UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS

Guilherme Prearo

Desenvolvimento e implementação de processador com arquitetura Harvard em tecnologia CMOS de 0,35μm

São Carlos

2018

Guilherme Prearo

Desenvolvimento e implementação de processador com arquitetura Harvard em tecnologia CMOS de 0,35μm

Monografia apresentada ao Curso de Engenharia de Computação, da Escola de Engenharia de São Carlos e Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo, como parte dos requisitos para obtenção do título de Engenheiro de Computação.

Orientador: Prof. Dr. João Navarro Soares Junior

São Carlos

2018

Ficha catalográfica elaborada pela Biblioteca Prof. Dr. Sérgio Rodrigues Fontes da EESC/USP com os dados inseridos pelo(a) autor(a).

Prearo, Guilherme
P953d Desenvolvimento e implementação de processador com arquitetura Harvard em tecnologia CMOS de 0,35um / Guilherme Prearo; orientador João Navarro Soares Júnior. São Carlos, 2018.

Monografia (Graduação em Engenharia de Computação) -- Escola de Engenharia de São Carlos e Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo, 2018.

1. processador. 2. VHDL. 3. sistemas digitais. 4. circuitos integrados digitais. 5. microcontrolador. 6. descrição de hardware. 7. organização de computadores. 8. arquitetura de computadores. I. Título.

Eduardo Graziosi Silva - CRB - 8/8907

FOLHA DE APROVAÇÃO

Nome: Guilherme Prearo

Título: "Desenvolvimento e implementação de processador com arquitetura Harvard em tecnologia CMOS de 0,35μm"

Trabalho de Conclusão de Curso defendido em 28 / 06 / 2019.

Comissão Julgadora:	**Resultado:**
Prof. Dr. João Navarro Soares Júnior (Orientador) - SEL/EESC/USP	APROVADO
Prof. Dr. João Paulo Pereira do Carmo SEL/EESC/USP	APROVADO
Prof. Dr. Maximiliam Luppe SEL/EESC/USP	Aprovado

Coordenador do Curso Interunidades Engenharia de Computação:

Prof. Dr. Maximiliam Luppe

RESUMO

PREARO, G. **Desenvolvimento e implementação de processador com arquitetura Harvard em tecnologia CMOS de 0,35μm** . 2018. 90p. Monografia (Trabalho de Conclusão de Curso) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2018.

Os processadores estão presentes em praticamente todas as tarefas que realizamos atualmente, e a tendência é que esta presença aumente cada vez mais. O objetivo deste trabalho é projetar, descrever em linguagem de descrição de hardware e implementar parcialmente um processador de 8 bits, assim, este material pode ser utilizado em disciplinas que estudam desde organização de computadores até projeto de circuitos integrados. Ao longo da etapa de projeto as instruções do processador são definidas, a microarquitetura é diagramada e a unidade de controle é projetada. Logo após é feita toda a descrição do processador em uma linguagem de descrição de hardware, VHDL, e este código simulado para garantir a coerência com o projeto. Com o código simulado é feito o *layout* de um dos blocos funcionais do processador, o somador de 16 bits. Como resultado, temos uma visão geral de todo o processo de criação de um circuito integrado digital, bem como o processo de projetar um processador simples.

Palavras-chave: Sistemas digitais. Circuitos integrados digitais. Processador. Microcontrolador. Descrição de Hardware. VHDL. Organização de computadores. Arquitetura de computadores.

ABSTRACT

PREARO, G. **Development and implementation of a Harvard architecture processor in 0.35μm CMOS** . 2018. 90p. Monografia (Trabalho de Conclusão de Curso) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2018.

Processors are present in almost all activities we do today, and the expectation is that this presence will increase more and more. The goal of this work is to design, to describe in hardware description language and partially implement a 8 bit processor, thus, this work can be used in courses of computer organization to integrated circuits design. Throughout the design stage the processor instructions are defined, the micro-architecture is diagrammed and the control unit is designed. Right after that, the processor description is made in a hardware description language, VHDL, and the code simulated, to ensure the consistency with the design. With the simulated code, the layout of one of the blocks is made, the 16 bit adder. As result, we have a overview of all the process of making a digital integrated circuit, as well the process of design a simple processor.

Keywords: Digital systems. Digital integrated circuits. Processor. Micro-controller. Hardware description. VHDL. Computer organization. Computer architecture.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ASIC	*Application-Specific Integrated Circuit*
CPU	*Central Processing Unit*
CPLD	*Complex Programmable Logic Device*
DRC	*Design Rules Check*
EDIF	*Electronic Design Interchange Format*
FPGA	*Field-Programmable Gate Array*
LVS	*Layout versus Schematic*
PC	*Program Counter*
ULA	Unidade Lógica Aritmética
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuit*

SUMÁRIO

1 INTRODUÇÃO

1.1 Motivação

Processadores estão presentes em praticamente todos equipamentos utilizados atualmente, tanto no lazer, quanto no estudo e no trabalho. Com o recente advento do IoT (*Internet of Things*), a tendência é que processadores, na forma de microcontroladores, e sistemas embarcados se tornem cada vez mais presentes em nossa vida.

Segundo (BORGES, 2011), o avanço na tecnologia dos processadores com arquitetura de 8 bits vem barateando dispositivos que apresentam melhor desempenho além de mais recursos de memória e periféricos. Isto é diretamente refletido no gráfico desta mesma pesquisa, que exibe os microcontroladores de 8 bits sendo utilizados em 31% das aplicações de sistemas embarcados no Brasil, perdendo apenas para os de 32 bits, com 34%, estes dados podem ser vistos na figura 1.

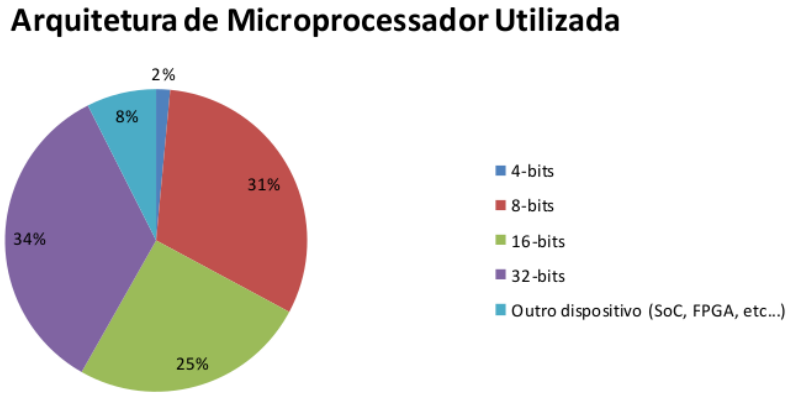


Figura 1: Arquitetura de microprocessadores utilizados nos projetos de sistemas embarcados (distribuição percentual). Fonte: (BORGES, 2011).

Uma ISA (*Instrction Set Architecture*) é um camada de abstração entre as camadas de software e hardware de um processador. A maioria das ISAs modernas podem ser classificadas em duas classes: CISC (*Complex Instruction Set Computer*) e RISC (*Reduced Instruction Set Computer*). As arquiteturas CISC são compostas por instruções mais complexas, com funções especializadas, tamanho de instrução variável, com decodificação não uniforme e vários modos de endereçamento. As RISC vão na linha contrária, com instruções simples e reduzidas, tamanho de instrução fixo, decodificação uniforme e número de modos de endereçamento reduzido (SHAHZEB et al., 2016). Segundo (GEORGE, 1990), a arquitetura RISC tem algumas vantagens em relação a CISC, como: unidade de controle simples, decodificação rápida, execução eficiente com *pipeline* e um processo rápido de projeto, implementação e teste de processador. Motivado por estas vantagens, este trabalho

será baseado na arquitetura RISC.

Um processador que utiliza arquitetura de *von Neumann* é aquele em que as instruções e os dados são armazenados em uma única memória, este tipo de arquitetura é distinto da arquitetura *Harvard*, que separa as memórias utilizadas para armazenar instruções e dados (EIGENMANN; LILJA, 1998). Quando as duas arquiteturas são comparadas, a Harvard tem uma largura de barramento maior, já que na de *von Neumann* instruções e dados são acessadas por um barramento compartilhado (ŠULIK; VASILKO; FUCHS, 2001). Como descrito anteriormente, a arquitetura RISC tira vantagem da implementação em *pipeline*, para que este *pipeline* seja mais eficiente, este trabalho implementará a arquitetura Harvard, assim, uma instrução pode ser lida em paralelo com um acesso à memória de dados.

Do ponto de vista didático, atualmente é comum o aprendizado de organização de computadores e arquitetura de computadores utilizando a arquitetura do processador RISC MIPS, de 32 bits, utilizado nos livros (GOODMAN; MILLER, 1993) e (PATTERSON; HENNESSY, 2013). Devido ao fato de ser um processador de 32 bits, implementações em hardware do MIPS acabam se tornando grandes, o que dificulta sua realização com fins didáticos. Um processador de 8 bits serviria melhor a este propósito, podendo ser utilizado em disciplinas que analisam processadores em alto nível, até em disciplinas de de projeto de circuitos integrados digitais.

1.2 Proposta

O intuito deste trabalho é propor um processador simples, de 8 bits, do tipo RISC, que possa servir para aplicações didáticas em disciplinas de organização de computadores, arquitetura de computadores, sistemas digitais e projetos de circuitos integrados digitais. Futuramente, ele pode ser aproveitado para a construção de microcontroladores de propósito geral, que poderão ser utilizados para fins didáticos, ou em alguns sistemas embarcados simples.

Neste trabalho será elaborado o processador completo, partindo da arquitetura (tamanho dos dados, tamanho das memorias, registradores, endereçamentos, instruções) e terminando na sua microarquitetura (com o caminho de dados e blocos). A microarquitetura será então descrita em linguagem de descrição de hardware (*Hardware Description Language - HDL*) para permitir sua simulação e validação. Em uma última etapa, será implementado um dos blocos da arquitetura na tecnologia CMOS de $0,35\mu m$.

1.3 Objetivos

Os objetivos deste trabalho são divididos nas etapas de elaboração do processador, de descrição em HDL microarquitetura, de simulação e de implementação.

- Elaboração do Processador

  Esta etapa tem por objetivo a definição da arquitetura do processador, a interface externa do mesmo, o mapeamento dos registradores, o conjunto de instruções a ser implementado, a codificação das operações, a construção da microarquitetura do processador e definição dos sinais de controle para cada instrução.

- Descrição em HDL

  Nesta etapa o objetivo é descrever a microarquitetura projetada no item anterior em linguagem de descrição de hardware. Esta descrição será feita de forma modularizada, para facilitar a simulação.

- Simulação

  Neste ponto é feita a simulação do HDL construído no item anterior. Esta simulação tem o objetivo de checar a conformidade do código de descrição com a microarquitetura e esta com a arquitetura proposta. Também é feita uma simulação para estimar a frequência máxima de operação do circuito.

- Implementação

  Esta etapa visa implementar uma pequena parte do circuito descrito na tecnologia CMOS de $0,35\mu m$. Assim, tem-se uma visão ampla do processo de criação de um circuito integrado digital.

2 PROPOSTA DE ARQUITETURA

2.1 Embasamento teórico

Nesta seção serão apresentados dois assuntos que fundamentam a elaboração deste trabalho: projeto de circuitos integrados digitais e computadores digitais.

2.1.1 Projeto de circuitos integrados digitais

O projeto de circuitos integrados digitais é dividido em várias etapas, e cada uma delas trabalha com um nível de abstração diferente. Esses níveis variam da especificação do sistema de maneira comportamental até a camada física.

Segundo (NELSON et al., 1995), o nível mais alto de abstração é aquele em que o sistema digital pode ser visto como um ou mais blocos funcionais conectados. Neste nível não é levado em consideração nenhum detalhe de implementação dos blocos, mas sim a função deles. O nível logo abaixo deste é composto por elementos armazenadores, chamados de *registradores*, e nele é indicado como são feitas as transferências dos dados entre os registradores. Em alguns casos, os dados podem sofrer alterações durante estas transferências. Abaixo deste nível, mas ainda sendo uma representação comportamental, tem-se o nível de portas lógicas. Estas são elementos que realizam as operações fundamentais da álgebra booleana. Os circuitos lógicos formados neste nível podem ser divididos em duas categorias, os combinacionais, que a saída depende apenas da entrada atual, e os sequenciais, que a saída depende de todas as entradas a partir de um momento inicial. Estas portas lógicas definem completamente o funcionamento do sistema e são implementadas, no nível inferior, por combinações de dispositivos. Atualmente os dispositivos básicos utilizadas na implementação de portas são os transistores MOS (*Metal-Oxide-Semiconductor*) que são construídos pela associação de diversos materiais.

Os dispositivos básicos aplicados no nível inferior de abstração sofreram diversas mudanças nas últimas décadas. Os componentes mecânicos das máquinas de calculo concebidas no século XIX foram substituídos por chaves elétricas (relés), que eram uma solução com melhor custo benefício. Estas chaves, por sua vez, foram substituídas por válvulas eletrônicas. Com a tecnologia de válvulas já se tornou possível a montagem de computadores completos, como o ENIAC e o UNIVAC I. O consumo elevado de energia, a dificuldade de integração e a falta de confiabilidade desses computadores fez esta tecnologia atingir seu limite (RABAEY, 1996). Em 1947, nos laboratórios da Bell, o primeiro transistor bipolar foi inventado, criando um substituto para as válvulas e a possibilidade de construir computadores mais eficientes. Ao fim da década de 60, tecnologias de integração de circuitos são pesquisadas e aprimoradas. Transistores bipolares e circuitos integrados

permitiram a fabricação de vários dispositivos em um único pedaço de semicondutor, *chip*, elevando o poder dos computadores desenvolvidos a um novo patamar. Em 1960 se obteve o primeiro transistor MOS funcional. Este dispositivo permitiu melhoras na implementação de circuitos digitais e o desenvolvimento de memorias maiores (AYERS, 2009).

2.1.2 Organização de Computadores

Um computador digital é um sistema composto por blocos funcionais tais como a unidade lógica e aritmética, a unidade de controle, memórias e unidades de armazenamento e entradas e saídas (NELSON et al., 1995). O ciclo de instrução de um computador digital pode ser dividido em passos. Uma divisão normalmente considerada é a seguinte: (1 - *Instruction Fetch* - *IF*) ler a próxima instrução a ser executada da memória de programas, (2 - *Instruction Decodification* - *ID*) identificar qual operação a instrução representa e buscar os operandos no banco de registradores, (3 - *Execution* - *EX*) executar as operações aritméticas necessárias, (4 - *Memory* - *MEM*) acessar a memória de dados e (5 - *Write Back* - WB) escrever os resultados nos registradores.

Uma das métricas para avaliar o desempenho de um processador é a quantidade de instruções que ele consegue realizar por segundo, ou o *throughput*. O valor do *throughput* pode ser estimado como sendo o inverso do tempo médio de execução das instruções. Para melhorar o valor desta métrica, existe uma técnica de implementação, chamada *pipeline*, em que os passos do ciclo de instrução são feitos paralelamente em estágios distintos do *pipeline*. Cada um dos estágios do *pipeline* é responsável pela execução de um dos passo das instruções (PATTERSON; HENNESSY, 2013). Cada um dos estágios é isolado por elementos de armazenamento dos outros. Quando a execução de uma instrução é terminada em um estágio, os dados gerados são conduzidos para o próximo estágio onde a instrução continuará sua execução. Ao mesmo tempo, o estagio recebe novos dados para processar uma nova instrução.

A técnica de *pipeline* permite um aumento considerável da velocidade nos circuitos, mas pode gerar alguns problemas na execução de instruções, os *hazard*. No geral, os *hazard* podem ser classificados em três tipos:

- Estrutural: aparece quando duas ou mais instruções necessitam usar o mesmo recurso do processador (memória, ULA, banco de registrador, etc.) no mesmo ciclo. Este tipo de *hazard* não nos incomodará.

- De Dados: aparece quando dados são escritos ou lidos em ordem diferente da que deveria ocorrer devido à execução pipeline. Considere, para exemplificar, o caso em que há uma instrução, I1, que altera um determinado registrador, seguida de outra instrução, I2, que lê a esse mesmo registrador. Considere ainda que as instruções são executadas em cinco passos como dito acima. Como é necessário que I1 percorra

todos os estágios antes do resultado ser escrito no registrador, pode ocorrer que I2, quando executando, leia um valor incorreto do registrador, que não foi ainda atualizado por I1.

- De Controle: aparece com instruções de desvio, condicional ou não, que forçam com que certas instruções já em execução sejam ignoradas. Considere, para exemplificar, o caso de uma instrução de salto condicional, I1. Ela deve passar por alguns estágios até que seja decido se o salto deve ou não ser executado. Caso haja o salto, decidido, digamos, no terceiro estagio do *pipeline*, duas outras instruções, I2 e I3, já estarão sendo executadas nos dois primeiros estágios. Estas instruções, I2 e I3, deverão ser canceladas ou ser instruções do tipo *NOP* (operação que não realiza nenhuma escrita), colocadas pelo compilador na sequencia a I1, para que não haja nenhuma escrita indesejada nos registradores ou nas memórias.

2.2 Arquitetura

2.2.1 Visão geral

Esta seção visa definir algumas características iniciais do processador, as limitações e nortear o desenvolvimento do projeto. A arquitetura e a microarquitetura do processador tem as seguintes características

- O processador será implementado com arquitetura Harvard, visando possível uso em microcontroladores;
- Os dispositivos de entrada e saída serão mapeados como memória de dados;
- O tamanho da instrução será de 16 bits;
- As operações e os operandos são de 8 bits;
- A leitura na memória de instruções será feita com 16 bits.
- A leitura ou escrita na memória de dados será feita com 8 bits.
- O processador será implementado com *pipeline*;
  - Os problemas de dependência de dados e saltos condicionais do *pipeline* devem ser resolvido por software;

2.2.2 Definição da interface com memórias

Como o projeto de nosso processador é baseado na arquitetura Harvard ao invés da Von Neumann, há uma memória de dados e uma de programas, separadas fisicamente e com barramentos independentes.

Para que o processo de referencia à memória seja feito de forma simples com registradores de 8 bits, decidiu-se utilizar endereços de 16 bits para ambas as memórias e endereçamento a byte. Com isso, é possível endereçar até 65536 bytes nas memórias de dados e de programa.

Devido ao tamanho da instrução ser de 16 bits, optou-se por um barramento de 16 bits para a memória de programas. Assim, uma instrução completa é obtida em apenas um acesso a memória. Como só serão efetuadas operações de leitura nesta memória, há apenas um barramento unidirecional. Para a memória de dados, como é realizada apenas uma leitura ou uma escrita de um byte em cada operação (e nunca ambos simultaneamente), optou-se por apenas um barramento de 8 bits bidirecional.

A figura 2 sumariza as escolhas realizadas ao exibir um diagrama de blocos com as duas memórias, os barramentos de dados e endereço e o processador. Cada memória também deve conter sinais de controle para selecionar se esta sendo feita leitura ou escrita. Estes sinais foram omitidos do diagrama, pois a intenção é mostrar apenas informações sobre os barramentos.

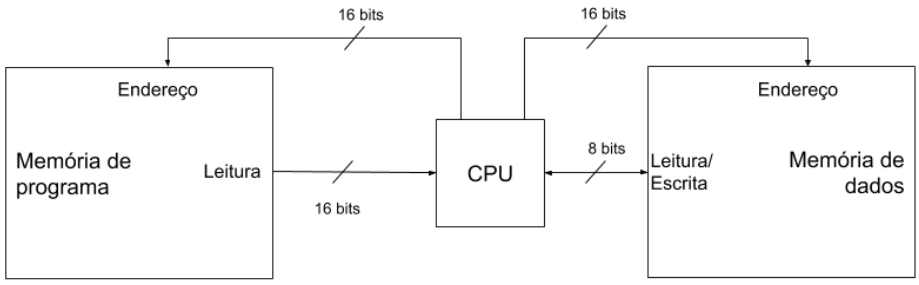


Figura 2: Diagrama das conexões entres as memórias de programa e dados e a CPU (*Central Processing Unit*).

