UNIVERSIDADE FEDERAL DE GOIÁS
ESCOLA DE ENGENHARIA ELÉTRICA, MECÂNICA E DE COMPUTAÇÃO

BEATRIZ CARVALHO DE BARROS DO VALE ROCHELLE

Implementação reduzida de MIPS-32 monociclo em SystemVerilog

Goiânia
2024

UNIVERSIDADE FEDERAL DE GOIÁS
ESCOLA DE ENGENHARIA ELÉTRICA, MECÂNICA E DE COMPUTAÇÃO

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO PARA DISPONIBILIZAR VERSÕES ELETRÔNICAS DE TRABALHO DE CONCLUSÃO DE CURSO DE GRADUAÇÃO NO REPOSITÓRIO INSTITUCIONAL DA UFG

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio do Repositório Institucional (RI/UFG), regulamentado pela Resolução CEPEC no 1240/2014, sem ressarcimento dos direitos autorais, de acordo com a Lei no 9.610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou download, a título de divulgação da produção científica brasileira, a partir desta data.

O conteúdo dos Trabalhos de Conclusão dos Cursos de Graduação disponibilizado no RI/UFG é de responsabilidade exclusiva dos autores. Ao encaminhar(em) o produto final, o(s) autor(a)(es)(as) e o(a) orientador(a) firmam o compromisso de que o trabalho não contém nenhuma violação de quaisquer direitos autorais ou outro direito de terceiros.

1. Identificação do Trabalho de Conclusão de Curso de Graduação (TCCG)

Nome(s) completo(s) do(a)(s) autor(a)(es)(as): Beatriz Carvalho de Barros do Vale Rochelle

Título do trabalho: Implementação reduzida de MIPS-32 monociclo em SystemVerilog

2. Informações de acesso ao documento (este campo deve ser preenchido pelo orientador) Concorda com a liberação total do documento [x] SIM [ ] NÃO[1]

[1] Neste caso o documento será embargado por até um ano a partir da data de defesa. Após esse período, a possível disponibilização ocorrerá apenas mediante: a) consulta ao(à)(s) autor(a)(es)(as) e ao(à) orientador(a); b) novo Termo de Ciência e de Autorização (TECA) assinado e inserido no arquivo do TCCG. O documento não será disponibilizado durante o período de embargo.

Casos de embargo:

- Solicitação de registro de patente;
- Submissão de artigo em revista científica;
- Publicação como capítulo de livro.

Obs.: Este termo deve ser assinado no SEI pelo orientador e pelo autor.

Documento assinado eletronicamente por **Tomas Antonio Costa Badan**, **Professor do Magistério Superior**, em 05/02/2024, às 14:42, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

Documento assinado eletronicamente por **Beatriz Carvalho De Barros Do Vale Rochelle**, **Discente**, em 05/02/2024, às 16:21, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

A autenticidade deste documento pode ser conferida no site https://sei.ufg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **4321732** e o código CRC **DD31EB56**.

Referência: Processo nº 23070.057896/2023-75 SEI nº 4321732

BEATRIZ CARVALHO DE BARROS DO VALE ROCHELLE

Implementação reduzida de MIPS-32 monociclo em SystemVerilog

Trabalho de Conclusão de Curso apresentado à banca examinadora da Escola de Engenharia Elétrica, Mecânica e de Computação da Universidade Federal de Goiás como parte dos requisitos para conclusão do Bacharelado em Engenharia de Computação.

Orientador: Prof. Dr. Tomás Antônio Costa Badan

Goiânia
2024

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFG.

ROCHELLE, Beatriz Carvalho de Barros do Vale
Implementação reduzida de MIPS-32 monociclo em SystemVerilog [manuscrito] / Beatriz Carvalho de Barros do Vale ROCHELLE. - 2024.
10 f.: il.

Orientador: Prof. Tomás Antônio Costa Badan.
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Goiás, Escola de Engenharia Elétrica, Mecânica e de Computação (EMC), Engenharia da Computação, Goiânia, 2024.
Bibliografia. Anexos.
Inclui tabelas.

1. Processadores. 2. MIPS. 3. Arquitetura de computadores. 4. Circuitos lógicos. 5. Linguagem de descrição de hardware. I. Badan, Tomás Antônio Costa, orient. II. Título.

CDU 621.3

UNIVERSIDADE FEDERAL DE GOIÁS
ESCOLA DE ENGENHARIA ELÉTRICA, MECÂNICA E DE COMPUTAÇÃO

ATA DE DEFESA DE TRABALHO DE CONCLUSÃO DE CURSO

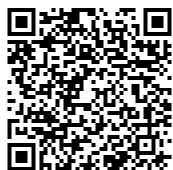
Ao(s) quinto (5º) dia(s) do mês de fevereiro do ano de 2024 iniciou-se a sessão pública de defesa do Trabalho de Conclusão de Curso (TCC) intitulado "Implementação reduzida de MIPS-32 monociclo em SystemVerilog", de autoria de Beatriz Carvalho de Barros do Vale Rochelle, do curso de Engenharia de Computação, do(a) Escola de Engenharia Elétrica, Mecânica e de Computação (EMC) da UFG. Os trabalhos foram instalados pelo(a) professor Dr. Tomás Antônio Costa Badan – orientador(a) (EMC) com a participação dos demais membros da Banca Examinadora: professor Dr. Carlos Galvão Pinheiro Júnior (EMC) e professor Dr. Sérgio Pires Pimentel (EMC). Após a apresentação, a banca examinadora realizou a arguição do(a) estudante. Posteriormente, de forma reservada, a Banca Examinadora atribuiu a nota final de 9.0 (nove) , tendo sido o TCC considerado aprovado.

Proclamados os resultados, os trabalhos foram encerrados e, para constar, lavrou-se a presente ata que segue assinada pelos Membros da Banca Examinadora.

Documento assinado eletronicamente por **Tomas Antonio Costa Badan**, **Professor do Magistério Superior**, em 05/02/2024, às 14:37, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

Documento assinado eletronicamente por **Sergio Pires Pimentel**, **Professora do Magistério Superior**, em 05/02/2024, às 16:08, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .

Documento assinado eletronicamente por **Carlos Galvao Pinheiro Junior**, **Professor do Magistério Superior**, em 05/02/2024, às 16:15, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020 .


A autenticidade deste documento pode ser conferida no site https://sei.ufg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **4321737** e o código CRC **6D1BD1DD**.

