL em baixo nível dentro da infraestrutura em nível sistêmico de PowerSC. Acontece que os requisitos de uso são muito parecidos com os necessários para adicionar um módulo em nível de arquitetura. ArchC fez uso desta classe e desta forma, conseguiu agregar ArchC, PowerSC e acPower em um tempo muito curto de desenvolvimento. Como um efeito colateral da adaptação, as simulações de ADLs contabilizam somente no campo de consumo de energia agregada, o que faz sentido para células RTL mas nem tanto para processadores em ADL. Apesar do problema conceitual, a informação é válida e os valores totais de consumo de energia do sistema permanecem corretos.

O procedimento ideal para este caso é a criação de uma classe inteiramente nova voltada para ADL. Este processo será realizado futuramente.

```
ArchC: Simulation statistics
    Times: 0.02 user, 0.00 system, 0.03 real
    Number of instructions executed: 1050054

-------------------------------- POWER REPORT ---------------------------------
L  - Cell  Name - Cell Type - Leakage Power - Internal Power - Aggregate Power
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
RT - sparcv8    - Processor - 0.000000e+00  - 0.000000e+00   - 1.633330e-01W
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Summary:
  Switching power :    0.000000e+00W
  Internal power  :    0.000000e+00W
  Leakage power   :    0.000000e+00W
  Aggregate power :    1.633330e-01W
                     ----------------
  TOTALS          :    1.633330e-01W
-------------------------------------------------------------------------------
```

Figura 3.7: Um exemplo de relatório padrão gerado por PowerSC com informações de um processador SPARCv8 descrito em ArchC

O segundo relatório é retornado em um arquivo CSV que apresenta janelas de consumo no tempo. A vantagem deste relatório é que ele fornece uma visão mais detalhada, não só um número fechado ao fim da execução. Com isto é possível não só saber quanta energia a execução de um dado programa alvo consumiu, mas também, de que forma, com que frequência e em qual momento este consumo apresentou vales e picos. Um trecho de exemplo deste tipo de relatório encontra-se na figura 3.8. A primeira coluna é

o perfil de execução em atividade no momento da estimativa no tempo, importante para informar mudanças de contexto caso haja mais de um na tabelas de consumo acPower da entrada. A segunda coluna é o valor de tempo em segundos estimado, calculado com base na informação de frequência também fornecido pelo arquivo de tabelas acPower; neste exemplo, a frequência de execução é de 100MHz. A terceira coluna apresenta o número da janela de instruções atual; neste exemplo, cada janela tem um tamanho fixo ajustável de instruções, neste caso, 1.000.000. A quarta e última coluna apresenta o valor de consumo de energia dentro daquela janela.

```
0,0.0100000000,1,0.2159880043
0,0.0200000000,2,0.2162745213
0,0.0300000000,3,0.2161019561
0,0.0400000000,4,0.2159379301
0,0.0500000000,5,0.2157549549
0,0.0600000000,6,0.2157320097
0,0.0700000000,7,0.2160721487
0,0.0800000000,8,0.2158551602
0,0.0900000000,9,0.2152191597
0,0.1000000000,10,0.2150747052
0,0.1100000000,11,0.2157043314
0,0.1200000000,12,0.2151162835
0,0.1300000000,13,0.2154721012
0,0.1400000000,14,0.2146962416
0,0.1500000000,15,0.2148985787
0,0.1600000000,16,0.2150786065
0,0.1700000000,17,0.2152264270
0,0.1799999999,18,0.2149242751
0,0.1899999999,19,0.2150620853
0,0.1999999999,20,0.2149782033
(...)
```

Figura 3.8: Exemplo de relatório no tempo

Esse relatório de energia no tempo traz um outro nível completamente diferente na discussão do consumo de energia sobre plataforma ArchC, trazendo a decisão do algoritmo adotado na execução de uma tarefa para dentro do campo de análise de consumo de energia.

É possível, por exemplo, simular situações em que há um teto de consumo máximo que não pode ser excedido e entender a diferença no tempo de execução de um algoritmo com e sem o teto. É possível ainda focar esforços em melhorias de arquitetura sobre as instruções mais utilizadas em uma aplicação ASIP com foco em reduzir o pico de consumo da execução destas instruções. Outra aplicação seria avaliar o comportamento de programas para definição de gatilhos para mudanças de frequência, voltagem ou política de uso, com o intuito de reduzir o consumo geral. Pode-se também estudar o consumo de conjuntos de instruções com funcionalidade equivalente no intuito de encontrar os conjuntos que consomem menos, direcionando compiladores a aplicar certos pré-processamentos em detrimento a outros, focando em redução do consumo.

Resultados concretos serão apresentadas no capítulo 4.

3.3 Desenvolvimento RTL com base em *templates* por acSynth

O acSynth funciona como uma ferramenta de integração com a possibilidade de agregar novas funcionalidades e permitir resultados interessantes do cruzamento das habilidades e recursos de cada ferramenta isolada.

Neste trabalho realizamos também a tarefa de integrar a ferramenta acRTL ao fluxo de desenvolvimento sobre acSynth. Uma vez agregado o aparato para análise de consumo, obtivemos um cenário novo no desenvolvimento RTL sobre ArchC. Com a atual plataforma é possível desenvolver um processador baseado nas ferramentas acRTL e gsc. Pautando o desenvolvimento por acPower, podemos direcionar os esforços na redução de consumo de energia.

A integração de acRTL à *framework* pode ser vista na figura 3.9. Note que se trata de uma expansão da *framework* inicialmente apresentada na figura 3.2.

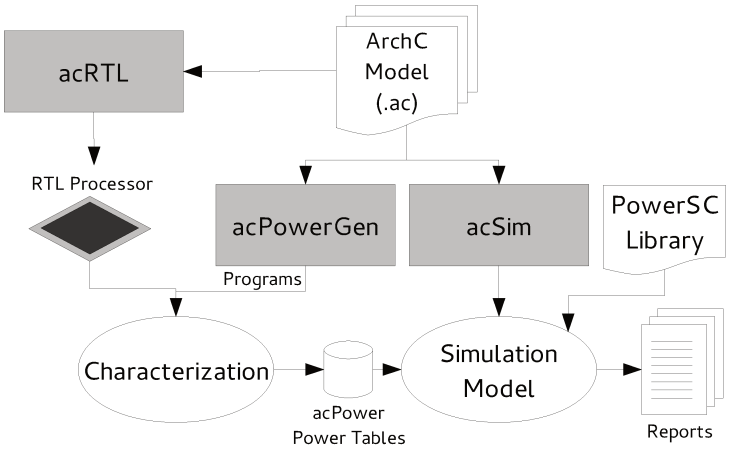


Figura 3.9: Integração do fluxo de desenvolvimento acRTL a acSynth. É possível, desta forma, gerar um processador a partir de ArchC e, ainda, obter informações de consumo de energia do circuito elaborado

O fluxo de desenvolvimento acRTL se baseia em duas ferramentas principais: a ferramenta acRTL propriamente, que gera um *template* SystemC com base nos arquivos ArchC de arquitetura e de conjunto de instruções; e a ferramenta gsc, um tradutor SystemC para Verilog especialmente focado no formato gerado no *template*. Este fluxo pode ser visto na figura 3.10.

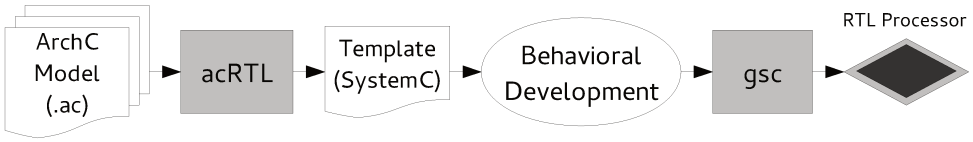


Figura 3.10: Fluxo de desenvolvimento acRTL. Como entrada, temos os arquivos ArchC padrões. Por fim, a ferramenta gsc gera Verilog sintetizável

Note que existe um trabalho por parte do projetista para desenvolver as descrições comportamentais das instruções. Essa atividade é realizada de forma simplificada em SystemC. Isto porque o *template* gerado retira a necessidade de desenvolver todo o controle de *fetch* e *decode*. São criadas instâncias específicas por instrução. Desta forma, é possível focar esforços, uma instrução por vez. Há ainda uma região independente e apropriada para tratar *hazards* e *forwarding*.

Uma vez gerado um Verilog válido, um processador desenvolvido pela metodologia acRTL e um processador genérico desenvolvido pelos métodos habituais recaem no mesmo cenário do ponto de vista das demais ferramentas em acSynth. Por este motivo, todo o fluxo de caracterização e simulação são válidos e aplicáveis aos processadores gerados por acRTL. A figura 3.11 apresenta um esquema simplificado de todas a etapas do processo.

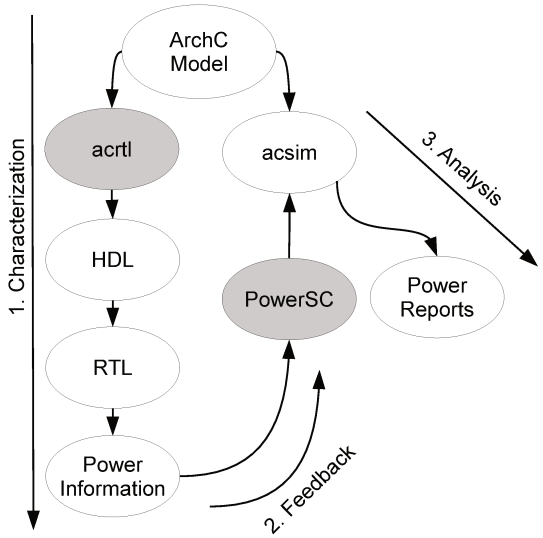


Figura 3.11: Fluxo simplificado das etapas da geração e análise de processador acRTL

Desta integração derivam aplicações interessantes, como realizar trabalhos comparativos entre processadores acRTL e processadores genéricos de mesma arquitetura. Ou ainda utilizar acSynth como base de testes para melhora incremental de um processador gerado por acRTL.

Resultados práticos serão apresentadas no capítulo 4.

Capítulo 4

Resultados Experimentais

Os trabalhos desenvolvidos em acSynth foram comprovados através de testes e experimentos realizados sobre as arquiteturas MIPS e SPARCv8. Foram realizados procedimentos de caracterização e simulação para processadores descritos em HDL:

- MIPS-I Plasma;
- SPARCv8 Leon3;
- MIPS-I desenvolvido por fluxo acRTL.

Os objetivos principais dos experimentos foram:

- Comprovar a eficácia de acSynth e seu fluxo de projeto;
- Validar seu uso com mais de uma ferramenta CAD;
- Validar seu uso com mais de uma arquitetura suportada por ArchC;
- Demonstrar a integração de acRTL ao fluxo ArchC com a geração de relatórios.

Os resultados encontram-se separados por processador estudado. Para comprovar a compatibilidade com múltiplas plataformas, o processador MIPS-I Plasma foi caracterizado tanto sob a plataforma Xilinx quanto a plataforma Altera. Para comprovar a compatibilidade com múltiplos modelos, o processador SPARCv8 Leon3 foi caracterizado sob plataforma Xilinx.

Em todos os casos e testes, apenas a energia dinâmica foi considerada pois este é o fator diretamente afetado pela arquitetura do processador e pelos programas executados sobre as simulações. No caso da plataforma Altera, há uma distinção entre a energia dinâmica interna e a energia das portas do módulo. Estas duas informações foram somadas para calcular a energia dinâmica total do sistema.

É importante dizer que não há nenhum impedimento para aplicação da metodologia de análise de processadores com uso da plataforma acSynth através de fluxos de desenvolvimento ASIC. Isto porque a metodologia de caracterização é independente da ferramenta CAD utilizada para se obter os dados de consumo de energia. Uma vez implementado o suporte a um CAD específico dentro da plataforma, acSynth poderá gerar relatórios automáticos seguindo os mesmo passos descritos para as ferramentas da Xilinx e da Altera.

4.1 Processador Plasma

Plasma é uma implementação HDL livre da arquitetura MIPS-I. O código do projeto encontra-se disponível no grupo OpenCore [43]. Seus desenvolvedores mantêm um servidor funcional sobre uma plataforma em Xilinx FPGA e sistema operacional Plasma RTOS para validar sua aplicação prática.

O processador Plasma foi escolhido para experimentos por ser de fácil acesso, além de ser um projeto previamente validado. Por ser uma implementação MIPS-I, Plasma é compatível com um modelo ADL ArchC estável.

Plasma foi sintetizado utilizando tanto a plataforma Xilinx quanto a plataforma Altera. O principal objetivo foi demonstrar que o fluxo de projeto sobre acSynth era factível para mais de uma ferramenta de síntese, demonstrando independência do fluxo de projeto escolhido.

Como o objetivo do projeto é analisar apenas o núcleo de processamento, a cache foi removida para análise. Ou seja, o *netlist* final foi gerado apenas com o *pipeline* principal e os coprocessadores como o mul/div. A tarefa foi simples uma vez que Plasma não é um processador de grande porte.

Para cada projeto foi gerado um *workspace* acSynth independente.

4.1.1 Plataforma Xilinx

Na plataforma Xilinx, foi escolhido desenvolver o sistema sobre a FPGA Spartan3E XC3S1200E a 100MHz. Após o *workspace* de acSynth ter sido criado, o pacote de Plasma foi baixado diretamente do site dos desenvolvedores para a pasta `char_target`. O projeto foi sintetizado com uso da ferramenta ISE gerando uma *netlist* pós-roteada. Modificações no código original só foram necessários para parametrizar o uso de tipo de memória para as compatíveis com plataforma Xilinx. Maiores detalhes podem ser encontrados na documentação própria de Plasma.

Após isto, um *testbench* em Verilog foi desenvolvido segundo as especificações relatadas na seção 3.1, sub-item `build`. Depois que ele foi alocado na pasta `char_target` e referenciado no arquivo de configuração, a caracterização pode ser executada. A fer-

ramenta de simulação para geração dos arquivos de atividade de transição utilizada foi Mentor Modelsim. O tempo de cada simulação nesta etapa flutuou entre alguns segundos para instruções simples como `nop`, até 5 minutos de simulação para instruções como `jump` e `branch`, com tempo médio de 1 minuto e 50 segundos. Após esta etapa, foram gerados os relatórios de potência. Isto foi feito com uso da ferramenta Xpower. Nesta etapa, o tempo de análise flutuou entre aproximadamente 30 segundos a até 4 minutos, com tempo médio de 1 minuto e 40 segundos. A descrição MIPS em ArchC possui 59 instruções. Destas, 13 instruções não foram processadas, ou por serem redundantes (como no caso dos diversos tipos de *branches*, muito parecidos do ponto de vista de consumo de energia) ou por não serem passíveis de análise em regime, caso das instruções `break` e `syscall`. Ao todo, foram realizadas 46 gerações de arquivos VCD e relatórios de potência. Para gerar todos os arquivos de atividade de transição, o sistema levou ao todo 1 hora e 35 minutos aproximadamente. Para gerar todos os relatórios de consumo com a ferramenta Xpower, o sistema levou aproximadamente 1 hora e 28 minutos. O tempo para a montagem do banco de dados é desconsiderável. O tempo total de caracterização foi, portanto, de aproximadamente 3 horas. Ao fim, restavam algumas lacunas a serem completadas manualmente.

- lwl e lwr foram considerados com o mesmo valor de lw;
- swl e swr foram considerados com o mesmo valor de sw;
- bne, blez, bgtz, bltz, bgez, bltzal e bgezal foram considerados com o mesmo valor de beq;
- syscall e break foram desconsiderados e mantidos em zero.

Ao fim, foram obtidos os dados de energia por instrução apresentados na tabela 4.1.

Uma simulação foi realizada com o sinal de `mem_pause` do processador Plasma ativo, o que faz com que o *pipeline* de Plasma pare de processar instruções. O resultado foi uma simulação com ciclos em *stall* e o consumo obtido foi de 0.21 nJ por ciclo para este cenário.

Todos estes dados foram compilados no banco de dados de instruções e utilizados em testes com os *benchmarks* Mibench [42], Mediabench [44] e acStone [45].

Os resultados de simulação em Mibench pode ser vistos na tabela 4.2. Os resultados para Mediabench podem ser visto das tabelas 4.3 a 4.8.

O *benchmark* acStone foi especialmente importante para comparar a eficiência da simulação em nível ADL com a metodologia tradicional de simulação em RTL. Os resultados são apresentados na tabela 4.9. Vê-se um comparativo entre o valor de potência obtido

Tabela 4.1: Consumo de energia dinâmica para instruções Plasma em FPGA Spartan3E 1200

Instruction	**Energy (nJ)**
add	1.83
addi	1.64
addiu	1.64
addu	1.03
andi	0.89
beq	3.08
bgez	3.08
bgezal	3.08
bgtz	3.08
blez	3.08
bltz	3.08
bltzal	3.08
bne	3.08
div	3.53
divu	3.47
instr_and	0.75
instr_break	0
instr_nor	1.53
instr_or	1.01
instr_xor	1.67
j	3.5
jal	0.85
jalr	6.37
jr	6.14
lb	4.44
lbu	2.47
lh	4.55
lhu	2.57
lui	0.88
lw	3.34

Instruction	**Energy (nJ)**
lwl	3.34
lwr	3.34
mfhi	0.61
mflo	0.61
mthi	0.99
mtlo	0.94
mult	1.43
multu	1.4
nop	0.51
ori	0.98
sb	5.39
sh	5.68
sll	0.76
sllv	0.76
slt	1.35
slti	1.23
sltiu	1.23
sltu	1.35
sra	0.75
srav	1.07
srl	0.74
srlv	0.75
sub	1.19
subu	1.19
sw	5.76
swl	5.76
swr	5.76
sys_call	0
xori	1.35

Tabela 4.2: Mediabench *benchmarks*: estimativas de consumo de energia

Program	**Instr (M)**	**Power (mW)**	**Energy (mJ)**	t_{Sim} **(s)**	$\approx t_{Sim}$ **RTL (day)**
timing	852	175	1489	26	726
rawcaudio	7	174	13	< 1	6
rawdaudio	6	181	11	< 1	5
toast	221	216	477	7	188
untoast	61	195	119	2	52
cjpeg	17	211	35	1	14
djpeg	5	194	10	< 1	4
mpeg2encode	11474	208	23832	364	9772
mpeg2decode	3772	219	8248	134	3212
pegwit gen	13	181	23	< 1	11
pegwit enc	31	182	57	1	27
pegwit dec	17	188	33	1	15

Tabela 4.3: Mibench *benchmarks*: estimativas de consumo para algoritmos *Consumer*

Program	**Test**	**Instr (M)**	**Power (mW)**	**Energy (mJ)**	t_{Sim} **(s)**	$\approx t_{Sim}$ **RTL (day)**
lame	small	8034	205.97	16547	276	6842
lame	large	94315	207.68	195872	3047	80324
cjpeg	small	29	211.42	62	1	25
djpeg	small	9	203.06	18	< 1	7
cjpeg	large	109	209.74	229	3	93
djpeg	large	29	207.54	61	1	25

Tabela 4.4: Mibench *benchmarks*: estimativas de consumo para algoritmos *Automotive*

Program	**Test**	**Instr (M)**	**Power (mW)**	**Energy (mJ)**	t_{Sim} **(s)**	$\approx t_{Sim}$ **RTL (day)**
basicmath	small	1361	195.16	2657	58	1159
basicmath	large	22269	208.70	46476	765	18966
bitcnts	small	46	195.74	89	1	39
bitcnts	large	684	195.69	1339	19	583
qsort	small	14	261.01	38	1	12
qsort	large	989	216.12	2138	31	843
susan	small/corner	3	202.66	7	< 1	3
susan	small/edge	7	193.06	13	< 1	6
susan	small/smooth	35	163.28	58	1	30
susan	large/corner	45	203.90	91	1	38
susan	large/edge	177	187.96	333	5	151
susan	large/smooth	423	155.55	659	12	361

Tabela 4.5: Mibench *benchmarks*: estimativas de consumo para algoritmos *Network*

Program	**Test**	**Instr (M)**	**Power (mW)**	**Energy (mJ)**	t_{Sim} (s)	$\approx t_{Sim}$ **RTL (day)**
dijkstra	small	59	194.18	115	2	51
dijkstra	large	285	192.36	549	8	243
patricia	small	289	212.05	613	9	246
patricia	large	1831	211.86	3879	65	1559

Tabela 4.6: Mibench *benchmarks*: estimativas de consumo para algoritmos *Office*

Program	**Test**	**Instr (M)**	**Power (mW)**	**Energy (mJ)**	t_{Sim} (s)	$\approx t_{Sim}$ **RTL (day)**
search	small	< 1	244.14	1	< 1	< 1
search	large	7	244.05	17	< 1	6

Tabela 4.7: Mibench *benchmarks*: estimativas de consumo para algoritmos *Security*

Program	**Test**	**Instr (M)**	**Power (mW)**	**Energy (mJ)**	t_{Sim} (s)	$\approx t_{Sim}$ **RTL (day)**
rijndael	large/decode	361	208.34	752	12	308
rijndael	small/encode	34	206.88	70	1	29
rijndael	small/decode	35	208.34	72	1	30
rijndael	large/encode	351	206.88	726	12	299
sha	small	13	181.76	24	< 1	11
sha	large	136	181.73	247	4	116

Tabela 4.8: Mibench *benchmarks*: estimativas de consumo para algoritmos *Telecommunication*

Program	**Test**	**Instr (M)**	**Power (mW)**	**Energy (mJ)**	t_{Sim} **(s)**	$\approx t_{Sim}$ **RTL (day)**
adpcm timing	small	644	174.76	1126	23	549
adpcm rawcaudio	large	689	172.87	1191	24	587
adpcm rawdaudio	large	539	179.06	965	20	459
adpcm timing	large	688	174.76	1203	25	586
adpcm rawcaudio	small	35	174.05	60	1	29
adpcm rawdaudio	small	27	180.54	49	1	23
crc_32	small	32	250.60	79	1	27
crc_32	large	615	250.60	1541	22	524
fft	4k	761	205.28	1561	29	648
fft	inv 8k	1823	205.00	3737	67	1553
fft	32k	15244	205.19	31280	534	12983
fft	inv 32k	14750	204.74	30201	474	12562
gsm toast	small	33	216.35	71	1	28
gsm untoast	small	10	199.35	19	< 1	8
gsm toast	large	1763	216.22	3813	58	1502
gsm untoast	large	523	199.27	1043	17	446

pelo acSim e o valor de potência obtido através de Xpower que utilizamos como referência. Na última coluna encontramos o erro inerente. Vamos aprofundar um pouco mais a respeito dos resultados apresentados em acStone. Estes testes demonstram os limites do método de simulação sobre acSim.