2.2.3 Definição dos registradores

A estrutura de registradores consiste basicamente de dois bancos de registradores, com alguns poucos registradores de propósito específico e o restante deles de propósito geral. Cada registrador tem a capacidade de armazenamento de um byte. A tabela 1 exibe informações sobre os dois bancos de registradores, com os nomes e endereços dos resgistradoes.

Os registradores RET_ADDR_L e RET_ADDR_H são utilizados para salvar o valor de PC (ver abaixo a função deste registrador) quando uma rotina é chamada. Neste documento, este dois registradores serão referenciados apenas como RET_ADDR, como mostra a tabela 2. Os registradores $DPTR\_H$ e $DPTR\_L$ são utilizados como endereço de base no acesso à memória de dados, tanto na escrita, quanto na leitura. Estes dois registradores serão referenciados como apenas $DPTR$ ao longo deste documento, como

mostra a tabela 3. Vale ressaltar que $DPTR$ contém apenas o endereço base, e para o acesso à memória este endereço será somado à um *offset* em registrador ou em imediato. Os registradores $R0$ até $R11$ são de propósito geral, porém, todas as instruções, exceto as de deslocamento, utilizam diretamente apenas os registradores do banco 0. Os registradores $R7$ ao $R11$ são úteis para armazenamento de dados, e não para o processamento.

Além dos dois bancos de registradores o processador também conta com o registrador PC (*Program Counter*) de 16 bits, que armazena o endereço da instrução a ser executada, e o registrador $FLAGS$ de três bits, que armazena as *flags* geradas pela ULA e esta ligado à unidade de controle. Estes registradores não são referenciados diretamente por nenhuma instrução. O registrador PC deve ter um sinal de *Reset*, para zerar seu conteúdo, permitindo inicializar a execução de programas quando o processador for ligado.

Tabela 1: Mapeamento dos registradores de cada banco.

Banco de registradores 0		**Banco de registradores 1**	
Nome	**Endereço**	**Nome**	**Endereço**
R0	0	R7	0
R1	1	R8	1
R2	2	R9	2
R3	3	R10	3
R4	4	R11	4
R5	5	DPTR_L	5
R6	6	DPTR_H	6
RET_ADDR_L	7	RET_ADDR_H	7

Tabela 2: Definição dos bits do registrador RET_ADDR, que é a concatenação dos registradores RET_ADDR_H (bits mais significativos) e RET_ADDR_L (bits menos significativos).

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
RET_ADDR_H								RET_ADDR_L							

Tabela 3: Definição dos bits do registrador $DPTR$, que é a concatenação dos registradores $DPTR\_H$ (bits mais significativos) e $DPTR\_L$ (bits menos significativos).

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
DPTR_H								DPTR_L							

2.2.4 Definição do conjunto de instruções

As instruções do processador tem 16 bits de comprimento. Destes 16 bits, os cinco mais significativos são o código da operação e os bits restantes identificam os operandos. Sempre que possível, foi utilizado o 11º bit para representar se a instrução é realizada

entre registradores ou com valor imediato (valor numérico na própria instrução). Abaixo são apresentadas as instruções. Os códigos de operação estão em ordem crescente. Nas descrições feitas, os bits marcados com *X*, *don't care*, podem assumir qualquer valor.

2.2.4.1 Deslocamento para esquerda

Assembly: `LSL Rd, Rm, #imediato`

A instrução deslocamento para esquerda recebe dois bits, *Bm* e *Bd*, dois registradores, *Rm* e *Rd*, e um valor numérico de 3 bits, *imediato*. O processador então atribui para *Rd* o valor $Rm \cdot 2^{imediato}$. O bit *Bm* determina de qual banco de registradores é *Rm* e *Bd*, de qual é *Rd* (*0* indica o banco de registradores *0* e *1*, o banco *1*). A tabela 4 exibe a função de cada um dos 16 bits desta instrução.

Tabela 4: Definição dos bits da instrução `LSL`, deslocamento para esquerda.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	0	0	0	0	Bm	Bd	imediato			Rm			Rd		

2.2.4.2 Deslocamento para direita

Assembly: `LSR Rd, Rm, #imediato`

A instrução deslocamento para direita recebe dois bits, *Bm* e *Bd*, dois registradores, *Rm* e *Rd*, e um valor numérico de 3 bits, *imediato*. O processador então atribui para *Rd* o valor $\frac{Rm}{2^{imediato}}$. O bit *Bm* determina de qual banco de registradores é *Rm* e *Bd*, de qual é *Rd* (*0* indica o banco de registradores *0* e *1*, o banco *1*). A tabela 5 exibe a função de cada um dos 16 bits desta instrução.

Tabela 5: Definição dos bits da instrução `LSR`, deslocamento para direita.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	0	0	0	1	Bm	Bd	imediato			Rm			Rd		

2.2.4.3 Adição de registradores

Assembly: `ADD Rd, Rm, Rn`

A instrução de adição recebe três registradores, *Rm*, *Rn* e *Rd*. O processador atribui a *Rd* o valor de (*Rm* + *Rn*) (os registradores são do banco *0*). A tabela 6 exibe a função de cada um dos 16 bits desta instrução.

Tabela 6: Definição dos bits da instrução `ADD`, soma de registradores.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	0	0	1	0	X	X	Rm			Rn			Rd		

2.2.4.4 Adição com imediato

Assembly: `ADD Rd, #imediato`

A instrução de adição com imediato recebe um registrador, *Rd*, e um valor numérico de 8 bits, *imediato*. O processador então atribui a *Rd* o valor de (*Rd* + *imediato*) (os registradores são do banco *0*). A tabela 7 exibe a função de cada um dos 16 bits desta instrução.

Tabela 7: Definição dos bits da instrução `ADD`, soma com imediato.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	0	0	1	1	imediato								Rd		

2.2.4.5 Subtração de registradores

Assembly: `SUB Rd, Rm, Rn`

A instrução de subtração recebe três registradores, *Rm*, *Rn* e *Rd*. O processador atribui a *Rd* o valor de (*Rm* - *Rn*) (os registradores são do banco *0*). A tabela 8 exibe a função de cada um dos 16 bits desta instrução.

Tabela 8: Definição dos bits da instrução `SUB`, subtração de registradores.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	0	1	0	0	X	X	Rm			Rn			Rd		

2.2.4.6 Subtração com imediato

Assembly: `SUB Rd, #imediato`

A instrução de subtração com imediato recebe um registrador, *Rd*, e um valor numérico de 8 bits, *imediato*. O processador então atribui a *Rd* o valor de (*Rd* - *imediato*) (os registradores são do banco *0*). A tabela 9 exibe a função de cada um dos 16 bits desta instrução.

Tabela 9: Definição dos bits da instrução `SUB`, subtração com imediato.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	0	1	0	1	imediato								Rd		

2.2.4.7 Adição de registradores com *carry*

Assembly: `ADC Rd, Rm, Rn`

A instrução de adição com *carry* recebe três registradores, *Rm*, *Rn* e *Rd*. O processador atribui a *Rd* o valor de (*Rm* + *Rn* + *flag Carry*) (os registradores são do banco *0*). A tabela 10 exibe a função de cada um dos 16 bits desta instrução.

Tabela 10: Definição dos bits da instrução `ADC`, soma de registradores com *carry*.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	0	1	1	0	X	X	Rm			Rn			Rd		

2.2.4.8 Adição com imediato e *carry*

Assembly: `ADC Rd, #imediato`

A instrução de adição com imediato e *carry* recebe um registrador, *Rd*, e um valor numérico de 8 bits, *imediato*. O processador então atribui a *Rd* o valor de (*Rd* + *imediato* + *flag Carry*) (os registradores são do banco *0*). A tabela 11 exibe a função de cada um dos 16 bits desta instrução.

Tabela 11: Definição dos bits da instrução `ADC`, soma com imediato e *carry*.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	0	1	1	1	imediato								Rd		

2.2.4.9 *AND* bit-a-bit de registradores

Assembly: `AND Rd, Rm, Rn`

A instrução *AND* bit-a-bit recebe três registradores, *Rm*, *Rn* e *Rd*. O processador então atribui a *Rd* o valor de (*Rm AND Rn*), *E* (tabela verdade 12) feito bit-a-bit (os registradores são do banco *0*). A tabela 13 exibe a função de cada um dos 16 bits desta instrução.

Tabela 12: Tabela verdade da operação *E*. Entradas *A* e *B* e saída *S*.

A	**B**	**S**
0	0	0
0	1	0
1	0	0
1	1	1

Tabela 13: Definição dos bits da instrução `AND`, operação *E* bit-a-bit.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	1	0	0	0	X	X	Rm			Rn			Rd		

2.2.4.10 *AND* bit-a-bit com imediato

Assembly: `AND Rd, #imediato`

A instrução *AND* bit-a-bit com imediato recebe um registrador, *Rd*, e um valor numérico de 8 bits, *imediato*. O processador então atribui a *Rd* o valor de (*Rm AND*

imediato), *E* (tabela verdade 12) feito bit-a-bit (os registradores são do banco *0*). A tabela 14 exibe a função de cada um dos 16 bits desta instrução.

Tabela 14: Definição dos bits da instrução `AND`, operação *E* bit-a-bit com imediato.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	1	0	0	1	imediato								Rd		

2.2.4.11 *OR* bit-a-bit de registradores

Assembly: `OR Rd, Rm, Rn`

A instrução *OR* bit-a-bit recebe três registradores, *Rm*, *Rn* e *Rd*. O processador então atribui a *Rd* o valor de (*Rm OR Rn*), *OU* (tabela verdade 15) feito bit-a-bit (os registradores são do banco *0*). A tabela 16 exibe a função de cada um dos 16 bits desta instrução.

Tabela 15: Tabela verdade da operação *OU*. Entradas *A* e *B* e saída *S*.

A	**B**	**S**
0	0	0
0	1	1
1	0	1
1	1	1

Tabela 16: Definição dos bits da instrução `OR`, operação *OU* bit-a-bit.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	1	0	1	0	X	X	Rm			Rn			Rd		

2.2.4.12 *OR* bit-a-bit com imediato

Assembly: `OR Rd, #imediato`

A instrução *OR* bit-a-bit com imediato recebe um registrador, *Rd*, e um valor numérico de 8 bits, *imediato*. O processador então atribui a *Rd* o valor de (*Rm OR imediato*), *OU* (tabela verdade 15) feito bit-a-bit (os registradores são do banco *0*). A tabela 17 exibe a função de cada um dos 16 bits desta instrução.

Tabela 17: Definição dos bits da instrução `OR`, operação *OU* bit-a-bit com imediato.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	1	0	1	1	imediato								Rd		

2.2.4.13 *XOR* bit-a-bit de registradores

Assembly: `XOR Rd, Rm, Rn`

A instrução *XOR* bit-a-bit recebe três registradores, *Rm*, *Rn* e *Rd*. O processador então atribui a *Rd* o valor de (*Rm XOR Rn*), *Ou Exclusivo* (tabela verdade 18) feito bit-a-bit (os registradores são do banco *0*). A tabela 19 exibe a função de cada um dos 16 bits desta instrução.

Tabela 18: Tabela verdade da operação *Ou Exclusivo*. Entradas *A* e *B* e saída *S*.

A	**B**	**S**
0	0	0
0	1	1
1	0	1
1	1	0

Tabela 19: Definição dos bits da instrução `XOR`, operação *Ou Exclusivo* bit-a-bit.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	1	1	0	0	X	X	Rm			Rn			Rd		

2.2.4.14 *XOR* bit-a-bit com imediato

Assembly: `XOR Rd, #imediato`

A instrução *XOR* bit-a-bit com imediato recebe um registrador, *Rd*, e um valor numérico de 8 bits, *imediato*. O processador então atribui a *Rd* o valor de (*Rm XOR imediato*), *Ou Exclusivo* (tabela verdade 18) feito bit-a-bit (os registradores são do banco *0*). A tabela 20 exibe a função de cada um dos 16 bits desta instrução.

Tabela 20: Definição dos bits da instrução `XOR`, operação *Ou Exclusivo* bit-a-bit com imediato.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	1	1	0	1	imediato								Rd		

2.2.4.15 *NOT* bit-a-bit

Assembly: `NOT Rd, Rm`

A instrução *NOT* bit-a-bit recebe dois registradores, *Rm* e *Rd*. O processador então atribui a *Rd* o valor de (*NOT Rm*), *Negação* (tabela verdade 21) feito bit-a-bit (os registradores são do banco *0*). A tabela 22 exibe a função de cada um dos 16 bits desta instrução.

Tabela 21: Tabela verdade da operação *NOT*. Entrada *A* e saída *S*.

A	**S**
0	1
1	0

Tabela 22: Definição dos bits da instrução `NOT`, operação de negação bit-a-bit.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	1	1	1	0	X	X	X	X	X	Rm			Rd		

2.2.4.16 Mover imediato para um registrador

Assembly: `MOV Rd, #imediato`

A instrução *MOV* com imediato recebe um registrador, *Rd*, e um valor numérico de 8 bits, *imediato*. O processador então atribui a *Rd* o valor de *imediato* (os registradores são do banco *0*). A tabela 23 exibe a função de cada um dos 16 bits desta instrução.

Tabela 23: Definição dos bits da instrução `MOV`, mover valor de um imediato para um registrador.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	1	1	1	1	imediato								Rd		

2.2.4.17 Salto incondicional

Assembly: `JMP` ***offset***

A instrução *JMP* recebe um valor numérico de 11 bits, $offset$. O processador então atribui ao registrador *PC* o valor $(PC + offset \cdot 2)$ [1], para assim avançar ou retroceder $offset$ instruções. Dessa forma é alterado o fluxo de execução de instruções. A tabela 24 exibe a função de cada um dos 16 bits desta instrução.

Tabela 24: Definição dos bits da instrução `JMP`, salto incondicional.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
1	0	0	0	0	*offset*										

2.2.4.18 Salto incondicional de rotina

Assembly: `JAL` ***offset***

A instrução *JAL* recebe um valor numérico de 11 bits, $offset$. O processador então atribui ao registrador RET_ADDR o valor de *PC* e ao *PC* o valor $(PC + offset \cdot 2)$,

[1] Cada instrução tem dois bytes de comprimento e a memória de programa é endereçada a byte, por isso a multiplicação por dois.

para assim avançar ou retroceder $offset$ instruções. Dessa forma é alterado o fluxo de execução de instruções, mas fica armazenado o endereço da instrução atual. A tabela 25 exibe a função de cada um dos 16 bits desta instrução.

Tabela 25: Definição dos bits da instrução `JAL`, salto incondicional de rotina.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
1	0	0	0	1	*offset*										

2.2.4.19 Desvio se igual

Assembly: `BEQ` ***offset***

A instrução *BEQ* recebe um valor numérico de 11 bits, $offset$. Caso a última operação realizada pela ULA tiver resultado zero (ver instrução `CMP`, secções 2.2.5.2 e 2.2.5.3) o processador então atribui ao registrador PC o valor $(PC + offset \cdot 2)$, para assim avançar ou retroceder $offset$ instruções. Dessa forma é alterado o fluxo de execução de instruções. A tabela 26 exibe a função de cada um dos 16 bits desta instrução.

Tabela 26: Definição dos bits da instrução `BEQ`, desvio se igual.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
1	0	0	1	0	*offset*										

2.2.4.20 Desvio para rotina se igual

Assembly: `BEL` ***offset***

A instrução *BEL* recebe um valor numérico de 11 bits, $offset$. Caso a última operação realizada pela ULA tiver resultado zero (ver instrução `CMP`, secções 2.2.5.2 e 2.2.5.3) o processador então atribui ao registrador RET_ADDR o valor de PC e ao registrador PC o valor de $(PC + offset \cdot 2)$, para assim avançar ou retroceder $offset$ instruções. Dessa forma é alterado o fluxo de execução de instruções, mas fica armazenado o endereço da instrução atual. A tabela 27 exibe a função de cada um dos 16 bits desta instrução.

Tabela 27: Definição dos bits da instrução `BEL`, desvio para rotina se igual.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
1	0	0	1	1	*offset*										

2.2.4.21 Desvio se menor

Assembly: `BLT` ***offset***

A instrução *BLT* recebe um valor numérico de 11 bits, $offset$. Caso a última operação realizada pela ULA tiver resultado negativo (ver instrução `CMP`, secções 2.2.5.2

e 2.2.5.3) o processador então atribui ao registrador PC o valor $(PC + offset \cdot 2)$, para assim avançar ou retroceder $offset$ instruções. Dessa forma é alterado o fluxo de execução de instruções. A tabela 28 exibe a função de cada um dos 16 bits desta instrução.

Tabela 28: Definição dos bits da instrução `BLT`, desvio se menor.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
1	0	1	0	0	*offset*										

2.2.4.22 Desvio para rotina se menor

Assembly: `BLL` ***offset***

A instrução BLL recebe um valor numérico de 11 bits, $offset$. Caso a última operação realizada pela ULA tiver resultado negativo (ver instrução `CMP`, secções 2.2.5.2 e 2.2.5.3) o processador então atribui ao registrador RET_ADDR o valor de PC e ao registrador PC o valor de $(PC + offset \cdot 2)$, para assim avançar ou retroceder $offset$ instruções. Dessa forma é alterado o fluxo de execução de instruções, mas fica armazenado o endereço da instrução atual. A tabela 29 exibe a função de cada um dos 16 bits desta instrução.

Tabela 29: Definição dos bits da instrução `BLL`, desvio para rotina se menor.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
1	0	1	0	1	*offset*										

2.2.4.23 Desvio se maior

Assembly: `BGT` ***offset***

A instrução BGT recebe um valor numérico de 11 bits, $offset$. Caso a última operação realizada pela ULA tiver resultado positivo (ver instrução `CMP`, secções 2.2.5.2 e 2.2.5.3) o processador então atribui ao registrador PC o valor $(PC + offset \cdot 2)$, para assim avançar ou retroceder $offset$ instruções. Dessa forma é alterado o fluxo de execução de instruções. A tabela 30 exibe a função de cada um dos 16 bits desta instrução.

Tabela 30: Definição dos bits da instrução `BGT`, desvio se maior.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
1	0	1	1	0	*offset*										

2.2.4.24 Desvio para rotina se maior

Assembly: `BGL` ***offset***

A instrução BGL recebe um valor numérico de 11 bits, $offset$. Caso a última operação realizada pela ULA tiver resultado positivo (ver instrução `CMP`, secções 2.2.5.2

e 2.2.5.3) o processador então atribui ao registrador RET_ADDR o valor de PC e ao registrador PC o valor de $(PC + offset \cdot 2)$, para assim avançar ou retroceder $offset$ instruções. Dessa forma é alterado o fluxo de execução de instruções, mas fica armazenado o endereço da instrução atual. A tabela 31 exibe a função de cada um dos 16 bits desta instrução.

Tabela 31: Definição dos bits da instrução `BGL`, desvio para rotina se maior.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
1	0	1	1	1	*offset*										

Assembly: `RET`

A instrução RET recebe o endereço de RET_ADDR_H, do banco de registradores 1 e RET_ADDR_L, do banco de registradores 0, estes endereços devem ser iguais e estão representados na instrução por $ADDR\_RET$. O processador atribui ao registrador PC o valor de RET_ADDR. A tabela 32 exibe a função de cada um dos 16 bits desta instrução.