Referência: Processo nº 23070.057896/2023-75 SEI nº 4321737

Implementação reduzida de MIPS-32 monociclo em SystemVerilog

Beatriz Carvalho de Barros do Vale Rochelle[1], graduanda em Engenharia de Computação. [1]EMC/UFG
Tomás Antônio Costa Badan[2], Professor Adjunto. [2]EMC/UFG E-mails: oblongata@egresso.ufg.br[1], tbadan@ufg.br[2]

***Resumo*—Este artigo apresenta uma implementação do núcleo principal de um processador de arquitetura derivada de MIPS 32, com conjunto reduzido de instruções e organização monociclo. Ele não inclui instruções privilegiadas e funcionalidades como suporte a interrupções e operações de E/S. Os módulos de memória foram substituídos por simulacros para execução dos testes de integração. Os testes foram escritos em Python e realizados utilizando o *framework cocotb*, devido à sua facilidade de uso. A arquitetura foi implementada utilizando a linguagem SystemVerilog e compilada tanto com o *Icarus Verilog* quanto com o *Verilator*. Apesar de o projeto não ter sido sintetizado, as simulações apresentaram funcionamento correto e resultados acurados.**

***Palavras-chave*—arquitetura de computadores, circuitos lógicos, linguagem de descrição de *hardware*, processadores, MIPS, *design* lógico e *design* de *hardware*.**

***Abstract*—This paper presents an implementation of the basic processing unit of a processor derived from MIPS 32, with a reduced instruction set and single-cycle design. It does not include privileged instructions and functionalities like interrupt handling and I/O support. The memory modules were substituted for mock units for the execution of the integration tests. The tests were coded in Python and executed using the framework *cocotb*, due to its ease of use. The architecture was implemented using the language SystemVerilog and compiled using both *Icarus Verilog* and *Verilator*. Despite the project not having been synthesized, the simulations demonstrated proper functioning and produced accurate results.**

***Index Terms*—computer architectures, logic circuits, hardware description language, processors, mips, logical design, hardware design.**

1. INTRODUÇÃO

Feche os olhos por um momento. Imagine o seguinte: Ada Lovelace, filha de Lorde Byron programando junto com Charles Babbage o que hoje chamamos de "o primeiro computador" [1]. Os *mainframes*, que tomavam salas inteiras. Programas em cartões, *jobs* em *batches*, fitas magnéticas. O começo do uso de CIs em computadores. Os primeiros telefones "portáteis"de 1 kg, considerados "peso-pena" nos anos 70. UNIX e seus *spin-offs* [2]. A popularização dos PCs. O "computador da casa". A conexão discada (e mais barata de madrugada!). O *bug* do milênio, o frenesi dos anos 2000. Os primeiros *smartphones*. Wi-Fi em quase lugar nenhum, aos poucos passando à onipresença. Automação, industrial e residencial. A integração transparente de sistemas embarcados. *Desktops*, *notebooks* e celulares em todos os lugares.

Agora, imagine que tudo isso aconteceu em um piscar de olhos. E se eu te dissesse que aconteceu mesmo? Se trocarmos de escala e considerarmos o tempo que passou entre a formação da Terra e agora como 24 horas, os seres humanos existem há um segundo [3]. Segue-se, logicamente, que o tempo que passou entre Babbage e agora foi isso - uma fração de segundo.

O que tudo isso tem em comum? Em cada um dos cenários, é necessária alguma forma de processamento. E sem projeto de circuitos digitais, não há processadores. Pelo menos não como os fazemos desde que começamos a usar circuitos integrados em computadores, em 1964. Desde que empresas como a IBM lançaram os primeiros *mainframes* que usavam *chips*, nunca mais voltamos atrás na produção em larga escala [2].

Na UFG, os alunos são imersos no mundo dos semicondutores em disciplinas como Materiais Elétricos (que toca no aspecto físico desses materiais, como o funcionamento interno de um transistor a nível molecular, por exemplo), Circuitos Lógicos (que apresenta as portas lógicas, como elas se comportam e os tipos de circuitos que podem constituir, como um circuito somador - assim como todos os outros componentes do processador). Além disso, em Arquitetura de Computadores, eles são instruídos sobre como selecionar, configurar e integrar esses componentes para construir sistemas mais complexos, como o núcleo básico de um processador - o que foi feito aqui, e que está detalhado adiante.

O projeto apresentado a seguir almeja a construção do núcleo básico de um processador de arquitetura derivada do MIPS 32, com conjunto reduzido de instruções e organização monociclo. Não estão inclusas, por exemplo, instruções privilegiadas, interrupções e operações de E/S. Os detalhes a respeito dos métodos empregados, da fase de testes e dos resultados obtidos serão abordados nas seções seguintes. Os trabalhos relacionados serão apresentados e em seguida, a fundamentação teórica do projeto e as ferramentas utilizadas. Serão descritas a implementação e a criação dos testes e, por fim, a conclusão do trabalho.

A razão da escolha do tema se deve (além de preferência pessoal) a diversos fatores. O projeto facilita e democratiza o entendimento da arquitetura de processadores e pode servir como base para novas implementações e pesquisas. A escolha de uma arquitetura aberta permite adaptações e modificações, além do desenvolvimento de processadores mais eficientes e possível redução do consumo energético. Existem diversas arquiteturas, algumas delas bastante célebres [4]. Poderia-se ter optado por alguma certamente mais presente na indústria, mas foi escolhido o desenvolvimento de um projeto em MIPS. Ela é uma arquitetura simples e clara, tem conjunto de instruções reduzido e licença aberta, além de ser bastante estudada e documentada [5, 6]. A abundância de recursos a respeito facilita bastante o processo de desenvolvimento e aprendizado.

2. Trabalhos Relacionados

No GitHub, é possível encontrar vários projetos de implementação de processadores MIPS 32 *bits*, dos quais seis foram selecionados. Eles serão apresentados e diferenciados em relação as seguintes especificidades: existência de memória *cache*, suporte a E/S e interrupções e separação das unidades de memória de dados e instruções (como definido em Arquiteturas Harvard [7]). Entre os projetos selecionados, existem três organizações monociclo, uma multiciclo e duas *pipeline*. Todos foram escritos em Verilog.

As implementações nomeadas MIPS-in-Verilog [8], Verilog-MIPSProcessor [9] e 32-bit-MIPS-Processor [10] têm as seguintes características em comum: são organizações monociclo, não possuem suporte a E/S, não lidam com interrupções e as unidades de memória são simuladas internamente ao processador. O projeto 32-bit-Multicycle-CPU [11], apesar de ser multiciclo, não apresenta qualquer uma das especificidades listadas. Dentre as implementações em *pipeline*, MIPS [12] apresenta memória *cache* a mais. A implementação mips-cpu [13] é um pouco mais sofisticada: além de *cache*, inclui as unidades de memória por completo. A Tabela I mostra as caracteristicas apresentadas de maneira condensada.

3. Referencial teórico

Nesta seção, está apresentada de maneira resumida a teoria usada para o desenvolvimento do projeto. São discutidas brevemente arquiteturas e organizações. Em especial, a organização dessa arquitetura no formato monociclo, seus elementos e como os seus tipos de instruções são decodificados.

3.1 Arquiteturas e Organizações

A arquitetura de um processador define não apenas o tamanho dos registradores como também a quantidade e o propósito deles, tanto de uso geral quanto de uso específico. Define também sua codificação nos formatos numérico e alfanumérico. Além disso, determina o conjunto de instruções, incluindo suas codificações e como elas funcionam.

A organização implementa a arquitetura. Ela define quais componentes[1] são utilizados, suas interfaces e a forma como eles se comunicam [14]. O desempenho e a eficiência do funcionamento de um processador estão diretamente atrelados à sua organização. Entre as formas de organização, pode-se citar monociclo, multiciclo e *pipeline*. Por estarem fora do escopo deste trabalho, as organizações multiciclo e pipeline não serão tratadas adiante.