Tabela 4.9: Resultados da simulação com acStone e estudo do erro efetivo

Program	*instr*	$instr_{eff}$	$acSim(mW)$	$W_{eff}(mW)$	$Xpower(mW)$	*err*	err_{eff}
000.main	19	0	198.21	21.00	21.59	818.07%	2.73%
011.const	34	15	271.35	131.45	146.77	84.88%	10.44%
012.const	34	15	271.35	131.45	146.77	84.88%	10.44%
013.const	34	15	278.18	134.46	145.6	91.06%	7.65%
014.const	34	15	278.18	134.46	145.6	91.06%	7.65%
015.const	37	18	257.08	135.85	157.61	63.11%	13.81%
016.const	37	18	257.08	135.85	157.61	63.11%	13.81%
017.const	56	37	277.11	190.21	178.44	55.29%	6.60%
018.const	56	37	277.11	190.21	178.44	55.29%	6.60%
021.cast	44	25	310.68	185.59	186.75	66.36%	0.62%
022.cast	44	25	312.32	186.52	186.56	67.41%	0.02%
023.cast	75	56	267.63	205.15	202.47	32.18%	1.32%
024.cast	44	25	315.95	188.59	185.54	70.29%	1.64%
025.cast	75	56	269.71	206.70	203.91	32.27%	1.37%
026.cast	68	49	292.79	216.85	214.6	36.44%	1.05%
027.cast	49	30	299.63	191.59	187.82	59.53%	2.01%
031.add	259	240	283.03	263.80	240.71	17.58%	9.59%
032.add	259	240	294.29	274.24	245.03	20.10%	11.92%
033.add	257	238	309.14	287.84	251.67	22.84%	14.37%

034.add	587	568	284.81	276.27	231.02	23.28%	19.59%
041.sub	259	240	284.51	265.18	243.22	16.98%	9.03%
042.sub	259	240	295.77	275.61	246.66	19.91%	11.74%
043.sub	257	238	310.02	288.65	251.35	23.34%	14.84%
044.sub	559	540	295.12	285.80	230.46	28.06%	24.01%
051.mul	96	77	307.93	251.14	102.28	201.06%	145.54%
052.mul	96	77	279.20	228.10	101.87	174.07%	123.91%
053.mul	129	110	322.29	277.92	105.16	206.48%	164.28%
054.mul	129	110	291.60	251.74	105.23	177.10%	139.23%
055.mul	231	212	295.42	272.85	93.73	215.18%	191.10%
056.mul	231	212	295.25	272.69	80.6	266.32%	238.33%
057.mul	261	242	250.45	233.75	83.19	201.06%	180.98%
058.mul	265	246	236.92	221.44	82.71	186.45%	167.73%
061.div	192	173	293.85	266.85	107.52	173.30%	148.18%
062.div	182	163	271.68	245.51	98.5	175.81%	149.25%
063.div	192	173	302.68	274.81	107.44	181.72%	155.78%
064.div	182	163	280.86	253.73	99.7	181.70%	154.49%
065.div	195	176	284.08	258.45	109.9	158.49%	135.17%
066.div	183	164	290.68	262.68	106	174.23%	147.81%
067.div	1521	1502	213.94	211.53	232.79	8.10%	9.13%
068.div	1170	1151	228.01	224.65	231.48	1.50%	2.95%
071.bool	281	262	284.73	266.90	240.51	18.39%	10.97%
072.bool	281	262	296.51	277.88	246.96	20.06%	12.52%
073.bool	285	266	314.09	294.56	255.36	23.00%	15.35%
074.bool	557	538	323.24	312.93	255.86	26.34%	22.31%
075.bool	125	106	254.10	218.67	223.97	13.45%	2.37%
081.shift	183	164	312.44	282.18	235.6	32.61%	19.77%
082.shift	183	164	325.22	293.64	237.28	37.06%	23.75%
083.shift	191	172	311.18	282.32	239.49	29.94%	17.88%
084.shift	423	404	290.97	278.84	234.71	23.97%	18.80%
085.shift	154	135	177.50	158.19	196.3	9.58%	19.41%
111.if	318	299	275.45	260.25	228.4	20.60%	13.94%
112.if	318	299	244.47	231.12	222.97	9.64%	3.66%
113.if	318	299	278.09	262.73	228.42	21.75%	15.02%
114.if	318	299	246.96	233.46	225.09	9.72%	3.72%
115.if	318	299	259.32	245.08	226.82	14.33%	8.05%
116.if	318	299	259.32	245.08	225.71	14.89%	8.58%
117.if	514	495	246.20	237.88	226.78	8.56%	4.89%
118.if	514	495	246.20	237.88	227.13	8.40%	4.73%
119.if	275	256	210.07	197.00	202.16	3.91%	2.55%
121.loop	959	940	261.06	256.30	223.72	16.69%	14.56%
122.loop	959	940	248.47	243.96	211.43	17.52%	15.38%
123.loop	1758	1739	221.81	219.64	227.59	2.54%	3.49%
124.loop	2199	2180	221.24	219.51	238.11	7.09%	7.81%
125.loop	1597	1578	243.21	240.56	205.93	18.10%	16.82%
126.loop	2241	2222	213.55	211.92	131.58	62.30%	61.05%
131.call	111	92	288.81	242.97	255.9	12.86%	5.05%
132.call	271	252	292.45	273.42	319.15	8.36%	14.33%
133.call	2887	2868	220.25	218.94	234.1	5.92%	6.48%
134.call	43261	43242	265.51	265.41	233.21	13.85%	13.81%
141.array	1602	1583	200.13	198.01	184	8.77%	7.61%
142.array	94771	94752	168.42	168.39	128.71	30.86%	30.83%
143.array	18506	18487	223.64	223.43	347.07	35.56%	35.62%
144.array	22042	22023	199.77	199.62	349.07	42.77%	42.81%
145.array	22640	22621	200.99	200.84	341.89	41.21%	41.26%

146.array	29248	29229	225.84	225.71	319.4	29.29%	29.33%

Em primeiro lugar, vale lembrar que o método adotado neste trabalho foca na energia dinâmica em regime de programas executados sobre as arquiteturas alvo. Ou seja, a energia do *setup* inicial do sistema não é tratada. É esperado que exista um comportamento totalmente diferenciado nos instantes iniciais do sistema, logo quando o processador é ligado. No entanto, este caso especial foi desconsiderado nas análises e o foco manteve-se no comportamento em regime.

Quando realizamos os testes com acStone o problema gerado por esta simplificação surgiu claramente porque vários dos programas de teste são pequenos. Há um número considerável de programas em acStone com menos de 100 instruções. Nestes casos, o comportamento inicial do sistema não é desprezível.

Houve outro aspecto da metodologia dos testes que provocou aumento representativo dos erros: o trecho de inicialização de sistema gerado pelo compilador. É sabido que todo programa compilado necessita de um trecho de código para inicializar variáveis, preparar recursos como *stack* e *heap*, ou configurar parâmetros para tratamento de chamadas de sistema. Este código existe nos programas compilados para execução em acSim e, portanto, são executados e contabilizados. Porém, o trecho de inicialização não pode ser comparado entre o resultado da simulação em nível ADL e a simulação em RTL. Fundalmentalmente, o código compilado para o processador decrito em HDL recebe um código de inicialização diferente daquele gerado pelo compilador para execução em acSim. Não há compromisso com correspondências neste aspecto e, portanto, não há possibilidade de casar estas informações de forma consistente. Portanto, ambos os trechos de código de inicialização precisam ser desconsiderados. De fato, o trecho de inicialização foi ignorado na simulação RTL. Isto foi possível porque os programas do *benchmark* acStone são muito simples, nem mesmo utilizando *stack* para funcionar. Em ArchC, é preciso ignorar a inicialização da mesma forma para, então, haver uma comparação coerente entre os dados.

Essa diferença criou uma disparidade nos dados, e os erros gerados por esta situação foram expressivos e precisaram ser tratados. Para tanto, utilizamos o programa `000.main` como base para definir o tamanho do trecho de código de inicialização gerado pelo compilador, uma vez que este programa não realiza nenhuma operação útil. O programa `000.main` pode ser visto no código 4.1 e as constantes `BEGINCODE` e `ENDCODE` não foram definidas. Constatamos que o número de instruções de inicialização era igual a 19 a partir do relatório gerado pela execução deste programa em acSim.

```
/* The file begin.h is included if compiler flag -DBEGINCODE is used */
#ifdef BEGINCODE
#include "begin.h"
#endif

int main() {
```

```
  return 0;
  /* Return 0 only */
}

/* The file end.h is included if compiler flag -DENDCODE is used */
#ifdef ENDCODE
#include "end.h"
#endif
```

Código 4.1: Código do programa 000.main de acStone

O número de instruções executadas do trecho de código de inicialização gerado pelo compilador impacta de forma regular, independente do programa executado em acSim. Para estas instruções da etapa de inicialização, foi atribuído um valor de consumo em *stall*, que para Plasma é igual a 0.21 nJ. Com isto, pode-se definir um **consumo efetivo**, paralelo ao consumo obtido diretamente. A fórmula para este cálculo é:

$W_{eff} = (i_{init} * W_{stall} + i_{stable} * W_{dyn})/(i_{init} + i_{stable})$

Onde W_{eff} é a potência efetivamente em regime de consumo, $i_{init} = 19$ é o número de instruções em estágio inicial. $W_{stall} = 21mW$, é definido com base na energia de uma instrução em *stall*, 0.21 nJ, frequência de operação simulada 100MHz e o número IPC (*Instructions Per Cycle*) igual a 1. W_{dyn} é o valor dinâmico obtido do relatório gerado por acSim, com erro. i_{stable} é o total de instruções executadas quando o sistema se encontra em regime estável.

O que se busca aqui é solucionar os dois problemas de metodologia ao mesmo tempo, atenuando o efeito das instruções de inicialização compiladas junto com o código útil e, juntamente, criando uma modelagem das instruções de inicialização com uso do dado de consumo em *stall*. Com isso, os resultados se aproximam dos resultados das simulações em RTL, mais coerentes com a realidade.

O erro cai sistematicamente em todos os casos, especialmente nos programas menores, corroborando as suposições da origem da disparidade. De fato, utilizando a formulação do erro efetivo tem-se uma simulação do efeito do estágio de *setup* do processor. Assim, a presente dissertação considera os valores efetivos apresentados na tabela 4.9 como os corretos para análise.

Sobre estes valores é necessário fazer mais algumas considerações. Os erros das instruções das categorias `mul` e `div` foram muito altos. Isto porque as simulações dos programas acStone estressaram o coprocessador e este gerou estágios de *stall* durante a execução. Como acSim não simula em nível de ciclo de *clock*, não é possível, por enquanto, capturar comportamentos como este. Apesar do consumo de energia em Joules ser consistente e independente deste fator, o valor de potência em Watts é afetado pois esta informação é dependente do tempo. O mesmo número de instruções é executada mas dentro de uma janela de tempo muito maior, diminuindo o valor de potência. O resultado é que o valor obtido da execução da simulação RTL é muito menor em comparação com os números

da simulação ADL. Qualquer solução para este problema passa pela detecção de eventos ciclo a ciclo na simulação e, portanto, requer modificações no funcionamento atual de acSim. Em suma, é preciso que acSim apresente informação de tempo. Desta forma, seria possível simular e contabilizar as ocorrências de *stall*, elevando assim a precisão dos resultados da caracterização.

Foi verificado que existe um erro alto, porém não na mesma proporção dos problemas com div/mul, em casos onde as instruções de `load` e `store` são estressadas, utilizadas repetidamente. Em RTL são constatadas atividades de transição muito maiores que as simuladas em nível de arquitetura. É fato que existe um erro grande na obtenção dos dados de consumo quando há acesso a memória externa pelo método de Tiwari. Há uma super-estimativa em cenários como este porque acessos externos totalmente aleatórios não são uma regra. Supõe-se que os erros sejam consideravelmente reduzidos se adicionar um modelo de cache a simulação.

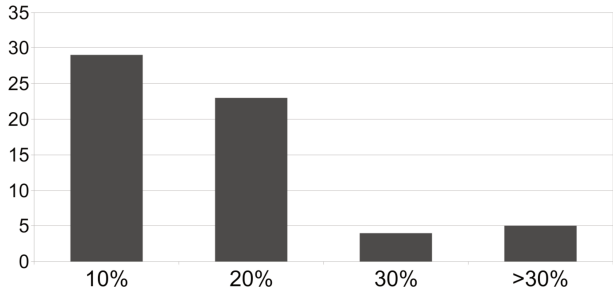


Figura 4.1: Distribuição do erro para os programas do *benchmark* acStone no processador Plasma, desconsiderando-se instruções do tipo `mul` e `div`

Um histograma com a distribuição do erro para o teste acStone pode ser visto na figura 4.1. Não foram incluídas instruções das categorias `div` e `mul` pelos motivos já citados de limitação do simulador. Gráficos com o perfil energético dos *benchmarks* executados encontram-se nas figuras 4.2 e 4.3, com exceção do *Mibench Search Small*, que é muito pequeno e não executou tempo suficiente para que fosse possível plotar um gráfico.

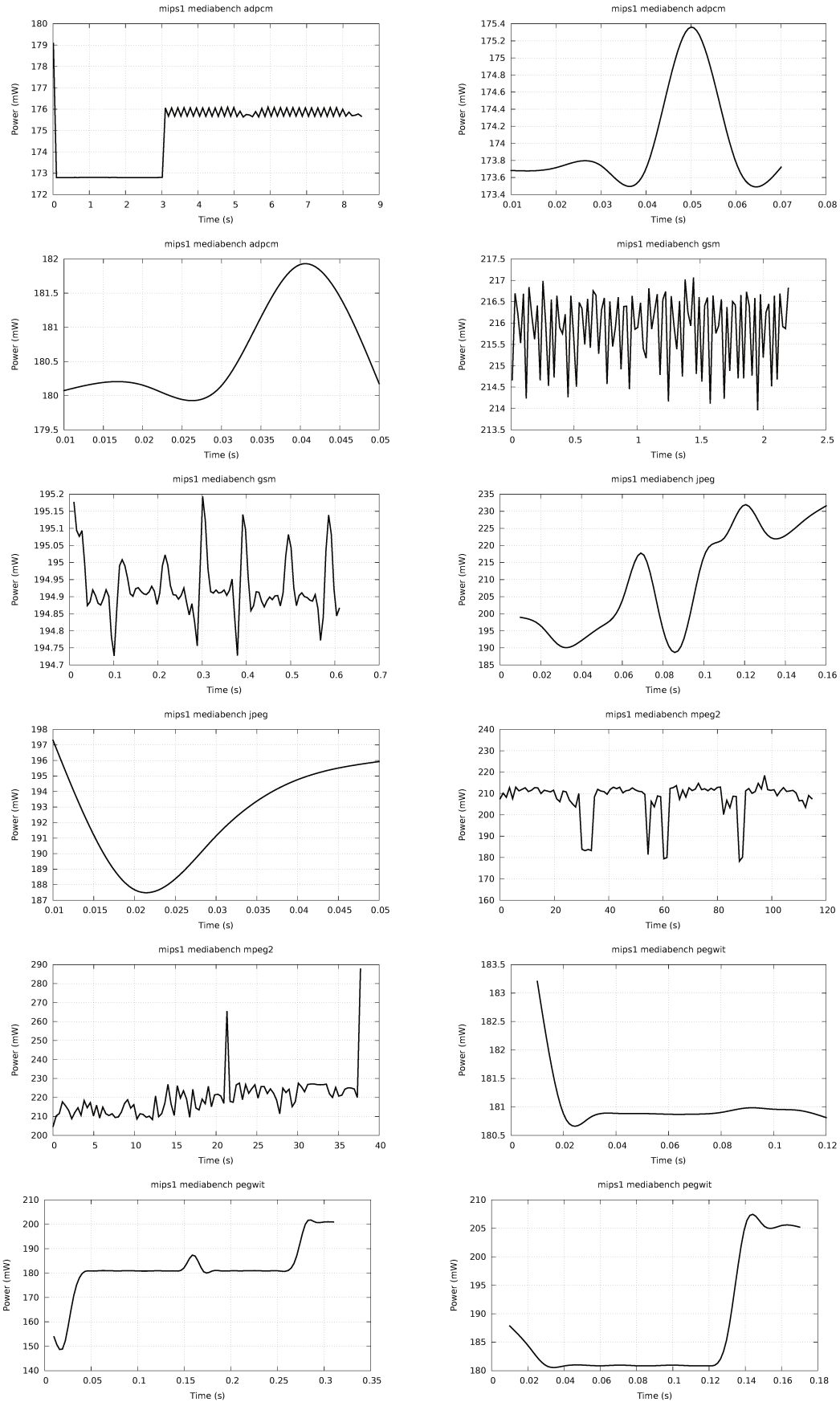
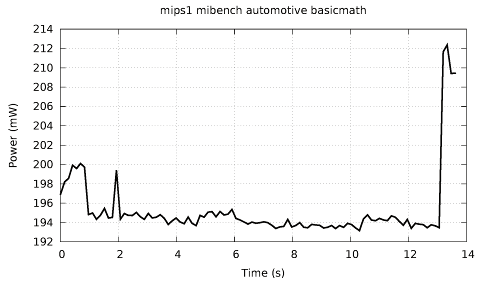
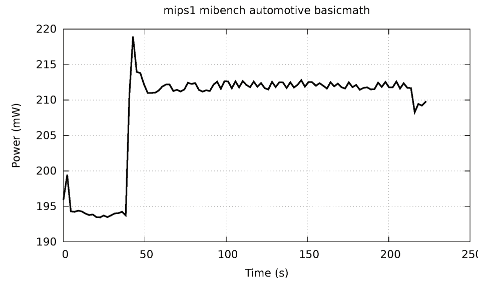
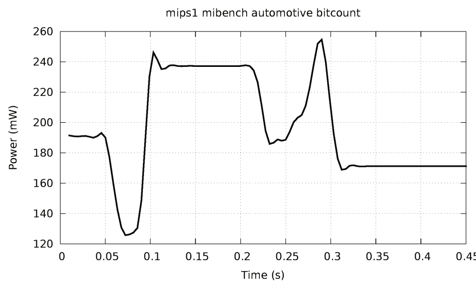
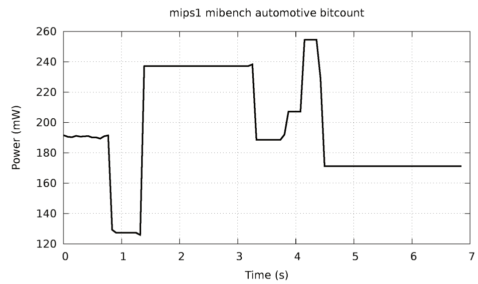


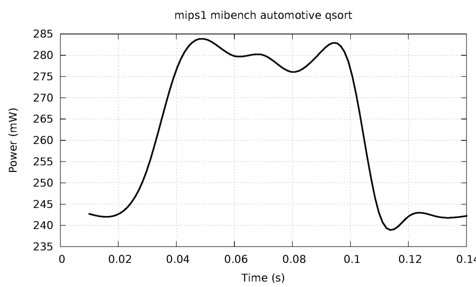
Figura 4.2: Perfis de energia do *benchmark* Mediabench

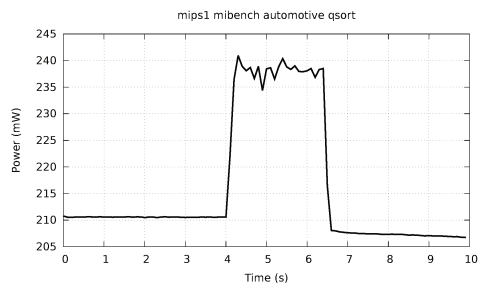

mips1 mibench automotive basicmath
Power (mW)
Time (s)

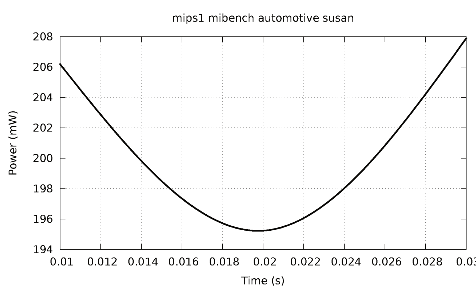

mips1 mibench automotive basicmath
Power (mW)
Time (s)

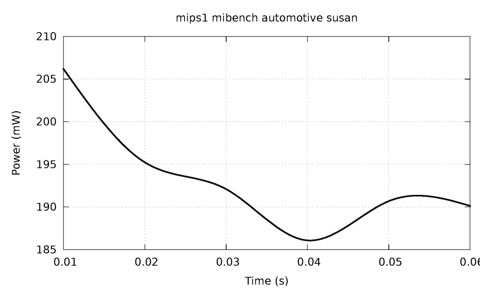

mips1 mibench automotive bitcount
Power (mW)
Time (s)

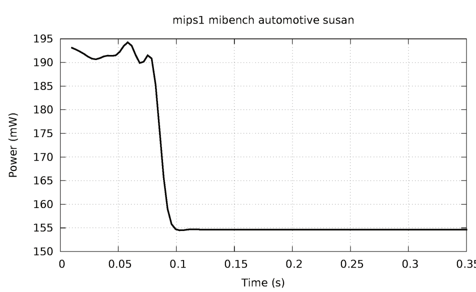

mips1 mibench automotive bitcount
Power (mW)
Time (s)

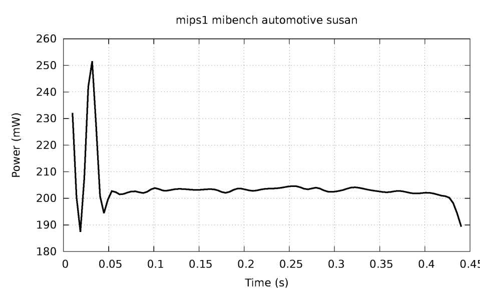

mips1 mibench automotive qsort
Power (mW)
Time (s)

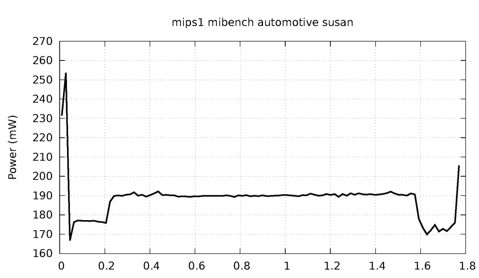

mips1 mibench automotive qsort
Power (mW)
Time (s)

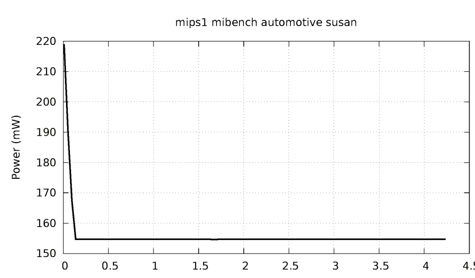

mips1 mibench automotive susan
Power (mW)
Time (s)

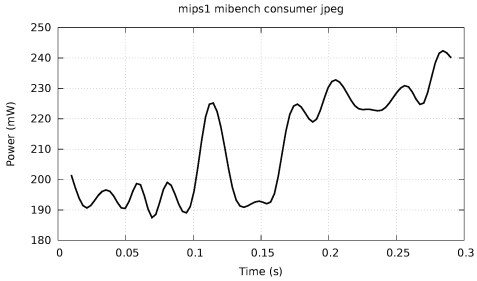

mips1 mibench automotive susan
Power (mW)
Time (s)

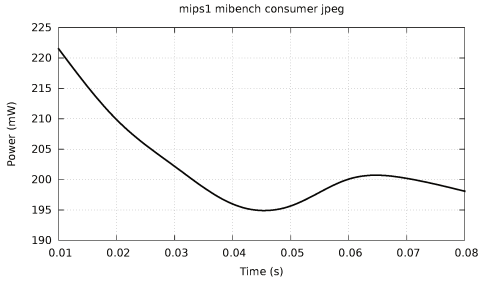

mips1 mibench automotive susan
Power (mW)
Time (s)

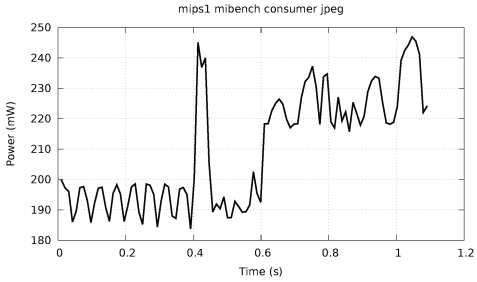

mips1 mibench automotive susan
Power (mW)
Time (s)

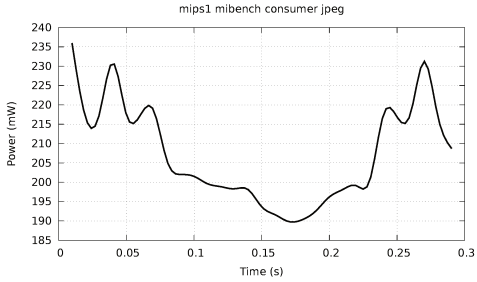

mips1 mibench automotive susan
Power (mW)
Time (s)

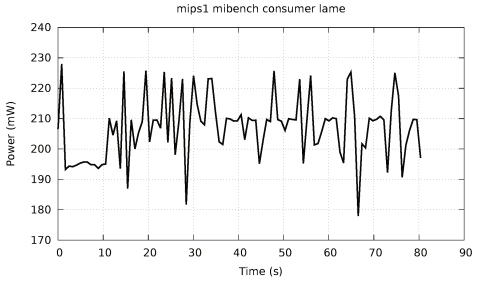

mips1 mibench automotive susan
Power (mW)
Time (s)