Tabela 32: Definição dos bits da instrução `RET`, retorno de rotina.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
1	1	0	0	0	X	X	X	X	X	X	X	X	*ADDR_RET*		

2.2.4.25 Armazenar registrador na memória de dados com *offset* imediato

Assembly: `STR Rd, offset`

A instrução STR com imediato recebe um registrador, Rd, e um valor numérico de 8 bits, $offset$. O processador atribui ao byte de endereço $(DPTR + offset)$ da memória de dados o valor de Rd (o registrador Rd é do banco 0). A tabela 33 exibe a função de cada um dos 16 bits desta instrução.

Tabela 33: Definição dos bits da instrução `STR`, mover valor de Rd para a posição da memória de dados apontado por $DPTR + offset$.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
1	1	0	0	1	*offset*								Rd		

2.2.4.26 Armazenar registrador na memória de dados com *offset* em registrador

Assembly: `STR Rd, Rm`

A instrução STR recebe dois registradores, Rm e Rd. O processador atribui ao byte de endereço $(DPTR + Rm)$ da memória de dados o valor de Rd (os registradores Rm e Rd são do banco 0). A tabela 34 exibe a função de cada um dos 16 bits desta instrução.

Tabela 34: Definição dos bits da instrução `STR`, mover valor de *Rd* para a posição da memória de dados apontado por $DPTR + Rm$.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
1	1	0	1	0	X	X	X	X	X	Rm			Rd		

2.2.4.27 Carregar byte da memória de dados com *offset* imediato

Assembly: `LDB Rd, offset`

A instrução *LDB* com imediato recebe um registrador, *Rd* e um valor numérico de 8 bits, $offset$. O processador atribui a *Rd* o valor do byte de endereço ($DPTR + offset$) da memória de dados (o registrador *Rd* é do banco *0*). A tabela 35 exibe a função de cada um dos 16 bits desta instrução.

Tabela 35: Definição dos bits da instrução `LDB`, mover valor do endereço $DPTR + offset$ da memória da dados para o registrador *Rd*.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
1	1	0	1	1	*offset*								Rd		

2.2.4.28 Carregar byte da memória de dados com *offset* em registrador

Assembly: `LDB Rd, Rm`

A instrução *LDB* recebe dois registradores, *Rm* e *Rd*. O processador atribui a *Rd* o valor byte de endereço ($DPTR + Rm$) da memória de dados (os registradores *Rm* e *Rd* são do banco *0*). A tabela 36 exibe a função de cada um dos 16 bits desta instrução.

Tabela 36: Definição dos bits da instrução `LDB`, mover valor de da posição $DPTR + Rm$ da memória de dados para o registrador *Rd*.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
1	1	1	0	0	X	X	X	X	X	Rm			Rd		

2.2.4.29 Nenhuma operação

Assembly: `NOP`

A instrução *NOP* (*No Operation*) não recebe nenhum parâmetro. O processador não realiza nenhuma escrita. A instrução é utilizada para resolver conflitos de dependência devido ao *pipeline*. A tabela 37 exibe a função de cada um dos 16 bits desta instrução.

Tabela 37: Definição dos bits da instrução `NOP`.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
1	1	1	1	1	X	X	X	X	X	X	X	X	X	X	X

2.2.5 Definição das Pseudo-Instruções

As pseudo-instruções são variações de instruções existentes e são tratadas como se fossem uma operação diferente (PATTERSON; HENNESSY, 2013). Como estas instruções são apenas variações, elas não são implementada em hardware. Geralmente são utilizadas para facilitar a programação em assembly.

2.2.5.1 Mover registrador para registrador

Assembly: `MOV Rd, Rm`

A instrução de mover o valor de um registrador para outro utiliza o mesmo código de operação da instrução de deslocamento para esquerda (ver seção 2.2.4.1). A diferença é que o imediato que representa a quantidade de deslocamento é zero, sendo assim, o valor de *Rm* é copiado em *Rd*.

2.2.5.2 Comparação de registradores

Assembly: `CMP Rd, Rm, Rn`

Esta instrução compara o valor de dois registradores, *Rm* e *Rn*, e utiliza o mesmo código de operação da instrução de subtração (ver seção 2.2.4.5). A comparação é realizada por meio de uma subtração, dependendo do resultado desta operação (positivo, zero ou negativo) uma *flag* diferente é acionada pela ULA. O resultado desta operação de subtração é armazenado em *Rd*.

2.2.5.3 Comparação de registrador com imediato

Assembly: `CMP Rd, #imediato`

Esta instrução compara o valor de um registrador, *Rd*, com um valor imediato, ela utiliza o mesmo código de operação da instrução de subtração (ver seção 2.2.4.6). A comparação é realizada por meio de uma subtração, dependendo do resultado desta operação (positivo, zero ou negativo) uma *flag* diferente é acionada pela ULA. O resultado desta operação de subtração é armazenado em *Rd*.

3 IMPLEMENTAÇÃO

3.1 Ferramentas utilizadas

Nesta seção são descritas brevemente as ferramentas utilizadas ao longo deste documento.

- LeonardoSpectrum

  Ferramenta da empresa *Mentor Graphics* que realiza síntese de alto nível. Nela entra-se com uma descrição do circuito em VHDL (*VHSIC Hardware Description Language*), Verilog ou EDIF (*Electronic Design Interchange Format*) e é sintetizado um circuito para FPGAs (*Field-Programmable Gate Array*), CPLDs (*Complex Programmable Logic Device*) ou ASICs (*Application-Specific Integrated Circuit*) (Mentor Graphics Company, 2001). Neste projeto foi utilizado para extrair estimativas de caminho crítico e sintetizar o circuito na biblioteca padrão da tecnologia CMOS de $0,35\mu m$.

- ModelSim SE

  Ferramenta da empresa *Mentor Graphics* que realiza a simulação de circuitos descritos VHDL, Verilog ou EDIF (Mentor Graphics,). Neste projeto foi utilizado para verificar a coerência do código criado com a microarquitetura e desta com a arquitetura proposta.

- IC Station

  Também da *Mentor Graphics*, permite criar *layouts* de circuitos integrados com base no esquemático e utilizar as ferramentas do Calibre para checar as se o *layout* atende as regras de construção e conferir se está de acordo com o esquemático (Mentor Graphics,). Neste projeto foi utilizado para criar o *layout* do somador de 16 bits.

- Design Architect

  Permite criar os esquemáticos dos circuitos lógicos com células pré-projetadas ou com elementos primários, como transistores. Também é possível extrair o *netlist* do esquemático para simulações elétricas (Mentor Graphics,). Neste trabalho é utilizado para importar a descrição estrutural dos blocos codificados em VHDL, gerada pelo LeonardoSpectrum.

- Calibre

  Permite verificações do *layout* do circuito integrado, como o DRC (*Design Rules Check*) (Mentor Graphics,), que confere se o tamanho e espaçamento das camadas está seguindo as regras do fabricante, e o LVS (*Layout Versus Schematic*) (Mentor

Graphics,), que confere se o *layout* está de acordo com o esquemático projetado. Neste trabalho é utilizado para realizar tais verificações.

3.2 Microarquitetura

A microarquitetura é a organização do processador, incluindo as unidades funcionais e a conexão entre elas, o chamado *DataPath*, e a Unidade de Controle, que gera sinais para controle do *DataPath* (PATTERSON; HENNESSY, 2013). Para projetar a microarquitera exibida na figura 9, todas as instruções da seção 2.2.4 foram analisadas, para assim, definir todos os blocos funcionais necessários e a ligação entre eles.

Na microarquitetura proposta utilizamos cinco estágios de *pipeline*, *IF* (*Instruction Fetch*), *ID* (*Instruction Decode*), *EX* (*Execution*) , *MEM* (*Memory*) e *WB* (*Write Back*), em que são realizadas as operações de busca da instrução, decodificação da instrução e busca dos operandos, execução da instrução, acesso à memória e escrita do resultado. Os estágios do *pipeline* são separadas pelos retângulos marcados com *IF/ID*, *ID/EX*, *EX/MEM* e *MEM/WB*. Estes retângulos representam registradores implementados em flip-flop, sensíveis à borda de descida de *clock*.

Praticamente todas as instruções do processador podem ser classificadas em três categorias, as de *Desvio*, as com *Registrador* e as com *Imediato*. As instruções classificadas como de desvio são as que alteram o valor do registrador *PC*. As instruções com registrador são aquelas realizadas com o valor de dois registradores. Por fim, as instruções com imediato são as operações realizadas com o valor de um registrador e um valor numérico incluso na própria instrução. A instrução `NOP` (seção 2.2.4.29), que não realiza operação alguma, é a única que não entra em nenhuma das três categorias. Esta classificação em categorias é refletida no projeto da unidade de controle, pois os sinais de controle de operações da mesma categoria se assemelham.

A seguir, será descrito o funcionamento dos blocos que compõe a microarquitetura do processador.

3.2.1 Multiplexadores

O multiplexador digital, ou seletor de dados, é um circuito lógico que recebe diversos dados digitais de entrada e seleciona um, em determinado instante, para transferi-lo para a saída. O envio do dado de entrada desejado para saída é controlado pelas entradas de *seleção* (frequentemente denominadas *endereço*) (TOCCI; WIDMER, 2003). A figura 3 exibe a representação de um multiplexador genérico. Cada linha de seleção tem apenas um bit de comprimento, entretanto, cada entrada e cada saída podem ter mais de um, com a condição de que sejam do mesmo tamanho.

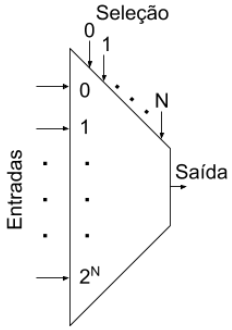


Figura 3: Representação de um multiplexador genérico, com N linhas de seleção, 2^N entradas e uma saída.

Para elaborar a microarquitetura do processador foram necessários 12 multiplexadores, de *M00* a *M11*. O sinal de controle deles é gerado pela unidade de controle e depende da instrução a ser executada. A função de cada multiplexador está listada abaixo.

- **M00**: Duas entradas e uma saída de 16 bits e apenas um bit de seleção. Define o valor de PC. Se o sinal de seleção for 1, o programa mantém o fluxo normal, ou seja, o registrador PC terá valor igual a $(PC + 2)$. Caso o sinal de seleção tenha valor 0, é executado um salto, então o registrador PC terá o valor do endereço da instrução a qual o salto foi direcionado.

- **M01**: Duas entradas e uma saída de 8 bits e apenas um bit de seleção. Define o valor que será escrito no banco de registradores 1. Se o sinal de seleção for 1, o programa executou um salto para rotina, então o byte mais significativo de PC deve ser salvo no banco de registradores 1, para que o programa possa retornar ao fluxo normal em algum ponto. Caso o sinal de seleção tenha valor 0, não ocorreu um salto para rotina, então deve-se apenas salvar o resultado da operação no banco de registradores 1.

- **M02**: Duas entradas e uma saída de 3 bits e apenas um bit de seleção. Define o endereço do registrador que será escrito no banco de registradores 1. Se o sinal de seleção for 1, o programa executou um salto para rotina, então o byte mais significativo de PC deve ser salvo no registrador *RET_ADDR_H* do banco de registradores 1, para isso, a saída do multiplexador é o endereço de *RET_ADDR_H*. Caso o sinal de seleção tenha valor 0, não ocorreu um salto para rotina, então deve-se apenas salvar o resultado da operação no endereço de *Rd* (bits de 2 à 0 da instrução) do banco de registradores 1.

- **M03**: Três entradas e uma saída de 3 bits e dois bits de seleção. Define o endereço de um dos registradores que será lido do banco de registradores 1. Se o sinal de seleção for 2, o programa executará uma operação de deslocamento com um operando do banco 1, deve-se ler o registrador que o endereço é composto bits de 5 à 3 da

instrução. Caso o sinal de seleção tenha valor 1, o programa está executando uma operação de retorno de rotina, deve-se ler o registrador *RET_ADDR_H*, cujo o endereço está nos bits de 2 à 0 da instrução. Caso o sinal tenha valor 0, o programa está realizando um acesso à memória, o endereço a ser lido deve ser o de *DPTR_L*.

- **M04**: Duas entradas e uma saída de 8 bits e apenas um bit de seleção. Define o valor que será escrito no banco de registradores 0. Se o sinal de seleção for 1, o programa executou um salto para rotina, então o byte menos significativo de *PC* deve ser salvo no banco de registradores 0, para que o programa possa retornar ao fluxo normal em algum ponto. Caso o sinal de seleção tenha valor 0, não ocorreu um salto para rotina, então deve-se apenas salvar o resultado da operação no banco de registradores 0.

- **M05**: Duas entradas e uma saída de 3 bits e apenas um bit de seleção. Define o endereço do registrador que será escrito no banco de registradores 0. Se o sinal de seleção for 1, o programa executou um salto para rotina, então o byte mais significativo de *PC* deve ser salvo no registrador *RET_ADDR_L* do banco de registradores 0, para isso, a saída do multiplexador é o endereço de *RET_ADDR_L*. Caso o sinal de seleção tenha valor 0, não ocorreu um salto para rotina, então deve-se apenas salvar o resultado da operação no endereço de *Rd* (bits de 2 à 0 da instrução) do banco de registradores 0.

- **M06**: Duas entradas e uma saída de 3 bits e apenas um bit de seleção. Define o endereço de um dos registradores que será lido do banco de registradores 0. Se o sinal de seleção for 1, o programa está executando uma operação com dois registradores, então deve-se buscar o operando *Rm*, cujo o endereço é composto pelos bits de 8 à 6 da instrução. Caso o sinal de seleção tenha valor 0, o programa está executando uma operação com um registrador e um imediato, dendo assim, deve-se buscar o operando *Rd*, cujo o endereço é composto pelos bits de 2 à 0 da instrução.

- **M07**: Duas entradas e uma saída de 8 bits e apenas um bit de seleção. Define qual será o banco de registrador do primeiro operando da ULA. Se o sinal de seleção for 1, o operando deve ser do banco de registradores 1. Caso contrário, ele será do banco de registradores 0.

- **M08**: Duas entradas e uma saída de 8 bits e apenas um bit de seleção. Define qual será o banco de registrador do segundo operando da ULA. Se o sinal de seleção for 1, o operando deve ser do banco de registradores 1. Caso contrário, ele será do banco de registradores 0.

- **M09**: Duas entradas e uma saída de 8 bits e apenas um bit de seleção. Define se o segundo operando da ULA será um dado lido do banco de registradores, ou um valor numérico contido na instrução (imediato). Se o sinal de seleção for 1, o operando será

composto dos bits 10 à 3 da instrução. Caso contrário, ele será o dado selecionado pelo multiplexador *M08*.

- **M10**: Duas entradas e uma saída de 16 bits e apenas um bit de seleção. Define o valor de PC em caso de saltos e retorno de rotina. Se o sinal de seleção for 1, o programa estará realizando uma operação de retorno de rotina, portanto, o valor de PC deve ser de RET_ADDR. Caso contrário, o valor de PC será $PC + (offset \cdot 2)$.

- **M11**: Duas entradas e uma saída de 8 bits e apenas um bit de seleção. Define a origem do valor a ser escrito no banco de registradores, ULA ou memória de dados. Se o sinal de seleção for 1, o valor a ser escrito é proveniente da memória de dados. Caso contrário, é o resultado da ULA.

3.2.2 Somadores

O somador digital é um componente capaz de efetuar a soma aritmética de dois valores numéricos em formato binário. Este bloco recebe duas entradas de comprimento igual e tem uma saída do mesmo comprimento das entradas, em alguns casos, o somador também pode ter uma saída de *carry*, de apenas um bit, responsável por sinalizar o *overflow*, ou seja, se o resultado da adição excede o tamanho da saída. A figura 4 exibe a representação de um somador.

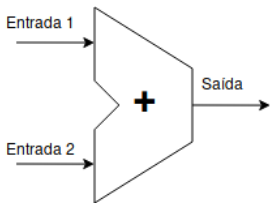
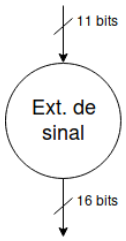


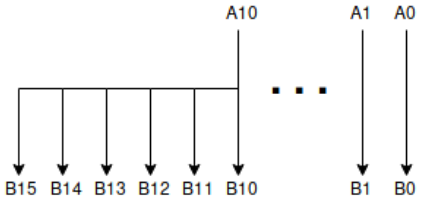
Figura 4: Representação de um somador com duas entradas e uma saída.

3.2.3 Extensor de sinal

O extensor de sinal é utilizado para converter um valor numérico de 11 bits em um de 16, para que assim, possa utilizá-lo em operações aritméticas com outros valores de 11 bits. Assume-se que o valor de 11 bits está representado em complemento de dois (método para representação de números negativos no sistema binário), então, para que o valor de 16 bits mantenha o sinal, é necessário que os bits mais significativos da saída sejam idênticos ao bit mais significativo da entrada. A figura 5 exibe a visão externa do bloco extensor de sinal (5a) e a visão da implementação deste bloco (5b).



(a) Visão externa do bloco de extensão de sinal. O bloco tem um entrada de 11 bits e uma saída de 16.



(b) Implementação do bloco de extensão de sinal. Entradas indicadas pelo prefixo A e saídas pelo prefixo B, sendo os bits $A0$ e $B0$ os menos significativos e $A10$ e $B15$ os mais significativos.

Figura 5: Visão externa do bloco de extensão de sinal na figura 5a e visão interna na figura 5b.

3.2.4 Banco de registradores

O banco de registradores é o bloco que armazena os operandos da maioria das instruções do processador. Cada banco de registradores armazena 8 registradores de 8 bits, cada um desses registradores tem um endereço de 3 bits. Este bloco tem duas entradas de endereço (*End 1* e *End 2*), de 3 bits cada, e quando o sinal de *clock* (*CLK*) está em nível alto, o valor dos registradores com endereço correspondente ao valor destas entradas é reproduzido nas saídas de dados *Dado 1* e *Dado 2*, respectivamente. Quando o sinal de *clock* (*CLK*) está em nível lógico baixo e o sinal de habilitação de escrita, *WR_REG*, está em nível alto, o valor do registrador endereçado pelo sinal *End Escrita* é alterado para o valor na entrada *Dado escrita*. Caso o sinal *WR_REG* esteja em nível lógico baixo nenhuma escrita é realizada. A figura 6 exibe a representação do banco de registradores.

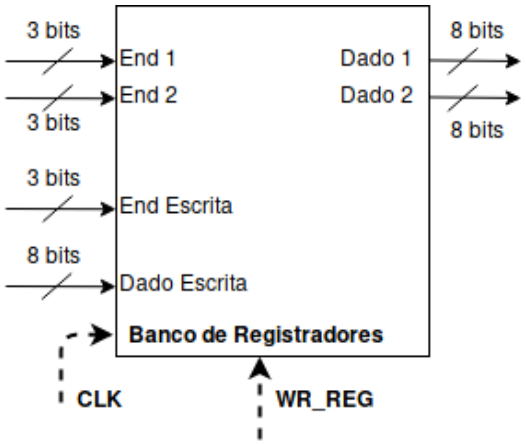


Figura 6: Representação do banco de registradores. As entradas *End 1* e *End 2* definem o valor das saídas *Dado 1* e *Dado 2*, respectivamente. O sinal *WR_REG* habilita ou desabilita a escrita. A entrada *End Escrita* define o registrador que receberá o valor da entrada *Dado Escrita* caso a escrita esteja habilitada. E o sinal *CLK* é o *clock* do processador, utilizado para sincronizar este bloco com os demais.