Independentemente do tipo de organização, toda e qualquer instrução envolve três etapas: busca, decodificação e execução. A parte de execução pode ou não incluir as seguintes etapas: `memory` (quando é necessário acesso à memória de dados) e `write back`, quando há escrita em registradores dentro do RegFile (banco de registradores).

De modo geral, uma instrução pode ser vista como uma série de etapas. Cada etapa é executada dentro de um dos componentes primários (ULA, RegFile, memórias e controle). Idealmente, todas gastam o mesmo tempo de execução.

Organizações monociclo executam, necessariamente, uma instrução por ciclo. Para que isso seja possível, o tempo de ciclo (que não muda) deve acomodar a execução completa de qualquer instrução. Isto é: o tempo do ciclo equivale ao tempo que a instrução mais demorada leva para ser executada. Esse tipo de arquitetura acaba tendendo a ser considerado ineficiente [15], pois a execução de instruções mais curtas (compostas por poucas etapas) envolve tempo ocioso da CPU. Isso acontece porque ela é calibrada para acomodar as instruções mais longas do conjunto (que necessitam do maior número de etapas possível para serem executadas).

3.2 Instruções

	opcode	rs	rt	rd	shamt	funct
R	31–26	25–21	20–16	15–11	10–6	5–0
I	opcode (31–26)	rs (25–21)	rt (20–16)	immediate (15–0)		
J	opcode (31–26)	address (25–0)				

Fig. 1. Formatos das instruções dos tipos R, I e J

3.2.1 Tipo R: Esse tipo de instrução requer três registradores: dois deles são utilizados para operação de leitura e o terceiro para escrita. O formato da instrução pode ser visto na Figura 1. Nele pode-se observar os campos `opcode`, os dois registradores de leitura, o registrador de escrita, o campo `funct`, o campo `shamt` (do qual não é falado aqui), e os seus respectivos espaços de endereçamento. Observe que para todas as instruções do tipo R, o `opcode` é `zero` e a operação é determinada pelo campo `funct`. A Figura 2 mostra como instruções desse tipo foram implementadas. Note que o caminho de dados utilizado está em destaque (negrito).

3.2.2 Tipo I: O início da execução das instruções desse tipo é similar às do tipo R, a diferença entre elas é que no tipo I, um dos valores lidos é um valor imediato. O seu formato, apresentado na Figura 1, contém quatro campos: o `opcode` e o `rs` (na mesma posição de instruções do tipo R), o `rt` (que pode ser de leitura ou escrita, a depender do tipo de instrução) e, finalmente, os 16 *bits* inferiores, que determinam um valor imediato. Note que devido à quantidade de *bits* que compõem o valor imediato, ele pode assumir valores limitados a um conjunto de 65,536 possibilidades.

3.2.3 Tipo J: A especificação MIPS inclui duas instruções que são codificadas como tipo J: `jump` e `jump and link`. Ambas implementam a forma de endereçamento absoluto[2]. Este projeto inclui a implementação da instrução `jump`. A Figura 1 mostra o formato da instrução do tipo J. Perceba que o campo `opcode` tem a mesma posição que nos formatos anteriores. O restante da instrução contém parte do endereço da próxima instrução a ser executada. Devido à diferença entre o tamanho desse campo (26 *bits*) e o tamanho do registrador *Program Counter* (PC), o valor a ser armazenado em PC precisa ser calculado. O cálculo é feito através da expressão 1.

[1] Em circuitos lógicos.

[2] Independente da posição atual do ponteiro PC.

Nome	Monociclo	Multiciclo	*Pipeline*	Memória Real	Interrupções	E/S	Cache
MIPS-IN-VERILOG	✓	x	x	x	x	x	x
Verilog-MIPSProcessor	✓	x	x	x	x	x	x
32-bit-MIPS-Processor	✓	x	x	x	x	x	x
32-bit-Multicycle-CPU	x	✓	x	x	x	x	x
MIPS	x	x	✓	x	x	x	✓
mips-cpu	x	x	✓	✓	x	x	✓