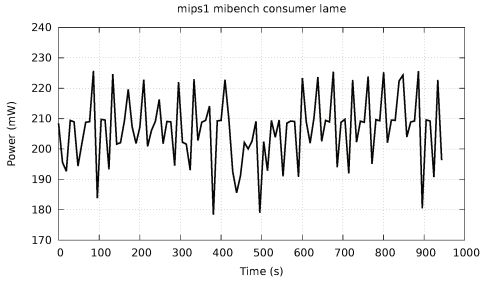

mips1 mibench consumer jpeg
Power (mW)
Time (s)

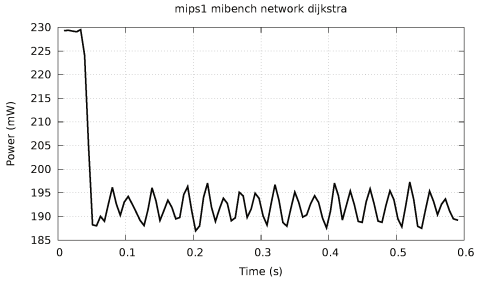

mips1 mibench consumer jpeg
Power (mW)
Time (s)

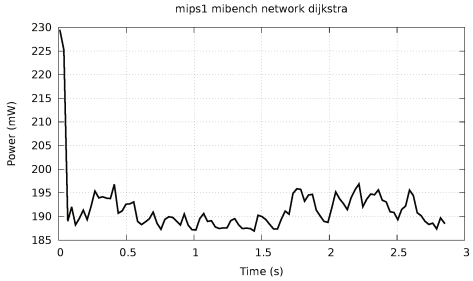

mips1 mibench consumer jpeg
Power (mW)
Time (s)

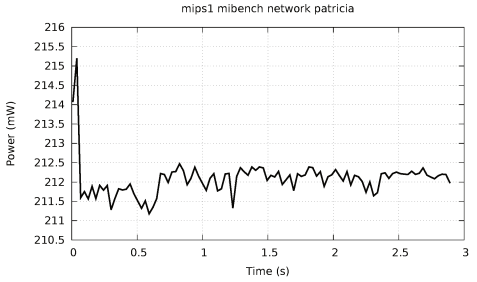

mips1 mibench consumer jpeg
Power (mW)
Time (s)

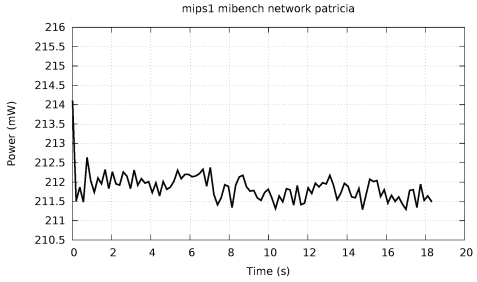

mips1 mibench consumer lame
Power (mW)
Time (s)

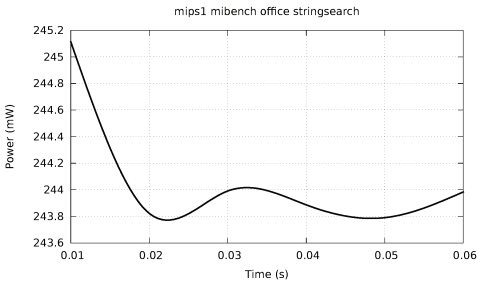

mips1 mibench consumer lame
Power (mW)
Time (s)

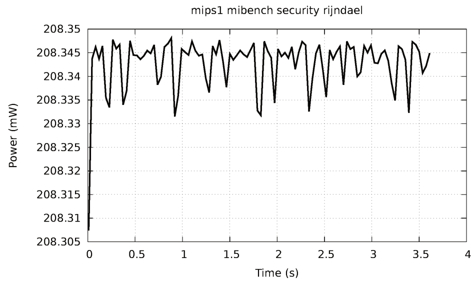

mips1 mibench network dijkstra
Power (mW)
Time (s)

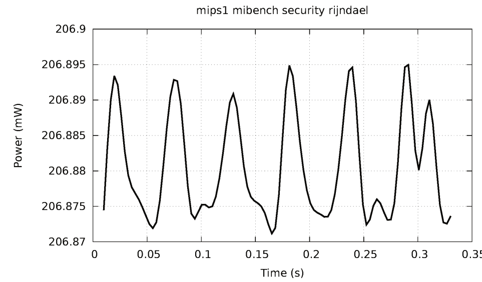

mips1 mibench network dijkstra
Power (mW)
Time (s)

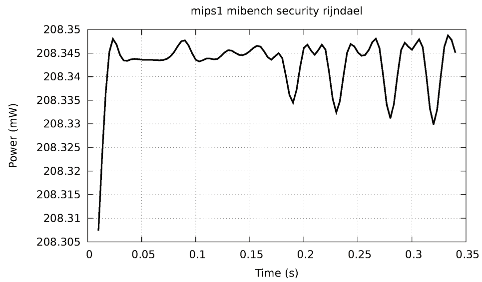

mips1 mibench network patricia
Power (mW)
Time (s)

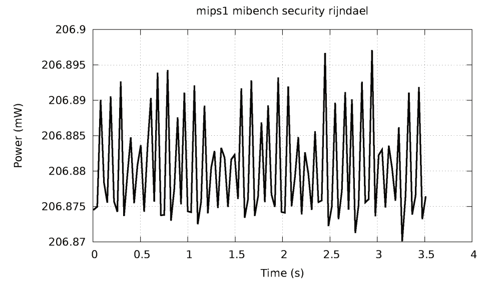

mips1 mibench network patricia
Power (mW)
Time (s)

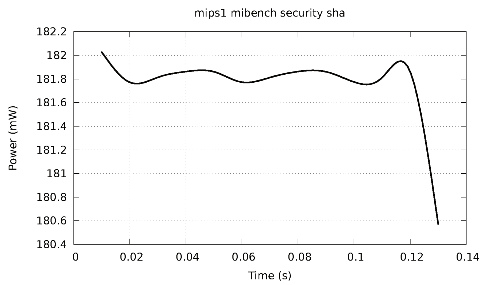

mips1 mibench office stringsearch
Power (mW)
Time (s)

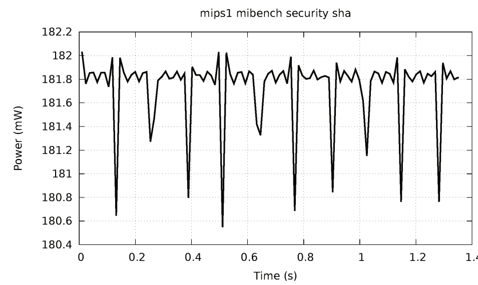

mips1 mibench security rijndael
Power (mW)
Time (s)

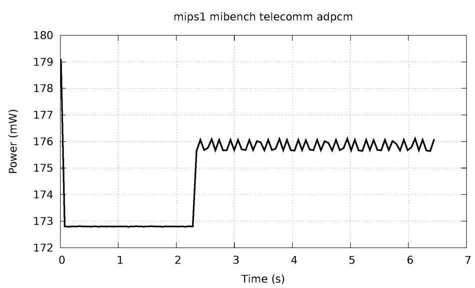

mips1 mibench security rijndael
Power (mW)
Time (s)

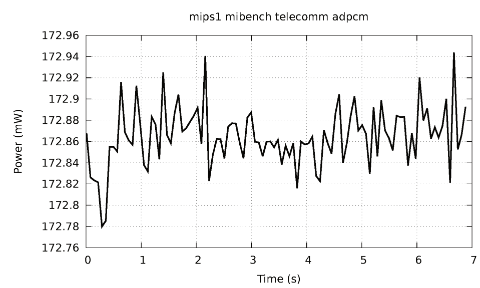

mips1 mibench security rijndael
Power (mW)
Time (s)

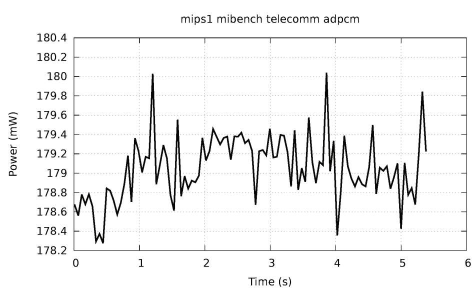

mips1 mibench security rijndael
Power (mW)
Time (s)

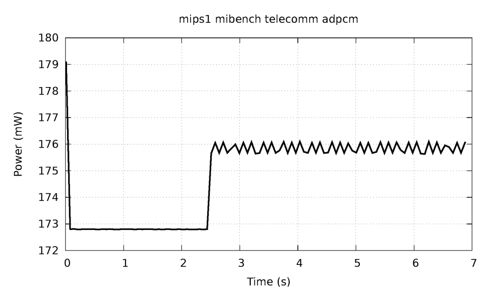

mips1 mibench security sha
Power (mW)
Time (s)

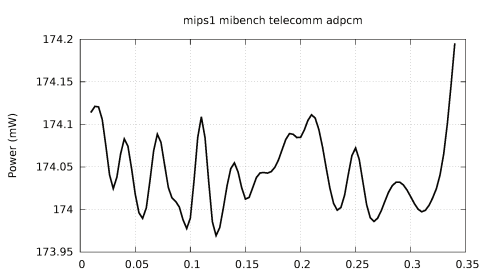

mips1 mibench security sha
Power (mW)
Time (s)

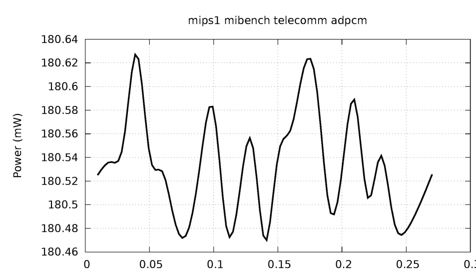

mips1 mibench telecomm adpcm
Power (mW)
Time (s)


mips1 mibench telecomm adpcm
Power (mW)
Time (s)


mips1 mibench telecomm adpcm
Power (mW)
Time (s)


mips1 mibench telecomm adpcm
Power (mW)
Time (s)


mips1 mibench telecomm adpcm
Power (mW)
Time (s)


mips1 mibench telecomm adpcm
Power (mW)
Time (s)

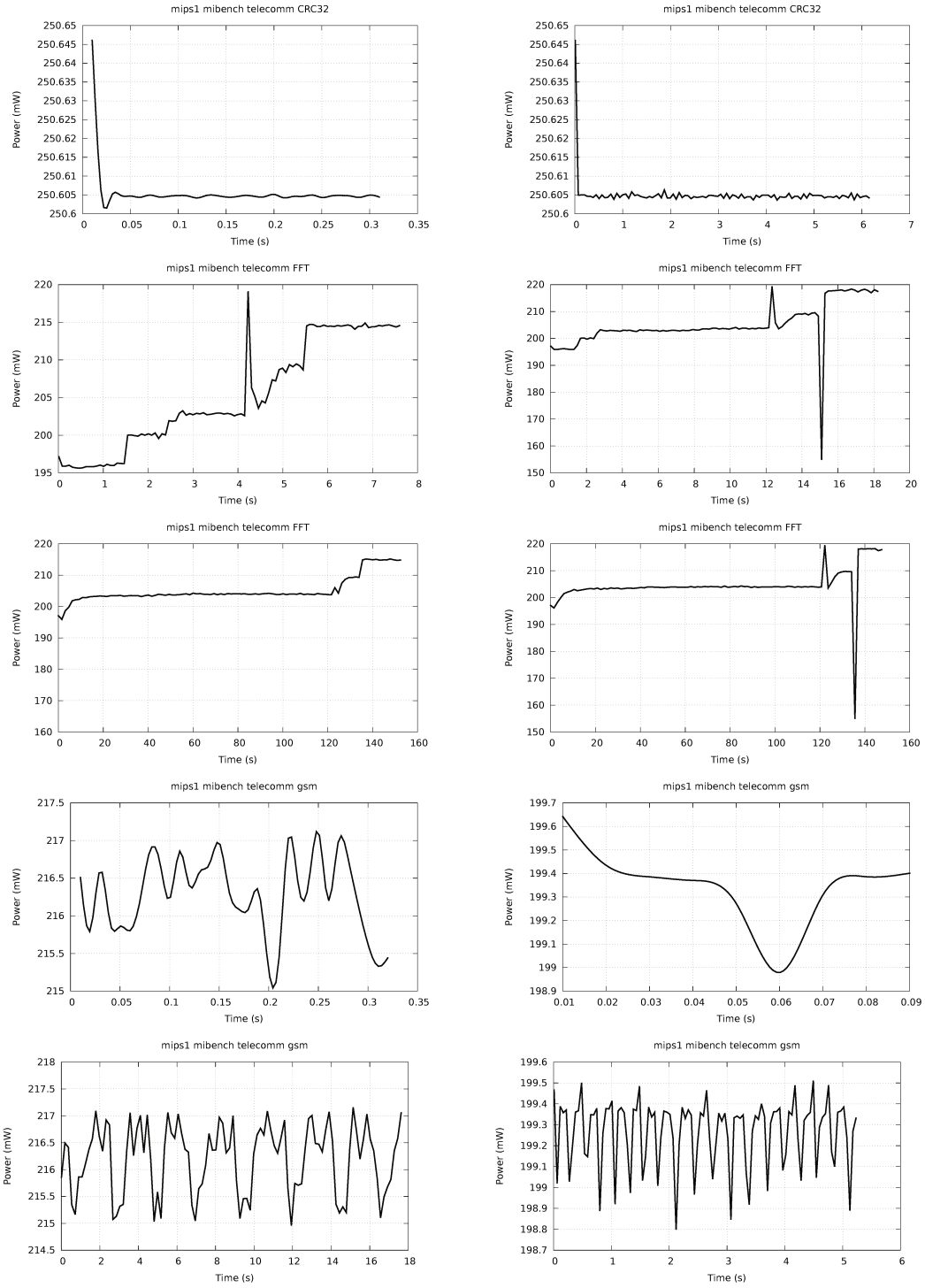


Figura 4.3: Perfis de energia do *benchmark* Mibench

4.1.2 Plataforma Altera

O mesmo desenvolvimento realizado sobre Plasma na plataforma Xilinx foi executado na plataforma Altera. Foi escolhido desenvolver o sistema sobre a FPGA Altera CycloneV 5CGXFC7C7F23C8 a 25MHz. O fluxo de desenvolvimento foi o mesmo que o apresentado para Xilinx, apenas configurando acSynth.conf adequadamente para utilizar as ferramentas Altera, Quartus e PowerPlay, ao invés de utilizar as ferramentas Xilinx. Da mesma forma, o código Plasma precisou ser parametrizado para utilizar memórias compatíveis com a arquitetura da família de FPGAs Altera.

O fluxo de simulação PowerPlay não faz uso de um *back-annotation* da *netlist* e, portanto, esta etapa não precisou ser realizada. A simulação foi aplicada diretamente do modelo HDL e o algoritmo de PowerPlay realizou o mapeamento adequado para o seus arquivos internos de projeto. Apesar de mais simples de utilizar, PowerPlay fornece um controle menor sobre o circuito em análise.

Como as simulações foram executadas em nível HDL, o tempo de geração dos arquivos de atividade de transição são praticamente desprezíveis. Todos os arquivos foram gerados em aproximadamente 20 minutos. O tempo de geração dos relatórios de consumo em RTL no PowerPlay levaram mais tempo, em torno de 40 minutos. Ao todo, foi consumida aproximadamente 1 hora para realizar a caracterização do sistema. Exatamente como no fluxo anterior, as lacunas de instruções `load` e `store` foram preenchidas manualmente. O mesmo ocorreu com os *branches*. O valor de consumo em *stall* foi auferido em 1.64 nJ.

Na tabela 4.10 é possível consultar os valores de consumo de energia de todas as instruções.

Com respeito aos *benchmarks*, um primeiro fato que se destaca é a qualidade dos resultados da simulação dos programas acStone em relação aos dados apresentados com plataforma Xilinx. Podemos atribuir esta melhora a qualidade de PowerPlay para gerar simulações mais consistentes.

A aplicação da correção da fase inicial em *stall* foi aplicada aqui também afetando significativamente os resultados do *benchmark*. A redução não foi tão consistente quanto com Xilinx, mas 64 dos 75 programas acStone executados tiveram uma redução de erro após o ajuste. E, mais importante, todos os erros ficaram abaixo de 20% de desvio. A tabela 4.11 apresenta os resultados completos. O gráfico com a distribuição do erro pode ser visto em 4.4.

É interessante notar também que os testes com operações `mul` e `div` não apresentaram os mesmos problemas de *stall* presentes na simulação Xilinx. Atribui-se este resultado ao fato da síntese ter sido realizada com um *clock* quatro vezes mais lento, o que afeta a temporização e deve interferir na geração de *stalls* do sistema. No caso da execução em Xilinx Spartan, apesar do *pipeline* principal atingir 100 MHz, o coprocessador mul/div não consegue manter este passo sempre e eventualmente gera *stall*.

Tabela 4.10: Consumo de energia dinâmica para instruções Plasma em FPGA Altera Cyclone V

Instruction	Energy (nJ)	Instruction	Energy (nJ)
add	2.21	lwl	2.98
addi	2.70	lwr	2.98
addiu	2.71	mfhi	2.44
addu	2.18	mflo	2.43
andi	2.84	mthi	2.07
beq	3.90	mtlo	2.05
bgez	3.90	mult	2.30
bgezal	3.90	multu	2.32
bgtz	3.90	nop	1.94
blez	3.90	ori	2.90
bltz	3.90	sb	3.41
bltzal	3.90	sh	3.47
bne	3.90	sll	2.39
div	3.24	sllv	2.15
divu	3.39	slt	2.20
instr_and	2.17	slti	2.94
instr_break	0	sltiu	2.99
instr_nor	2.20	sltu	2.21
instr_or	2.25	sra	2.36
instr_xor	2.24	srav	2.20
j	3.93	srl	2.38
jal	3.19	srlv	2.14
jalr	3.05	sub	2.18
jr	2.76	subu	2.17
lb	3.27	sw	3.56
lbu	2.57	swl	3.56
lh	3.31	swr	3.56
lhu	2.60	sys_call	0
lui	3.16	xori	2.87
lw	2.98		

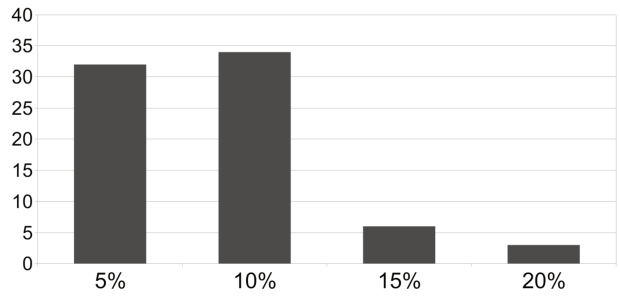


Figura 4.4: Distribuição do erro para os programas do *benchmark* acStone no processador Plasma

Os resultados dos *benchmarks* Mibench e Mediabench são apresentados nas tabelas 4.12 a 4.18. Adicionalmente, são apresentados os gráficos 4.5 e 4.6. Assim como antes, todos os programas estão representados nos gráficos, exceto *Mibench Search Small* pois este não executa por tempo suficiente para gerar dados válidos.

Tabela 4.11: Resultados da simulação com acStone e estudo do erro efetivo

$Program$	$instr$	$instr_{eff}$	$acSim(mW)$	$W_{eff}(mW)$	$PowerPlay(mW)$	err	err_{eff}
000.main	19	0	66.66	41.00	43.32	53.87%	5.36%
011.const	34	15	71.26	54.35	65.87	8.18%	17.49%
012.const	34	15	71.26	54.35	65.87	8.18%	17.49%
013.const	34	15	71.61	54.50	63.86	12.14%	14.65%
014.const	34	15	71.61	54.50	63.86	12.14%	14.65%
015.const	37	18	73.39	56.75	62.04	18.29%	8.52%
016.const	37	18	73.39	56.75	62.04	18.29%	8.52%
017.const	56	37	74.61	63.20	57.98	28.68%	9.01%
018.const	56	37	74.61	63.20	57.98	28.68%	9.01%
021.cast	44	25	72.28	58.77	64.51	12.04%	8.90%
022.cast	44	25	72.74	59.03	65.01	11.89%	9.19%
023.cast	75	56	69.17	62.03	66.65	3.78%	6.93%
024.cast	44	25	72.97	59.16	65.01	12.24%	8.99%
025.cast	75	56	69.29	62.12	66.65	3.96%	6.80%
026.cast	68	49	71.52	62.99	66.65	7.31%	5.49%
027.cast	49	30	71.86	59.90	66.65	7.82%	10.13%
031.add	259	240	68.94	66.89	65.69	4.95%	1.83%
032.add	259	240	69.57	67.47	64.01	8.68%	5.41%
033.add	257	238	73.23	70.85	66.61	9.94%	6.36%
034.add	587	568	71.23	70.25	66.34	7.37%	5.89%
041.sub	259	240	68.92	66.87	65.69	4.92%	1.80%
042.sub	259	240	69.54	67.45	64.01	8.65%	5.37%
043.sub	257	238	73.22	70.83	66.61	9.92%	6.34%
044.sub	559	540	72.05	70.99	66.34	8.60%	7.01%
051.mul	96	77	72.24	66.05	65.88	9.65%	0.27%
052.mul	96	77	69.68	64.01	65.88	5.78%	2.84%
053.mul	129	110	73.33	68.56	63.97	14.62%	7.18%
054.mul	129	110	70.57	66.22	63.97	10.32%	3.51%
055.mul	231	212	72.06	69.50	64.29	12.08%	8.11%
056.mul	231	212	72.09	69.53	64.29	12.13%	8.15%
057.mul	261	242	67.85	65.90	64.15	5.77%	2.72%
058.mul	265	246	66.68	64.84	64.15	3.94%	1.07%
061.div	192	173	72.93	69.77	67.33	8.31%	3.62%
062.div	182	163	71.49	68.31	67.33	6.18%	1.45%
063.div	192	173	73.44	70.23	65.84	11.55%	6.67%
064.div	182	163	72.00	68.76	65.5	9.92%	4.98%
065.div	195	176	73.22	70.08	66.02	10.91%	6.15%
066.div	183	164	74.26	70.81	66.02	12.48%	7.25%
067.div	1521	1502	68.48	68.13	64.54	6.10%	5.57%
068.div	1170	1151	69.40	68.93	65.39	6.12%	5.42%
071.bool	281	262	67.23	65.45	67.77	0.80%	3.42%
072.bool	281	262	67.89	66.08	65.71	3.32%	0.56%
073.bool	285	266	71.16	69.15	66.01	7.80%	4.76%
074.bool	557	538	72.00	70.94	62.91	14.44%	12.76%
075.bool	125	106	72.11	67.38	57.56	25.27%	17.06%
081.shift	183	164	71.60	68.43	67.29	6.41%	1.69%
082.shift	183	164	72.31	69.06	65.47	10.45%	5.48%