3.2.5 Unidade Lógica e Aritmética

A Unidade Lógica e Aritmética, ou ULA, como o nome diz, é bloco responsável por realizar as operações aritméticas e lógicas. Este bloco recebe dois operandos de 8 bits, *op1* e *op2*, um valor de 3 bits para operações de deslocamento, *shift_amnt*, e o código da operação a ser realizada, *ULA_OP*, de 4 bits. E tem duas saídas, *out*, de 8 bits que é o valor resultante da operação realizada e *FLAGS*, de 3 bits, com algumas informações sobre o resultado. A tabela 38 apresenta a função de cada bit da saída *FLAGS*, enquanto a tabela 39 exibe a relação entre a saída *out* e o valor de *ULA_OP*. A figura 7 exibe a representação da ULA.

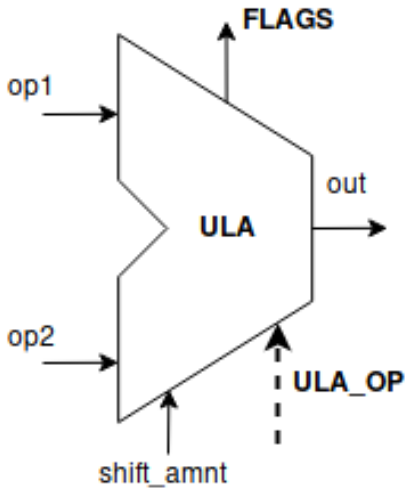


Figura 7: Representação da ULA. As entradas *op*1, *op*2 tem 8 bits de comprimento, *shift_amnt* tem 3, *ULA_OP* tem 4. A saída *FLAGS* tem 3 bits, e *out* tem 8. A saída *out* é o resultado da operação definida pela entrada *ULA_OP*, enquanto a saída *FLAGS* fornece algumas informações sobre o resultado.

Tabela 38: Informação fornecida por cada bit da saída *FLAGS* da ULA. O bit mais significativo tem valor 1 quando o resultado da operação é maior do que o tamanho da saída, o bit 2 tem valor 1 quando o resultado é 0 e o bit 0 tem valor 1 quando o resultado da operação é negativo, ou seja, quando o bit mais significativo de *out* tem valor 1.

2	1	0
overflow	zero	negativo

Tabela 39: Relação entre o código de operação da ULA (*ULA_OP*), a operação realizada e o valor da saída (*OUT*).

ULA_OP	**Operação**	**OUT**
0	Deslocamento para esquerda	*op2* deslocada *shift_amnt* bits para esquerda
1	Deslocamento para direita	*op2* deslocada *shift_amnt* bits para direita
2	Soma	$op1 + op2$
3	Subtração	$op1 - op2$
4	Soma com *carry*	$op1 + op2 + 1$
5	*E* bit-a-bit	*op1 AND op2*
6	*OU* bit-a-bit	*op1 OR op2*
7	*Ou Exclusivo* bit-a-bit	*op1 XOR op2*
8	Nenhuma operação	*op2*
9	*Negação* bit-a-bit	*NOT op2*

3.2.6 Unidade de controle

A Unidade de Controle do processador se encarrega de garantir que os dados certos sejam selecionados pelos multiplexadores, que a ULA realize a operação desejada e que as escritas e leituras sejam efetuadas. Para isto, ela conta com três entradas: o código da operação (*opcode*), de 5 bits, a seleção do banco de registradores dos operando (*reg_sel*), de 2 bits e as *flags* da ULA (*FLAGS*), de 3 bits. E 12 saídas de controle, que são listadas e descritas abaixo. A representação da Unidade de Controle pode ser vista na figura 8. A entrada das *flags* é atrasada em um ciclo de *clock* com um registrador implementado em flip-flop sensível a borda de descida, para que assim, seja possível utilizar as *flags* da última operação na decisão de saltos. Com a microarquitetura projetada (figura 9) e analisando o caminho dos dados de cada instrução, gerou-se a tabela 40. Cada linha desta tabela é uma instrução e cada coluna um sinal da Unidade de Controle. As instruções estão divididas em categorias, como mencionado anteriormente e os sinais estão divididos pelo estágio do *pipeline* em que estão aplicados.

- **PC_SRC**: Sinal de seleção do multiplexador *M00*, de apenas um bit. Este sinal recebe nível lógico alto em caso de saltos, e nível baixo nas demais operações.

- **WR_RET**: Sinal de seleção dos multiplexadores *M01*, *M02*, *M04* e *M05*, de apenas um bit. Este sinal recebe nível lógico alto no caso em que algum salto para rotina foi efetuado, e nível baixo nas demais operações.

- **REG1_SRC**: Sinal de seleção do multiplexador *M03*, de 2 bits. Este sinal recebe valor 2 quando o operando *Rm* da instrução é do banco de registradores 1, recebe 1 na instrução de retorno de rotina e recebe 0 em acessos a memória.

- **REG0_SRC**: Sinal de seleção do multiplexador *M06*, de apenas um bit. Este sinal recebe nível lógico alto quando é executada alguma operação entre dois registradores, e recebe nível lógico baixo quando a operação envolve um valor numérico imediato.

- **WR_REG1**: Sinal de habilitação de escrita do banco de registradores 1, de apenas um bit. Este sinal recebe nível lógico alto quando é permitida a escrita de dados no banco de registradores 1 e recebe nível lógico baixo caso contrário.

- **WR_REG0**: Sinal de habilitação de escrita do banco de registradores 0, de apenas um bit. Este sinal recebe nível lógico alto quando é permitida a escrita de dados no banco de registradores 0 e recebe nível lógico baixo caso contrário.

- **DATA1**: Sinal de seleção do multiplexador *M08*, de apenas um bit. Este sinal recebe nível lógico alto quando o segundo operando da ULA deve vir do banco de registradores 1 e recebe nível lógico baixo caso contrário.

- **DATA0**: Sinal de seleção do multiplexador *M09*, de apenas um bit. Este sinal recebe nível lógico alto quando o primeiro operando da ULA deve vir do banco de registradores 0 e recebe nível lógico baixo caso contrário.

- **ALU_SRC**: Sinal de seleção do multiplexador *M10*, de apenas um bit. Este sinal recebe nível lógico alto quando o segundo operando da ULA é um valor numérico inserido na instrução e recebe nível lógico baixo caso contrário.

- **ALU_OP**: Sinal de seleção da operação da ULA, de 4 bits. Este sinal recebe um valor entre 0 e 9, seguindo a tabela 39, que indica para a ULA a operação necessária para a execução da instrução.

- **JMP_SRC**: Sinal de seleção de multiplexador *M10*, de apenas um bit. Este sinal recebe nível lógico alto caso o salto seja um retorno de uma rotina e recebe nível lógico baixo caso contrário.

- **WR_MEM**: Sinal de habilitação de escrita da memória de dados, de apenas um bit. Este sinal recebe nível lógico alto caso a instrução execute uma operação de escrita na memória de dados, e recebe nível lógico baixo caso contrário.

- **WB_SRC**: Sinal de seleção do multiplexador *M11*, de apenas um bit. Este sinal recebe nível lógico alto quando o dado a ser escrito no banco de registradores é proveniente da memória de dados, ou seja, quando a instrução é de leitura da memória, e recebe nível lógico baixo caso o dado a ser escrito é o resultado da ULA.

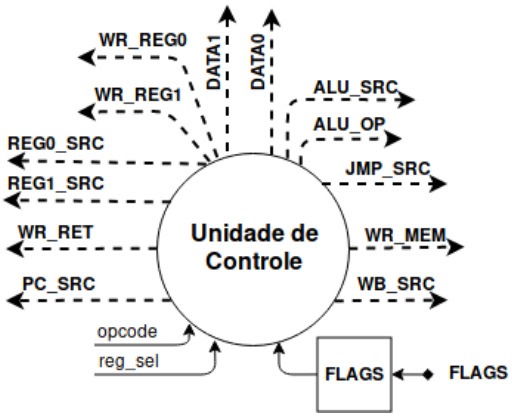


Figura 8: Representação da Unidade de Controle.

Tabela 40: Definição dos sinais da unidade de controle para cada instrução. Os sinais que não interferem na execução da operação estão marcados com *X*. Nas instruções de salto condicional, é indicado o sinal em caso verdadeiro (V) e falso (F). As instruções de deslocamento, que tem opção de banco de registrador para *Rm* e *Rd* (ver seções 2.2.4.1 e 2.2.4.2) estão indicadas com $Bm \rightarrow Bd$.

		IF	ID				EX			MEM	WB			
Tipo	**Instrução**	**PC_SRC**	**REG0_SRC**	**REG1_SRC**	**DATA1**	**DATA2**	**ALU_SRC**	**JMP_SRC**	**ALU_OP**	**WR_MEM**	**WB_SRC**	**WR_REG0**	**WR_REG1**	**WR_RET**
Desvio	**JMP**	1	X	X	X	X	X	0	X	0	X	0	0	0
	JAL	1	X	X	X	X	X	0	X	0	X	1	1	1
	BEQ (V/F)	1/0	X	X	X	X	X	0	X	0	X	0	0	0
	BEL (V/F)	1/0	X	X	X	X	X	0	X	0	X	1/0	1/0	1
	BLT (V/F)	1/0	X	X	X	X	X	0	X	0	X	0	0	0
	BLL (V/F)	1/0	X	X	X	X	X	0	X	0	X	1/0	1/0	1
	BGT (V/F)	1/0	X	X	X	X	X	0	X	0	X	0	0	0
	BGL (V/F)	1/0	X	X	X	X	X	0	X	0	X	1/0	1/0	1
	RET	1	0	1	0	1	X	1	X	0	X	0	0	0
Imediato	**LSL 0 → 0**	0	X	X	X	0	0	X	0	0	0	1	0	0
	LSL 0 → 1	0	X	X	X	0	0	X	0	0	0	0	1	0
	LSL 1 → 0	0	X	2	X	1	0	X	0	0	0	1	0	0
	LSL 1 → 1	0	X	2	X	1	0	X	0	0	0	0	1	0
	LSR 0 → 0	0	X	X	X	0	0	X	1	0	0	1	0	0
	LSR 0 → 1	0	X	X	X	0	0	X	1	0	0	0	1	0
	LSR 1 → 0	0	X	2	X	1	0	X	1	0	0	1	0	0
	LSR 1 → 1	0	X	2	X	1	0	X	1	0	0	0	1	0
	ADD	0	0	X	0	X	1	X	2	0	0	1	0	0
	SUB	0	0	X	0	X	1	X	3	0	0	1	0	0
	ADC	0	0	X	0	X	1	X	4	0	0	1	0	0
	AND	0	0	X	0	X	1	X	5	0	0	1	0	0
	OR	0	0	X	0	X	1	X	6	0	0	1	0	0
	XOR	0	0	X	0	X	1	X	7	0	0	1	0	0
	MOV	0	0	X	0	X	1	X	8	0	0	1	0	0
	STR	0	0	0	1	X	1	X	2	1	X	0	0	0
	LDB	0	X	0	1	X	1	X	2	0	1	1	0	0
Registrador	**ADD**	0	1	X	0	0	0	X	2	0	0	1	0	0
	SUB	0	1	X	0	0	0	X	3	0	0	1	0	0
	ADC	0	1	X	0	0	0	X	4	0	0	1	0	0
	AND	0	1	X	0	0	0	X	5	0	0	1	0	0
	OR	0	1	X	0	0	0	X	6	0	0	1	0	0
	XOR	0	1	X	0	0	0	X	7	0	0	1	0	0
	NOT	0	X	X	X	0	0	X	9	0	0	1	0	0
	STR	0	0	0	1	0	0	X	2	1	X	0	0	0
	LDB	0	X	0	X	0	0	X	2	0	1	1	0	0
	NOP	0	X	X	X	X	X	X	X	0	0	0	0	0

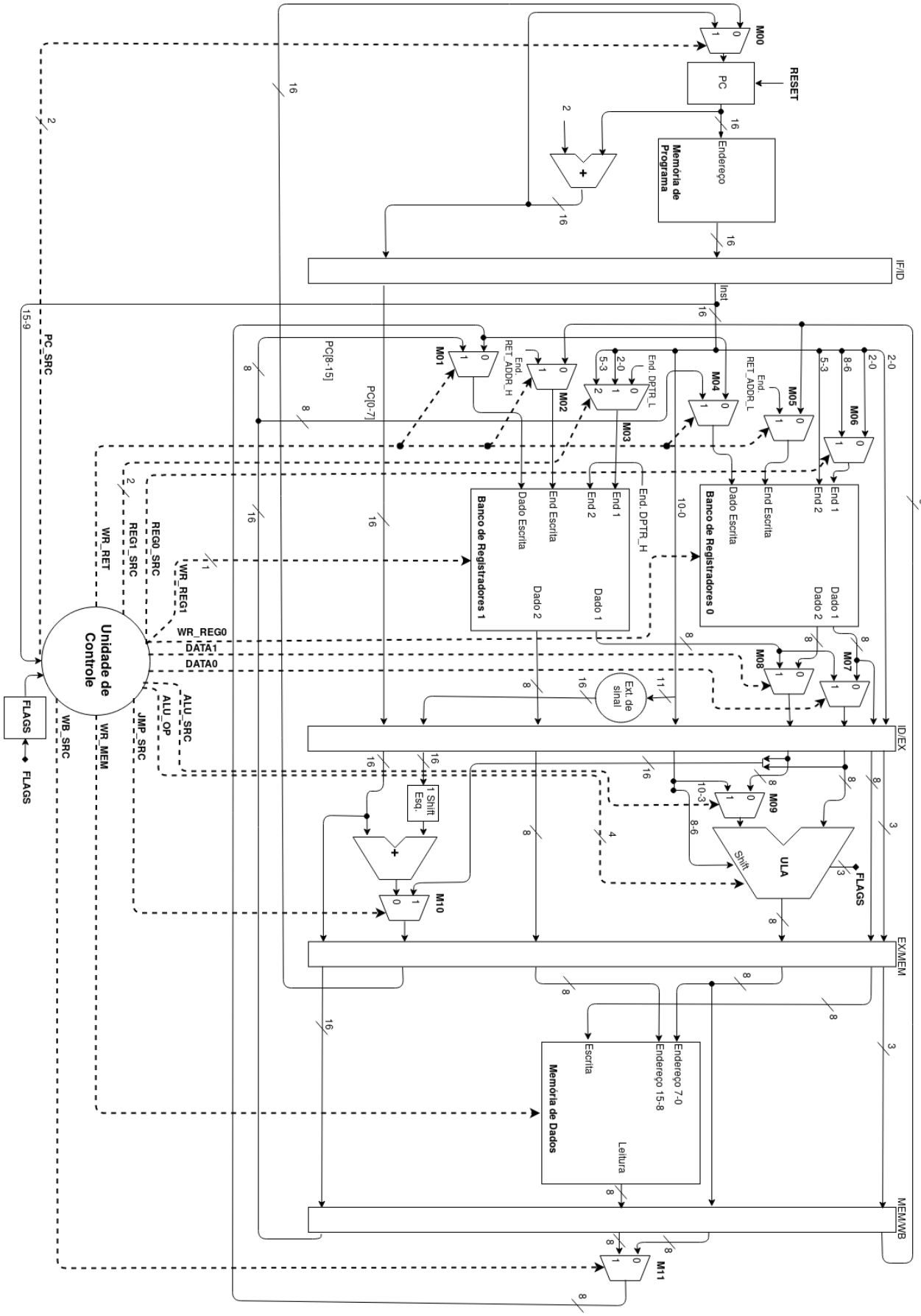


Figura 9: Microarquitetura do processador. Blocos funcionais do processador a ligação entre eles e os sinais de controle.

3.3 Descrição do hardware e simulação

Com a arquitetura e a microarquitetura do processador especificadas é possível descrever o circuito em uma linguagem de descrição de hardware. Pode-se, assim, realizar simulações para garantir que a descrição feita e a microarquitetura desenvolvida estão corretas em relação a arquitetura proposta e adquirir dados estimados sobre a performance do hardware. A linguagem escolhida para a descrição foi o VHDL.

Para descrever o processador, optou-se por descrever separadamente os blocos funcionais mais complexos e que se repetem mais de uma vez na microarquitetura e então integrá-los em outro código, assim, torna-se mais fácil o processo de simulação e de implementação em hardware, pois este processos poderão ser feitos em passos menores. O código VHDL se divide em quatro blocos, o somador de 16 bits, o banco de registradores, a ULA e a Unidade de Controle. Estes blocos são integrados pelo código do processador.

3.3.1 Somador de 16 bits

O código que descreve este bloco, em alto nível, está apresentado em A.1. Para verificar a descrição deste bloco foi realizada uma simulação que força a propagação da soma em todos os bits, colocando uma das entradas com todos os bits "1" (na base hexadecimal "FFFF") e a outra com o valor 1, e algumas outros valores aleatórios. Os resultados da simulação estão apresentados na figura 10. A tabela 41 apresenta as entradas da simulação, o valor esperado e o valor simulado, transcrito da imagem. Como podemos observar, para os casos de teste apresentado o somador se saiu como esperado, não havendo então, algum erro na descrição dele.

Tabela 41: Entradas de teste para a simulação do somador, *Entrada 1* (*input_1*) e *Entrada 2* (*input_2*), saída esperada da simulação e saída simulada.

Entrada 1	**Entrada 2**	**Saída esperada**	**Saída simulada**
0001	FFFF	0000	0000
70E1	338F	A470	A470
FFFF	0001	0000	0000
0000	FFFF	FFFF	FFFF
0001	338F	3390	3390
70E1	0001	70E2	70E2
FFFF	FFFF	FFFE	FFFE
0000	338F	338F	338F
0001	0001	0002	0002
70E1	FFFF	70E0	70E0

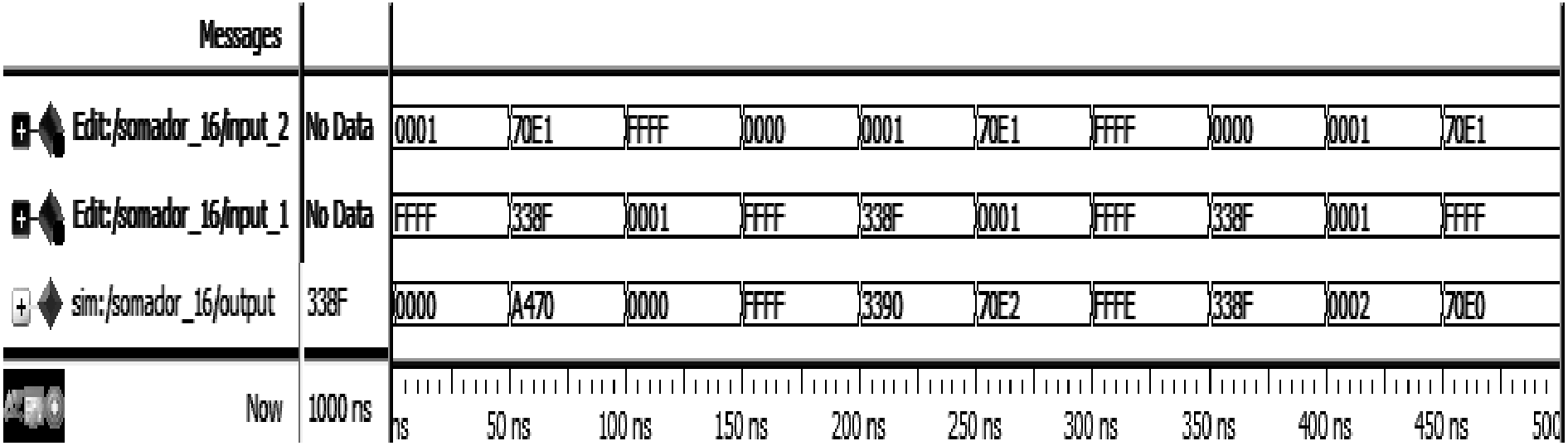


Figura 10: Resultado da simulação do somador de 16 bits. Os sinais *input_1* e *input_2* são entradas de teste e o sinal *output* é resultado da simulação. Os valores apresentado na simulação estão na base hexadecimal.