Tabela I. Projetos selecionados

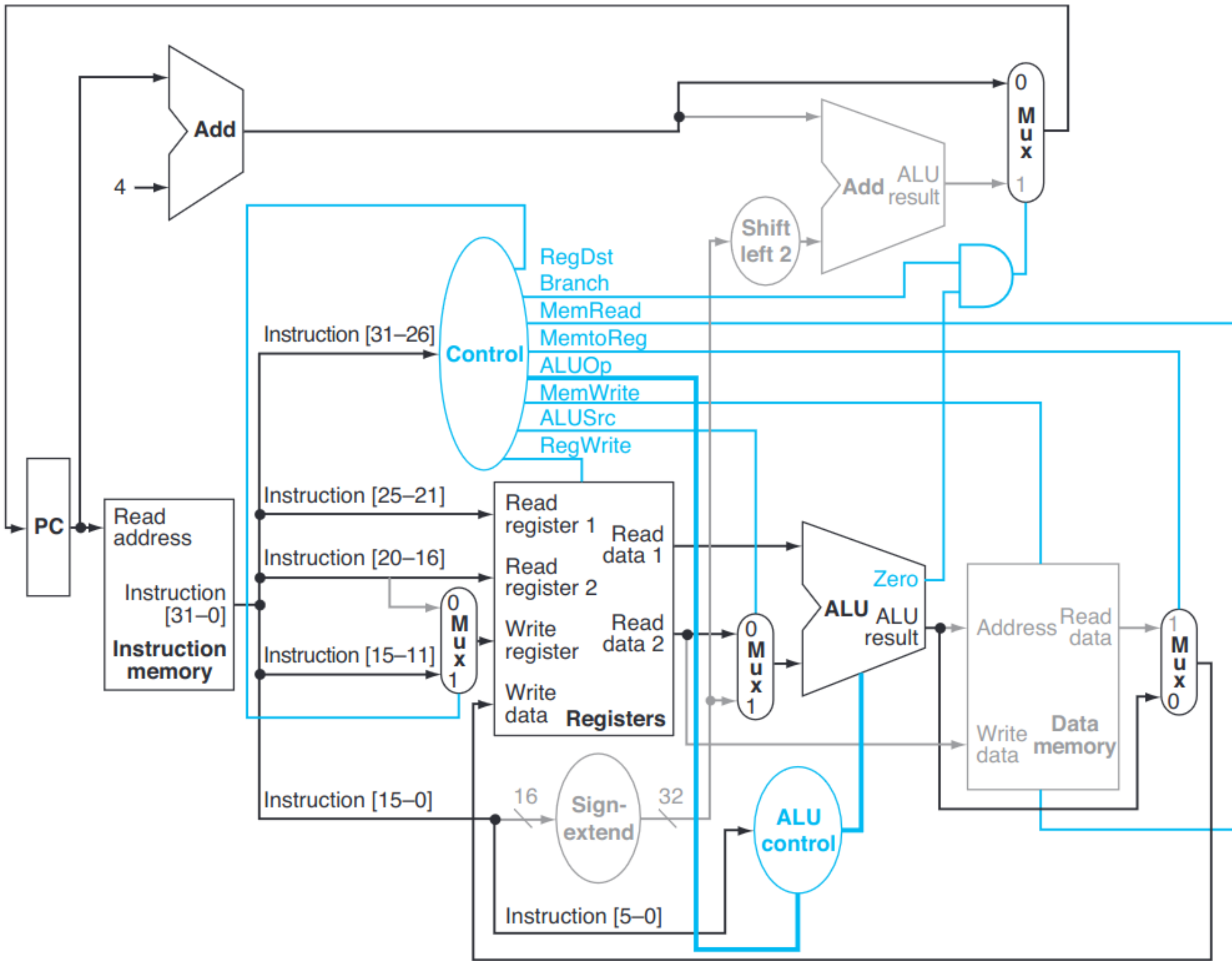


Fig. 2. Caminho de dados para instruções do tipo R em destaque

$$PC = \{(PC + 4)[31 : 28], address, 00)\} \qquad (1)$$

Na expressão, é possível identificar a concatenação de três valores: os 4 *bits* mais significativos do valor atual de PC acrescido de 4, o endereço fornecido pela instrução e dois *bits* zero.

4. Metodologia

Nesta seção está apresentada a metodologia utilizada para a implementação do projeto. O desenvolvimento se deu em iterações sucessivas do ciclo codificar-testar-debugar, não necessariamente nessa ordem.

Antes de explicar o processo em detalhes, convém apresentar uma visão geral da estrutura do projeto, assim como as ferramentas usadas para construí-lo.

4.1 Ferramentas

A programação dos componentes e testes foi feita no *Visual Studio Code* [16], um editor de texto de código aberto da Microsoft conhecido por ser leve e rápido. As possibilidades de personalização são um atrativo, pois oferece uma abundante seleção de extensões. Além disso, ele tem suporte a controle de versão, depuração e colaboração. Sua interface é limpa e adaptável às necessidades do usuário.

O projeto foi codificado em SystemVerilog. Devido à diferença considerável entre a descrição de *hardware* e programação de *software* em geral, foi necessário um tempo de adaptação com SystemVerilog. Para isso, foi utilizado o site *HDL bits* [17], uma plataforma educacional projetada para ajudar estudantes e entusiastas a aprender e praticar Verilog. O site tem uma interface interativa e oferece exercícios e desafios em linguagens como Verilog e VHDL, focados no desenvolvimento de circuitos digitais.

Um recurso que poderia ter sido utilizado para treinamento em SystemVerilog é o *ChipVerify* [18], uma plataforma educacional focada em *design* de *hardware* digital e verificação de circuitos integrados em Verilog. Oferece tutoriais, exemplos práticos e recursos educativos, abrangendo uma diversa gama de tópicos variando em complexidade, de conceitos básicos a técnicas avançadas.

Existem diversas ferramentas que podem ser utilizadas para compilação de código em SystemVerilog. Neste projeto foram utilizados *Icarus Verilog* [19] e *Verilator* [20].

O código foi compilado com maior frequência com o *Icarus Verilog*, uma ferramenta destinada à compilação de código em Verilog e um subconjunto de SystemVerilog. Tem suporte a uma grande quantidade de funcionalidades e é conhecido por ser flexível e fácil de usar. É uma escolha popular na comunidade de *design* de *hardware* por ter código aberto, possibilitando personalização e contribuição dos usuários.

Algumas compilações foram feitas com o *Verilator*, que foi capaz de encontrar erros que não haviam sido identificados com o *Icarus*. *Verilator* é uma ferramenta de código aberto voltada para a simulação rápida de projetos em Verilog, graças à geração de código C++ a partir do código Verilog, que resulta em simulações mais rápidas. Além disso, o *Verilator* é escalável e capaz de suportar projetos complexos.

Para os testes, poderia ter sido utilizado o método de referência na indústria disponível no site *ChipVerify*. Por não haver necessidade de se seguir o padrão da indústria à risca, uma vez que este trabalho se encontra no meio acadêmico, optou-se por recorrer ao *framework cocotb* [21]: comparado a codificação de testes em SystemVerilog, sua curva de aprendizagem e simplicidade são bastante atraentes. Ao contrário de SystemVerilog, que é tradicionalmente utilizada para verificação, o *cocotb* oferece uma alternativa baseada em Python. Graças à utilização de corrotinas (que simplificam a gestão do fluxo de controle), ele facilita a criação de *testbenches* concisos para verificação de circuitos.

Como ferramenta para auxílio no *debugging* foi escolhido o *GTKWave* [22], que permite verificar as formas de onda dos sinais ao longo dos ciclos de relógio. É uma ferramenta de código aberto, com suporte para múltiplos formatos de arquivo, como VCD (*Value Change Dump*) e FST (*Fast Signal Trace*). Possui uma interface gráfica intuitiva, com recursos como *zoom*, busca e filtros.

Decidiu-se usar um programa que calcula a sequência de Fibonacci para o teste de integração (verificação do funcionamento dos componentes trabalhando em conjunto). Ele foi escrito em C e depois convertido para Assembly usando o *MARS* [23] *(MIPS Assembler and Runtime Simulator)*. Trata-se de um ambiente de desenvolvimento criado para ser usado na implementação de projetos com arquitetura MIPS como apresentada por Patterson e Hennessy [15]. Ele inclui um montador, um simulador de tempo de execução e uma interface de depuração para programação em linguagem Assembly MIPS.

4.2 *Estrutura*

O projeto é composto por dois diretórios principais: um diretório `/src`, contendo o código em SystemVerilog e um diretório `/tst`, com subdiretórios de teste específicos para cada diretório, em Python.

Foi criado um *ambiente virtual* chamado `venv`, um diretório com instalações Python e *cocotb* isoladas e independentes do exterior. Dessa maneira, caso ocorram instalações ou atualizações de pacotes relacionados na máquina hospedeira, o ambiente virtual e o projeto permanecem isolados. O diretório `venv` é populado com conteúdo relacionado à implementação do ambiente virtual, portanto não foi descrito em mais detalhes.[3]

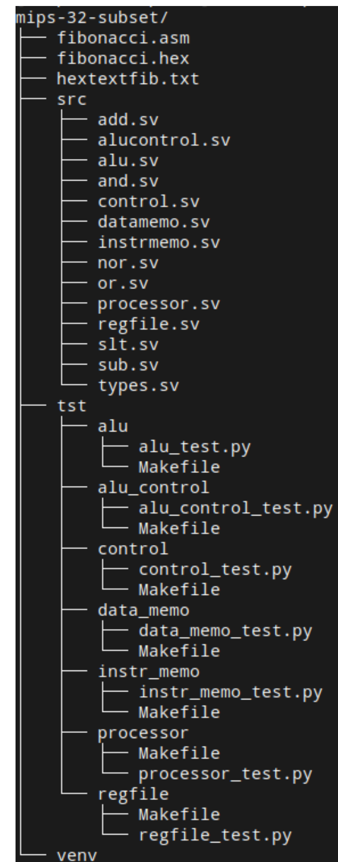


Fig. 3. Estrutura dos diretórios

4.3 *Implementação*

Agora que estabeleceu-se a estrutura do projeto, seus fundamentos e as ferramentas que ele usa, é hora de apresentar os detalhes de sua implementação. A produção dos componentes foi altamente modular: eles foram construídos e testados um a um, separadamente. Somente após a certeza de que todos funcionavam corretamente, o arquivo `processor.sv` foi

[3] Devido à extensão do seu conteúdo, ele foi omitido na Figura 3.

criado, onde todos foram instanciados e interligados. A seguir estão descritos cada um dos módulos que o compõem.