083.shift	191	172	73.95	70.67	65.69	12.57%	7.58%
084.shift	423	404	71.87	70.49	62.98	14.12%	11.92%
085.shift	154	135	65.56	62.53	59.08	10.97%	5.84%
111.if	318	299	70.91	69.12	66.65	6.39%	3.71%
112.if	318	299	68.17	66.55	66.65	2.29%	0.15%
113.if	318	299	71.11	69.31	66.71	6.60%	3.90%
114.if	318	299	68.34	66.71	66.71	2.44%	0.01%
115.if	318	299	69.89	68.16	66.42	5.22%	2.62%
116.if	318	299	69.90	68.18	66.42	5.24%	2.64%
117.if	514	495	69.52	68.46	65.48	6.16%	4.55%
118.if	514	495	69.53	68.47	65.48	6.18%	4.57%
119.if	275	256	69.59	67.61	66.35	4.88%	1.91%
121.loop	959	940	71.89	71.28	65.67	9.48%	8.55%
122.loop	959	940	71.14	70.54	66.96	6.24%	5.35%
123.loop	1758	1739	68.57	68.27	65.76	4.28%	3.82%
124.loop	2199	2180	69.70	69.45	65.78	5.96%	5.58%
125.loop	1597	1578	69.35	69.02	60.73	14.20%	13.64%
126.loop	2241	2222	67.38	67.16	62.61	7.62%	7.27%
131.call	111	92	71.14	65.98	64.45	10.39%	2.38%
132.call	271	252	71.19	69.07	64.41	10.53%	7.24%
133.call	2887	2868	67.22	67.05	65.6	2.47%	2.21%
134.call	43261	43242	70.67	70.66	65.6	7.73%	7.71%
141.array	1602	1583	65.46	65.17	65.5	0.07%	0.51%
142.array	94771	94752	62.51	62.50	65.74	4.92%	4.93%
143.array	18506	18487	66.49	66.47	64.6	2.93%	2.89%
144.array	22042	22023	64.65	64.63	65.72	1.63%	1.66%
145.array	22640	22621	65.00	64.98	64.32	1.06%	1.03%
146.array	29248	29229	66.56	66.54	65.54	1.55%	1.53%

Tabela 4.12: Mediabench *benchmarks*: estimativas de consumo de energia

Program	**Instr (M)**	**Power (mW)**	**Energy (mJ)**	t_{Sim} **(s)**	$\approx t_{Sim}$ **RTL (day)**
timing	812	70	2284	26	692
rawcaudio	7	69	21	< 1	6
rawdaudio	6	73	17	< 1	5
toast	221	68	596	7	188
untoast	61	70	171	2	52
cjpeg	17	69	46	1	14
djpeg	5	65	14	< 1	4
mpeg2encode	11474	69	31579	394	9772
mpeg2decode	3772	70	10628	125	3212
pegwit gen	13	67	34	< 1	11
pegwit enc	31	66	82	1	26
pegwit dec	17	67	47	1	15

Tabela 4.13: Mibench *benchmarks*: estimativas de consumo para algoritmos *Consumer*

Program	**Test**	**Instr (M)**	**Power (mW)**	**Energy (mJ)**	t_{Sim} **(s)**	$\approx t_{Sim}$ **RTL (day)**
lame	small	8034	69.79	22426	269	6842
lame	large	94315	69.90	263693	3043	80324
cjpeg	small	29	68.78	81	1	25
djpeg	small	9	66.27	23	< 1	7
cjpeg	large	109	68.45	299	3	93
djpeg	large	29	66.42	78	1	25

Tabela 4.14: Mibench *benchmarks*: estimativas de consumo para algoritmos *Automotive*

Program	**Test**	**Instr (M)**	**Power (mW)**	**Energy (mJ)**	t_{Sim} **(s)**	$\approx t_{Sim}$ **RTL (day)**
basicmath	small	1361	68.66	3738	44	1159
basicmath	large	22269	69.21	61646	735	18966
bitcnts	small	46	70.84	129	1	39
bitcnts	large	684	70.84	1939	19	583
qsort	small	14	73.04	42	< 1	12
qsort	large	989	70.24	2780	32	843
susan	small/corner	3	68.38	9	< 1	3
susan	small/edge	7	67.75	19	< 1	6
susan	small/smooth	35	63.94	90	1	30
susan	large/corner	45	67.35	120	1	38
susan	large/edge	177	67.48	479	6	151
susan	large/smooth	423	62.64	1061	16	361

Tabela 4.15: Mibench *benchmarks*: estimativas de consumo para algoritmos *Network*

Program	**Test**	**Instr (M)**	**Power (mW)**	**Energy (mJ)**	t_{Sim} **(s)**	$\approx t_{Sim}$ **RTL (day)**
dijkstra	small	59	67.39	160	2	51
dijkstra	large	285	67.16	766	8	243
patricia	small	289	69.25	801	10	246
patricia	large	1831	69.22	5069	60	1559

Tabela 4.16: Mibench *benchmarks*: estimativas de consumo para algoritmos *Office*

Program	**Test**	**Instr (M)**	**Power (mW)**	**Energy (mJ)**	t_{Sim} **(s)**	$\approx t_{Sim}$ **RTL (day)**
search	small	< 1	73.75	1	< 1	< 1
search	large	7	73.66	21	< 1	6

Tabela 4.17: Mibench *benchmarks*: estimativas de consumo para algoritmos *Security*

Program	**Test**	**Instr (M)**	**Power (mW)**	**Energy (mJ)**	t_{Sim} **(s)**	$\approx t_{Sim}$ **RTL (day)**
rijndael	large/decode	361	66.36	959	13	308
rijndael	small/encode	34	66.06	89	1	29
rijndael	small/decode	35	66.36	92	1	30
rijndael	large/encode	351	66.06	928	13	299
sha	small	13	64.58	34	< 1	11
sha	large	136	64.57	351	4	116

Tabela 4.18: Mibench *benchmarks*: estimativas de consumo para algoritmos *Telecommunication*

Program	**Test**	**Instr (M)**	**Power (mW)**	**Energy (mJ)**	t_{Sim} **(s)**	$\approx t_{Sim}$ **RTL (day)**
adpcm timing	small	756	70.31	2126	26	644
adpcm rawcaudio	large	689	69.29	1910	23	587
adpcm rawdaudio	large	539	72.88	1570	18	459
adpcm timing	large	850	70.31	2392	27	724
adpcm rawcaudio	small	35	69.47	96	1	29
adpcm rawdaudio	small	27	73.10	80	1	23
crc_32	small	32	71.88	91	1	27
crc_32	large	615	71.88	1768	20	524
fft	4k	761	69.37	2110	26	648
fft	inv 8k	1823	69.37	5058	62	1553
fft	32k	15244	69.33	42278	511	12983
fft	inv 32k	14750	69.34	40910	489	12562
gsm toast	small	33	67.63	88	1	28
gsm untoast	small	10	70.08	27	< 1	8
gsm toast	large	1763	67.59	4768	58	1502
gsm untoast	large	523	70.08	1467	16	446

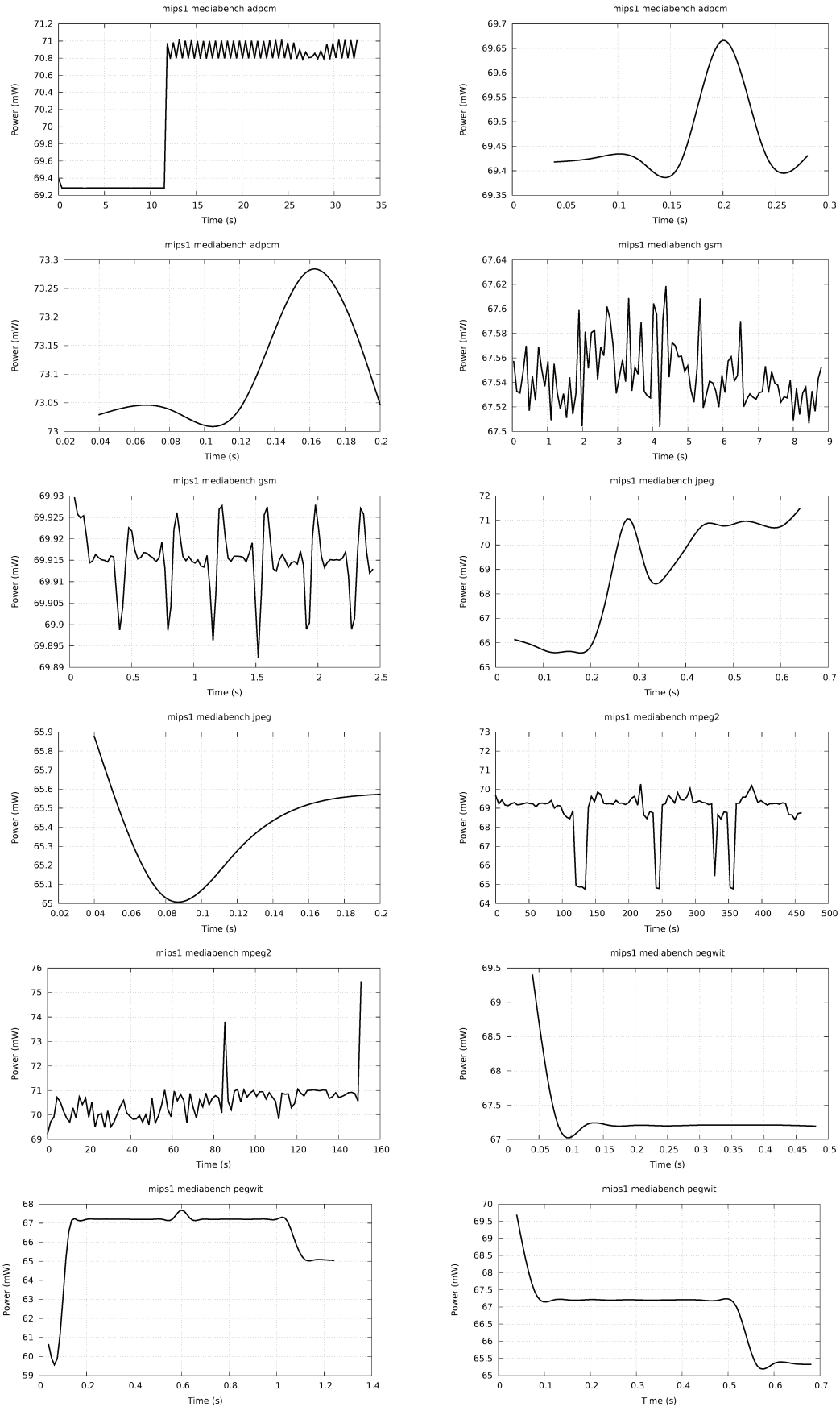
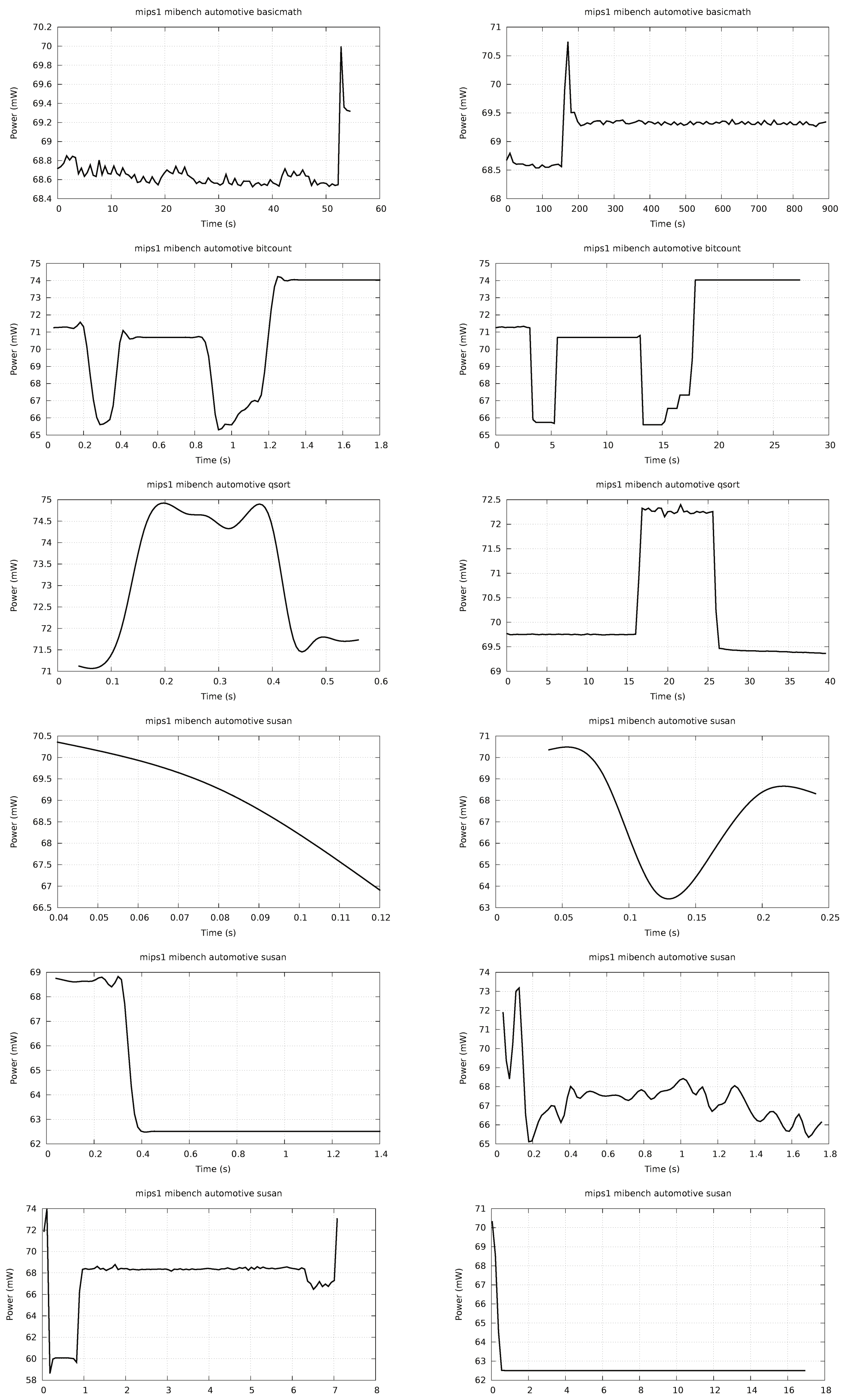
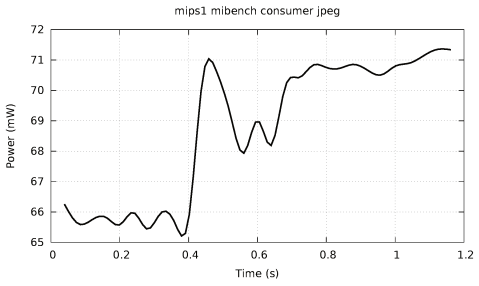
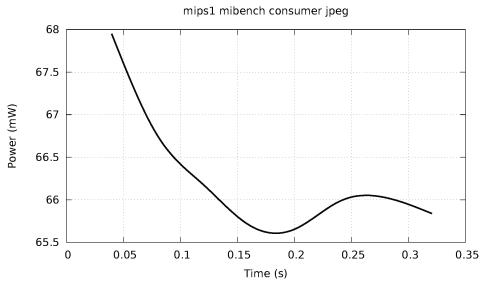
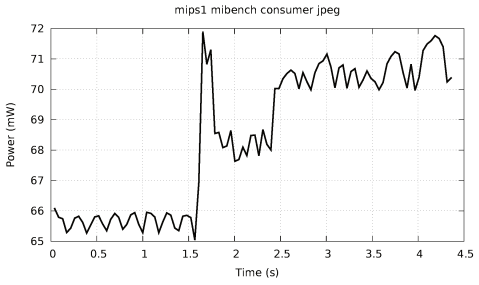


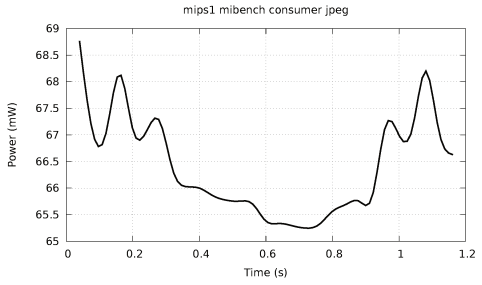
Figura 4.5: Perfis de energia do *benchmark* Mediabench

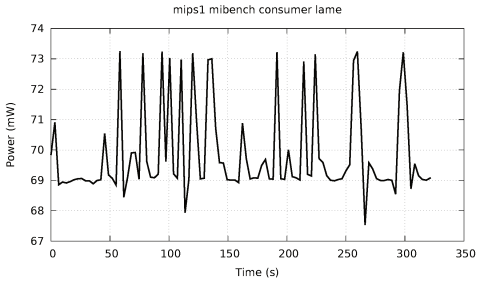

mips1 mibench automotive basicmath
Power (mW)
Time (s)
mips1 mibench automotive basicmath
Power (mW)
Time (s)
mips1 mibench automotive bitcount
Power (mW)
Time (s)
mips1 mibench automotive bitcount
Power (mW)
Time (s)
mips1 mibench automotive qsort
Power (mW)
Time (s)
mips1 mibench automotive qsort
Power (mW)
Time (s)
mips1 mibench automotive susan
Power (mW)
Time (s)
mips1 mibench automotive susan
Power (mW)
Time (s)
mips1 mibench automotive susan
Power (mW)
Time (s)
mips1 mibench automotive susan
Power (mW)
Time (s)
mips1 mibench automotive susan
Power (mW)
Time (s)
mips1 mibench automotive susan
Power (mW)
Time (s)

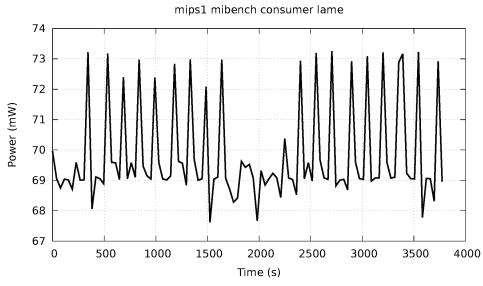

mips1 mibench consumer jpeg
Power (mW)
Time (s)

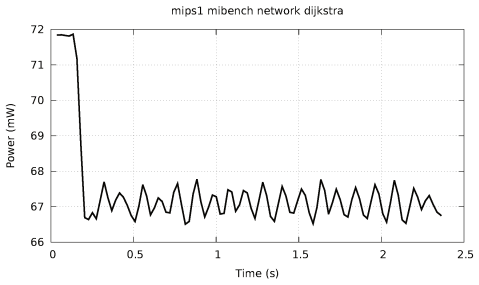

mips1 mibench consumer jpeg
Power (mW)
Time (s)

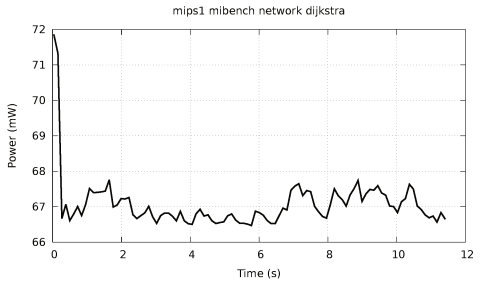

mips1 mibench consumer jpeg
Power (mW)
Time (s)

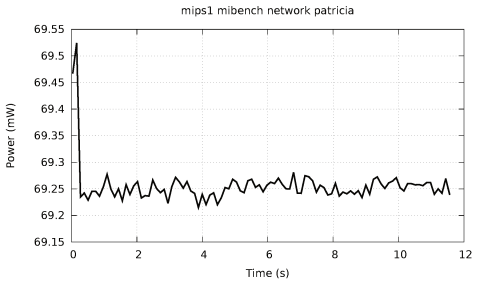

mips1 mibench consumer jpeg
Power (mW)
Time (s)

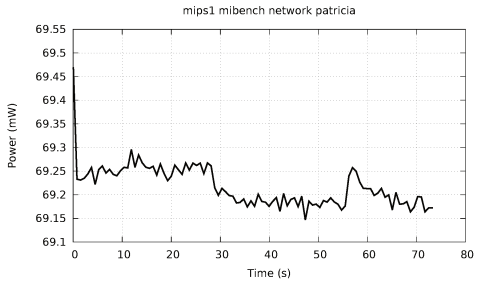

mips1 mibench consumer lame
Power (mW)
Time (s)

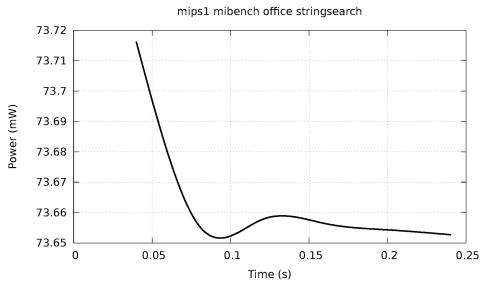

mips1 mibench consumer lame
Power (mW)
Time (s)

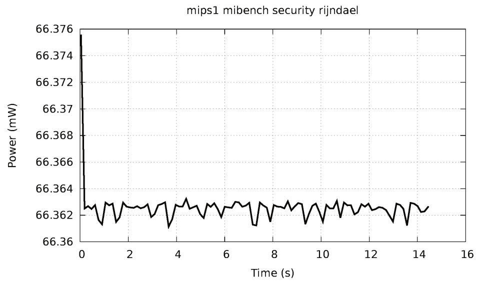

mips1 mibench network dijkstra
Power (mW)
Time (s)

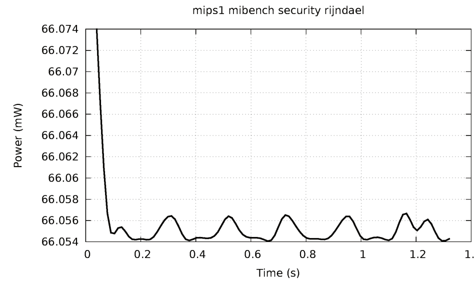

mips1 mibench network dijkstra
Power (mW)
Time (s)

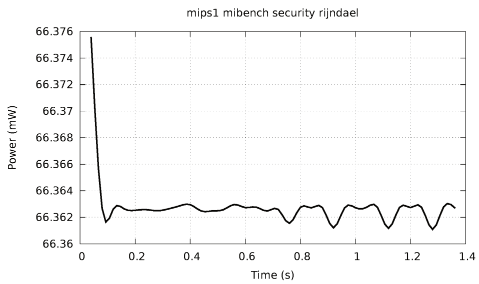

mips1 mibench network patricia
Power (mW)
Time (s)

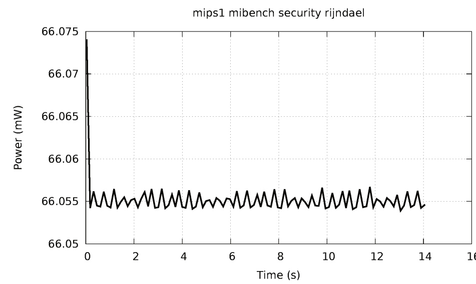

mips1 mibench network patricia
Power (mW)
Time (s)

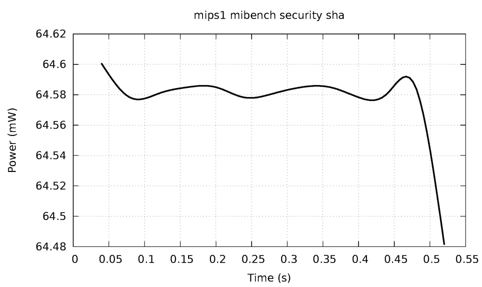

mips1 mibench office stringsearch
Power (mW)
Time (s)

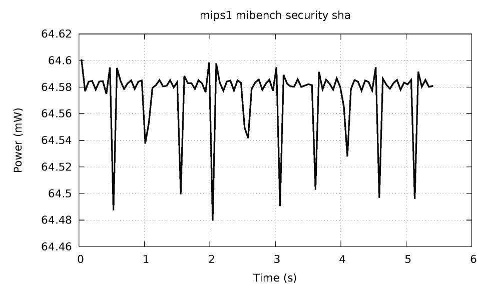

mips1 mibench security rijndael
Power (mW)
Time (s)

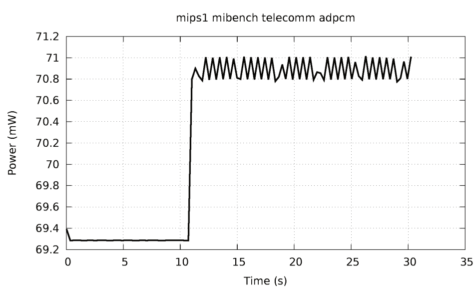

mips1 mibench security rijndael
Power (mW)
Time (s)

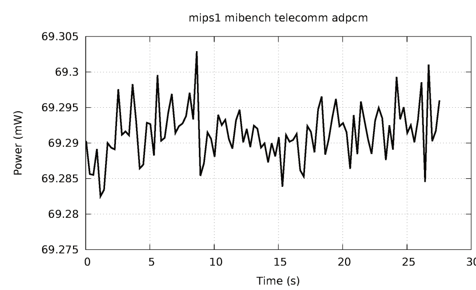

mips1 mibench security rijndael
Power (mW)
Time (s)