3.3.2 Banco de registradores

O código que descreve este bloco está apresentado em A.2. Para verificar a descrição deste bloco foram geradas seis sinais de entradas, o endereço de escrita (*wr_addr*), o primeiro endereço de leitura (*rd_addr1*), o segundo endereço de leitura (*rd_addr2*), o dado de escrita (*wr_data*), o sinal de *clock* (*clk*) e o sinal de habilitação de escrita (*wr_enable*). O formato de cada onda é descrito abaixo. Os resultados desta simulação estão apresentados na figura 11. Pela imagem podemos notar que na primeira metade do tempo de simulação, quando a escrita é habilitada, os valores escritos no banco de registradores são lidos com sucesso na saída *data1*, e o valor do registrador de endereço zero, lido na saída *data2* se mantém constante enquanto nenhuma escrita é realizada nele. Na segunda metade da simulação, quando a saída não é habilitada, o valor dos registradores não são mais alterados. Este resultado condiz com o funcionamento esperado do bloco, portanto podemos assumir que a descrição dele está coerente.

- **wr_addr**: Foi gerado um sinal que se inicia com valor zero e é incrementado a cada dois períodos de *clock*.
- **rd_addr1**: Foi gerado um sinal que se inicia com valor zero, mantém este valor por $100ns$ e então é incrementado a cada $50ns$.
- **rd_addr2**: Foi gerado um sinal constante em zero.
- **wr_data**: Foi gerado um sinal que assume um valor aleatório a cada 2 períodos de *clock*.
- **clk**: Foi gerado um sinal de *clock* com período de $20ns$.
- **wr_enable**: Foi gerado um sinal que começa em nível lógico alto e em $500ns$ (metade do tempo de simulação) é alterado para nível lógico baixo.

3.3.3 Unidade Lógica e Aritmética

O código que descreve este bloco, em alto nível, esta representado em A.3. Para verificar a descrição deste bloco foi realizada uma simulação com operandos aleatórios, todos os códigos de operação em sequência e um valor de deslocamento constante. Os resultados desta simulação estão apresentados na figura 12. Utilizando a tabela 39 para consultar qual operação equivale a cada código e a tabela 38 para verificar o significado de cada bit de *flags*, pode-se observar que o resultado de cada operação da ULA está condizente com o funcionamento esperado, assim, podemos concluir que a descrição deste bloco está correta.

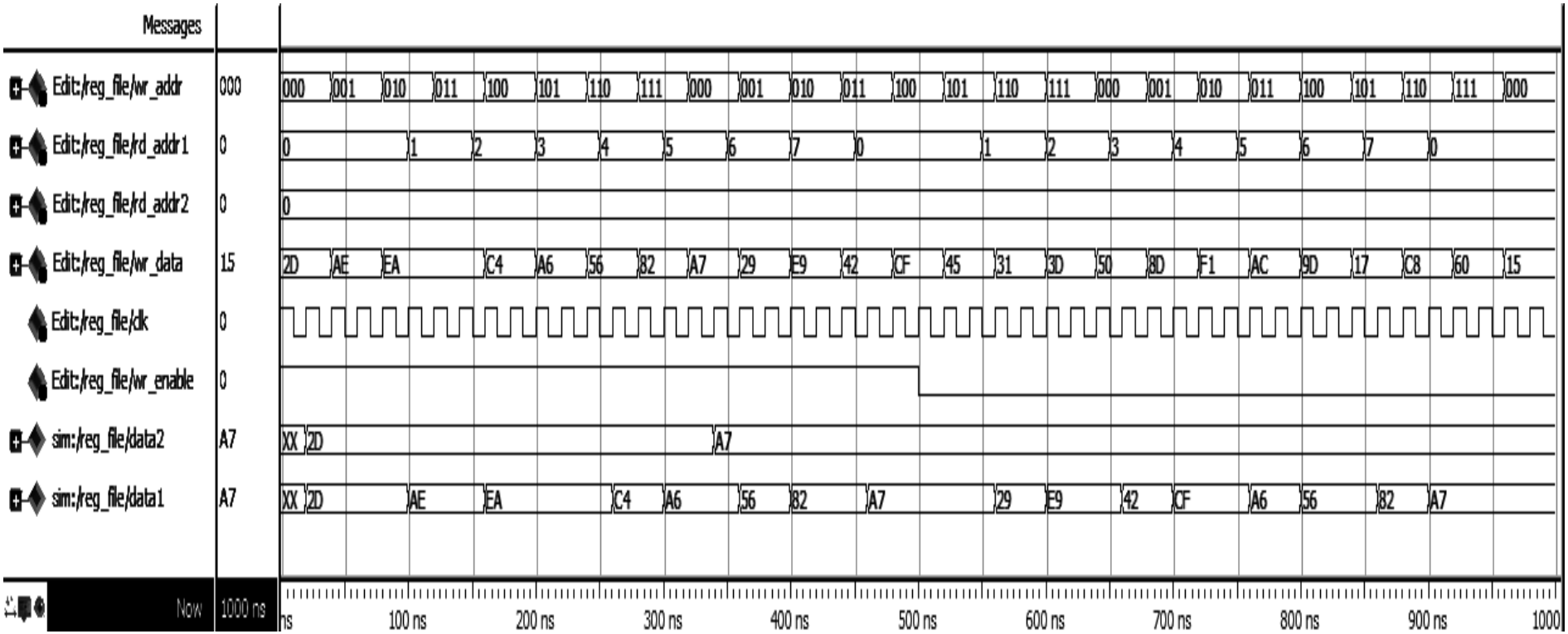


Figura 11: Sinais resultantes da simulação do banco de registradores com MODELSIM no tempo. O sinal *wr_addr* é o endereço de escrita (na base binária), *rd_addr1* e *rd_addr2* são endereços de leitura (na base hexadecimal), *wr_data* é o dado de escrita (na base hexadecimal). *clk* é o sinal de *clock*, *wr_enable* é o sinal para habilitar a escrita, todos estes são entradas de teste. Os sinais *data1* e *data2* são os dados lidos e são resultados da simulação (na base hexadecimal).

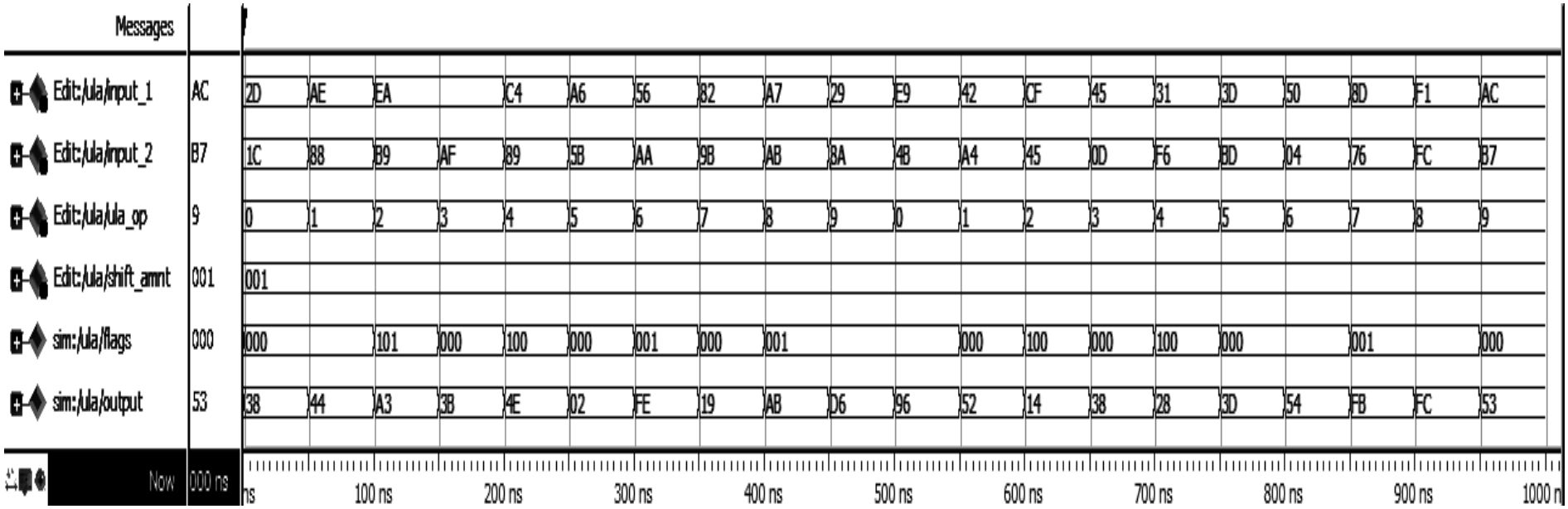


Figura 12: Resultado da simulação da ULA. Os sinais *input_1* e *input_2* são os operandos, o sinal *ula_op* é o código de operação e *shift_amnt* é o valor de deslocamento, todos estes são entradas de teste. Os sinais *output*, que é o resultado da operação, e *flags*, que são as *flags* resultantes, são resultados da simulação. Todos os valores estão representados na base hexadecimal, com exceção da entrada *shift_amnt* e da saída *flags* que estão na base binária.

3.3.4 Unidade de Controle

A descrição deste bloco está apresentada em A.4. Para verificar a descrição deste bloco foi realizada uma simulação com *flags* e bancos de registradores aleatórios, e todos os códigos de operação em ordem crescente. O resultado desta simulação está apresentado na figura 13. A saída desta simulação deve conter os mesmo valores da tabela 40 para cada sinal e cada instrução, porém, com um período de *clock* de defasagem para cara estágio do *pipeline*. É possível comprar a saída da simulação com a tabela 40 e observar que os sinais de controle estão sendo gerados corretamente, portanto, podemos assumir que a descrição deste bloco está correta.

3.3.5 Processador

A descrição deste bloco segue o diagrama da microarquitetura, figura 9, e o código é apresentado em A.5. Para simular o processador, foi criado um código simples para emular a memória de programa e que pode ser visto em A.6. O conteúdo dessa memória é um programa que escreve os valores 0, 1 e 2 nos primeiros endereços pares da memória de dados. O programa em *assembly* esta mostrado no código 3.1. Escolheu-se este programa por ele utilizar pelo menos uma instrução de cada categoria (desvio, imediato, registrador, e também acesso a memória), dessa forma aumenta-se a abrangência do teste. Foi criado um código em VHDL para descrever a conexão do processador com a memória de programa, este pode ser visto em A.6. O resultado da simulação pode ser observado na figura 14. Esta simulação consiste basicamente em uma escrita na memória de dados dentro de um laço de repetição, então, no resultado da simulação devemos observar estas operações. Para facilitar, foi inserido um “W” ou um “J” na parte superior da imagem, para indicar uma escrita ou salto, respectivamente. É possível observar que na primeira escrita a escrita na memória está habilitada, o endereço de escrita ($DPTR$) é 0000 e o valor é 00. Na segunda escrita, ae escrita na memória está habilitada, o endereço de escrita é 0002 e o valor é 01. Na terceira escrita, a escrita na memória está habilitada, o endereço de escrita é 0004 e o valor é 02. E nenhuma outra escrita é realizada, o programa fica em *loop* na última instrução. Este resultado condiz com o objetivo do programa, sendo assim, podemos ter um certo grau de confiança quanto a descrição do processador. Para garantir que não há nenhum erro na descrição seriam necessárias simulações mais aprofundadas que fogem do escopo deste trabalho.

Código 3.1: Código utilizado para testar o processador.

```
    --Insere os números de 0 a 2 nas primeira posições pares da memória de dados

MOV R0, 0
MOV R1, 0
MOV R2, 3
NOP         --(Espera WB de R0)
NOP
```

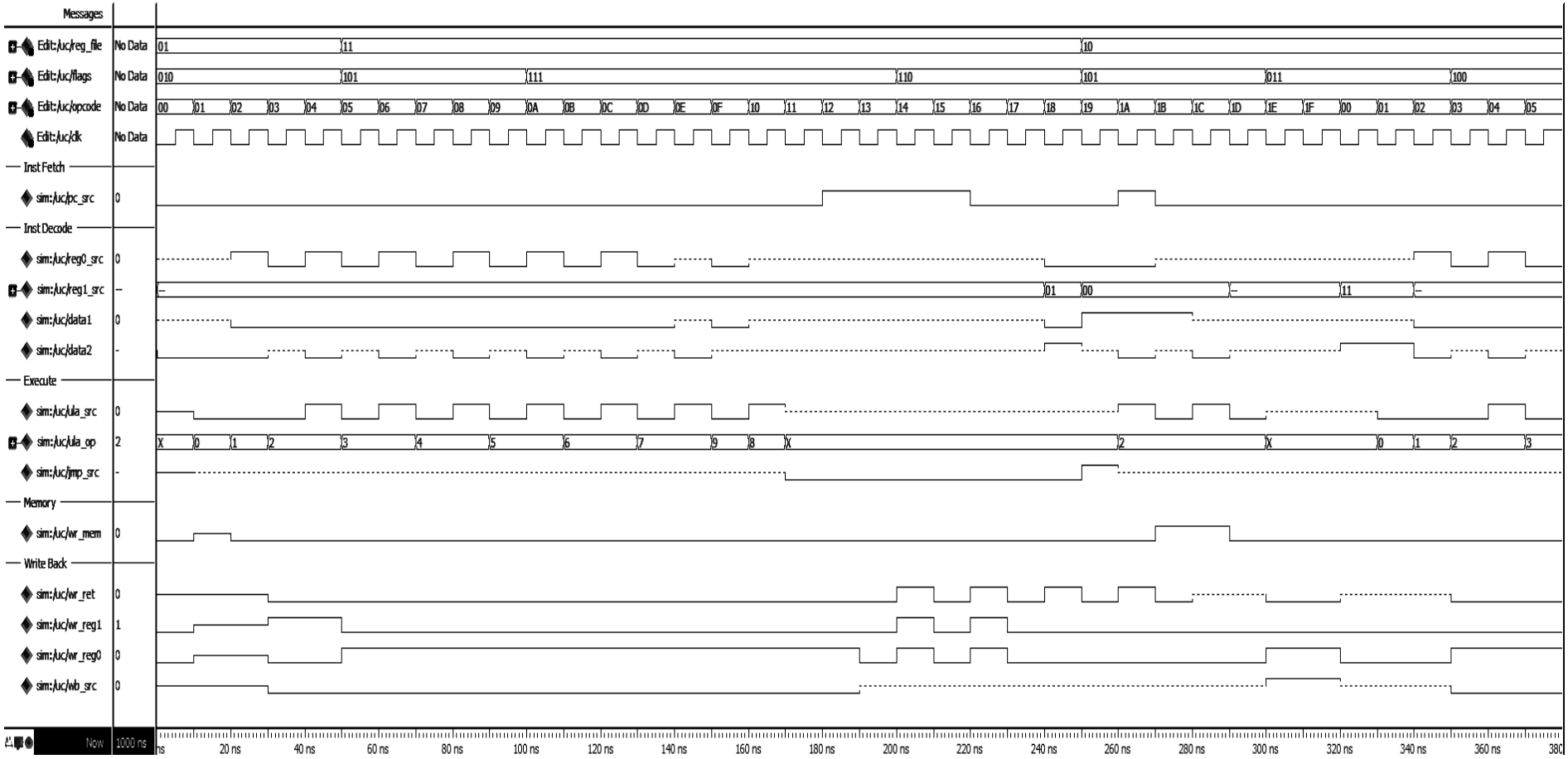


Figura 13: Sinais resultantes da simulação da Unidade de Controle com MODELSIM no tempo. O sinal *reg_file* é a seleção de banco de registrador, *flags* são as *flags* geradas pela ULA, *opcode* é o código de operação da instrução e *clk* é o *clock* do processador, todos estes são entradas de teste. Os sinais restantes, separados por estágio de *pipeline* são os sinais de controle gerados pela simulação.

```
MOV DPTR_L, R0
MOV DPTR_H, R0
NOP          --(Espera WB de DPTR_H)
NOP
NOP
NOP
LOOP: STR R1, R0
ADD R0, 1
ADD R1, 2
NOP          --(Espera WB de R0)
NOP
NOP
CMP R0, R2, R3
BLT LOOP
NOP          --(Espera a decisao do salto)
NOP
NOP
NOP
HERE: JMP HERE
```

O caminho crítico de um circuito é uma de suas características mais importantes, pois ele vai determinar a frequência máxima de operação. No software LeonardoSpectrum (Mentor Graphics Company, 2001) é possível estimar o caminho crítico para uma implementação uma vez fornecida uma biblioteca da tecnologia em que o hardware será desenvolvido. A estimativa do atraso no caminho crítico do processador está em torno de $8ns$, o que nos dá uma frequência de operação de aproximadamente $125MHz$. Esta estimativa não leva em conta os tempos de acesso às memórias, o que pode reduzir drasticamente a frequência de operação. Segundo (HENNESSY; PATTERSON, 2003), o tempo de acesso de uma memória DRAM (*Dynamic Random-Acces Memory*), implementada em CMOS (*Complementary Metal-Oxide-Semiconductor*) varia entre 50 e $250ns$, o que reduziria a frequência de operação, no melhor dos casos, a $20MHz$. Uma solução para esta limitação seria utilizar uma memória SRAM (*Static Random-Acces Memory*) *on-chip*, ou seja, construída no mesmo *chip* do processador, assim, também segundo (HENNESSY; PATTERSON, 2003), o tempo de acesso pode ser reduzido para até $0,5ns$.