A ULA recebe dois operandos de 32 *bits* e um código de 4 *bits* chamado operation, que indica a operação que deve ser executada. Munida dessas informações, ela calcula o resultado e o exibe na saída imediatamente. Caso o resultado seja zero, ela ativa a *flag* `zero`. Ela trabalha assincronamente: está sempre calculando, independente dos sinais de *clock*. Ela é composta de vários submódulos, que implementam as seguintes operações: `AND`, `OR`, `NOR`, `ADD`, `SUB`, `SLT`, `SLTU` [4]. A Figura 4 representa sua interface.

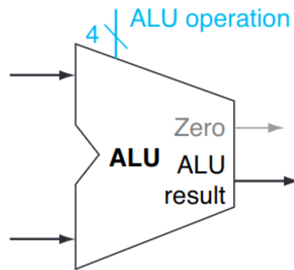


Fig. 4. Interface da ULA

RegFile (do inglês, *Register File*) é um conjunto de 32 registradores com 32 *bits* cada. Todos eles permitem leitura e escrita, com exceção do registrador `zero`, que sempre contém o valor 0 e apesar de aceitar entrada de valores para escrita, não os armazena. Caso haja tentativa de sobrescrita, o RegFile não registrará a mudança.

Trata-se da memória *interna* do processador, e o acesso a ele é muito mais rápido do que as unidades de memória adicionais. O acesso precisa ser de fato mais rápido, pois dentre as informações que armazena, as que são geradas durante o processamento são imediatamente necessárias. A convenção de uso sugerida pela arquitetura está apresentada na Figura 5.

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

Fig. 5. Convenção de uso dos registradores em MIPS

Toda leitura é assíncrona e toda escrita síncrona. Isso significa que um registrador pode ser consultado a qualquer momento, ao passo que depende dos ciclos de *clock* para receber valores novos. Para leitura é necessário fornecer os endereços de leitura, como mostra a Figura 6. Os valores lidos ficam disponíveis nas saídas `read data 1 e 2` imediatamente. Para escrita, é preciso fornecer o endereço do registrador que vai ter seu valor sobrescrito e, claro, habilitar a escrita com o sinal RegWrite. A escrita só vai surtir efeito no próximo ciclo de *clock*.

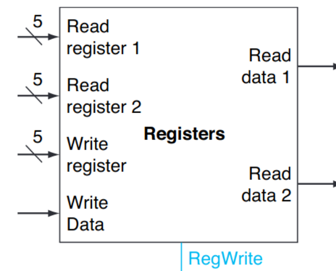


Fig. 6. Interface do RegFile

O controle é subdividido em dois módulos: o primeiro é denominado `control`, que recebe o `opcode` da instrução. A partir dela, ele identifica o tipo de instrução (I, R ou J), decodifica o que cada campo contém e ativa as *linhas de controle* correspondentes. Esses sinais são roteados para diversos outros componentes e determinam como eles funcionam (por exemplo: a escrita na memória de dados só é permitida se `memWrite` estiver ativo).

Caso a instrução seja do tipo R, o controle principal delega para o módulo `ALUcontrol` (controle da ULA), a configuração correta da ULA. `ALUcontrol` recebe o campo `funct` extraído da instrução, que se refere ao tipo de operação requisitada. Após computá-lo, transmite essa informação à ULA através do campo `operation`. A Figura 7 mostra onde cada sinal de controle é inserido, com exceção do sinal de `jump`.

Utilizada por operações que necessitam de um espaço para armazenamento volátil de informações, a memória de dados recebe um endereço de leitura, busca o valor contido nele e o retorna em seguida. Nessa implementação, apesar de depender do sinal `memRead` para ser habilitada, essa operação de leitura ocorre independentemente do relógio. Além disso, a memória de dados permite também escrita, porém isso só ocorre nas bordas positivas do sinal de *clock*, se e somente se o sinal de controle `memWrite` estiver ativo. Nesse caso, ela recebe o valor a ser armazenado, assim como o endereço onde ele deve ser guardado. O valor escrito pode ser lido no próximo ciclo de relógio. A Figura 8 apresenta a interface utilizada nessa memória.

A memória de instruções pode ser visualizada como um subcaso da memória de dados, em que a escrita é desabilitada. Constituída por um conjunto de registradores de 8 *bits*, ela é destinada ao armazenamento das instruções que o processador executa. Usualmente, as memórias são endereçadas em *bytes*, mas no caso da arquitetura MIPS, devido ao fato de o tamanho das *words* ser 32 *bits*, é necessário concatenar *chunks* de 8 *bits* para formar cada instrução antes de apresentá-la na interface de saída[5], pois todos os outros componentes trabalham com *words* de 32 *bits*. A memória de instruções é um módulo que permite apenas leitura, portanto funciona assincronamente. Ela recebe o endereço de leitura, busca a instrução no endereço solicitado e a disponibiliza na saída imediatamente.

[4]Unsigned.

[5]Problema de alinhamento de memória.