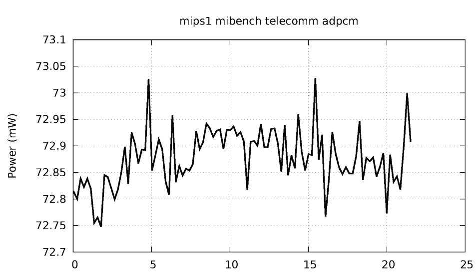

mips1 mibench security rijndael
Power (mW)
Time (s)

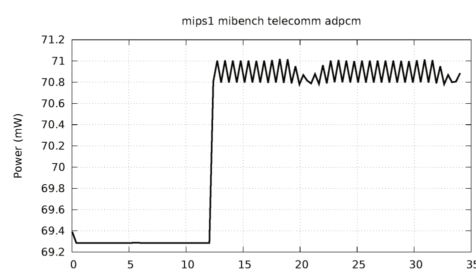

mips1 mibench security sha
Power (mW)
Time (s)

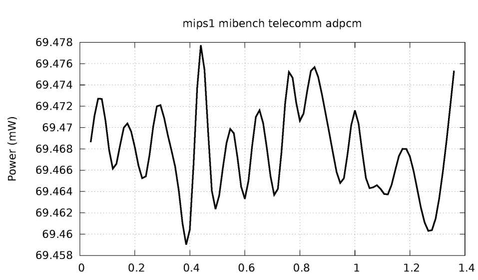

mips1 mibench security sha
Power (mW)
Time (s)

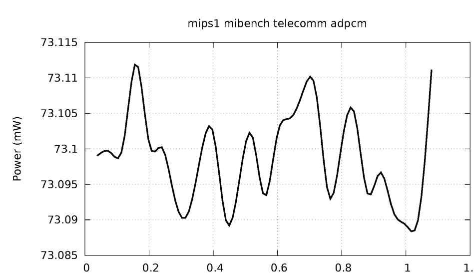

mips1 mibench telecomm adpcm
Power (mW)
Time (s)


mips1 mibench telecomm adpcm
Power (mW)
Time (s)


mips1 mibench telecomm adpcm
Power (mW)
Time (s)


mips1 mibench telecomm adpcm
Power (mW)
Time (s)


mips1 mibench telecomm adpcm
Power (mW)
Time (s)


mips1 mibench telecomm adpcm
Power (mW)
Time (s)

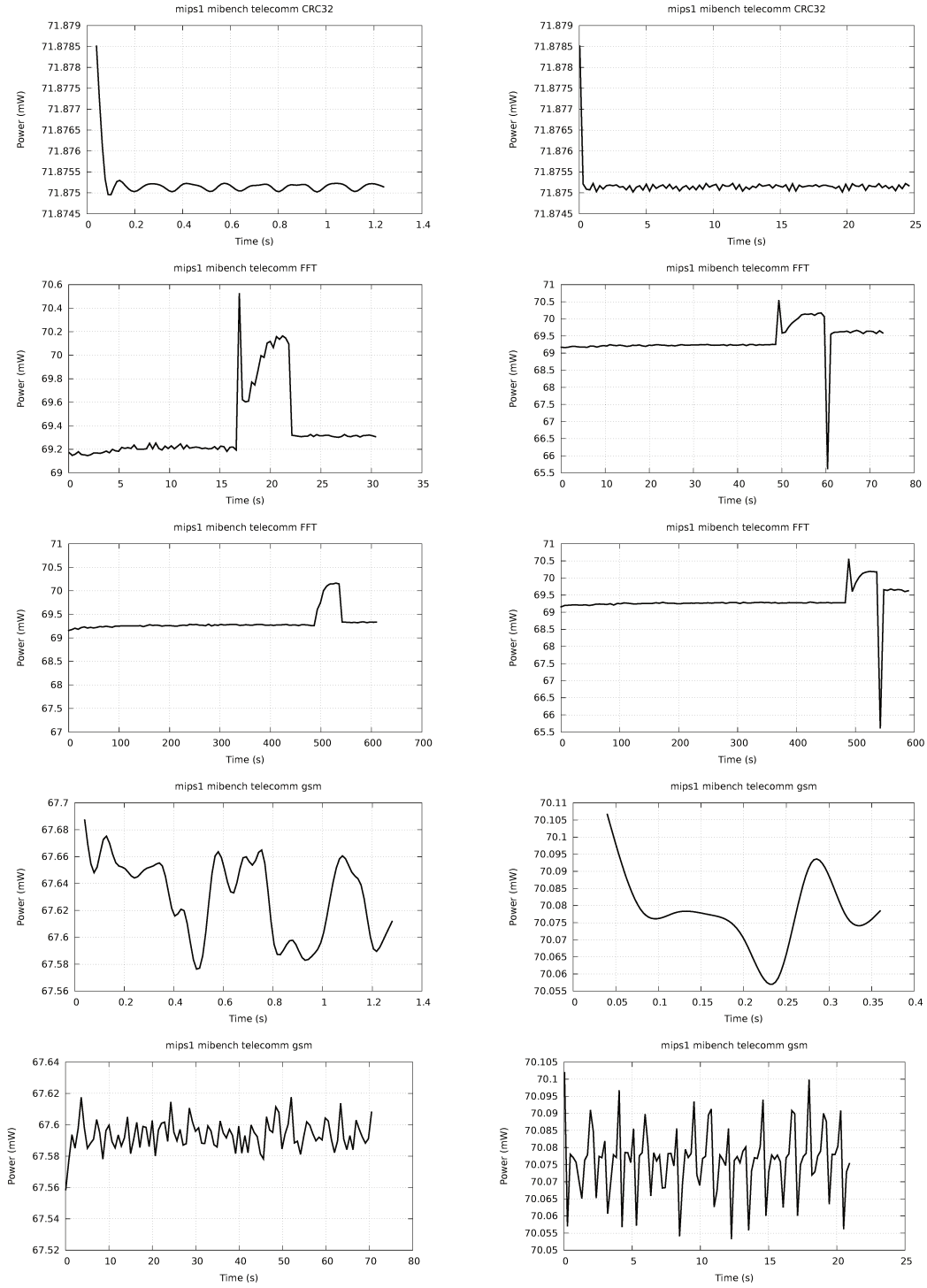


Figura 4.6: Perfis de energia do *benchmark* Mibench

4.2 Processador Leon3

Leon3 é uma implementação HDL da arquitetura SPARCv8 desenvolvido pela *Gaisler Research* [46]. O processador é distribuído sob duas licenças distintas, uma comercial e uma LGPL. Esta segunda licença permite acesso ao código fonte e livre utilização para fins acadêmicos. O processador Leon3 é um processador maduro utilizado comercialmente, com suporte a Linux e a diversas plataformas RTOS além de uma ampla gama de módulos acessórios suportados. Sua versão *fault-tolerant* Leon3FT é especialmente importante e aplica-se a programas aeroespaciais.

A licença LGPL, a compatibilidade SPARCv8 com um modelo ADL ArchC estável e a possibilidade de avaliar o fluxo sobre acSynth com um IP comercial foram os motivos para a escolha deste processador.

O processador foi caracterizado sob a plataforma Xilinx para uma FPGA Spartan3 xc3s1000 a 50 MHz. Com respeito a sequência de procedimentos para a caracterização, não houve grandes diferenças em relação ao que já foi apresentado nos fluxos com Plasma. Gerou-se o *netlist* e, então, foi desenvolvido um *testbench* funcional para a tríade de arquivos `init.txt`, `rom.txt` e saída `tb.vcd`. Desta forma foi possível realizar a etapa de caracterização automática de acSynth. Vale citar que existe uma plataforma de configuração criada pela Gaisler, facilitando a configuração de memória e adição ou remoção de recursos.

Podemos apontar a dificuldade superior como o grande diferencial nos experimentos com Leon3. Para começar, Leon3 é estruturado em uma plataforma AMBA 2.0 definida pela Gaisler, conectado a diversos periféricos. O funcionamento do processador é dependente estruturalmente dos módulos e da montagem desse sistema, o que torna difícil isolar o processador. Simplesmente separar a CPU (nomeada leon3s no sistema) não se mostrou uma solução adequada porque Leon3 necessita de um fluxo de *boot* e depende de certos parâmetros da MMU e da cache para funcionar apropriadamente. Sem estes dois módulos adequadamente configurados, o processador não executa em modo *burst* e deixa de utilizar o potencial pleno do *pipeline*. Outro problema apresentado foi o fato de Leon3 ser inteiramente descrito com uso de tipos estruturados, ao invés dos tipos padrões de sinais VHDL, std_logic e std_logic_vector. A ferramenta Modelsim não lida bem com estes casos pois não consegue mapear adequadamente sinais para tipos lógicos padrões e, quando isso acontece, os sinais são desconsiderados e nada é gerado como atividade de transição. Foi preciso criar uma casca separando cada sinal individualmente, permitindo Modelsim identificar apropriadamente e gerar as atividades de transição corretamente.

Para este teste, a inserção dos códigos de caracterização foi feita com uso de uma memória AHB modificada, não interferindo na estrutura do sistema AMBA pré-montada, apenas substituindo o módulo AHBRAM existente na plataforma da Gaisler por outra

memória de mesma assinatura que foi chamada de AHBCUSTOMRAM. Normalmente a plataforma de *testbench* fornecida pela Gaisler lida bem com a execução de testes ao fornecer uma memória acessível para a plataforma através de uma controladora de memória externa (módulo MCTRL). Porém, devido ao método Tiwari e a estrutura intrusiva da execução em modo forçado necessário para os testes deste trabalho, o *testbench* padrão fornecido pela Gaisler não foi utilizado. Isto serviu para demonstrar que o aumento de complexidade da abordagem por controle externo (apresentado na figura 3.5) é uma desvantagem maior do que previamente imaginado.

Como tarefa adicional, a memória ROM de *boot* (AHBROM) foi alterada para saltar para o início desta memória RAM modificada assim que terminado seu fluxo, ao contrário de buscar informações da controladora de memória externa. Isto é feito alterando-se a última linha do código na ROM (um `jump`). A lógica de endereçamento forçado foi implementada no *top level* da arquitetura, no módulo leon3mp. As atividades de transição registradas correspondem apenas ao processador. A cache foi desligada para garantir similaridade com o modelo ArchC em estudo.

A complexidade de Leon3 impactou também na geração dos arquivos de atividade de transição. Eles cresceram consideravelmente devido ao número maior de sinais e registradores. Consequentemente, o tempo de execução da caracterização subiu. O maior número de instruções no conjunto SPARCv8 também influenciou no aumento do tempo de simulação.

Outra dificuldade foi devido a incompatibilidade entre o formato dos nomes dos sinais na geração da atividade de transição pelo Modelsim com o formato de nome esperado por Xpower. O problema ocorre porque Modelsim gera um formato para tipos estruturados que é diferente do formato de nome que Xpower considera válido, afetando o pareamento de fios com as atividades de transição na simulação de consumo. Foi necessário um procedimento para corrigir os nomes dentro de todos os arquivos VCD, o que elevou mais ainda o tempo de testes.

Cada instrução levou entre 5 a 10 minutos para executar a geração de atividade de transição e mais 10 a 15 minutos para geração dos relatórios por Xpower e correção dos nomes. Foram executados 97 testes, totalizando mais de 30 horas de caracterização em um computador padrão.

Outro problema encontrado nos trabalhos com Leon3 refere-se ao endereçamento de memória. SPARCv8 Leon3 é um processador complexo e comercial, portanto, apresenta restrições mais rigorosas de uso e execução. Isto se estende ao acesso a memória do sistema. No MIPS pudemos realizar acessos aleatórios de memória e forçar dados corretos garantindo que o *pipeline* sempre executaria comandos válidos. Aqui esta abordagem é inviável porque a chamada de endereços inválidos para o barramento faz com que o sistema AMBA pare de funcionar. Foi preciso, portanto, gerar códigos personalizados

manualmente para caracterizar funções de *load* e *store*, garantindo dados corretos sempre. Isso foi bastante dispendioso.

Problema semelhante ocorreu com instruções de *jump*, uma vez que um PC inválido gerou o mesmo tipo de bloqueio de acesso inválido no barramento AMBA, impedindo o sistema de prosseguir com o teste.

Todas estas dificuldades geraram muitas tarefas na criação do *testbench*, exigindo a criação de novos módulos de memória, além de elaboração adicional de *scripts* para corrigir os problemas de compatibilidade.

Após a caracterização automática, algumas instruções ainda precisaram ser completadas manualmente no banco de dados:

- Todas as instruções de *load* em registradores foram preenchidos com dados dos seus equivalentes baseados em imediatos. Assim, `ldsb_reg` foi considerado com o mesmo consumo de `ldsb_imm`, assim por diante. Isto porque se observou um desvio muito baixo em outros pares imm/reg no restante do conjunto de instruções e isto reduziu a necessidade de mais códigos personalizados.

- O mesmo processo ocorreu para instruções *store* e *swap*, pelas mesmas razões.

- As instruções de divisão foram consideradas com o mesmo consumo das instruções equivalentes de multiplicação.

- Para o caso das instruções de *jump*, `jmpl_imm` e `jmpl_reg`, foi considerado o mesmo consumo de energia das instruções `ba`, que realizam saltos semelhantes, porém, localizados, o que permite testes com dados aleatórios.

- Instruções `trap_imm`, `trap_reg` e `unimplemented` não foram consideradas e, portanto, mantidas em zero.

O resultado final do consumo por instrução é apresentado na tabela 4.19 em pico-Joules (pJ). A mudança na escala deve-se ao menor consumo por instruções em relação ao processador Plasma. O consumo em *stall* foi estimado em 84 pJ.

Assim como com o processador Plasma, foi realizado um trabalho comparativo entre a simulação em RTL e a simulação em nível ADL utilizando o *benchmark* acStone. Os resultados obtidos podem ser vistos na tabela 4.20. Note que novamente houve uma redução substancial do erro ao aplicar a correção com a informação de *stall* na etapa inicial de simulação. O parâmetro $i_{init} = 16$ foi utilizado. A melhora dos resultados foi consistente neste teste. Os erros foram altos nas instruções de `mul` e `div` e atribui-se o motivo, novamente, a geração de *stall* provocado pelo coprocessador em operações como esta. Os demais casos de altos índices de erro são os relacionados a altos processos de `load`

Tabela 4.19: Consumo de energia dinâmica para instruções Leon3 em FPGA Xilinx Spartan3

Instruction	**E (pJ)**
add_imm	272.00
add_reg	396.00
addcc_imm	286.00
addcc_reg	396.80
addx_imm	282.20
addx_reg	394.60
addxcc_imm	282.80
addxcc_reg	397.40
and_imm	188.20
and_reg	174.20
andcc_imm	191.00
andcc_reg	175.00
andn_imm	205.60
andn_reg	176.20
andncc_imm	209.80
andncc_reg	177.40
ba	102.80
bcc	106.20
bcs	151.60
be	102.40
bg	151.80
bge	106.40
bgu	151.80
bl	151.80
ble	102.40
bleu	102.20
bn	150.00
bne	150.40
bneg	151.60
bpos	106.60
bvc	106.40
bvs	151.80
call	103.80
jmpl_imm	102.80
jmpl_reg	102.80
ld_imm	111.00
ld_reg	111.00
ldd_imm	126.40
ldd_reg	169.80
ldsb_imm	118.20

Instruction	**E (pJ)**
ldsb_reg	118.20
ldsh_imm	117.00
ldsh_reg	117.00
ldstub_imm	149.60
ldstub_reg	149.60
ldub_imm	111.00
ldub_reg	111.00
lduh_imm	111.40
lduh_reg	111.40
mulscc_imm	256.40
mulscc_reg	278.80
nop	149.60
or_imm	217.60
or_reg	182.60
orcc_imm	217.00
orcc_reg	183.00
orn_imm	222.20
orn_reg	193.00
orncc_imm	224.60
orncc_reg	194.40
rdy	160.00
restore_imm	170.00
restore_reg	169.60
save_imm	171.00
save_reg	171.40
sdiv_imm	171.00
sdiv_reg	116.00
sdivcc_imm	144.40
sdivcc_reg	115.20
sethi	244.80
sll_imm	195.20
sll_reg	183.40
smul_imm	171.00
smul_reg	116.00
smulcc_imm	167.60
smulcc_reg	115.20
sra_imm	190.60
sra_reg	176.20
srl_imm	193.60
srl_reg	175.80

Instruction	**E (pJ)**
st_imm	189.40
st_reg	189.40
stb_imm	229.80
stb_reg	235.20
std_imm	167.20
std_reg	167.20
sth_imm	220.40
sth_reg	220.40
sub_imm	298.60
sub_reg	405.20
subcc_imm	302.20
subcc_reg	407.80
subx_imm	299.40
subx_reg	405.20
subxcc_imm	300.00
subxcc_reg	407.40
swap_imm	137.60
swap_reg	137.60
trap_imm	0.00
trap_reg	0.00
udiv_imm	133.20
udiv_reg	115.00
udivcc_imm	144.40
udivcc_reg	113.60
umul_imm	133.20
umul_reg	115.00
umulcc_imm	144.40
umulcc_reg	113.60
unimplemented	0.00
wry_imm	324.00
wry_reg	218.80
xnor_imm	345.40
xnor_reg	375.60
xnorcc_imm	347.20
xnorcc_reg	380.00
xor_imm	257.00
xor_reg	288.00
xorcc_imm	265.40
xorcc_reg	281.60

e `store` confirmando novamente o problema de super-estimativa por parte do método de Tiwari para acesso de memória.

Tabela 4.20: Resultados da simulação com acStone e estudo do erro efetivo

$Program$	$instr$	$instr_{eff}$	$acSim(mW)$	$W_{eff}(mW)$	$XPower(mW)$	err	err_{eff}
000.main	16	0	8.56	8.43	8.48	0.97%	0.59%
011.const	31	15	9.87	9.13	8.72	13.20%	4.67%
012.const	31	15	9.87	9.13	8.72	13.20%	4.67%
013.const	34	18	9.97	9.25	8.99	10.91%	2.84%
014.const	34	18	9.97	9.25	8.99	10.91%	2.84%
015.const	34	18	9.62	9.06	8.63	11.44%	4.97%
016.const	34	18	9.62	9.06	8.63	11.44%	4.97%
017.const	43	27	9.63	9.18	8.64	11.43%	6.28%
018.const	43	27	9.63	9.18	8.64	11.43%	6.28%
021.cast	45	29	9.47	9.10	8.56	10.59%	6.29%
022.cast	45	29	9.17	8.90	8.52	7.59%	4.52%
023.cast	63	47	9.42	9.17	8.51	10.70%	7.74%
024.cast	46	30	9.21	8.94	8.52	8.06%	4.89%
025.cast	64	48	9.45	9.19	8.51	10.99%	8.01%
026.cast	56	40	9.29	9.05	8.51	9.22%	6.32%
027.cast	45	29	9.34	9.02	8.81	6.07%	2.38%
031.add	222	206	10.91	10.73	9.56	14.12%	12.25%
032.add	226	210	10.74	10.58	9.46	13.57%	11.83%
033.add	222	206	9.91	9.81	8.62	15.02%	13.78%
034.add	324	308	10.70	10.59	9.14	17.04%	15.82%
041.sub	222	206	10.96	10.78	9.48	15.65%	13.73%
042.sub	226	210	10.80	10.63	9.66	11.76%	10.03%
043.sub	222	206	9.95	9.84	8.62	15.38%	14.11%
044.sub	324	308	10.76	10.65	9.39	14.62%	13.39%
051.mul	381	365	11.46	11.33	8.80	30.22%	28.77%
052.mul	282	266	10.53	10.41	9.15	15.04%	13.74%
053.mul	555	539	11.52	11.43	8.70	32.36%	31.34%
054.mul	555	539	11.52	11.43	8.70	32.36%	31.34%
055.mul	1982	1966	11.98	11.95	8.52	40.57%	40.24%
056.mul	1905	1889	12.01	11.98	8.52	40.97%	40.61%
057.mul	917	901	11.14	11.09	8.70	28.06%	27.52%
058.mul	913	897	11.14	11.09	8.70	28.02%	27.48%
061.div	769	753	10.46	10.42	9.46	10.60%	10.15%
062.div	645	629	10.16	10.12	9.25	9.82%	9.35%
063.div	816	800	10.51	10.47	9.25	13.61%	13.17%
064.div	762	746	10.31	10.27	9.16	12.59%	12.16%
065.div	900	884	10.58	10.54	8.59	23.15%	22.70%
066.div	983	967	10.54	10.50	8.59	22.69%	22.29%
067.div	3427	3411	13.42	13.40	8.85	51.65%	51.39%
068.div	2613	2597	13.66	13.63	8.91	53.35%	52.99%
071.bool	238	222	10.21	10.09	9.12	11.91%	10.60%
072.bool	240	224	10.03	9.93	9.07	10.62%	9.44%
073.bool	242	226	9.49	9.42	8.60	10.37%	9.55%
074.bool	390	374	10.29	10.21	9.02	14.05%	13.20%
075.bool	93	77	10.18	9.88	9.07	12.21%	8.90%
081.shift	186	170	10.44	10.27	9.42	10.84%	9.00%
082.shift	196	180	10.30	10.14	9.40	9.53%	7.91%
083.shift	160	144	9.77	9.63	8.58	13.85%	12.29%
084.shift	356	340	9.46	9.41	8.94	5.79%	5.28%
085.shift	144	128	9.38	9.27	8.67	8.17%	6.96%

111.if	381	365	9.56	9.51	8.79	8.75%	8.21%
112.if	331	315	9.49	9.44	8.71	8.98%	8.39%
113.if	381	365	9.55	9.50	8.93	6.90%	6.37%
114.if	381	365	9.55	9.50	8.93	6.90%	6.37%
115.if	281	265	9.42	9.36	8.58	9.79%	9.13%
116.if	281	265	9.42	9.36	8.58	9.79%	9.13%
117.if	455	439	9.39	9.36	8.61	9.05%	8.66%
118.if	455	439	9.39	9.36	8.61	9.05%	8.66%
119.if	283	267	9.07	9.03	8.62	5.21%	4.79%
121.loop	703	687	9.60	9.57	8.51	12.82%	12.51%
122.loop	755	739	9.18	9.16	8.51	7.88%	7.70%
123.loop	1606	1590	8.84	8.84	8.64	2.34%	2.30%
124.loop	2053	2037	8.73	8.72	8.64	0.98%	0.96%
125.loop	1489	1473	8.57	8.57	8.56	0.15%	0.14%
126.loop	7126	7110	11.87	11.86	8.43	40.75%	40.66%
131.call	94	78	8.54	8.52	8.50	0.50%	0.28%
132.call	260	244	9.29	9.23	8.54	8.74%	8.12%
133.call	7430	7414	11.49	11.49	8.50	35.20%	35.12%
134.call	27920	27904	9.08	9.08	8.50	6.85%	6.84%
141.array	2546	2530	10.56	10.55	8.87	19.05%	18.90%
142.array	187078	187062	11.60	11.60	8.61	34.74%	34.74%
143.array	15682	15666	9.40	9.40	9.70	3.11%	3.12%
144.array	17938	17922	10.44	10.44	10.09	3.45%	3.43%
145.array	16460	16444	9.36	9.36	8.68	7.83%	7.82%
146.array	21288	21272	9.06	9.06	10.04	9.80%	9.80%

Tabela 4.21: Mediabench *benchmarks*: estimativas de consumo de energia

Program	**Instr (M)**	**Power (mW)**	**Energy (mJ)**	$\boldsymbol{t_{Sim}}$ **(s)**	$\approx t_{Sim}$ **RTL (day)**
timing	868	12	208	27	739
rawcaudio	7	12	2	< 1	6
rawdaudio	6	11	1	< 1	5
toast	157	11	35	5	134
untoast	85	12	21	3	72
cjpeg	14	12	3	< 1	12
djpeg	4	12	1	< 1	4
mpeg2encode	10834	11	2299	380	9227
mpeg2decode	3452	10	664	123	2940
pegwit gen	13	12	3	< 1	11
pegwit enc	30	11	7	1	26
pegwit dec	17	11	4	1	14