Para implementar o hardware em circuito integrado utilizando as ferramentas da Mentor Graphics (LeonardoSpectrum, Design Architect, IC Station, Calibre), o primeiro passo é sintetizar o circuito descrito em VHDL com o LeonardoSpectrum. Como resultado é gerado uma descrição estrutural do circuito em VERILOG, utilizando apenas as células disponíveis na biblioteca da tecnologia utilizada. Nesta descrição temos, basicamente, a informação de quais células da biblioteca são utilizados e as interconexões entre elas. Em seguida, esta descrição em VERILOG é importada pela ferramenta Design Architect, gerando automaticamente o esquemático do processador. No terceiro passo é feita a criação do layout orientado pelo esquemático. Para isso, inicialmente se carregam todas as células utilizadas no circuito. Em seguida se estima a área que o circuito ocupa, o chamado *floorplaning* (quando as células são posicionadas em linhas, o que é nosso caso,

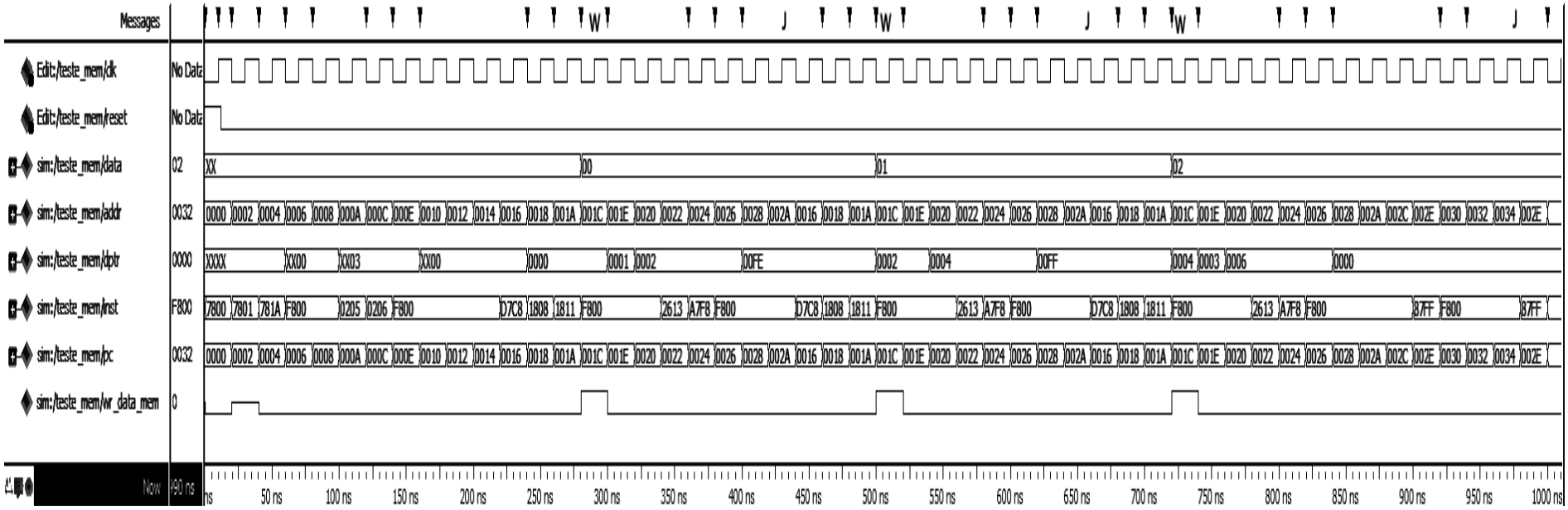


Figura 14: Resultado da simulação do processador. O sinal *clk* é o *clock* do processador, *reset* é o sinal utilizado para zerar o registrador *PC*, estes são entradas de teste. Os sinais *data*, valor lido da memória de dados ou a ser escrito nela, *addr*, endereço da memória de programa, *dptr*, endereço da memória de dados, *inst*, valor lido da memória de programa, *pc*, endereço da memória de programa e *wr_data_mem*, habilita escrita na memória de dados, são resultado da simulação.

determinam-se quantas linhas devem ser usadas e o espaçamento entre elas). Depois, é feito o posicionamento das células, *placement*. A última etapa na criação do layout é realizar as ligações entre as células, o chamado *routing*. Estas etapas são realizadas de forma automática no IC Station, mas pode-se realizar algumas interferências manuais para auxiliar a ferramenta.

Dada a complexidade da tarefa de gerar uma implementação em circuitos integrados, ferramentas adicionais são utilizadas para garantir a correção do resultado final. Entre elas temos a ferramenta Calibre DRC (*Design Rule Check*), que verifica se as regras da tecnologia estão sendo respeitadas e a ferramenta Calibre LVS (*Layout Versus Schematic*), que compara os esquemáticos com o layout.

Para completar o processo de criação de um circuito integrado digital e ter uma visão geral de todo o processo, o último passo é projetar o *layout*. Sendo assim, foi feito o *layout* do somador de 16 bits, pois é um bloco simples e consegue exemplificar o processo da criação de *layout* para qualquer um dos outros blocos. Entre as interferências feitas no processo automático do IC Station está a escolha da distancia entre as linhas de célula, que foi aumentada, pois como o circuito tem muitas entradas e saídas (32 sinais de entrada e 16 de saída), é necessário um espaço maior entre as linhas para a permitir a passagem de várias trilhas de metal entres elas. Outra interferência foi realizada no roteamento dos sinais de alimentação, em que o comprimento das trilhas foi aumentado para $1,8\mu m$, para uma resistência menor, o que garante uma atenuação menor no sinal, e uma capacitância maior, que garante estabilidade da alimentação. Além disso, o roteamento foi feito manualmente, para que as trilhas de alimentação não ficassem sobrepostas à linhas de sinais, a sobreposição aumentaria a capacitância nestas trilhas, diminuindo a frequência máxima de operação do circuito. A célula gerada está exibida na figura 15. O tamanho desta célula é de $118,2\mu m$ por $125,8\mu m$, resultando em uma área de $14869,6\mu m^2$.

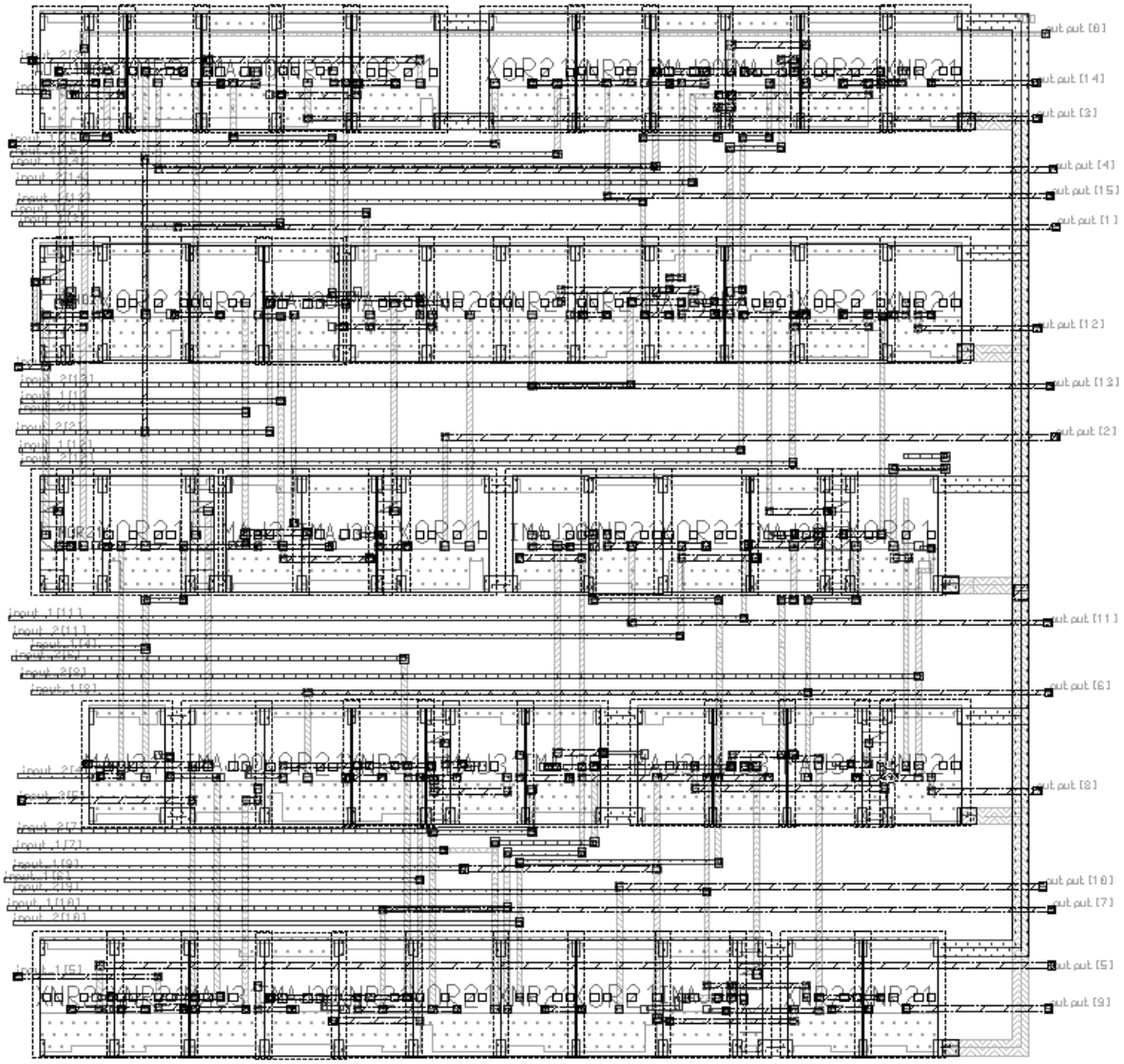


Figura 15: *Layout* do somador de 16 bits. Sinais de entrada posicionados à esquerda e de saída posicionados à direita.

4 CONCLUSÃO

Neste trabalho foi proposta a arquitetura de um processador de 8 bits RISC, com 29 instruções e arquitetura Havard. Os objetivos do processador são ser utilizado em ensino e servir de núcleo para o desenvolvimento de futuros microcontroladores e processadores de *ultra-low power* . Posteriormente se desenvolveu uma microarquitetura para implementar a arquitetura proposta. Esta microarquitetura foi descrita em VHDL e simulada no software ModelSim. Para isso se emulou a memória de programa com um programa escrito nela. A simulação mostrou que, numa primeira analise, o processador funciona corretamente. A partir do código em VHDL gerou-se o esquemático do circuito do somador de 16 bits utilizando as células padrão da tecnologia CMOS de $0,35\mu m$ e então criou-se o *layout* deste circuito para completar o processo de ciração de um circuito integrado digital e dar uma visão geral de todo o processo.

A velocidade máxima do circuito estimada pelo LeonardoSpectrum foi de $125MHz$, sem considerar a velocidade de acesso às memórias, que não fazem parte do escopo deste trabalho. Dependendo da memória utilizada esta frequência pode cair para faixa de algumas dezenas de mega-hertz. Ainda é um resultado satisfatório, já que muitos microcontroladores de 8 bits atualmente atuam nesta faixa de frequência (Atmel, 2016). O *layout* do somador de 16 bits ocupou uma área de aproximadamente $14869,6\mu m^2$.

Este projeto permitiu o aprofundamento na organização de computadores e na linguagem de descrição de hardware VHDL. Também proporcionou a experiência no projeto de um processador desde a definição da arquitetura, partindo das definições iniciais, a escolha das instruções, sua codificação até e desenvolvimento da microarquitetura, até sua implementação, passando pela descrição do hardware, simulação e elaboração, em parte, do *layout*, englobando assim, todo o processo de criação de um circuito integrado digital.

Como sugestão para trabalhos futuros estão:

- Simulações mais detalhadas do processador para verificar seu funcionamento;
- Desenvolvimento de mais blocos funcionais a nível de *layout* para verificar métodos de implementação;
- Implementação completa do circuito a nível de *layout*;
- Analise do conjunto de instruções para otimizá-lo;
- Adição de instruções de multiplicação e divisão;
- Melhora na microarquitetura para tratar problemas de *hazard* a nível de hardware;

- Otimização do código VHDL;
- Desenvolvimento de *Assebler* para o processador;
- Desenvolvimento de compilador de linguagem de alto nível para o processador.
- Desenvolvimento de periféricos, para que, futuramente possa-se construir um microcontrolador completo;
- Estudo de técnicas de distribuição de *clock*;
- Construção de um sistema de sensoriamento de um milimetro cúbico, como em (SEOK et al., 2008).

REFERÊNCIAS

Atmel. **Datasheet ATmega328P**. 2016. <http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf>.

AYERS, J. E. **Digital Integrated Circuits: Analysis and Design, Second Edition**. 2nd. ed. Boca Raton, FL, USA: CRC Press, Inc., 2009. ISBN 142006987X, 9781420069877.

BORGES, R. W. **Aplicabilidade de Sistemas Operacionais de Tempo Real (RTOS) para sistemas embarcados de baixo custo e pequeno porte**. 2011.

EIGENMANN, R.; LILJA, D. J. Von neumann computers. 1998.

GEORGE, A. D. An overview of risc vs. cisc. In: **[1990] Proceedings. The Twenty-Second Southeastern Symposium on System Theory**. [S.l.: s.n.], 1990. p. 436–438. ISSN 0094-2898.

GOODMAN, J.; MILLER, K. **A Programmer's View of Computer Architecture with Assembly Language Examples from the MIPS RISC Architecture**. Philadelphia, PA, USA: Saunders College Publishing, 1993. ISBN 0-03-097219-1.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture: A Quantitative Approach**. 3. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. ISBN 1558607242.

Mentor Graphics. **Calibre nmDRC**. <https://www.mentor.com/products/ic_nanometer_design/verification-signoff/physical-verification/calibre-nmdrc/>.

Mentor Graphics. **Calibre nmLVS**. <https://www.mentor.com/products/ic_nanometer_design/verification-signoff/circuit-verification/calibre-nmlvs/>.

Mentor Graphics. **Design Architect Userś Manual**. <http://pages.cs.wisc.edu/~sohi/cs552/Handouts/MentorDocs/design_architect_users_manual.pdf>.

Mentor Graphics. **ModelSim PE Student Edition**. <https://www.mentor.com/company/higher_ed/modelsim-student-edition>.

Mentor Graphics. **Physix Layout**. <https://www.mentor.com/products/ic_nanometer_design/custom-ic-design/pyxis-layout/>.

Mentor Graphics Company. **LeonardoSpectrum for Altera HDL Synthesis Manual**. 2001. <http://www.eng.auburn.edu/~agrawvd/COURSE/E6200_Spr09/PROJECT/VHDLSynthesisGuide.pdf>.

NELSON, V. P. et al. **Digital Logic Circuit Analysis and Design**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1995. ISBN 0-13-463894-8.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design, Fifth Edition: The Hardware/Software Interface**. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN 0124077269, 9780124077263.

RABAEY, J. M. **Digital Integrated Circuits: A Design Perspective**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996. ISBN 0-13-178609-1.

SEOK, M. et al. Phoenix : an ultra-low power processor for cubic millimeter sensor systems. In: . [S.l.: s.n.], 2008.

SHAHZEB et al. Comparative study of risc and cisc architectures. **International Journal of Computer Applications Technology and Research(IJCATR)**, v. 5, p. 417 – 503, 2016.

ŠULIK, D.; VASILKO, M.; FUCHS, P. Design of a risc microcontroller core in 48 hours. **Journal of ELECTRICAL ENGINEERING**, v. 52, n. 5-6, p. 171–176, 2001.

TOCCI, R. J.; WIDMER, N. S. **Sistemas digitais: princípios e aplicações**. [S.l.]: Prentice Hall, 2003. v. 8.

Apêndices

APÊNDICE A – CÓDIGO FONTE VHDL

Código A.1: Código utilizado para descrever os somadores de 16 bits.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL ;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Somador_16 is


  port (
    input_1 : in std_logic_vector(15 downto 0);
    input_2 : in std_logic_vector(15 downto 0);
    output : out std_logic_vector(15 downto 0)
  );


end Somador_16;


architecture arch of Somador_16 is
begin
  process (input_1, input_2)
  begin
    output <= input_1 + input_2 ;
  end process;
end;
```

Código A.2: Código utilizado para descrever os bancos de registradores.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL ;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Reg_file is

  port (
    rd_addr1 : in std_logic_vector(2 downto 0);
    rd_addr2 : in std_logic_vector(2 downto 0);
    wr_addr: in std_logic_vector(2 downto 0);
    wr_enable : in std_logic;
    wr_data : in std_logic_vector(7 downto 0) ;
    clk : in std_logic;
    data1 : out std_logic_vector(7 downto 0) ;
    data2 : out std_logic_vector(7 downto 0)
  );


end Reg_file;


architecture arch of Reg_file is
```

```
  type registers_type is array(0 to 7) of std_logic_vector(7 downto 0);
  signal registers : registers_type ;
begin
  process (clk, rd_addr1, rd_addr2, wr_addr, wr_enable, wr_data, registers)
  begin
    if clk = '0' then
      if wr_enable = '1' then
        registers(to_integer(unsigned(wr_addr))) <= wr_data ;
      end if ;
      data1 <= "--------" ;
      data2 <= "--------" ;
    else
      data1 <= registers(to_integer(unsigned(rd_addr1))) ;
      data2 <= registers(to_integer(unsigned(rd_addr2))) ;
    end if ;
  end process ;
end;
```

Código A.3: Código utilizado para descrever a ULA.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL ;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ULA is
    generic (
    CONSTANT S_LEFT : std_logic_vector(3 downto 0) := "0000";
    CONSTANT S_RIGHT : std_logic_vector(3 downto 0) := "0001";
    CONSTANT ADD : std_logic_vector(3 downto 0) := "0010";
    CONSTANT SUB : std_logic_vector(3 downto 0) := "0011";
    CONSTANT ADC : std_logic_vector(3 downto 0) := "0100"; --Add with carry flag
    CONSTANT BW_AND : std_logic_vector(3 downto 0) := "0101"; --Bitwise AND
    CONSTANT BW_OR : std_logic_vector(3 downto 0) := "0110";  --Bitwise OR
    CONSTANT BW_XOR : std_logic_vector(3 downto 0) := "0111"; --Bitwise XOR
    CONSTANT BYPASS : std_logic_vector(3 downto 0) := "1000"; --No operation
    CONSTANT BW_NOT : std_logic_vector(3 downto 0) := "1001" --Bitwise NOT
  );

  port (
    input_1 : in std_logic_vector(7 downto 0);
    input_2 : in std_logic_vector(7 downto 0);
    shift_amnt: in std_logic_vector(2 downto 0);
    ula_op : in std_logic_vector(3 downto 0);
    flags : out std_logic_vector(2 downto 0); --flag(2) = carry; flag(1) = zero; flag(0)
        = negativo
    output : out std_logic_vector(7 downto 0)
  );


end ULA;


architecture arch of ULA is
  signal aux : std_logic_vector(8 downto 0) ;
begin
  process (ula_op, input_1, input_2, shift_amnt)
  begin
    case ula_op is
```