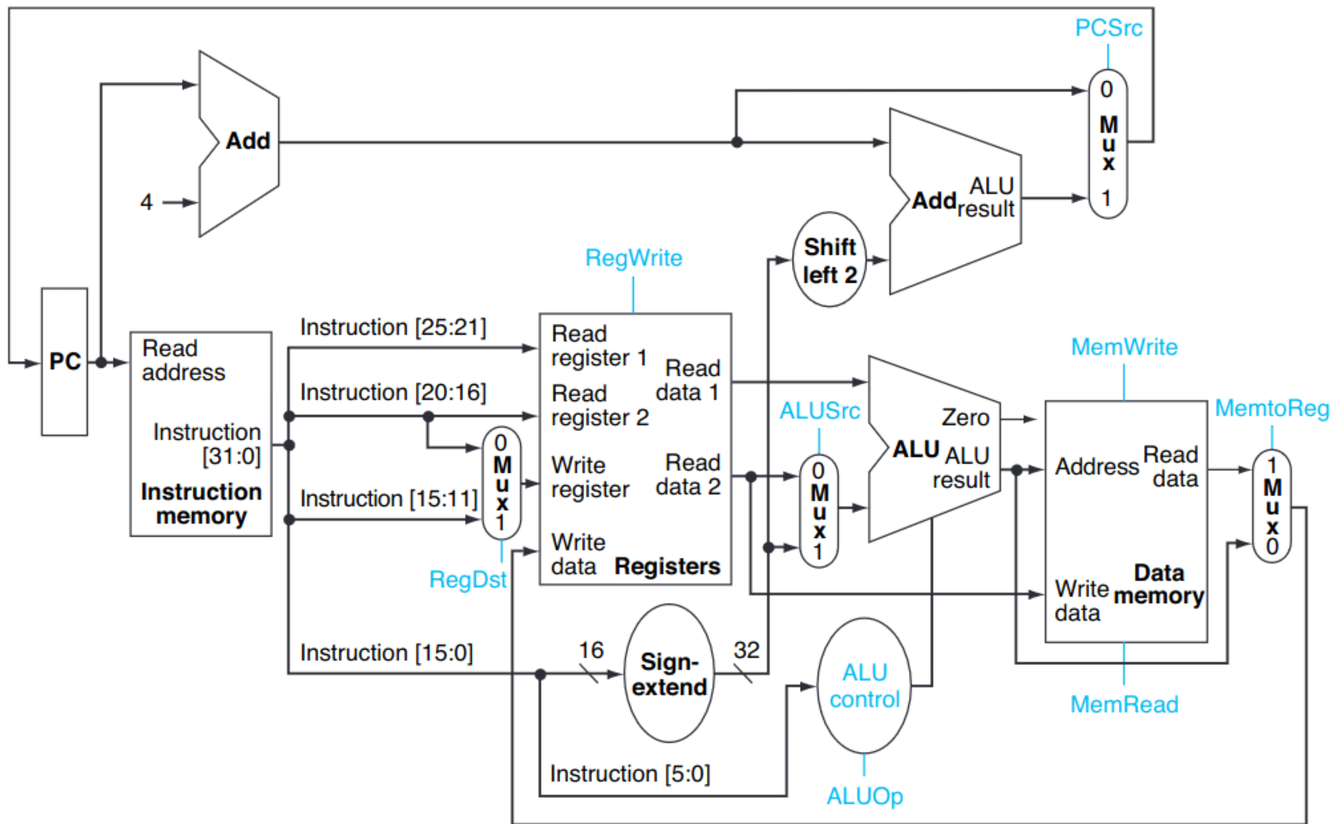


Fig. 7. Caminho de dados com linhas de controle em destaque

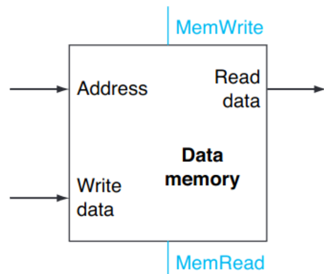


Fig. 8. Interface da Memória de dados

O processador é um módulo hierarquicamente acima dos outros, no qual são instanciados e interconectados. Conforme apresentado na Figura 2, para completar o núcleo principal do processador, além de instanciar os módulos ULA, RegFile, as unidades de memória e a unidade de controle, é necessário também adicionar outros componentes, como extensor de sinal, multiplexadores e somadores.

4.3.1 Implementando Jumps: Conforme explicado anteriormente, `jump` altera o valor de PC conforme a expressão 1 apresentada na seção 3. Esse caminho de dados pode ser observado na parte superior da Figura 9.

4.4 Instruções Agregadas ao Núcleo Básico

A seguir, estão listadas as instruções que foram adicionadas posteriormente como adições ao núcleo principal[6]. É interessante notar que o arquivo `types.sv` foi preenchido com todos os códigos de `opcode` e do campo `funct` previstos na referência MIPS [24]. Isso facilita consideravelmente a adição de novas instruções, que exige apenas alterações pontuais nos módulos de controle.

4.4.1 ADDI - ADD Immediate: O comportamento da instrução `ADDI` é semelhante à instrução `ADD`, mas é uma instrução do tipo I. Isso significa que a maior parte do caminho de dados já implementado para a instrução `ADD` também pode ser utilizado para execução do `ADDI`. A diferença entre as duas é que, em vez da soma do conteúdo de dois registradores, a soma é feita entre o conteúdo de *um* registrador e uma constante, proveniente dos 16 *bits* menos significativos da instrução. Esses dois valores são transmitidos à ULA, juntamente com o código que força a ULA a ser um módulo somador (proveniente de `ALU Control`). Os sinais de controle são exatamente iguais aos da instrução `ADD`, com exceção de `aluSrc` e `aluOp`.

4.4.2 BNE - Branch on Not Equal: Essa instrução foi adicionada para facilitar a escrita e a execução do programa `fibonacci.asm`. Para implementá-la, aproveitou-se a implementação da instrução `BEQ`, notando-se que a instrução `BNE`

[6] Necessário para a execução de programas simples.