Tabela 4.22: Mibench *benchmarks*: estimativas de consumo para algoritmos *Consumer*

Program	**Test**	**Instr (M)**	**Power (mW)**	**Energy (mJ)**	t_{Sim} **(s)**	$\approx t_{Sim}$ **RTL (day)**
lame	small	7641	10.37	1584	322	6508
lame	large	89899	10.37	18637	3400	76564
cjpeg	small	25	11.57	6	1	21
djpeg	small	7	11.52	2	< 1	6
cjpeg	large	92	11.65	21	3	78
djpeg	large	25	11.44	6	1	22

Tabela 4.23: Mibench *benchmarks*: estimativas de consumo para algoritmos *Automotive*

Program	**Test**	**Instr (M)**	**Power (mW)**	**Energy (mJ)**	t_{Sim} **(s)**	$\approx t_{Sim}$ **RTL (day)**
basicmath	small	1305	10.57	276	57	1112
basicmath	large	21365	10.46	4471	909	18196
bitcnts	small	50	11.34	11	2	42
bitcnts	large	50	11.34	11	2	42
qsort	small	14	10.11	3	1	12
qsort	large	792	10.33	164	29	675
susan	small/corner	3	10.79	1	< 1	3
susan	small/edge	7	11.41	2	< 1	6
susan	small/smooth	30	11.57	7	1	26
susan	large/corner	42	11.26	10	2	36
susan	large/edge	174	11.84	41	7	148
susan	large/smooth	349	11.94	83	12	297

Tabela 4.24: Mibench *benchmarks*: estimativas de consumo para algoritmos *Network*

Program	**Test**	**Instr (M)**	**Power (mW)**	**Energy (mJ)**	t_{Sim} **(s)**	$\approx t_{Sim}$ **RTL (day)**
dijkstra	small	51	10.25	10	3	43
dijkstra	large	244	10.27	50	10	208
patricia	small	278	10.60	59	11	237
patricia	large	1761	10.60	373	83	1500

Tabela 4.25: Mibench *benchmarks*: estimativas de consumo para algoritmos *Office*

Program	**Test**	**Instr (M)**	**Power (mW)**	**Energy (mJ)**	t_{Sim} **(s)**	$\approx t_{Sim}$ **RTL (day)**
search	small	< 1	10.54	< 1	< 1	< 1
search	large	7	10.54	1	< 1	6

Tabela 4.26: Mibench *benchmarks*: estimativas de consumo para algoritmos *Security*

Program	**Test**	**Instr (M)**	**Power (mW)**	**Energy (mJ)**	t_{Sim} **(s)**	$\approx t_{Sim}$ **RTL (day)**
rijndael	large/decode	338	10.71	72	25	288
rijndael	small/encode	33	10.70	7	1	28
rijndael	small/decode	32	10.71	7	1	28
rijndael	large/encode	339	10.70	73	12	289
sha	small	13	11.14	3	< 1	11
sha	large	138	11.14	31	5	118

Tabela 4.27: Mibench *benchmarks*: estimativas de consumo para algoritmos *Telecommunication*

Program	**Test**	**Instr (M)**	**Power (mW)**	**Energy (mJ)**	t_{Sim} **(s)**	$\approx t_{Sim}$ **RTL (day)**
adpcm timing	small	681	12.01	164	32	580
adpcm rawcaudio	large	679	12.27	167	29	578
adpcm rawdaudio	large	585	11.40	133	25	498
adpcm timing	large	496	12.01	119	23	422
adpcm rawcaudio	small	34	12.22	8	2	29
adpcm rawdaudio	small	30	11.35	7	2	25
crc_32	small	30	9.59	6	1	26
crc_32	large	588	9.59	113	25	501
fft	4k	716	10.36	148	34	610
fft	inv 8k	1718	10.33	355	77	1463
fft	32k	14401	10.37	2985	660	12265
fft	inv 32k	13928	10.35	2883	532	11862
gsm toast	small	23	10.99	5	1	20
gsm untoast	small	13	12.16	3	< 1	11
gsm toast	large	1253	11.02	276	58	1067
gsm untoast	large	705	12.16	171	28	600

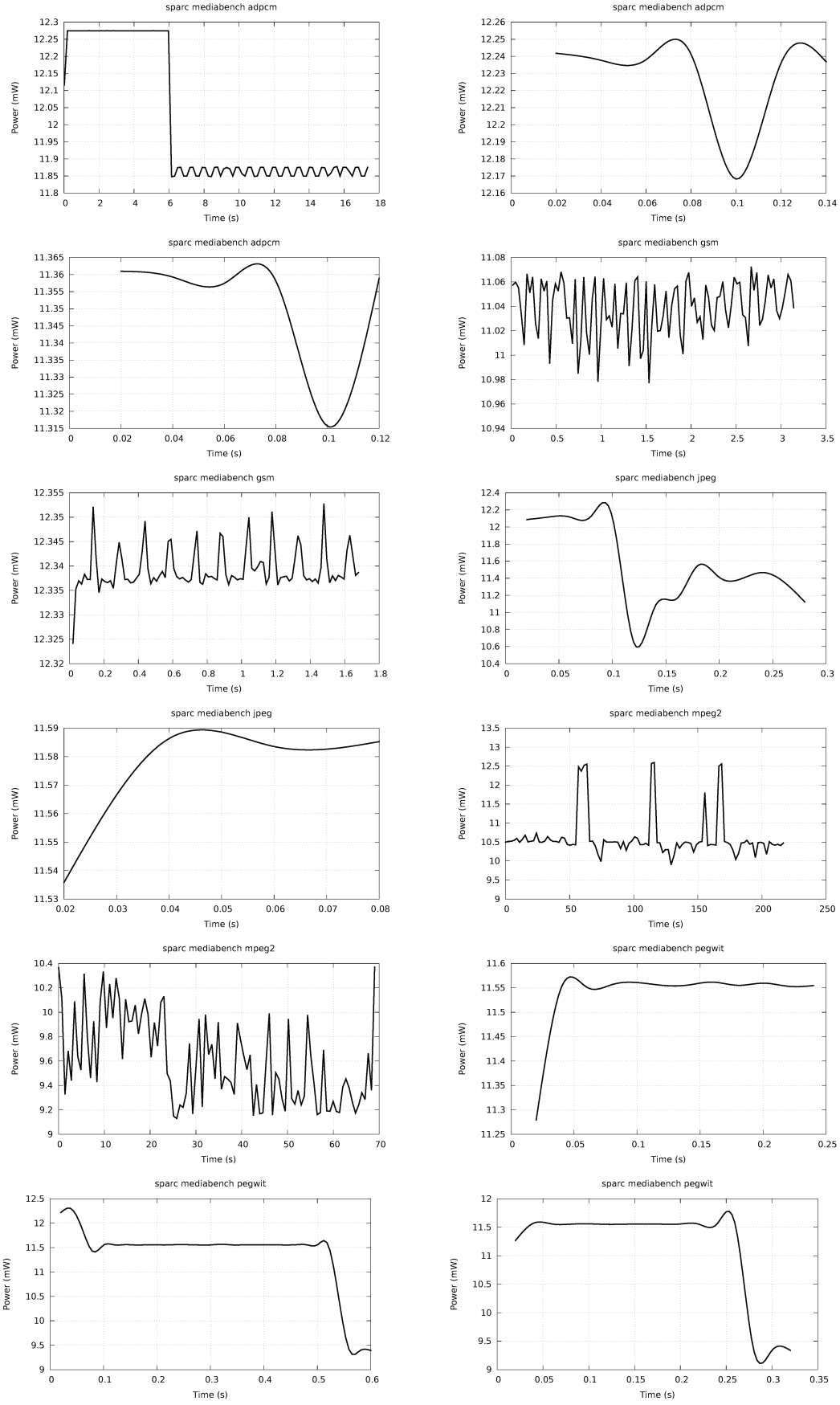
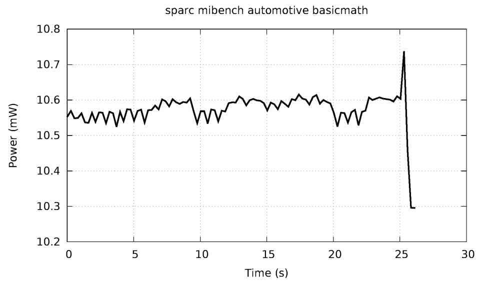
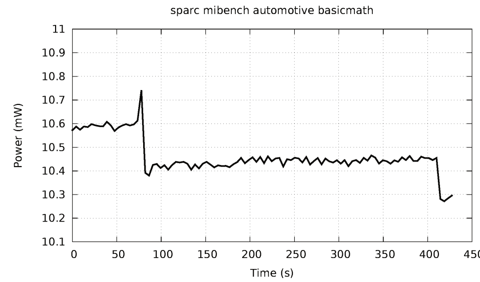
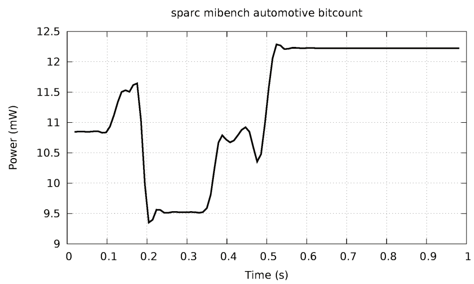
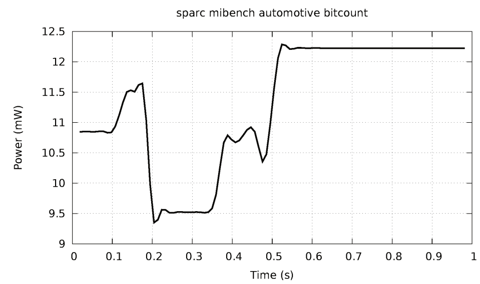


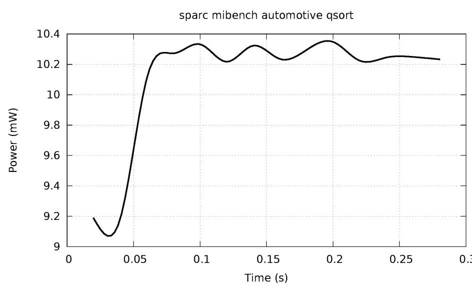
Figura 4.7: Perfis de energia do *benchmark* Mediabench

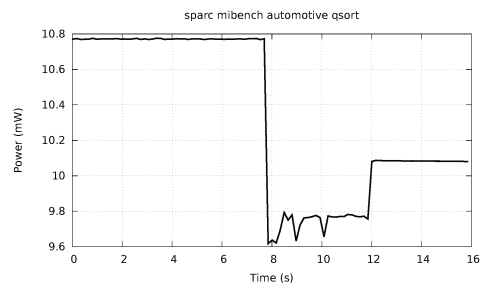

sparc mibench automotive basicmath
Power (mW)
10.2 10.3 10.4 10.5 10.6 10.7 10.8
0 5 10 15 20 25 30
Time (s)

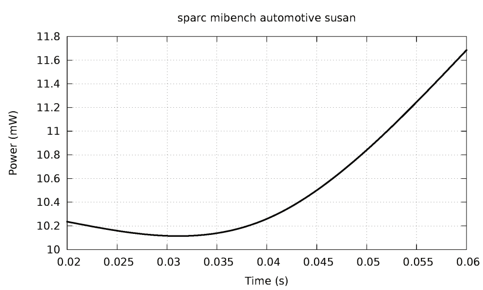

sparc mibench automotive basicmath
Power (mW)
10.1 10.2 10.3 10.4 10.5 10.6 10.7 10.8 10.9 11
0 50 100 150 200 250 300 350 400 450
Time (s)

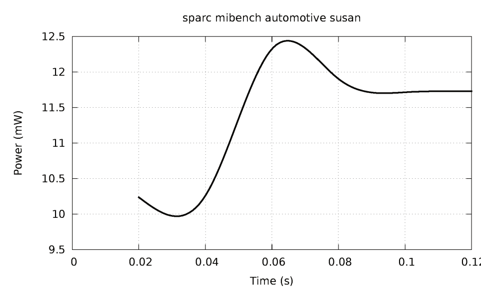

sparc mibench automotive bitcount
Power (mW)
9 9.5 10 10.5 11 11.5 12 12.5
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
Time (s)

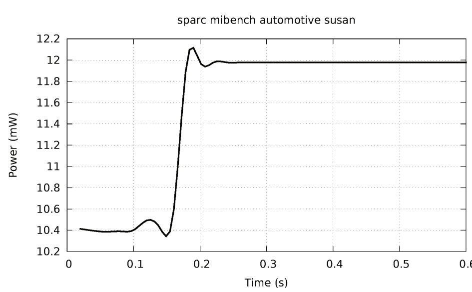

sparc mibench automotive bitcount
Power (mW)
9 9.5 10 10.5 11 11.5 12 12.5
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
Time (s)

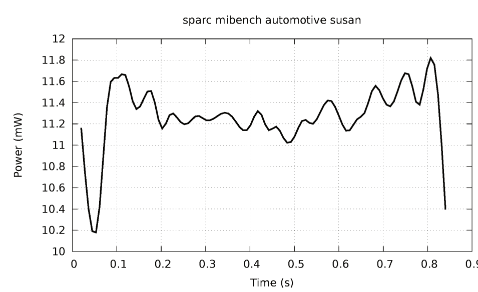

sparc mibench automotive qsort
Power (mW)
9 9.2 9.4 9.6 9.8 10 10.2 10.4
0 0.05 0.1 0.15 0.2 0.25 0.3
Time (s)

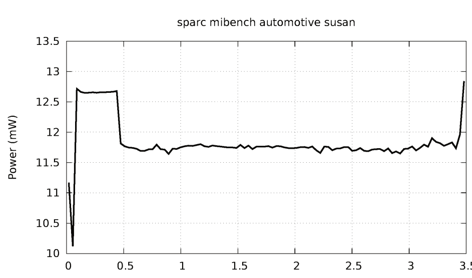

sparc mibench automotive qsort
Power (mW)
9.6 9.8 10 10.2 10.4 10.6 10.8
0 2 4 6 8 10 12 14 16
Time (s)

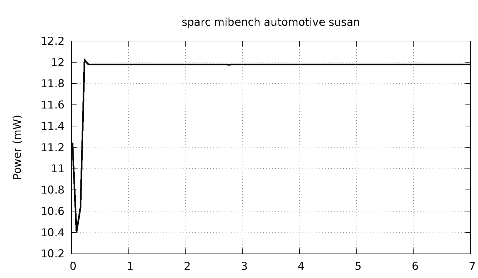

sparc mibench automotive susan
Power (mW)
10 10.2 10.4 10.6 10.8 11 11.2 11.4 11.6 11.8
0.02 0.025 0.03 0.035 0.04 0.045 0.05 0.055 0.06
Time (s)

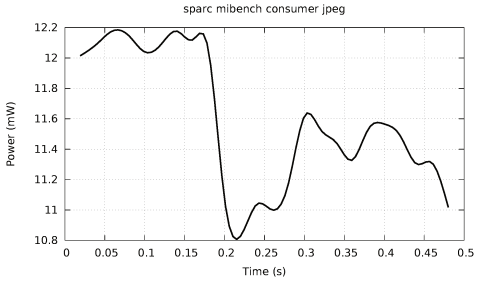

sparc mibench automotive susan
Power (mW)
9.5 10 10.5 11 11.5 12 12.5
0 0.02 0.04 0.06 0.08 0.1 0.12
Time (s)

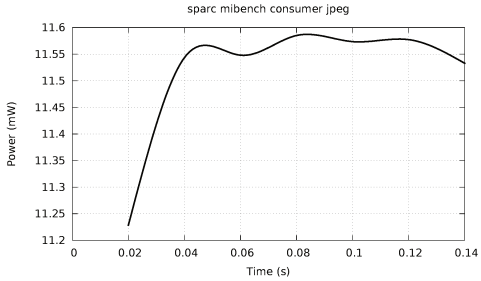

sparc mibench automotive susan
Power (mW)
10.2 10.4 10.6 10.8 11 11.2 11.4 11.6 11.8 12 12.2
0 0.1 0.2 0.3 0.4 0.5 0.6
Time (s)


sparc mibench automotive susan
Power (mW)
10 10.2 10.4 10.6 10.8 11 11.2 11.4 11.6 11.8 12
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
Time (s)

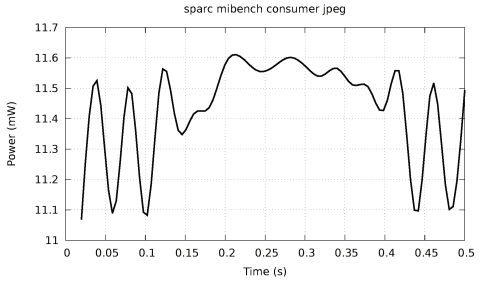

sparc mibench automotive susan
Power (mW)
10 10.5 11 11.5 12 12.5 13 13.5
0 0.5 1 1.5 2 2.5 3 3.5
Time (s)

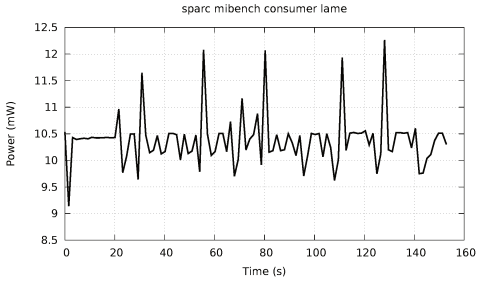

sparc mibench automotive susan
Power (mW)
10.2 10.4 10.6 10.8 11 11.2 11.4 11.6 11.8 12 12.2
0 1 2 3 4 5 6 7
Time (s)

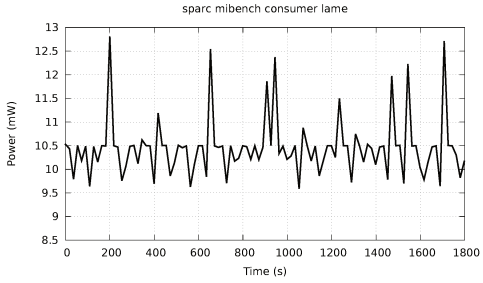

sparc mibench consumer jpeg
Power (mW)
Time (s)

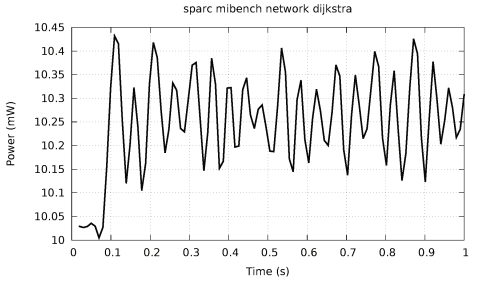

sparc mibench consumer jpeg
Power (mW)
Time (s)

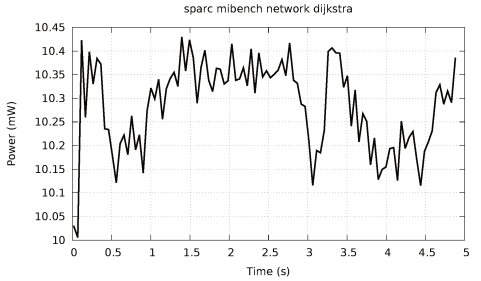

sparc mibench consumer jpeg
Power (mW)
Time (s)

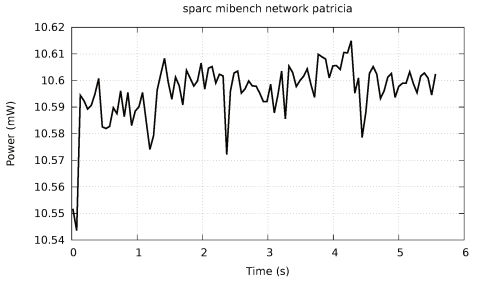

sparc mibench consumer jpeg
Power (mW)
Time (s)

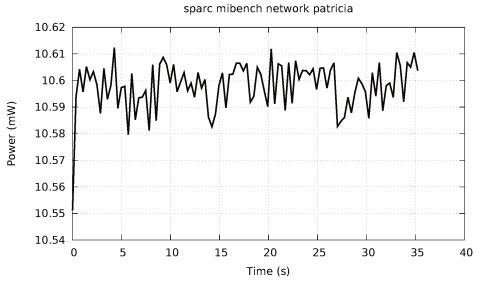

sparc mibench consumer lame
Power (mW)
Time (s)

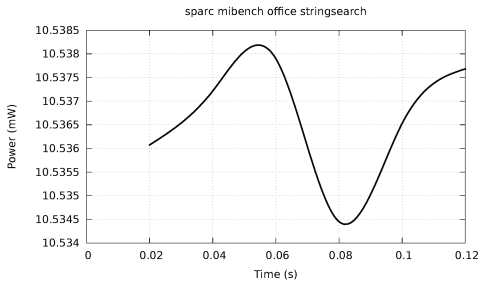

sparc mibench consumer lame
Power (mW)
Time (s)

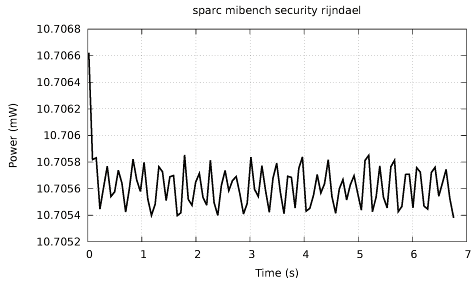

sparc mibench network dijkstra
Power (mW)
Time (s)

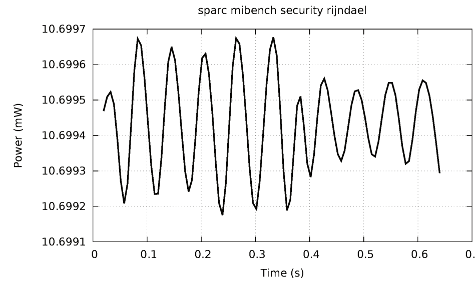

sparc mibench network dijkstra
Power (mW)
Time (s)

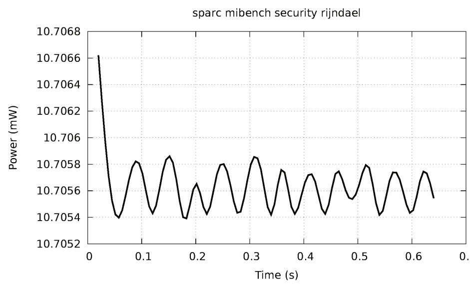

sparc mibench network patricia
Power (mW)
Time (s)

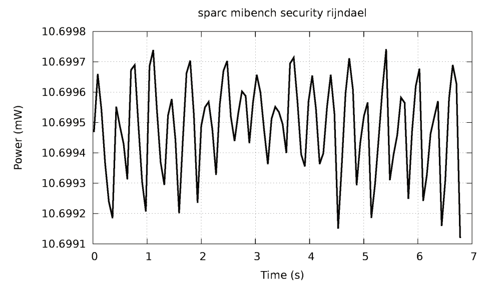

sparc mibench network patricia
Power (mW)
Time (s)

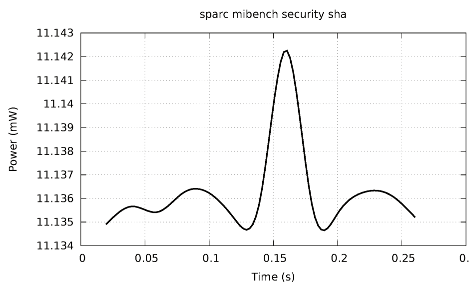

sparc mibench office stringsearch
Power (mW)
Time (s)

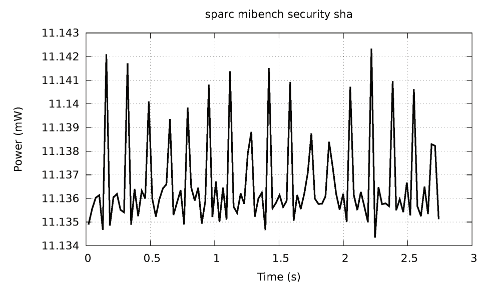

sparc mibench security rijndael
Power (mW)
Time (s)