```
      when S_LEFT => aux <= std_logic_vector(unsigned('0' & input_2) sll to_integer(
          unsigned(shift_amnt)));
      when S_RIGHT => aux <= std_logic_vector(unsigned('0' & input_2) srl to_integer(
          unsigned(shift_amnt)));
      when ADD => aux <= std_logic_vector('0' & unsigned(input_1) + unsigned(input_2));
      when SUB => aux <= '0' & std_logic_vector(signed(input_1) - signed(input_2));
      when ADC => aux <= std_logic_vector('0' & unsigned(input_1) + unsigned(input_2) +
          1);
      when BW_AND => aux <= '0' & (input_1 and input_2) ;
      when BW_OR => aux <= '0' & (input_1 or input_2) ;
      when BW_XOR => aux <= '0' & (input_1 xor input_2) ;
      when BYPASS => aux <= '0' & (input_2) ;
      when BW_NOT => aux <= '0' & (not input_2) ;
           when others =>
    end case ;
  end process;

  process (aux)
  begin
    output <= aux(7 downto 0) ;

    flags(0) <= aux(7) ;
    flags(2) <= aux(8) ;
    if to_integer(unsigned(aux)) = 0 then
      flags(1) <= '1' ;
    else
      flags(1) <= '0' ;
    end if ;
  end process ;
end;
```

Código A.4: Código utilizado para descrever a unidade de controle.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL ;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity UC is
  generic (
    CONSTANT LSL : std_logic_vector(4 downto 0) := "00000" ;
    CONSTANT LSR : std_logic_vector(4 downto 0) := "00001" ;
    CONSTANT ADD_REG : std_logic_vector(4 downto 0) := "00010" ;
    CONSTANT ADD_IMM : std_logic_vector(4 downto 0) := "00011" ;
    CONSTANT SUB_REG : std_logic_vector(4 downto 0) := "00100" ;
    CONSTANT SUB_IMM : std_logic_vector(4 downto 0) := "00101" ;
    CONSTANT ADC_REG : std_logic_vector(4 downto 0) := "00110" ;
    CONSTANT ADC_IMM : std_logic_vector(4 downto 0) := "00111" ;
    CONSTANT AND_REG : std_logic_vector(4 downto 0) := "01000" ;
    CONSTANT AND_IMM : std_logic_vector(4 downto 0) := "01001" ;
    CONSTANT OR_REG: std_logic_vector(4 downto 0) := "01010" ;
    CONSTANT OR_IMM: std_logic_vector(4 downto 0) := "01011" ;
    CONSTANT XOR_REG : std_logic_vector(4 downto 0) := "01100" ;
    CONSTANT XOR_IMM : std_logic_vector(4 downto 0) := "01101" ;
    CONSTANT NOT_REG : std_logic_vector(4 downto 0) := "01110" ;
    CONSTANT MOV_IMM : std_logic_vector(4 downto 0) := "01111" ;
    CONSTANT JMP : std_logic_vector(4 downto 0) := "10000" ;
    CONSTANT JAL : std_logic_vector(4 downto 0) := "10001" ;
    CONSTANT BEQ : std_logic_vector(4 downto 0) := "10010" ;
```

```
    CONSTANT BEL : std_logic_vector(4 downto 0) := "10011" ;
    CONSTANT BLT : std_logic_vector(4 downto 0) := "10100" ;
    CONSTANT BLL : std_logic_vector(4 downto 0) := "10101" ;
    CONSTANT BGT : std_logic_vector(4 downto 0) := "10110" ;
    CONSTANT BGL : std_logic_vector(4 downto 0) := "10111" ;
    CONSTANT RET : std_logic_vector(4 downto 0) := "11000" ;
    CONSTANT STR_IMM : std_logic_vector(4 downto 0) := "11001" ;
    CONSTANT STR_REG : std_logic_vector(4 downto 0) := "11010" ;
    CONSTANT LDB_IMM : std_logic_vector(4 downto 0) := "11011" ;
    CONSTANT LDB_REG : std_logic_vector(4 downto 0) := "11100" ;
    CONSTANT NOP : std_logic_vector(4 downto 0) := "11111" ;




    CONSTANT S_LEFT : std_logic_vector(3 downto 0) := "0000";
    CONSTANT S_RIGHT : std_logic_vector(3 downto 0) := "0001";
    CONSTANT ADD : std_logic_vector(3 downto 0) := "0010";
    CONSTANT SUB : std_logic_vector(3 downto 0) := "0011";
    CONSTANT ADC : std_logic_vector(3 downto 0) := "0100"; --Add with carry flag
    CONSTANT BW_AND : std_logic_vector(3 downto 0) := "0101"; --Bitwise AND
    CONSTANT BW_OR : std_logic_vector(3 downto 0) := "0110";  --Bitwise OR
    CONSTANT BW_XOR : std_logic_vector(3 downto 0) := "0111"; --Bitwise XOR
    CONSTANT BYPASS : std_logic_vector(3 downto 0) := "1000"; --No operation
    CONSTANT BW_NOT : std_logic_vector(3 downto 0) := "1001" --Bitwise NOT
  );

  port (
    opcode     : in std_logic_vector(4 downto 0) ;
    reg_file   : in std_logic_vector(1 downto 0) ;
    flags      : in std_logic_vector(2 downto 0) ;
    clk      : in std_logic ;
    pc_src     : out std_logic := '0';
    reg0_src   : out std_logic ;
    reg1_src   : out std_logic_vector (1 downto 0) ;
    data1    : out std_logic ;
    data2      : out std_logic ;
    ula_src    : out std_logic ;
    jmp_src    : out std_logic ;
    ula_op     : out std_logic_vector(3 downto 0) ;
    wr_mem     : out std_logic := '0';
    wb_src     : out std_logic ;
    wr_reg0    : out std_logic := '0';
    wr_reg1    : out std_logic := '0';
    wr_ret     : out std_logic
  );


end UC;


architecture arch of UC is
  signal ula_op_id : std_logic_vector(3 downto 0) ;
  signal ula_src_id : std_logic ;
  signal wr_mem_id : std_logic ;
  signal wb_src_id : std_logic ;
  signal wr_reg0_id : std_logic ;
  signal wr_reg1_id : std_logic ;
  signal wr_ret_id : std_logic ;
  signal jmp_src_id : std_logic ;
```

```
  signal jmp_id : std_logic ;
  signal bz_id : std_logic ;
  signal bp_id : std_logic ;
  signal bn_id : std_logic ;
  signal wr_mem_ex : std_logic ;
  signal wb_src_ex : std_logic ;
  signal wr_reg0_ex : std_logic ;
  signal wr_reg1_ex : std_logic ;
  signal wr_reg0_ex2 : std_logic ;
  signal wr_reg1_ex2 : std_logic ;
  signal wr_ret_ex : std_logic ;
  signal jmp_ex : std_logic ;
  signal bz_ex : std_logic ;
  signal bp_ex : std_logic ;
  signal bn_ex : std_logic ;
  signal true_branch: std_logic ;
  signal wb_src_mem : std_logic ;
  signal wr_reg0_mem : std_logic ;
  signal wr_reg1_mem : std_logic ;
  signal wr_ret_mem : std_logic ;



  signal last_flag : std_logic_vector(2 downto 0) ;
begin


  process (clk)
  begin
    if (falling_edge(clk)) then
      last_flag <= flags ;

      wb_src  <= wb_src_mem;
      wr_reg0 <= wr_reg0_mem;
      wr_reg1 <= wr_reg1_mem;
      wr_ret  <= wr_ret_mem;

      wr_mem <= wr_mem_ex ;
      wb_src_mem <= wb_src_ex ;
      wr_reg0_mem <= wr_reg0_ex2 ;
      wr_reg1_mem <= wr_reg1_ex2 ;
      wr_ret_mem <= wr_ret_ex ;
      pc_src <= true_branch ;

      ula_op <= ula_op_id;
      ula_src <= ula_src_id;
      wr_mem_ex <= wr_mem_id;
      wb_src_ex <= wb_src_id;
      wr_reg0_ex <= wr_reg0_id;
      wr_reg1_ex <= wr_reg1_id;
      wr_ret_ex <= wr_ret_id;
      jmp_src <= jmp_src_id;
      jmp_ex <= jmp_id;
      bz_ex <= bz_id;
      bp_ex <= bp_id;
      bn_ex <= bn_id;
    end if ;
  end process ;

  process (jmp_ex, bz_ex, bp_ex, bn_ex, last_flag)
```

```
begin
  if (jmp_ex='1' or (last_flag(1)='1' and bz_ex='1') or (last_flag(0)='1' and bn_ex
      ='1') or (last_flag(0)='0' and last_flag(1)='0' and bp_ex='1')) then
    true_branch <= '1' ;
  else
    true_branch <= '0' ;
  end if ;
end process ;

process (true_branch, wr_ret_ex, wr_reg0_ex, wr_reg1_ex)
begin
  if (true_branch='1' and wr_ret_ex='1') then
    wr_reg0_ex2 <= '1' ;
    wr_reg1_ex2 <= '1' ;
  else
    wr_reg0_ex2 <= wr_reg0_ex ;
    wr_reg1_ex2 <= wr_reg1_ex ;
  end if ;
end process ;

process (opcode, reg_file)
begin
  case opcode is
    when LSL => ula_src_id <= '0' ;
          ula_op_id <= S_LEFT ;
          wr_mem_id <= '0' ;
          wb_src_id <= '0' ;
          jmp_id <= '0' ;
          bz_id <= '0' ;
          bp_id <= '0' ;
          bn_id <= '0' ;
          jmp_src_id <= '-' ;
          wr_ret_id <= '0' ;
          reg0_src <= '-' ;
          data1 <= '-' ;
          if reg_file(1) = '0' then
            data2 <= '0' ;
            reg1_src <= "--" ;
                    else
            data2 <= '1' ;
            reg1_src <= "11" ;
          end if ;

          if reg_file(0) = '0' then
            wr_reg0_id <= '1' ;
            wr_reg1_id <= '0' ;
          else
            wr_reg0_id <= '0' ;
            wr_reg1_id <= '1' ;
          end if ;

    when LSR => ula_src_id <= '0' ;
          ula_op_id <= S_RIGHT ;
          wr_mem_id <= '0' ;
          wb_src_id <= '0' ;
          jmp_id <= '0' ;
          bz_id <= '0' ;
          bp_id <= '0' ;
          bn_id <= '0' ;
          jmp_src_id <= '-' ;
```

```
        wr_ret_id <= '0' ;
        reg0_src <= '-' ;
        data1 <= '-' ;
        if reg_file(1) = '0' then
          data2 <= '0' ;
          reg1_src <= "--" ;
                    else
          data2 <= '1' ;
          reg1_src <= "11" ;
        end if ;

        if reg_file(0) = '0' then
          wr_reg0_id <= '1' ;
          wr_reg1_id <= '0' ;
        else
          wr_reg0_id <= '0' ;
          wr_reg1_id <= '1' ;
        end if ;

    when ADD_REG => reg0_src <= '1' ;
          reg1_src <= "--" ;
          data1 <= '0' ;
          data2 <= '0' ;
          ula_src_id <= '0' ;
          jmp_src_id <= '-' ;
          ula_op_id <= ADD ;
          wr_mem_id <= '0' ;
          wb_src_id <= '0' ;
          wr_reg0_id <= '1' ;
          wr_reg1_id <= '0' ;
          wr_ret_id <= '0' ;
          jmp_id <= '0' ;
          bz_id <= '0' ;
          bp_id <= '0' ;
          bn_id <= '0' ;

    when ADD_IMM => reg0_src <= '0' ;
          reg1_src <= "--" ;
          data1 <= '0' ;
          data2 <= '-' ;
          ula_src_id <= '1' ;
          jmp_src_id <= '-' ;
          ula_op_id <= ADD ;
          wr_mem_id <= '0' ;
          wb_src_id <= '0' ;
          wr_reg0_id <= '1' ;
          wr_reg1_id <= '0' ;
          wr_ret_id <= '0' ;
          jmp_id <= '0' ;
          bz_id <= '0' ;
          bp_id <= '0' ;
          bn_id <= '0' ;

    when SUB_REG => reg0_src <= '1' ;
          reg1_src <= "--" ;
          data1 <= '0' ;
          data2 <= '0' ;
          ula_src_id <= '0' ;
          jmp_src_id <= '-' ;
          ula_op_id <= SUB ;
```

```
            wr_mem_id <= '0' ;
            wb_src_id <= '0' ;
            wr_reg0_id <= '1' ;
            wr_reg1_id <= '0' ;
            wr_ret_id <= '0' ;
            jmp_id <= '0' ;
            bz_id <= '0' ;
            bp_id <= '0' ;
            bn_id <= '0' ;

        when SUB_IMM => reg0_src <= '0' ;
            reg1_src <= "--" ;
            data1 <= '0' ;
            data2 <= '-' ;
            ula_src_id <= '1' ;
            jmp_src_id <= '-' ;
            ula_op_id <= SUB ;
            wr_mem_id <= '0' ;
            wb_src_id <= '0' ;
            wr_reg0_id <= '1' ;
            wr_reg1_id <= '0' ;
            wr_ret_id <= '0' ;
            jmp_id <= '0' ;
            bz_id <= '0' ;
            bp_id <= '0' ;
            bn_id <= '0' ;

        when ADC_REG => reg0_src <= '1' ;
            reg1_src <= "--" ;
            data1 <= '0' ;
            data2 <= '0' ;
            ula_src_id <= '0' ;
            jmp_src_id <= '-' ;
            ula_op_id <= ADC ;
            wr_mem_id <= '0' ;
            wb_src_id <= '0' ;
            wr_reg0_id <= '1' ;
            wr_reg1_id <= '0' ;
            wr_ret_id <= '0' ;
            jmp_id <= '0' ;
            bz_id <= '0' ;
            bp_id <= '0' ;
            bn_id <= '0' ;

        when ADC_IMM => reg0_src <= '0' ;
            reg1_src <= "--" ;
            data1 <= '0' ;
            data2 <= '-' ;
            ula_src_id <= '1' ;
            jmp_src_id <= '-' ;
            ula_op_id <= ADC ;
            wr_mem_id <= '0' ;
            wb_src_id <= '0' ;
            wr_reg0_id <= '1' ;
            wr_reg1_id <= '0' ;
            wr_ret_id <= '0' ;
            jmp_id <= '0' ;
            bz_id <= '0' ;
            bp_id <= '0' ;
            bn_id <= '0' ;
```

```
when AND_REG => reg0_src <= '1' ;
        reg1_src <= "--" ;
        data1 <= '0' ;
        data2 <= '0' ;
        ula_src_id <= '0' ;
        jmp_src_id <= '-' ;
        ula_op_id <= BW_AND ;
        wr_mem_id <= '0' ;
        wb_src_id <= '0' ;
        wr_reg0_id <= '1' ;
        wr_reg1_id <= '0' ;
        wr_ret_id <= '0' ;
        jmp_id <= '0' ;
        bz_id <= '0' ;
        bp_id <= '0' ;
        bn_id <= '0' ;

when AND_IMM => reg0_src <= '0' ;
        reg1_src <= "--" ;
        data1 <= '0' ;
        data2 <= '-' ;
        ula_src_id <= '1' ;
        jmp_src_id <= '-' ;
        ula_op_id <= BW_AND ;
        wr_mem_id <= '0' ;
        wb_src_id <= '0' ;
        wr_reg0_id <= '1' ;
        wr_reg1_id <= '0' ;
        wr_ret_id <= '0' ;
        jmp_id <= '0' ;
        bz_id <= '0' ;
        bp_id <= '0' ;
        bn_id <= '0' ;

when OR_REG =>  reg0_src <= '1' ;
        reg1_src <= "--" ;
        data1 <= '0' ;
        data2 <= '0' ;
        ula_src_id <= '0' ;
        jmp_src_id <= '-' ;
        ula_op_id <= BW_OR ;
        wr_mem_id <= '0' ;
        wb_src_id <= '0' ;
        wr_reg0_id <= '1' ;
        wr_reg1_id <= '0' ;
        wr_ret_id <= '0' ;
        jmp_id <= '0' ;
        bz_id <= '0' ;
        bp_id <= '0' ;
        bn_id <= '0' ;

when OR_IMM => reg0_src <= '0' ;
        reg1_src <= "--" ;
        data1 <= '0' ;
        data2 <= '-' ;
        ula_src_id <= '1' ;
        jmp_src_id <= '-' ;
        ula_op_id <= BW_OR ;
        wr_mem_id <= '0' ;
```

```
            wb_src_id <= '0' ;
            wr_reg0_id <= '1' ;
            wr_reg1_id <= '0' ;
            wr_ret_id <= '0' ;
            jmp_id <= '0' ;
            bz_id <= '0' ;
            bp_id <= '0' ;
            bn_id <= '0' ;

        when XOR_REG => reg0_src <= '1' ;
            reg1_src <= "--" ;
            data1 <= '0' ;
            data2 <= '0' ;
            ula_src_id <= '0' ;
            jmp_src_id <= '-' ;
            ula_op_id <= BW_XOR ;
            wr_mem_id <= '0' ;
            wb_src_id <= '0' ;
            wr_reg0_id <= '1' ;
            wr_reg1_id <= '0' ;
            wr_ret_id <= '0' ;
            jmp_id <= '0' ;
            bz_id <= '0' ;
            bp_id <= '0' ;
            bn_id <= '0' ;

        when XOR_IMM => reg0_src <= '0' ;
            reg1_src <= "--" ;
            data1 <= '0' ;
            data2 <= '-' ;
            ula_src_id <= '1' ;
            jmp_src_id <= '-' ;
            ula_op_id <= BW_XOR ;
            wr_mem_id <= '0' ;
            wb_src_id <= '0' ;
            wr_reg0_id <= '1' ;
            wr_reg1_id <= '0' ;
            wr_ret_id <= '0' ;
            jmp_id <= '0' ;
            bz_id <= '0' ;
            bp_id <= '0' ;
            bn_id <= '0' ;

        when MOV_IMM => reg0_src <= '0' ;
            reg1_src <= "--" ;
            data1 <= '0' ;
            data2 <= '-' ;
            ula_src_id <= '1' ;
            jmp_src_id <= '-' ;
            ula_op_id <= BYPASS ;
            wr_mem_id <= '0' ;
            wb_src_id <= '0' ;
            wr_reg0_id <= '1' ;
            wr_reg1_id <= '0' ;
            wr_ret_id <= '0' ;
            jmp_id <= '0' ;
            bz_id <= '0' ;
            bp_id <= '0' ;
            bn_id <= '0' ;
```

```
when NOT_REG => reg0_src <= '-' ;
        reg1_src <= "--" ;
        data1 <= '-' ;
        data2 <= '0' ;
        ula_src_id <= '0' ;
        jmp_src_id <= '-' ;
        ula_op_id <= BW_NOT ;
        wr_mem_id <= '0' ;
        wb_src_id <= '0' ;
        wr_reg0_id <= '1' ;
        wr_reg1_id <= '0' ;
        wr_ret_id <= '0' ;
        jmp_id <= '0' ;
        bz_id <= '0' ;
        bp_id <= '0' ;
        bn_id <= '0' ;

when NOP => reg0_src <= '-' ;
        reg1_src <= "--" ;
        data1 <= '-' ;
        data2 <= '-' ;
        ula_src_id <= '-' ;
        jmp_src_id <= '-' ;
        ula_op_id <= "----" ;
        wr_mem_id <= '0' ;
        wb_src_id <= '-' ;
        wr_reg0_id <= '0' ;
        wr_reg1_id <= '0' ;
        wr_ret_id <= '-' ;
        jmp_id <= '0' ;
        bz_id <= '0' ;
        bp_id <= '0' ;
        bn_id <= '0' ;

when STR_REG => reg0_src <= '0' ;
        reg1_src <= "00" ;
        data1 <= '1' ;
        data2 <= '0' ;
        ula_src_id <= '0' ;
        jmp_src_id <= '-' ;
        ula_op_id <= ADD ;
        wr_mem_id <= '1' ;
        wb_src_id <= '-' ;
        wr_reg0_id <= '0' ;
        wr_reg1_id <= '0' ;
        wr_ret_id <= '-' ;
        jmp_id <= '0' ;
        bz_id <= '0' ;
        bp_id <= '0' ;
        bn_id <= '0' ;

when STR_IMM => reg0_src <= '0' ;
        reg1_src <= "00" ;
        data1 <= '1' ;
        data2 <= '-' ;
        ula_src_id <= '1' ;
        jmp_src_id <= '-' ;
        ula_op_id <= ADD ;
        wr_mem_id <= '1' ;
        wb_src_id <= '-' ;
```

```
            wr_reg0_id <= '0' ;
            wr_reg1_id <= '0' ;
            wr_ret_id <= '-' ;
            jmp_id <= '0' ;
            bz_id <= '0' ;
            bp_id <= '0' ;
            bn_id <= '0' ;

        when LDB_REG => reg0_src <= '-' ;
            reg1_src <= "00" ;
            data1 <= '-' ;
            data2 <= '0' ;
            ula_src_id <= '0' ;
            jmp_src_id <= '-' ;
            ula_op_id <= ADD ;
            wr_mem_id <= '0' ;
            wb_src_id <= '1' ;
            wr_reg0_id <= '1' ;
            wr_reg1_id <= '0' ;
            wr_ret_id <= '0' ;
            jmp_id <= '0' ;
            bz_id <= '0' ;
            bp_id <= '0' ;
            bn_id <= '0' ;

        when LDB_IMM => reg0_src <= '-' ;
            reg1_src <= "00" ;
            data1 <= '1' ;
            data2 <= '-' ;
            ula_src_id <= '1' ;
            jmp_src_id <= '-' ;
            ula_op_id <= ADD ;
            wr_mem_id <= '0' ;
            wb_src_id <= '1' ;
            wr_reg0_id <= '1' ;
            wr_reg1_id <= '0' ;
            wr_ret_id <= '0' ;
            jmp_id <= '0' ;
            bz_id <= '0' ;
            bp_id <= '0' ;
            bn_id <= '0' ;

        when JMP =>    reg0_src <= '-' ;
            reg1_src <= "--" ;
            data1 <= '-' ;
            data2 <= '-' ;
            ula_src_id <= '-' ;
            jmp_src_id <= '0' ;
            ula_op_id <= "----" ;
            wr_mem_id <= '0' ;
            wb_src_id <= '-' ;
            wr_reg0_id <= '0' ;
            wr_reg1_id <= '0' ;
            wr_ret_id <= '0' ;
            jmp_id <= '1' ;
            bz_id <= '0' ;
            bp_id <= '0' ;
            bn_id <= '0' ;

        when JAL =>    reg0_src <= '-' ;
```

```
            reg1_src <= "--" ;
            data1 <= '-' ;
            data2 <= '-' ;
            ula_src_id <= '-' ;
            jmp_src_id <= '0' ;
            ula_op_id <= "----" ;
            wr_mem_id <= '0' ;
            wb_src_id <= '-' ;
            wr_reg0_id <= '0' ;
            wr_reg1_id <= '0' ;
            wr_ret_id <= '1' ;
            jmp_id <= '1' ;
            bz_id <= '0' ;
            bp_id <= '0' ;
            bn_id <= '0' ;

        when BEQ =>    reg0_src <= '-' ;
            reg1_src <= "--" ;
            data1 <= '-' ;
            data2 <= '-' ;
            ula_src_id <= '-' ;
            jmp_src_id <= '0' ;
            ula_op_id <= "----" ;
            wr_mem_id <= '0' ;
            wb_src_id <= '-' ;
            wr_reg0_id <= '0' ;
            wr_reg1_id <= '0' ;
            wr_ret_id <= '0' ;
            jmp_id <= '0' ;
            bz_id <= '1' ;
            bp_id <= '0' ;
            bn_id <= '0' ;

        when BEL =>    reg0_src <= '-' ;
            reg1_src <= "--" ;
            data1 <= '-' ;
            data2 <= '-' ;
            ula_src_id <= '-' ;
            jmp_src_id <= '0' ;
            ula_op_id <= "----" ;
            wr_mem_id <= '0' ;
            wb_src_id <= '-' ;
            wr_reg0_id <= '0' ;
            wr_reg1_id <= '0' ;
            wr_ret_id <= '1' ;
            jmp_id <= '0' ;
            bz_id <= '1' ;
            bp_id <= '0' ;
            bn_id <= '0' ;

        when BLT =>    reg0_src <= '-' ;
            reg1_src <= "--" ;
            data1 <= '-' ;
            data2 <= '-' ;
            ula_src_id <= '-' ;
            jmp_src_id <= '0' ;
            ula_op_id <= "----" ;
            wr_mem_id <= '0' ;
            wb_src_id <= '-' ;
            wr_reg0_id <= '0' ;
```

```
            wr_reg1_id <= '0' ;
            wr_ret_id <= '0' ;
            jmp_id <= '0' ;
            bz_id <= '0' ;
            bp_id <= '0' ;
            bn_id <= '1' ;

        when BLL =>    reg0_src <= '-' ;
            reg1_src <= "--" ;
            data1 <= '-' ;
            data2 <= '-' ;
            ula_src_id <= '-' ;
            jmp_src_id <= '0' ;
            ula_op_id <= "----" ;
            wr_mem_id <= '0' ;
            wb_src_id <= '-' ;
            wr_reg0_id <= '0' ;
            wr_reg1_id <= '0' ;
            wr_ret_id <= '1' ;
            jmp_id <= '0' ;
            bz_id <= '0' ;
            bp_id <= '0' ;
            bn_id <= '1' ;

        when BGT =>    reg0_src <= '-' ;
            reg1_src <= "--" ;
            data1 <= '-' ;
            data2 <= '-' ;
            ula_src_id <= '-' ;
            jmp_src_id <= '0' ;
            ula_op_id <= "----" ;
            wr_mem_id <= '0' ;
            wb_src_id <= '-' ;
            wr_reg0_id <= '0' ;
            wr_reg1_id <= '0' ;
            wr_ret_id <= '0' ;
            jmp_id <= '0' ;
            bz_id <= '0' ;
            bp_id <= '1' ;
            bn_id <= '0' ;

        when BGL =>    reg0_src <= '-' ;
            reg1_src <= "--" ;
            data1 <= '-' ;
            data2 <= '-' ;
            ula_src_id <= '-' ;
            jmp_src_id <= '0' ;
            ula_op_id <= "----" ;
            wr_mem_id <= '0' ;
            wb_src_id <= '-' ;
            wr_reg0_id <= '0' ;
            wr_reg1_id <= '0' ;
            wr_ret_id <= '1' ;
            jmp_id <= '0' ;
            bz_id <= '0' ;
            bp_id <= '1' ;
            bn_id <= '0' ;

        when RET =>    reg0_src <= '0' ;
            reg1_src <= "01" ;
```

```
                data1 <= '0' ;
                data2 <= '1' ;
                ula_src_id <= '-' ;
                jmp_src_id <= '1' ;
                ula_op_id <= "----" ;
                wr_mem_id <= '0' ;
                wb_src_id <= '-' ;
                wr_reg0_id <= '0' ;
                wr_reg1_id <= '0' ;
                wr_ret_id <= '0' ;
                jmp_id <= '1' ;
                bz_id <= '0' ;
                bp_id <= '0' ;
                bn_id <= '0' ;

      when others =>  reg0_src <= '-' ;
                reg1_src <= "--" ;
                data1 <= '-' ;
                data2 <= '-' ;
                ula_src_id <= '-' ;
                jmp_src_id <= '-' ;
                ula_op_id <= "----" ;
                wr_mem_id <= '0' ;
                wb_src_id <= '-' ;
                wr_reg0_id <= '0' ;
                wr_reg1_id <= '0' ;
                wr_ret_id <= '-' ;
                jmp_id <= '0' ;
                bz_id <= '0' ;
                bp_id <= '0' ;
                bn_id <= '0' ;
    end case ;
  end process ;
end;
```