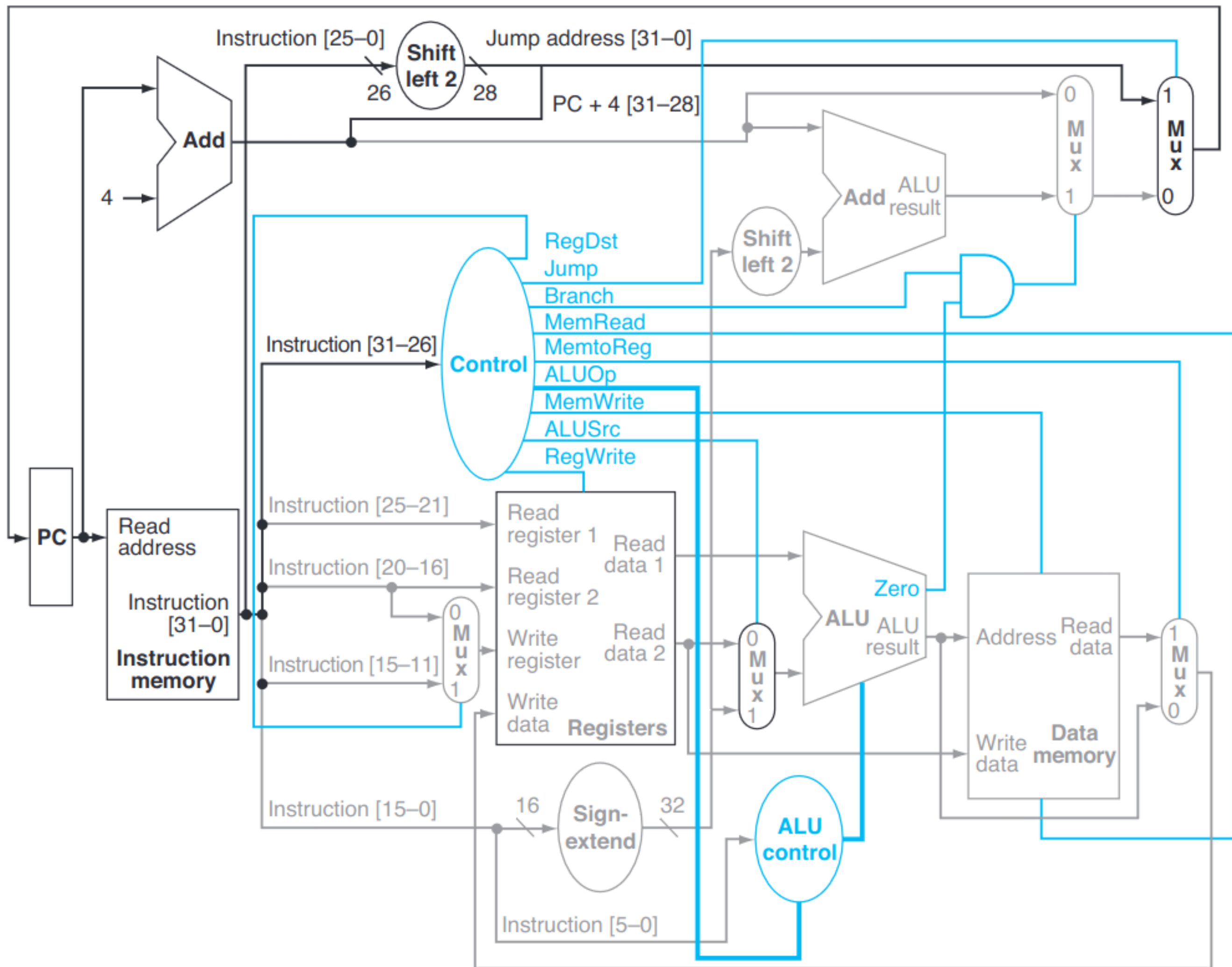


Fig. 9. Caminho de dados da instrução *jump* em destaque

é apenas a negação do resultado do teste feito na execução da instrução BEQ.

5. TESTES

Não houve um módulo sequer que tenha sido codificado sem que o componente anterior tivesse sido testado satisfatoriamente. Na verdade, desde sua concepção, todos os módulos foram construídos tendo em vista a fase de testes. Isso se deve ao fato de ter sido adotada uma abordagem de desenvolvimento guiada por testes (TDD, ou *Test Driven Development*, em inglês).

Essa abordagem tem uma vantagem no contexto deste projeto: devido à alta interdependência dos componentes, se os testes unitários[7] houverem sido suficientemente rigorosos, quando os módulos forem conectados, os novos erros provavelmente estarão no escopo do módulo do processador.

No *framework cocotb* os testes foram estuturados da seguinte forma: um diretório contendo um arquivo de teste em Python e um arquivo de configuração e execução no formato *makefile* [25]. A estrutura do diretório de testes pode ser visualizada também na Figura 3.

A metodologia de testes no *cocotb* pode ser descrita da seguinte forma: primeiro, instancia-se o módulo a ser testado, que é comumente chamado de dut (*Device Under Test*, ou dispositivo sob teste). O próximo passo consiste em configurar o teste a ser realizado utilizando-se da sua interface de entrada. Em sequência, passa-se o controle para o dut para que o teste possa ser executado. Por fim, captura-se os resultados fornecidos pelo dut e compara-se com os valores esperados.

Em geral, são realizados vários testes em um módulo particular e esse procedimento é realizado para cada um deles. Normalmente, dentro deste conjunto de testes, são realizados alguns que verificam o funcionamento rotineiro do módulo, e, principalmente, os que testam os casos mais extremos.

Por exemplo: no procedimento de teste da memória de instruções, ela inicialmente recebe um endereço (de 32 *bits*) chamado read_addr. Na saída, exibe o valor do dado armazenado, chamado de instrução. Para testá-lo, é necessário verificar se exibe a instrução correta quando solicitado: basta inicializar a memória com um conjunto de instruções previamente definido,

[7]Teste unitário é a verificação do funcionamento de um módulo isolado.

em seguida acessar um de cada vez e comparar o valor obtido com o esperado. Caso em algum momento alguma saída divirja do esperado, uma mensagem é exibida sinalizando exatamente onde a divergência ocorreu.

Um procedimento análogo a esse deve ser executado para verificação de todos os módulos. Na ULA, o foco é a verificação de resultados de operações aritméticas. O RegFile e as duas memórias (instrução e dados), por poderem ser entendidos como bancos de registradores e deverem armazenar dados, foram verificados quanto à capacidade de reter informações e de retornar a informação correta no caso de consultas. Os dois módulos de controle foram testados variando o `opcode` das instruções passadas na entrada e checando quais linhas de controle foram ativadas.

Por último passou-se ao teste de integração, ou à checagem do funcionamento do processador com todos os componentes integrados. A verificação foi feita por um algoritmo que usa as instruções disponíveis e as instruções adicionadas *a posteriori*, descritas anteriormente. Decidiu-se calcular alguns termos da sequência de Fibonacci e armazenar os valores calculados sequencialmente em espaços contíguos na memória de dados para verificação futura. Começou-se com a escrita de um programa em C e sua tradução para Assembly através do assembler Mars MIPS.

A Figura 10 apresenta os comandos Assembly, os valores calculados e as formas de onda no início da execução. Os sinais apresentados são *clock*, a instrução executada em cada ciclo, o sinal de controle `memWrite` (que habilita a escrita na memória de dados), o valor presente na entrada da memória de dados (somente relevante quando a escrita estiver habilitada - memWrite ativo) em finalmente, o valor do registrador *Program Counter*, contendo o endereço da instrução sendo executada em cada ciclo.

Embora os módulos tenham sido escritos em SystemVerilog em RTL (*Register Transfer Level*), não houve tempo hábil para sintetizá-los. Com isso, essa etapa passa a ser uma sugerida como trabalho futuro.

6. Conclusão

Projetar circuitos digitais é uma tarefa complexa e trabalhosa. Por isso, realizar testes constantemente mostrou-se um processo essencial para o sucesso do projeto. Pôde-se obter prematuramente a confirmação de conformidade dos módulos que estavam sendo projetados com segurança ao aumentar a complexidade, graças à construção bem-sucedida das fases anteriores.

Conclui-se que o objetivo do projeto foi alcançado através da aplicação sistemática de metodologias de projeto, principalmente da escrita do *hardware* em HDL. Atesta-se assim o sucesso da nossa empreitada.

Como possibilidades de trabalhos futuros, são sugeridas as seguintes: aumentar o conjunto de instruções; implementar instruções privilegiadas; sintetizar o projeto; inserir um co-processador aritmético de modo a aumentar a capacidade do sistema; fazer versões multiciclo e *pipeline*. As possibilidades são inúmeras. Em suma, computadores são maneiros.

Agradecimentos

Esse projeto não existiria se não fosse por muita gente.

À minha família, agradeço pelas oportunidades. Pela educação, dentro e fora de casa. Pelo imensurável apoio. Quando eu precisei, sempre esteve lá por mim. Obrigada.

Ao professor Badan, pelas inúmeras vezes que me ajudou. Com explicações, sugestões e revisões. Obrigada por me orientar.

Tive a sorte de ter tido excelentes professores. Aos bons professores com quem já cruzei caminhos - obrigada. Entusiasmo é contagiante e palavras de encorajamento fazem a diferença. Lembro de vocês com grande estima.

Agradeço aos que me disseram pra não desistir. Aos que me ensinaram a me responsabilizar por minha vida. Aos que deram o exemplo e me lembraram que ser produto do meio é opcional. Obrigada pela inspiração.

Referências

[1] Máquina analítica. Acessado em: 28/01/2024. [Online]. Available: https://pt.wikipedia.org/wiki/M%C3%A1quina_anal%C3%ADtica

[2] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Pearson, 2014.