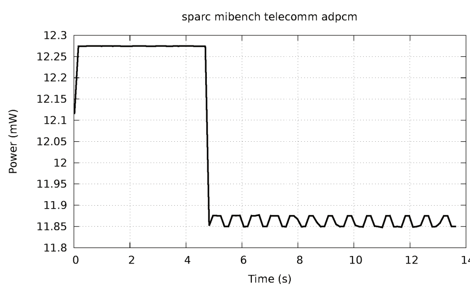

sparc mibench security rijndael
Power (mW)
Time (s)

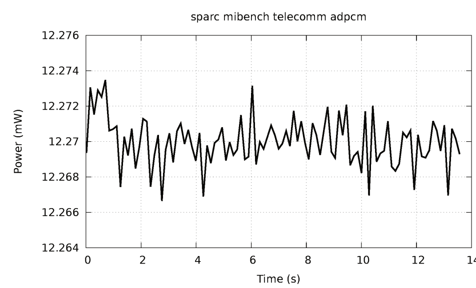

sparc mibench security rijndael
Power (mW)
Time (s)

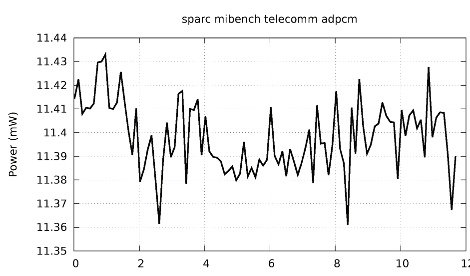

sparc mibench security rijndael
Power (mW)
Time (s)

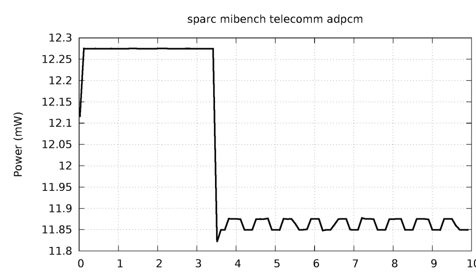

sparc mibench security sha
Power (mW)
Time (s)

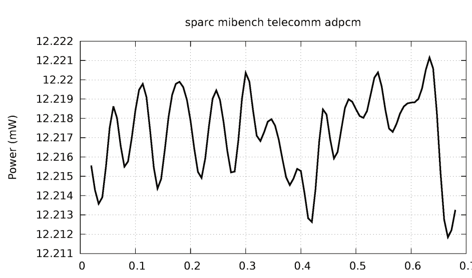

sparc mibench security sha
Power (mW)
Time (s)

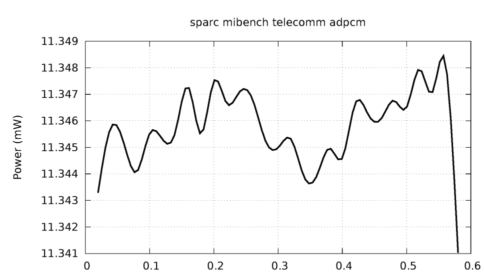

sparc mibench telecomm adpcm
Power (mW)
Time (s)


sparc mibench telecomm adpcm
Power (mW)
Time (s)


sparc mibench telecomm adpcm
Power (mW)
Time (s)


sparc mibench telecomm adpcm
Power (mW)
Time (s)


sparc mibench telecomm adpcm
Power (mW)
Time (s)


sparc mibench telecomm adpcm
Power (mW)
Time (s)

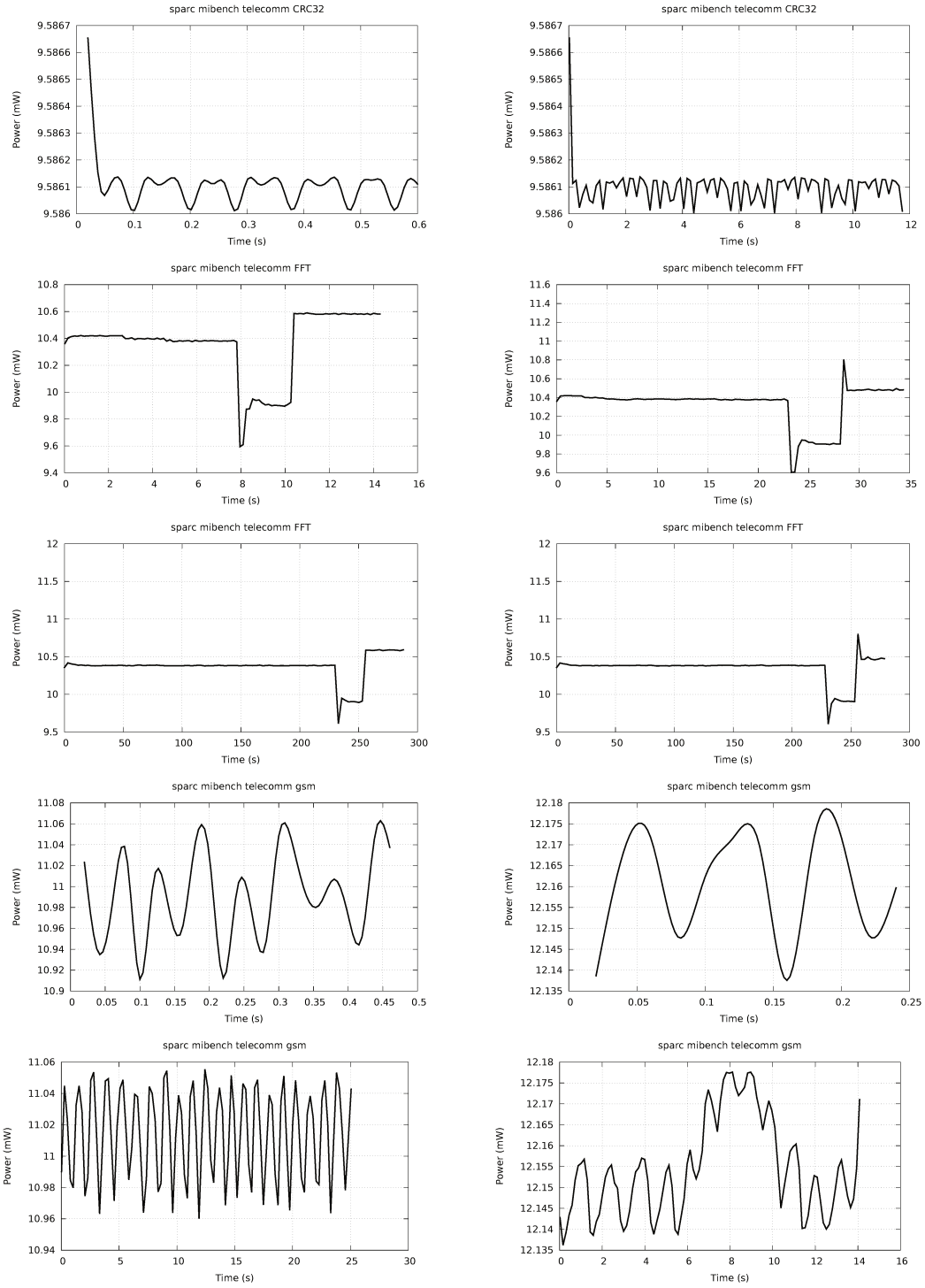


Figura 4.8: Perfis de energia do *benchmark* Mibench

4.3 Desenvolvimento e análise de consumo de MIPS-I em acRTL

Um dos experimentos realizados neste trabalho foi adicionar acRTL à plataforma acSynth. Este experimento foi representativo pois:

- Exigiu o fluxo completo de desenvolvimento, avaliando os prós e contras do atual estágio de acRTL;
- Permitiu uma análise de consumo e a possibilidade de comparação com o comportamento de um processador de referência, neste caso, o Plasma;

O resultado final foi um processador MIPS-I *pipelined* funcional, com exceção de instruções de multiplicação e divisão. O processador Plasma foi utilizado como *golden model*. O desenvolvimento, em especial o tratamento de *hazards* e *forwarding* foi embasado teoricamente pelas explicações apresentadas no livro didático *"Computer Organization and Design"* de Patterson e Hennessy [47], que detalha no capítulo 4 o desenvolvimento de um processador MIPS-I.

Como Plasma foi utilizado como referência, as instruções de *load* e *store* desalinhadas não foram implementadas. Isto porque Plasma não pôde implementar estas instruções no passado por questões de patentes da MIPS [43, 48]. Apesar da patente ter expirado em 2006, a funcionalidade não foi totalmente implementada por conta desta antiga restrição.

4.3.1 O desenvolvimento em acRTL

O fluxo de trabalho acRTL exige a configuração de uma plataforma de trabalho composta pelo SystemC, ArchC, o acRTL propriamente e o gsc. Assume-se para um projeto acRTL que exista um modelo ArchC completo e funcional que será o molde para elaboração dos *templates*. Para projetos *pipelined*, o parâmetro `ac_pipe` precisa ser definido com os nomes dos estágios requeridos.

Tendo os arquivos de descrição de arquitetura, basta invocar acRTL a partir do diretório com os arquivos ARCH e ISA:

```
$ acrtl mips_pipeline.cpp
```

Neste exemplo, os arquivos ArchC utilizavam o prefixo `mips_pipeline`. Uma vez gerado o *template* em SystemC, o projetista pode iniciar a tarefa de implementação das instruções. Cada instrução descrita em ArchC recebe no *template* uma função correspondente. Além disso, são geradas funções para cada tipo de *opcode* declarado, além de uma função geral. Elas são importantes para trechos de código que tenham interferência em

aspectos globais do processador. Estas funções gerais acabam sendo o local perfeito para a implementação de tratamento de *hazards* e *forwarding*.

Podemos ver no trecho de código 4.2 o formato de uma função gerada por acRTL. Neste caso, se trata da função para instrução `add`. Todas as demais funções relatadas são semelhantes a esta, inclusive as globais. Cada `case` refere-se a um do estágios de *pipeline* descritos no comando `ac_pipe`.

```
void ac_behavior(add) {
  switch (stage) {
    case IF: {
      break;
    }
    case ID: {
      break;
    }
    case EX: {
      break;
    }
    case MEM: {
      break;
    }
    case WB: {
      break;
    }
  }
}
```

Código 4.2: Função gerada pelo *template* para instrução `add` do processador MIPS-I

Para testes, foi utilizado o processador Plasma como modelo de referência e programas gerados pelo acPowerGen para explorar o funcionamento das instruções. Um testbench precisou ser elaborado com o intuito de ler os programas, executar sobre Plasma e retornar relatórios com o *status* dos registradores a cada ciclo de execução.

O *testbench* executa um mesmo código sobre o processador em desenvolvimento e o *golden model* e, então, os resultados comparados ciclo a ciclo. Esse fluxo formata um processo padrão de teste de código de hardware, como apresentado na figura 4.9. Como *source*, foram utilizados pequenos programas desenvolvidos em MIPS-I com o objetivo de analisar a execução padrão e encontrar erros comuns de *hazard*. O *driver* foi o *script* de compilação e geração de ROM em formato texto criado para alimentar o DUV (*Design Under Verification*). O monitor e o verificador foram implementados com simples *diff* dos arquivos de saída.

É importante dizer que o *golden model* poderia ser outro qualquer, independente de linguagem. Seria até mesmo interessante que fosse possível no futuro utilizar simulações sobre o próprio modelo ArchC como referência, dado que já há suporte para descrição comportamental na ADL.

Com esta plataforma de teste, iniciaram-se os trabalhos de descrição das instruções. Os desenvolvedores da ferramenta acRTL não definiram um fluxo de trabalho sobre o

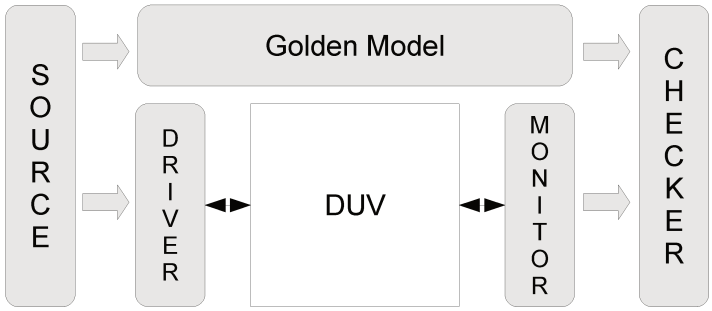


Figura 4.9: Modelo padrão de *testbench* sobre módulos de hardware

template, ficando a cargo do usuário da plataforma definir o modo de trabalho e o escalonamento das tarefas. Decidiu-se iniciar as atividades pelas instruções aritméticas, mais simples de serem elaboradas e por sofrerem menor interferência de *hazards*.

Os tratamentos de *hazard* foram gerados de forma iterativa ao longo da implementação, conforme impactavam na instrução a ser elaborada. O livro de referência do Patterson serviu como mapa na aplicação das soluções.

De um ponto de vista prático, o trabalho se resumiu a preencher as lacunas do *template*, em linguagem SystemC.

Um exemplo da instrução `add` preenchida encontra-se no trecho de código 4.3. O sinal de *overflow* foi gerado mas não tratado. A implementação de `beq` encontra-se no código 4.4. Note o aumento de complexidade no código para *branches*, provocado pela necessidade de tratamentos de *control hazards*. Esta solução de código não é ótima, porém, funcional.

```
void ac_behavior(add) {
  switch (stage) {
    case IF: {
      break;
    }
    case ID: {
      ID_EX.regwrite = 1;
      break;
    }
    case EX: {
      // auxiliar variables to check overflow
      sc_uint<33> add_alures = 0;

      // add calculation
      add_alures = id_value1 + id_value2;

      // return
      EX_MEM.alures = add_alures.range(31,0);

      // overflow checking
      if (add_alures.range(32,32) == 1) {
        epc = address;
        cause = 12;
```

```
      }
      break;
    }
    case MEM: {
      break;
    }
    case WB: {
      break;
    }
  }
}
```

Código 4.3: Instrução `add` do processador MIPS-I *pipelined* em acRTL em SystemC

```
void ac_behavior(beq) {
  switch (stage) {
    case IF: {
      break;
    }
    case ID: {
      /* Stall if ALU or load just before */
      if ((ID_EX.regwrite.read() == 1) &&
        (ID_EX.rd.read() != 0) &&
        ((ID_EX.rd.read() == rs) || ID_EX.rd.read() == rt)) {

        // Stall
        ac_ID.ac_bubble = 1;
        ac_pc = ac_pc.read();
        ID_EX.npc = ac_pc.read() + 4;
      }
      else if (br_value1 == br_value2) { // No ALU or load just before and brach
taken
        ac_ID.ac_bubble = 1;
        branch_taken = 1;
        ac_pc = ac_pc.read() + offset.read() - 4;
      }
      else
        ac_EX.ac_bubble = 1; // Branch not taken and no need for further actions
      break;
    }
    case EX: {
      /* Branch solved here for ALU or load just before */
      if ((EX_MEM.regwrite.read() == 1) &&
        (EX_MEM.rdest.read() != 0) &&
        ((EX_MEM.rdest.read() == rs) || EX_MEM.rdest.read() == rt)) {

        // Stall
        ac_ID.ac_bubble = 1;
        ac_pc = ac_pc.read();
        ID_EX.npc = ac_pc.read() + 4;
      }
      else if (branch_taken) { // No ALU or load just before
        ac_ID.ac_bubble = 1;
        ac_MEM.ac_bubble = 1; // Branch taken and no need for further actions
      }
      break;
    }
    case MEM: {
```

```
      /* Branch solved here for ALU or load just before */
      if ((MEM_WB.regwrite.read() == 1) &&
        (MEM_WB.rdest.read() != 0) &&
        ((MEM_WB.rdest.read() == rs) || MEM_WB.rdest.read() == rt)) {
          if (ex_value1 == ex_value2) { // branch
            branch_taken = 1;
            ac_pc = ac_pc.read() + offset.read() - 4;
          }
      }

      ac_ID.ac_bubble = 1;
      break;
    }
    case WB: {
      if (branch_taken)
        ac_ID.ac_bubble = 1;
      break;
    }
  }
}
```

Código 4.4: Instrução `beq` do processador MIPS-I *pipelined* em acRTL em SystemC

Ao longo do processo, as instruções de *load* e *store* mostraram-se as mais complexas e não foram otimizadas, com uso de bolhas no *pipeline*. Isso porque para usufruir do máximo oferecido pela arquitetura, seria preciso implementar uma cache, algo fora do atual escopo.

Todas as instruções da arquitetura padrão MIPS-I foram elaboradas, com exceção das que exigiam acessos especiais, como MUL/DIV dependentes de co-processamento, e aquelas não presentes na descrição Plasma.

A principal conclusão deste processo foi a comprovação de que acRTL apresenta-se como uma alternativa de fluxo de projeto factível. No entanto, complexo em vários aspectos.

4.3.2 Desafios em projetos acRTL

O desenvolvimento em acRTL e a posterior síntese apresentaram desafios, o que é comum quando se estabelece e utiliza uma metodologia nova. Começando pelo tempo de implementação de solução.

Um dos conceitos principais em torno de acRTL é que ele possa fornecer um meio rápido de criar um processador sintetizável diretamente de ArchC. O problema é que a linguagem ArchC não fornece hoje diretamente todas as informações necessárias para isto, o que limita acRTL a gerar apenas um *template* de desenvolvimento com lacunas para o desenvolvedor preencher.

Há um ganho de tempo de desenvolvimento aqui, uma vez que a estrutura básica inerente a CPUs encontra-se toda presente, havendo até a prévia elaboração das etapas de

fetch e *decode*. No entanto, precisar implementar todas as instruções desde o estágio inicial não é uma tarefa rápida. Especialmente quando o desenvolvedor está elaborando um processador *pipeline*, lidando com *hazard* e *forwarding*. Foi também uma tarefa repetitiva, exigindo trechos de código muito semelhantes em mais de um caso, sem que houvessem alternativas suportadas pela plataforma para reuso de código.

Foi preciso um longo ciclo de testes e retestes ainda, não fugindo totalmente do paradigma de desenvolvimento com linguagens HDL. Mais que isso, entender a estrutura de acRTL foi complexo porque não há uma separação clara entre o código que deve ser mantido estático e o código onde se espera que o desenvolvedor atue. Outro problema surge na questão de compatibilidade. O código gerado por acRTL é muito particular e foi desenhado para funcionar com o tradutor **gsc**, também elaborado durantes os trabalhos de Goto [9]. O problema é que isto limita o uso de outras ferramentas. Um problema adicional é que o tradutor gsc não possui mais suporte, dependendo de bibliotecas antigas e sem uma documentação clara para prosseguir os trabalhos. Isso cria um empecilho muito alto para aprimorar o fluxo acRTL com uso de gsc. Apesar disso, acRTL é muito mais claro e limpo, permitindo adição de melhorias. O presente autor sugere que para prosseguir com a evolução desta vertente de trabalhos sobre ArchC deve-se trabalhar sobre acRTL, permitindo-o ser compatível com outros tradutores SystemC ou, idealmente, traduzir códigos diretamente para linguagens sintetizáveis como VHDL e Verilog.

Seria interessante também haver uma forma clara em ArchC para realizar descrições com precisão de ciclos onde fosse possível informar detalhes de um pipeline, mapeando o processamento de instruções aos estágios. Se este método de descrição existisse, acRTL poderia fazer uso para gerar processadores sintetizados com uma redução do esforço de desenvolvimento em linguagens de níveis mais baixos. Adicionalmente, ArchC poderia ainda fornecer uma execução comportamental como *golden model*, oferendo uma leitura paralela da descrição inicial como contra-prova.

Outro problema apresentado foi que o *template* gerado por acRTL não era totalmente sintetizável. O *template* é todo estruturado com base em uma cascata de *process* e *tasks*. Como raízes, existem os processos principais, um para cada estágio do *pipeline*. No caso do MIPS-I, acRTL gerou 5 processos, chamados IF_prc, ID_prc, EX_prc, MEM_prc e WB_prc, correspondendo respectivamente aos estágios de *fetch*, *decode*, *execute*, *memory* e *write-back*.

Essa estrutura de código naturalmente faz com que mais de um processo tenha o potencial de escrever ao mesmo tempo sobre um mesmo registrador. De fato, isso não ocorre em acRTL, mas a descrição dá abertura para esta interpretação. O exemplo mais claro é do PC, que pode receber sinais de outros estágios além do *fetch*, no caso de *branches* e *jumps*. Em nível SystemC, o código gerado é factível para simulação, mas no momento que gsc traduz para Verilog, o código não é aceitável. Foi preciso detectar este

problema e resolver manualmente. O código problemático inicialmente gerado por gsc encontra-se parcialmente no excerto de código 4.5, destacando os trechos importantes. Ele foi modificado para a estrutura apresentada no código 4.6.

Os conflitos surgiam por conta das sub-tarefas chamadas pela tarefa principal **isa**. O processo de síntese agrupava as subpartes das tarefas para cada um destes processos e se um mesmo sinal fosse modificado em mais de um estágio, o que é natural acontecer, a simulação falhava. A solução foi criar apenas um ponto de chamada ISA e, desta forma, permitir ao sintetizador identificar que não haviam múltiplos *drives* ao mesmo tempo.

```
always @( posedge clk or negedge resetn )
begin : IF_prc
    (...)
        if (match( rdata )) begin
          ac_ID_ac_buffer  <=  rdata;
          ac_ID_ac_bubble  <=  0;
          isa(IF, rdata);
        end
    (...)
end

always @( posedge clk or negedge resetn )
begin : ID_prc
  (...)
      if (match( ac_ID_ac_buffer )) begin
        ac_behavior__instruction(ID, ac_ID_ac_buffer);
        if (!ac_ID_ac_bubble) begin
          isa(ID, ac_ID_ac_buffer);
        end
      end
  (...)
end

always @( posedge clk or negedge resetn )
begin : EX_prc
  (...)
      if (match( ac_EX_ac_buffer )) begin
        ac_behavior__instruction(EX, ac_EX_ac_buffer);
        if (!ac_EX_ac_bubble) begin
          isa(EX, ac_EX_ac_buffer);
        end
      end
  (...)
end

always @( posedge clk or negedge resetn )
begin : MEM_prc
  (...)
      if (match( ac_MEM_ac_buffer )) begin
        ac_behavior__instruction(MEM, ac_MEM_ac_buffer);
        if (!ac_MEM_ac_bubble) begin
          isa(MEM, ac_MEM_ac_buffer);
        end
      end
  (...)
end
```

```
always @( posedge clk or negedge resetn )
begin : WB_prc
  (...)
      if (match( ac_WB_ac_buffer )) begin
        ac_behavior__instruction(WB, ac_WB_ac_buffer);
        if (!ac_WB_ac_bubble) begin
          isa(WB, ac_WB_ac_buffer);
        end
      end
  (...)
end
```

Código 4.5: Trecho de controle originalmente gerado por gsc

```
always @( posedge clk or negedge resetn )
begin : IF_prc
  (...)
        if (match( rdata )) begin
          ac_ID_ac_buffer  <=  rdata;
          ac_ID_ac_bubble  <=  0;
          isa(IF, rdata);
        end
        else begin
          epc  =  ac_pc;
          cause  =  1;
        end

        ac_EX_ac_buffer  <=  ac_ID_ac_buffer;
        ac_EX_ac_bubble  <=  ac_ID_ac_bubble;
        if (match( ac_ID_ac_buffer )) begin
          ac_behavior__instruction(ID, ac_ID_ac_buffer);
          if (!ac_ID_ac_bubble) begin
            isa(ID, ac_ID_ac_buffer);
          end
        end

        ac_MEM_ac_buffer  <=  ac_EX_ac_buffer;
        ac_MEM_ac_bubble  <=  ac_EX_ac_bubble;
        if (match( ac_EX_ac_buffer )) begin
          ac_behavior__instruction(EX, ac_EX_ac_buffer);
          if (!ac_EX_ac_bubble) begin
            isa(EX, ac_EX_ac_buffer);
          end
        end

        ac_WB_ac_buffer  <=  ac_MEM_ac_buffer;
        ac_WB_ac_bubble  <=  ac_MEM_ac_bubble;
        if (match( ac_MEM_ac_buffer )) begin
          ac_behavior__instruction(MEM, ac_MEM_ac_buffer);
          if (!ac_MEM_ac_bubble) begin
            isa(MEM, ac_MEM_ac_buffer);
          end
        end

        if (match( ac_WB_ac_buffer )) begin
          ac_behavior__instruction(WB, ac_WB_ac_buffer);
          if (!ac_WB_ac_bubble) begin
```

```
            isa(WB, ac_WB_ac_buffer);
          end
        end
  (...)
end
```

Código 4.6: Trecho de controle modificado com chamadas *task isa* agrupadas

Com respeito ainda a estrutura de múltiplas tarefas, vale notar que o sintetizador tem um bom trabalho computacional para gerar o circuito. Houve um aumento visível no tempo de síntese e quando não há casamento entre lógica e a FPGA escolhida, como em casos de insuficiência de recursos, o processamento do sintetizador é massivo ao tentar mapear celúlas de FPGA. No caso da ferramenta Xilinx ISE 14.2, a versão mais recente no momento, o programa nem mesmo conseguiu gerar uma resposta aos erros provocados pela síntese, caindo em falhas catastróficos de execução (nenhum retorno para depuração e, eventualmente, *segmentation fault*).