[3] Putting Time In Perspective. Acessado em: 28/01/2024. [Online]. Available: https://waitbutwhy.com/2013/08/putting-time-in-perspective.html

[4] Comparison of instruction set architectures. Acessado em: 28/01/2024. [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_instruction_set_architectures

[5] Introduction to the MIPS32 Architecture. Acessado em: 28/01/2024. [Online]. Available: https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00082-2B-MIPS32INT-AFP-06.01.pdf

[6] The MIPS32 Instruction Set Manual. Acessado em: 28/01/2024. [Online]. Available: https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00086-2B-MIPS32BIS-AFP-6.06.pdf

[7] Arquitetura Harvard. Acessado em: 28/01/2024. [Online]. Available: https://pt.wikipedia.org/wiki/Arquitetura_Harvard

[8] MIPS-in-Verilog. Acessado em: 28/01/2024. [Online]. Available: https://github.com/alok-upadhyay/MIPS-in-Verilog/tree/master

[9] Verilog-MIPSProcessor. Acessado em: 28/01/2024. [Online]. Available: https://github.com/waleedmohamedme/Verilog-MIPSProcessor/tree/master

[10] 32-bit-MIPS-Processor. Acessado em: 28/01/2024. [Online]. Available: https://github.com/sevvalmehder/32-bit-MIPS-Processor/tree/master

[11] 32-bit-Multicycle-CPU. Acessado em: 28/01/2024. [Online]. Available: https://github.com/johnc219/32-bit-Multicycle-CPU/tree/master

[12] MIPS. Acessado em: 28/01/2024. [Online]. Available: https://github.com/valar1234/MIPS/tree/master

[13] mips-cpu. Acessado em: 28/01/2024. [Online]. Available: https://github.com/sxtyzhangzk/mips-cpu/tree/master

[14] W. Stallings, *Computer Organization and Architecture*, 9th ed. Pearson, 2012.

[15] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design*, 4th ed. Morgan Kaufmann, 2011.

[16] Visual Studio Code. Acessado em: 28/01/2024. [Online]. Available: https://code.visualstudio.com/

[17] HDLBits — Verilog Practice. Acessado em: 28/01/2024. [Online]. Available: https://hdlbits.01xz.net/

[18] ChipVerify. Acessado em: 28/01/2024. [Online]. Available: https://www.chipverify.com/

[19] Icarus Verilog. Acessado em: 28/01/2024. [Online]. Available: https://github.com/steveicarus/iverilog
[20] Verilator. Acessado em: 28/01/2024. [Online]. Available: https://github.com/verilator/verilator
[21] cocotb - Python verification framework. Acessado em: 28/01/2024. [Online]. Available: https://www.cocotb.org/
[22] GTKWave. Acessado em: 28/01/2024. [Online]. Available: https://github.com/gtkwave/gtkwave
[23] MARS MIPS Simulator. Acessado em: 28/01/2024. [Online]. Available: https://courses.missouristate.edu/kenvollmar/mars/
[24] MIPS Encoding Reference. Acessado em: 28/01/2024. [Online]. Available: https://student.cs.uwaterloo.ca/~isg/res/mips/opcodes
[25] Make - Makefiles. Acessado em: 28/01/2024. [Online]. Available: https://en.wikipedia.org/wiki/Make_(software)#Makefiles

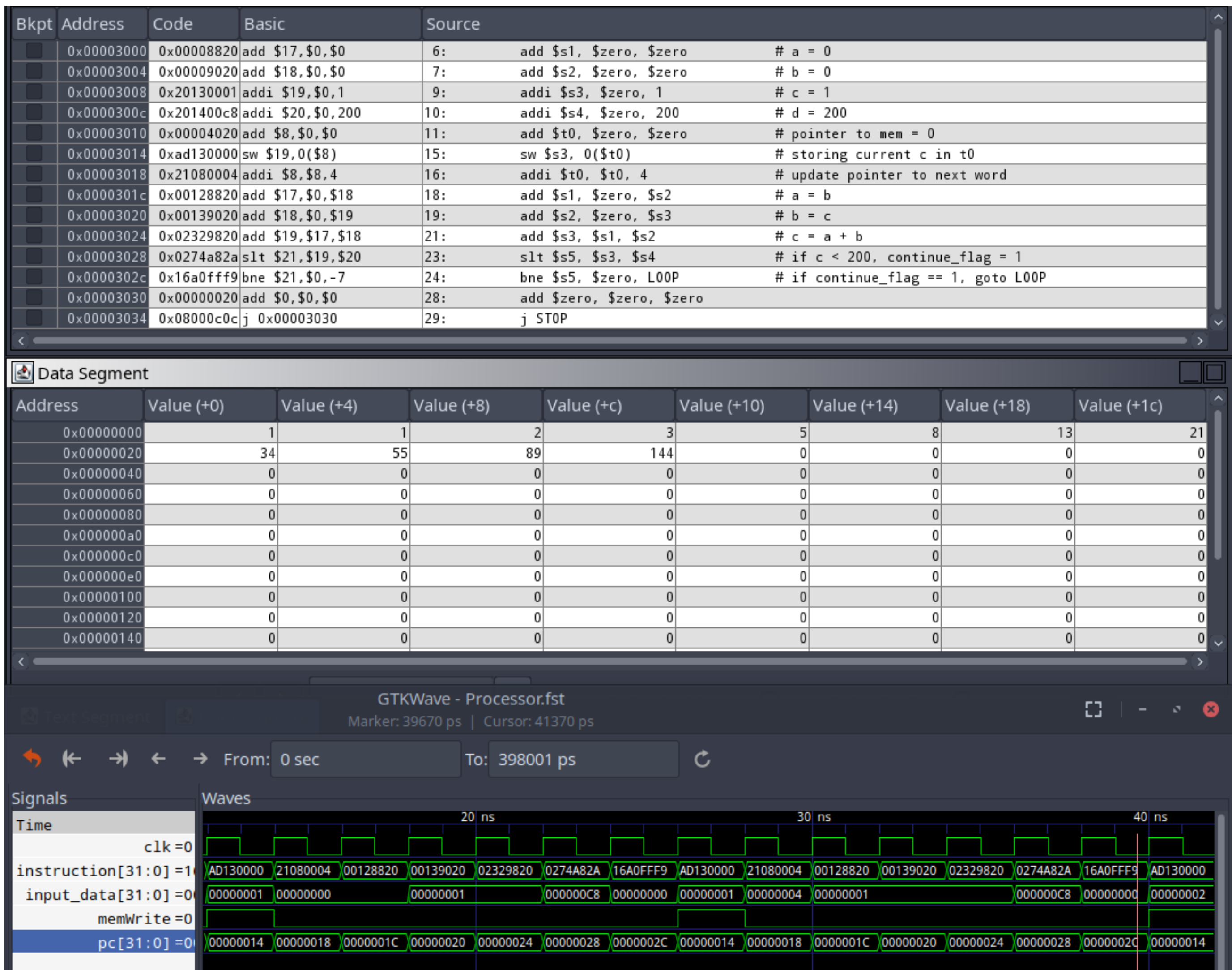


Fig. 10. Código em Assembly e formas de onda