Adicionalmente, o processador MIPS gerado por acRTL consome uma área significantemente maior, de modo que sínteses na mesma FPGA em que Plasma foi sintetizada se torna impossível. Para efeito de teste, realizamos a síntese numa FPGA grande e comparamos o número de LUTs utilizadas. O modelos escolhido foi o Virtex 6 xc6vlx75t-fft784 especificamente por conta do volume de recursos. Com os dados desta síntese, pudemos realizar uma análise comparativa. Os relatórios Xilinx não utilizam métricas consistentes entre famílias diferentes de FPGAs, o que impossibilita comparar todos os aspectos de utilização de recursos. Como o processador Plasma foi elabora apenas com suporte de pinagem a Spartan3E, um remapeamento completo seria requerido para uma comparação equivalente em um mesmo modelo de FPGA. Apesar disto, fica muito evidente pelos parâmetros apresentados na tabela 4.28 o quão maior acRTL é em relação a Plasma, mesmo executando menos instruções do que este, pois não há coprocessador de multiplicação e divisão no MIPS acRTL. Um agravante é o fato de que as LUTs presentes no Virtex6 são maiores do que as presentes em Spartan3E.

Tabela 4.28: Comparação de uso de recursos entre MIPS-I acRTL e Plasma

	acRTL	Plasma
Number of Slice Registers	1820	281
Number of LUTs	16142	2956
Number of occupied Slices	7349	1629
Number of bonded IOBs	131	136
Number of BUFGs	1	1

Outro problema foi o fato do processador acRTL não ter atingido *timing* a 100 MHz, mesmo em uma FPGA maior. O sistema foi então sintetizado em 50 MHz.

Em suma, acRTL não consegue gerar um processador eficiente do ponto de vista de recursos e *timing*. Além disso, mantém o nível de complexidade de desenvolvimento alto. Isto demonstra que, apesar de factível, a síntese direta com base em ADL ainda possui um longo caminho de estudo para se tornar tão prático e aplicável quanto soluções em níveis mais baixos de desenvolvimento. A elevação do nível de abstração é uma tendência no desenvolvimento acadêmico do setor e os desafios são muitos para qualquer equipe que enverede nesta tarefa.

4.3.3 Resultados e comparativo com Plasma

O desenvolvimento do processador levou em torno de 6 meses, com dois desenvolvedores em tempo parcial.

As simulações ocorreram com o código em nível HDL, após a execução de gsc, portanto, sobre código sintetizável. O pipeline foi inteiramente desenvolvido, com exceção de instruções associadas a periféricos ou coprocessadores. O conceito se mostrou aplicável e coerente, apesar da necessidade de melhoras em diversos aspectos. Do ponto de vista técnico, as ferramentas precisam ser refinados para que acRTL seja largamente utilizado.

Para comprovar o uso de acRTL dentro do fluxo em acSynth, foi realizada a caracterização do novo processador. Este foi, então, comparado com os dados obtidos anteriormente para Plasma na plataforma Xilinx. Os valores estão apresentados na tabela 4.29. Como os dados correspondem a energia dinâmica por instrução, é possível discutir e comparar os consumos entre os dois processadores uma vez que a energia dinâmica é fortemente atrelada a arquitetura do sistema.

Em relação ao consumo, os resultados foram mistos, com algumas instruções mais econômicas e outras mais custosas. Salta aos olhos o menor consumo de acRTL em relação a Plasma para acesso a memória externa. Isto porque acRTL foi implementado com um acesso a memória muito simples, através de *stall* no barramento no momento da captura dos dados, sem prever a presença de cache. Com isto, acRTL consome mais ciclos do que Plasma para o mesmo número de instruções *load* e *store*. É um problema semelhante ao que observamos no caso das instruções de coprocessador mul/div e a geração de *stall*. Como não foi realizada uma análise mais fina sobre os ciclos de clock por instrução, supondo-se que todas as instruções executam em um ciclo, distorções como esta surgem.

O presente trabalho contribui neste aspecto ao demonstrar como um desenvolvedor de *hardware* pode utilizar acRTL no projeto de processadores. Dados os resultados aqui apresentados, concluímos que acRTL não se encontra hoje em um estado maduro suficiente para desenvolvimento de projetos. No entanto, a ferramenta representa uma reflexão acadêmica bem-vinda na busca da solução do problema do aumento do tempo de desenvolvimento de SoCs. O conceito do uso de *templates* foi uma ideia acertada, precisando

Tabela 4.29: Comparação entre o consumo do MIPS-I acRTL e Plasma

	acRTL (nJ)	Plasma (nJ)		acRTL (nJ)	Plasma (nJ)
add	1.01	1.83	lwl	1.05	3.34
addi	2.39	1.64	lwr	1.43	3.34
addiu	2.42	1.64	mfhi	0.85	0.61
addu	1.73	1.03	mflo	0.84	0.61
andi	1.29	0.89	mthi	0.85	0.99
beq	2.92	3.08	mtlo	0.85	0.94
bgez	3.07	3.08	nop	0.72	0.51
bgezal	3.29	3.08	ori	1.46	0.98
bgtz	3.03	3.08	sb	1.03	5.39
blez	3.12	3.08	sh	1.54	5.68
bltz	2.69	3.08	sll	0.97	0.76
bltzal	3.1	3.08	sllv	1.03	0.76
bne	2.72	3.08	slt	1.01	1.35
instr_and	1.02	0.75	slti	1.38	1.23
instr_nor	2.57	1.53	sltiu	2.85	1.23
instr_or	1.02	1.01	sltu	1.02	1.35
instr_xor	1.54	1.67	sra	0.98	0.75
j	3.28	3.5	srav	1.03	1.07
jal	3.62	0.85	srl	0.98	0.74
jalr	1.82	6.37	srlv	1.01	0.75
jr	1.63	6.14	sub	1.01	1.19
lb	0.92	4.44	subu	1.78	1.19
lbu	1.71	2.47	sw	1.46	5.76
lh	0.86	4.55	swl	2.59	5.76
lhu	1.8	2.57	swr	0.98	5.76
lui	0.95	0.88	xori	2.06	1.35
lw	2.19	3.34			

apenas de ajustes, como uma definição mais clara do que é código estático e quais as lacunas que o usuário deve preencher. O formato dos *templates* também precisa evoluir para uma estrutura mais legível para ferramentas de tradução, fugindo da dependência técnica da ferramenta gsc. Seria interessante que estes *templates* ainda mirassem na mitigação dos problemas de área e *timing*, criando estruturas que forcem uma arquitetura mais eficiente. Apesar da geração automática não mirar no caso ótimo, a solução atual está aquém da necessária para ser considerada útil em projetos práticos. A aplicação destas melhorias tornariam acRTL uma ferramenta prática, permitindo a rápida implementação de arquiteturas de processadores e a exploração de novos paradigmas.

Capítulo 5

Conclusões

Esta dissertação apresentou a plataforma de desenvolvimento acSynth, voltada para integração de ferramentas ArchC com o objetivo de auxiliar o desenvolvimento e estudo de processadores em nível de arquitetura. Grande parte do mérito do trabalho deve-se a integração de outras ferramentas compatíveis com a ADL ArchC, permitindo o intercâmbio de dados e recursos. Com isto, novos resultados interessantes são obtidos. Paralelamente, acSynth apresentou uma metodologia de integração que pode ser reaplicada em outras soluções da infra-estrutura de projetos em ArchC. Os resultados práticos apresentados demonstram que a ferramenta pode ser utilizada em aplicações práticas. As atividades em acRTL serviram para analisar a capacidade desta ferramenta na geração automática de processadores. Conclui-se que acRTL possui um grande potencial e encontra-se alinhada a uma tendência na busca de redução de tempo de engenharia de projetos, porém, não é hoje uma ferramenta suficientemente prática para largo uso. Com melhorias, a ferramenta acRTL pode ser muito poderosa, rebaixando IPs de processadores simples a categoria de *commodities* e permitindo a exploração de novas arquiteturas com rápido salto de ADL para RTL.

Como contribuições principais da presente dissertação, podemos citar:

- Integrou as ferramentas PowerSC, acPower e acSim para criar um sistema unificado de análise de consumo de energia, capacitado para futuras expansões tanto em nível ADL através de melhorias em acSim com simulações ciclo a ciclo, quanto em nível SDL com a adição de novos módulos em SoC e agrutinação dos dos dados de consumo de energia através da biblioteca PowerSC;

- Apresentou uma metodologia completa de caracterização de processadores sob plataforma acSynth, em um fluxo independente de qualquer ferramentas de síntese específica, passível de expansão de funcionalidades e melhorias com uso de algoritmos e procedimentos mais complexos e modernos;

- Exemplificou seu uso prático, com resultados coerentes para dois processadores, Plasma e Leon3, de duas arquiteturas diferentes MIPS-I e SPARCv8, utilizando duas plataformas de síntese diferentes, Altera e Xilinx, o que comprova a independência da ferramenta e exemplifica sua capacidade de integração com fluxo de projeto diversos;

- Apresentou métodos para expandir acSim para abarcar novos aspectos de simulação em nível de arquitetura, com alta reusabilidade através de arquivos de bancos de dados, o que permite que dados gerados por um projetista possam ser distribuídos para outros reduzindo o tempo de engenharia geral requerido dos envolvidos na implementação de soluções;

- Demonstrou o uso de acRTL na elaboração de um MIPS-I funcional, *pipelined* de cinco estágios, com todas as instruções puramente dependentes do *pipeline* implementadas, exibindo os prós e os contras do estágio atual desse paradigma de desenvolvimento de processadores;

- Apresentou os resultados dos testes sob acSynth de um processador inteiramente novo desenvolvido em acRTL, com um comparativo entre o processador gerado automaticamente e um processador desenvolvido pelos métodos tradicionais, considerando os aspectos de área, *timing* e complexidade de projeto.

Como resultado destes trabalhos dois artigos científicos internacionais foram elaborados e submetidos.

1. *"An ArchC approach for automatic energy consumption characterization of processors"* [2], submetido a *Rapid Speed Prototyping Symposium*, IEEE, em colaboração com Rafael Auler, Ricardo Borin e Rodolfo Azevedo.

2. *"ESLBench: A benchmark suite for evaluating electronic system level tools and methodologies"* [3], submetido a *Embedded Systems Letters*, IEEE, tendo como autora principal Liana Duenha, em colaboração com Matheus Boy e Rodolfo Azevedo.

Um terceiro artigo está em processo, expandindo os resultados previamente apresentados em [2].

Como trabalhos futuros sugeridos para acSynth, podemos citar:

- Elaboração do mesmo fluxo de caracterização e simulação para as demais arquiteturas suportadas por ArchC, idealmente, com pelo menos um banco de dados de consumo de energia por arquitetura;

- Expandir acSim para simulações com precisão de ciclo, permitindo ao conjunto de ferramentas de análise de energia de acSynth considerar fatores como *stall*.
- Elaborar um modelo de cache funcional em ArchC, permitindo que seja possível simular o consumo de energia de acesso a memória. Através de PowerSC seria possível contabilizar este dado, e com uso de modelos apropriados, reduzir consideravelmente os erros nos dados de relatório de energia.
- Desenvolver um algoritmo mais moderno para contabilizar o consumo, considerando que o método de Tiwari foi proposto em 1994 e apresenta problemas conhecidos.
- Integrar acSynth a plataforma ARP, apresentando formas de análise de consumo para cada componente do SoC elaborado. É certo que haverão algoritmos distintos para cada componente, contudo, isto é previsto por PowerSC. Assim, a elaboração de relatórios de SoCs é factível.
- Aprimorar os relatórios gerados através da biblioteca PowerSC, criando novas classes dentro da biblioteca e corrigindo erros conceituais no texto atual quando há referência a processadores ArchC.
- Expandir a linguagem e criar uma ferramenta em ArchC para geração de *testbench* automático, o que reduziria consideravelmente o tempo de desenvolvimento das atividades aqui apresentadas.
- Elaborar métodos menos intrusivos de caracterização, reduzindo a complexidade deste procedimento.
- Aplicar a metodologia sobre fluxo de projeto *full-custom*, expandindo acSynth para além do âmbito das FPGAs.
- Solucionar os problemas relacionado ao acPowerGen para a geração de alguns tipos de códigos de teste, especialmente para *branches*, encontrando formas e novos algoritmos que permitam completa confiabilidade sobre os programas gerados. Idealmente, o projetista não deveria intervir em momento algum nesta etapa.

Como trabalhos futuros sugeridos para acRTL, podemos citar:

- Reformar o modelo dos *templates* no intuito de tornar o mais claro possível a separação entre código estático e os trechos que necessitam de intervenção do projetista.
- Explorar métodos de descrição que apresentem maior reuso de código. Um exemplo seria criar meios para derivar as funções padrões de ALU, de tal forma que não fosse

preciso repetir todo o código de tratamento destes casos em cada uma das intruções. Isso é factível, uma vez que instruções ALU recebem parâmetros idênticos e estão sujeitos as mesmas condições de *hazard*.

- Encontrar formas de tornar acRTL independente da ferramenta gsc, gerando *templates* mais legíveis para outras ferramentas acadêmicas ou comerciais.

- Buscar ganhos de eficiência de área e *timing* pois, apesar do caso ótimo não ser o objetivo aqui, os resultados atuais estão muito aquém do uso prático.

- Expandir o suporte para outros paradigmas de processadores, como suporte a *multi-core* tendo em vista a complexidade de sincronia de memória, ou ainda na elaboração de processadores vetoriais e DSPs.

- Utilizar informações comportamentais descritas em ArchC para a geração de métodos e funções. Isto poderia reduzir o tempo de engenharia do projetista. Idealmente, o *template* poderia apresentar sugestões de implementação para casos simples, como das operações aritméticas básicas.

Assim, espera-se que acSynth funcione como uma base para expandir o universo de possibilidades de exploração e desenvolvimento de arquitetura em ArchC, seja pelas seus resultados, por suas funções, ou pelas ideias aqui apresentadas. Quanto mais funcionalidades forem agregadas e integradas sob a mesma plataforma, permitindo a comunicação de diferentes trabalhos, algoritmos e resultados, mais ArchC se caracterizará como uma alternativa viável e robusta para acelerar o desenvolvimento de sistemas ASIC.

Referências Bibliográficas

[1] "International technology roadmap for semiconductors - design report," tech. rep., ITRS, 2011.

[2] M. Guedes, R. Auler, E. Borin, and R. Azevedo, "An archc approach for automatic energy consumption characterization of processors," in *Rapid System Prototyping (RSP), 2012*, 2012.

[3] L. Duenha, M. Boy, M. Guedes, and R. Azevedo, "Eslbench: A benchmark suite for evaluating electronic system level tools and methodologies," in *Embedded Systems Letters*, 2012.

[4] "International technology roadmap for semiconductors - design report," tech. rep., ITRS, 2007.

[5] S. Rigo, R. J. Azevedo, and G. Araujo, "The ArchC architecture description language," Tech. Rep. IC-03-15, Institute of Computing, University of Campinas, June 2003.

[6] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step towards software power minimization," *IEEE Transactions on VLSI Systems*, vol. 2, pp. 437–445, 1994.

[7] V. Tiwari, S. Malik, A. Wolfe, and M. T. chien Lee, "Instruction level power analysis and optimization of software," *Journal of VLSI Signal Processing*, vol. 13, pp. 1–18, 1996.

[8] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "Estimating android applications' cpu energy usage via bytecode profiling," in *First International Workshop on Green and Sustainable Software (GREENS), in conjunction with ICSE 2012*, June 2012.

[9] S. S. F. Goto and R. J. de Azevedo, "Síntese de linguagens de descrição de arquitetura," Master's thesis, Universidade Estadual de Campinas . Instituto de Computação, 2010.

[10] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, (New York, NY, USA), pp. 105–114, ACM, 2010.

[11] "A power monitor for android-based mobile platforms," Out 2012.

[12] "Monsoon power monitor."

[13] R. Azevedo, B. Albertini, and S. Rigo, "Arp: Um gerenciador de pacotes para sistemas embarcados com processadores modelados em archc," in *Workshop de Sistemas Embarcados - WSE*, SBC, 2010. In Portuguese.

[14] R. A. Walker and D. E. Thomas, "A model of design representation and synthesis," 1985. Las Vegas, Nevada, United States. ACM Press.

[15] V. Degalahal and T. Tuan, "Methodology for high level estimation of fpga power consumption," in *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, vol. 1, pp. 657 – 660 Vol. 1, jan. 2005.

[16] A. Stammermann, L. Kruse, W. Nebel, A. Pratsch, E. Schmidt, M. Schulte, and A. Schulz, "System level optimization and design space exploration for low power," in *Proceedings of the 14th international symposium on Systems synthesis*, ISSS '01, (New York, NY, USA), pp. 142–146, ACM, 2001.

[17] S. Pees, "Modeling embedded processors and generating fast simulators using the machine description language lisa," 2002.

[18] O. Schliebusch, A. Chattopadhyay, R. Leupers, G. Ascheid, H. Meyr, M. Steinert, G. Braun, and A. Nohl, "Rtl processor synthesis for architecture exploration and implementation," in *Proceedings of the conference on Design, automation and test in Europe - Volume 3*, DATE '04, (Washington, DC, USA), pp. 30156–, IEEE Computer Society, 2004.

[19] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "Mparm: Exploring the multi-processor soc design space with systemc," *The Journal of VLSI Signal Processing*, vol. 41, pp. 169–182, 2005. 10.1007/s11265-005-6648-1.

[20] M. Loghi, M. Poncino, and L. Benini, "Cycle-accurate power analysis for multiprocessor systems-on-a-chip," in *Proceedings of the 14th ACM Great Lakes symposium on VLSI*, GLSVLSI '04, (New York, NY, USA), pp. 410–406, ACM, 2004.

[21] P. Mishra and N. Dutt, "Architecture description languages for programmable embedded systems," in *In IEEE Proceedings on Computers and Digital Techniques*, pp. 285–297, 2005.

[22] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo, "Archc: a systemc-based architecture description language," in *16th Symposium on Computer Architecture and High Performance Computing, 2004 - SBAC-PAD 2004*, pp. 66 – 73, Oct. 2004. Best Paper Award.

[23] S. Rigo, R. J. Azevedo, and G. Araujo, "The ArchC architecture description language," Tech. Rep. IC-03-15, Institute of Computing, University of Campinas, June 2003.

[24] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros, "The archc architecture description language and tools," *International Journal of Parallel Programming*, vol. 33, pp. 453–484, 2005. 10.1007/s10766-005-7301-0.

[25] A. Baldassin, P. Centoducatte, and S. Rigo, "Extending the archc language for automatic generation of assemblers," *17th International Symposium on Computer Architecture and High Performance Computing. SBAC-PAD 2005.*, pp. 60 – 67, oct. 2005.

[26] A. Baldassin and P. Centoducatte, "Geração automatica de montadores em archc," Master's thesis, Universidade Estadual de Campinas . Instituto de Computação, 2005.

[27] R. Auler and P. Centoducatte, "Dotando ArchC com infraestrutura para geração de montadores e simuladores ARM," in *WSCAD-WIC '09: X Simpósio em Sistemas Computacionais - Workshop de Iniciação Científica*, 2009. In Portuguese.

[28] R. Auler and P. C. Centoducatte, "Geração automática de backend de compiladores baseada em adls," Master's thesis, Universidade Estadual de Campinas . Instituto de Computação, 2011.

[29] M. Abbaspour and J. Zhu, "Retargetable binary utilities," in *Proceedings of the 39th annual Design Automation Conference*, DAC '02, (New York, NY, USA), pp. 331–336, ACM, 2002.

[30] A. Baldassin, P. C. Centoducatte, S. Rigo, D. Casarotto, L. C. V. Santos, M. R. O. Schultz, and O. J. V. Furtado, "Automatic retargeting of binary utilities for embedded code generation," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pp. 253–258, IEEE Computer Society, 2007.

[31] J. T. H. Ma and R. J. de Azevedo, "Estimativa de consumo de energia em nível de instrução para processadores modelados em archc," Master's thesis, Outubro 2007.

[32] J. Ma and R. Azevedo, "Estimativa de consumo de energia em nivel de instrucao para processadores modelados em archc," in *Workshop de Sistemas Computacionais - WSCAD-SSC*, pp. 119–126, SBC, 2009. In Portuguese.

[33] F. V. Klein, G. Araújo, and R. J. de Azevedo, "Powersc: Uma extensão de systemc para a captura de atividade de transição," Abril 2005. Tese de Mestrado.

[34] F. Klein, R. Azevedo, and G. Araujo, "Enabling high-level switching activity estimation using systemc," Tech. Rep. IC-05-17, Institute of Computing, University of Campinas, August 2005.

[35] F. Klein, G. Araujo, and R. Azevedo, "Powersc: A systemc framework for power estimation." 6th NASCUG, San Jose, USA, Feb 2007.

[36] F. Klein, G. Araujo, R. Azevedo, R. Leao, and dos Luiz Santos, "An efficient framework for high-level power exploration," in *MWSCAS 2007: Proceedings of the 50th Midwest Symposium on Circuits and Systems*, pp. 1046–1049, Aug 2007.

[37] IEEE Computer Society Press, *IEEE Standard SystemC Language Reference Manual*, version 2.1 ed., Março 2006.

[38] F. Klein, R. Azevedo, and G. Araujo, "High-level switching activity prediction through sampled monitored simulation," pp. 161 –166, nov. 2005.

[39] F. V. Klein, G. Araújo, and R. J. de Azevedo, "Técnicas avançadas de modelagem, análise e otimização de potência em sistemas digitais," Outubro 2009. Tese de Doutorado.

[40] T. Gupta, C. Bertolini, O. Heron, N. Ventroux, T. Zimmer, and F. Marc, "High level power and energy exploration using archc," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*, pp. 25 –32, oct. 2010.

[41] L. Po and W. Ma, "A novel four-step search algorithm for fast block motion estimation," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 6, no. 3, pp. 313–317, 1996.

[42] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite,"

in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, (Washington, DC, USA), pp. 3–14, IEEE Computer Society, 2001.

[43] S. Rhoads, "Plasma-most mips i (tm) opcodes: overview," *Internet: http://opencores. org/project, plasma [May 2, 2012]*, 2006.

[44] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, (Washington, DC, USA), pp. 330–335, IEEE Computer Society, 1997.

[45] R. Alencar, S. Rigo, and R. Azevedo, "Software co-verification based on program traces from different processors," in *3rd Workshop on Infrastructures for Software/Hardware co-design - WISH*, 2011.

[46] A. Jiri Gaisler, "Aeroflex gaisler." http://www.gaisler.com/, Oct 2012.

[47] D. Patterson and J. Hennessy, *Computer organization and design: the hardware/-software interface*. Morgan Kaufmann, 2009.

[48] C. Hansen and T. Riordan, "Risc computer with unaligned reference handling and method for the same," Mar. 21 1989. US Patent 4,814,976.