Código A.5: Código utilizado para descrever o processador.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL ;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

library work;
use work.all;

entity datapath is

  generic (
    CONSTANT ADDR_RET_L : std_logic_vector(2 downto 0) := "111";
    CONSTANT ADDR_RET_H : std_logic_vector(2 downto 0) := "111";
    CONSTANT ADDR_DPTR_L : std_logic_vector(2 downto 0) := "101";
    CONSTANT ADDR_DPTR_H : std_logic_vector(2 downto 0) := "110"
  );

  PORT (
    pc : out std_logic_vector(15 downto 0) := "0000000000000000";
    inst : in std_logic_vector(15 downto 0) ;
    reset : in std_logic ;
    dptr : out std_logic_vector(15 downto 0) ;
```

```
    data : inout std_logic_vector(7 downto 0) ;
    wr_data_mem : out std_logic ;
    clk : in std_logic
  ) ;
end datapath;


architecture arch of datapath is

  --IF STAGE SIGNALS
  signal pc_in : std_logic_vector(15 downto 0) := "0000000000000000" ;
  signal pc_out : std_logic_vector(15 downto 0) := "0000000000000000" ;
  signal inst_if : std_logic_vector(15 downto 0) ;
  signal added_pc : std_logic_vector(15 downto 0) ;
  signal dois : std_logic_vector(15 downto 0) ;


  --ID STAGE SIGNALS
  signal inst_id : std_logic_vector(15 downto 0) ;
  signal rd_addr1_reg0 : std_logic_vector(2 downto 0) ;
  signal rd_addr2_reg0 : std_logic_vector(2 downto 0) ;
  signal data1_reg0 : std_logic_vector(7 downto 0) ;
  signal data2_reg0 : std_logic_vector(7 downto 0) ;
  signal rd_addr1_reg1 : std_logic_vector(2 downto 0) ;
  signal rd_addr2_reg1 : std_logic_vector(2 downto 0) ;
  signal data1_reg1 : std_logic_vector(7 downto 0) ;
  signal data2_reg1 : std_logic_vector(7 downto 0) ;
  signal data1 : std_logic_vector(7 downto 0) ;
  signal data2 : std_logic_vector(7 downto 0) ;
  signal imm : std_logic_vector(10 downto 0) ;
  signal ext_imm : std_logic_vector(15 downto 0) ;
  signal added_pc_id : std_logic_vector(15 downto 0) ;

  --EX STAGE SIGNALS
  signal data1_reg0_ex : std_logic_vector(7 downto 0) ;
  signal data1_ex : std_logic_vector(7 downto 0) ;
  signal data2_ex : std_logic_vector(7 downto 0) ;
  signal imm_ex : std_logic_vector(10 downto 0) ;
  signal ext_imm_ex : std_logic_vector(15 downto 0) ;
  signal sft_ext_imm : std_logic_vector(15 downto 0) ;
  signal added_pc_ex : std_logic_vector(15 downto 0) ;
  signal pc_offset_sum : std_logic_vector(15 downto 0) ;
  signal dptr_h_ex : std_logic_vector(7 downto 0) ;
  signal jmp_addr_ex : std_logic_vector(15 downto 0) ;
  signal ula_op2 : std_logic_vector(7 downto 0) ;
  signal flags : std_logic_vector(2 downto 0) ;
  signal ula_result : std_logic_vector(7 downto 0) ;
  signal wr_addr_ex : std_logic_vector(2 downto 0) ;


  --MEM STAGE SIGNALS
    signal ula_result_mem : std_logic_vector(7 downto 0) ;
  signal dptr_h_mem : std_logic_vector(7 downto 0) ;
  signal data1_reg0_mem : std_logic_vector(7 downto 0) ;
  signal mem_read : std_logic_vector(7 downto 0) ;
  signal added_pc_mem : std_logic_vector(15 downto 0) ;
  signal jmp_addr_mem : std_logic_vector(15 downto 0) ;
  signal wr_addr_mem : std_logic_vector(2 downto 0) ;
```

```
--WB STAGE SIGNALS
signal wr_addr : std_logic_vector(2 downto 0) ;
signal mem_read_wb : std_logic_vector(7 downto 0) ;
signal ula_result_wb : std_logic_vector(7 downto 0) ;
signal added_pc_wb : std_logic_vector(15 downto 0) ;
signal wr_data : std_logic_vector(7 downto 0) ;
signal wr_addr_reg0 : std_logic_vector(2 downto 0) ;
signal wr_addr_reg1 : std_logic_vector(2 downto 0) ;
signal wr_data_reg0 : std_logic_vector(7 downto 0) ;
signal wr_data_reg1 : std_logic_vector(7 downto 0) ;

--CONTROL SIGNALS
signal pc_src      :  std_logic ;
signal reg0_src    :  std_logic ;
signal reg1_src    :  std_logic_vector (1 downto 0) ;
signal data1_sel   :  std_logic ;
signal data2_sel   :  std_logic ;
signal ula_src     :  std_logic ;
signal jmp_src     :  std_logic ;
signal ula_op    :  std_logic_vector(3 downto 0) ;
signal wr_mem    :  std_logic ;
signal wb_src      :  std_logic ;
signal wr_reg0     :  std_logic ;
signal wr_reg1     :  std_logic ;
signal wr_ret      :  std_logic ;


component Reg_file is
  port (
    rd_addr1 : in std_logic_vector(2 downto 0);
    rd_addr2 : in std_logic_vector(2 downto 0);
    wr_addr: in std_logic_vector(2 downto 0);
    wr_enable : in std_logic;
    wr_data : in std_logic_vector(7 downto 0) ;
    clk : in std_logic;
    data1 : out std_logic_vector(7 downto 0) ;
    data2 : out std_logic_vector(7 downto 0)
  );
end component;

component Somador_16 is
  port (
    input_1 : in std_logic_vector(15 downto 0);
    input_2 : in std_logic_vector(15 downto 0);
    output : out std_logic_vector(15 downto 0)
  );
end component;

component UC is
  port (
    opcode     : in std_logic_vector(4 downto 0) ;
    reg_file   : in std_logic_vector(1 downto 0) ;
    flags      : in std_logic_vector(2 downto 0) ;
    clk      : in std_logic ;
    pc_src     : out std_logic := '0';
    reg0_src   : out std_logic ;
    reg1_src   : out std_logic_vector (1 downto 0) ;
    data1    : out std_logic ;
    data2      : out std_logic ;
    ula_src    : out std_logic ;
```

```
      jmp_src    : out std_logic ;
      ula_op     : out std_logic_vector(3 downto 0) ;
      wr_mem     : out std_logic := '0';
      wb_src     : out std_logic ;
      wr_reg0    : out std_logic := '0';
      wr_reg1    : out std_logic := '0';
      wr_ret     : out std_logic
    );
  end component;

  component ULA is
    port (
      input_1 : in std_logic_vector(7 downto 0);
      input_2 : in std_logic_vector(7 downto 0);
      shift_amnt: in std_logic_vector(2 downto 0);
      ula_op : in std_logic_vector(3 downto 0);
      flags : out std_logic_vector(2 downto 0); --flag(2) = carry; flag(1) = zero; flag
          (0) = negativo
      output : out std_logic_vector(7 downto 0)
    );


  end component;

begin

  pc_adder : Somador_16 port map (input_1 => pc_out,
                  input_2 => dois,
                  output => added_pc);

  banco_reg0 : Reg_file port map (rd_addr1 => rd_addr1_reg0,
                  rd_addr2 => rd_addr2_reg0,
                  wr_addr => wr_addr_reg0,
                  wr_enable => wr_reg0,
                  wr_data => wr_data_reg0,
                  clk => clk,
                  data1 => data1_reg0,
                  data2 => data2_reg0);

  banco_reg1 : Reg_file port map (rd_addr1 => rd_addr1_reg1,
                  rd_addr2 => rd_addr2_reg1,
                  wr_addr => wr_addr_reg1,
                  wr_enable => wr_reg1,
                  wr_data => wr_data_reg1,
                  clk => clk,
                  data1 => data1_reg1,
                  data2 => data2_reg1);

  inst_ula : ULA port map (input_1 => data1_ex,
                input_2 => ula_op2,
                shift_amnt => imm_ex(8 downto 6),
                ula_op => ula_op,
                flags => flags,
                output => ula_result) ;

  jmp_adder : Somador_16 port map(input_1 => added_pc_ex,
                  input_2 => sft_ext_imm,
                  output => pc_offset_sum) ;

  unidade_controle : UC port map( opcode     => inst_id(15 downto 11) ,
```

```
                reg_file  => inst_id(10 downto 9) ,
                flags     => flags ,
                clk       => clk ,
                pc_src    => pc_src ,
                reg0_src  => reg0_src ,
                reg1_src  => reg1_src ,
                data1     => data1_sel ,
                data2     => data2_sel ,
                ula_src   => ula_src ,
                jmp_src   => jmp_src ,
                ula_op    => ula_op ,
                wr_mem    => wr_mem ,
                wb_src    => wb_src ,
                wr_reg0   => wr_reg0 ,
                wr_reg1   => wr_reg1,
                wr_ret    => wr_ret) ;

process (clk)
begin
  if (falling_edge(clk)) then

    --MEM TO WB
    wr_addr <= wr_addr_mem ;
    ula_result_wb <= ula_result_mem ;
    mem_read_wb <= mem_read ;
    added_pc_wb <= added_pc_mem ;

    --EX TO MEM
    ula_result_mem <= ula_result ;
    dptr_h_mem <= dptr_h_ex ;
    added_pc_mem <= added_pc_ex ;
    data1_reg0_mem <= data1_reg0_ex ;
    wr_addr_mem <= wr_addr_ex ;
    jmp_addr_mem <= jmp_addr_ex ;

    --ID TO EX
    data1_ex <= data1 ;
    data2_ex <= data2 ;
    imm_ex <= imm ;
    ext_imm_ex <= ext_imm ;
    added_pc_ex <= added_pc_id ;
    wr_addr_ex <= inst_id(2 downto 0) ;
    data1_reg0_ex <= data1_reg0 ;
    dptr_h_ex <= data2_reg1 ;

    --IF TO ID
    inst_id <= inst_if ;
    added_pc_id <= added_pc ;

  end if ;

end process ;

inst_if <= inst ;

pc <= pc_out ;



process (clk,reset, pc_in)
```

```
begin
  if (reset = '1') then
    pc_out <= "0000000000000000" ;
  else
    if (falling_edge(clk)) then
      pc_out <= pc_in ;
    end if ;
  end if ;
end process ;

dois <= X"0002" ;
rd_addr2_reg1 <= ADDR_DPTR_H ;
rd_addr2_reg0 <= inst_id(5 downto 3) ;

process (added_pc, jmp_addr_mem, pc_src)
begin
  if (pc_src = '0') then
    pc_in <= added_pc ;
  else
    pc_in <= jmp_addr_mem ;
  end if ;
end process ;


process (inst_id(2 downto 0), inst_id(8 downto 6), reg0_src)
begin
  if (reg0_src = '0') then
    rd_addr1_reg0 <= inst_id(2 downto 0) ;
  else
    rd_addr1_reg0 <= inst_id(8 downto 6) ;
  end if ;
end process ;

process (wr_ret, wr_addr)
begin
  if (wr_ret = '0') then
    wr_addr_reg0 <= wr_addr ;
    wr_addr_reg1 <= wr_addr ;
  else
    wr_addr_reg0 <= ADDR_RET_L ;
    wr_addr_reg1 <= ADDR_RET_H ;
  end if ;
end process ;

process (wr_ret, added_pc_wb, wr_data)
begin
  if (wr_ret = '1') then
    wr_data_reg0 <= added_pc_wb(7 downto 0) ;
    wr_data_reg1 <= added_pc_wb(15 downto 8) ;
  else
    wr_data_reg0 <= wr_data ;
    wr_data_reg1 <= wr_data ;
  end if ;
end process ;

process (inst_id(5 downto 0), reg1_src)
begin
  if (reg1_src = "00") then
    rd_addr1_reg1 <= ADDR_DPTR_L ;
  elsif (reg1_src = "01") then
```

```
    rd_addr1_reg1 <= inst_id(2 downto 0) ;
  else
    rd_addr1_reg1 <= inst_id(5 downto 3) ;
  end if ;
end process ;

process (inst_id(10 downto 0))
begin
  if (inst_if(10) = '0') then
    ext_imm <= "00000" & inst_id(10 downto 0) ;
  else
    ext_imm <= "11111" & inst_id(10 downto 0) ;
  end if ;
end process ;


process (data1_reg0, data1_reg1, data1_sel)
begin
  if (data1_sel = '0') then
    data1 <= data1_reg0 ;
  else
    data1 <= data1_reg1 ;
  end if ;
end process ;

process (data2_reg0, data1_reg1, data2_sel)
begin
  if (data2_sel = '0') then
    data2 <= data2_reg0 ;
  else
    data2 <= data1_reg1 ;
  end if ;
end process ;

imm <= inst_id(10 downto 0) ;

process (ula_src, imm_ex(10 downto 3), data2_ex)
begin
  if (ula_src = '0') then
    ula_op2 <= data2_ex ;
  else
    ula_op2 <= imm_ex(10 downto 3) ;
  end if ;
end process ;

process (ext_imm_ex)
begin
  sft_ext_imm <= std_logic_vector(signed(ext_imm_ex) sll 1) ;
end process ;

process (data1_ex, data2_ex, pc_offset_sum, jmp_src)
begin
  if (jmp_src = '0') then
    jmp_addr_ex <= pc_offset_sum ;
  else
    jmp_addr_ex <= data2_ex & data1_ex ;
  end if ;
end process ;

dptr <= dptr_h_mem & ula_result_mem ;
```

```
  process (wr_mem, data1_reg0_mem , data)
  begin
    if (wr_mem = '0') then
      mem_read <= data ;
    else
      data <= data1_reg0_mem ;
      mem_read <= "----------" ;
    end if ;
  end process ;

  process (wb_src, ula_result_wb, mem_read_wb)
  begin
    if (wb_src = '0') then
      wr_data <= ula_result_wb ;
    else
      wr_data <= mem_read_wb ;
    end if ;
  end process ;

  wr_data_mem <= wr_mem ;
end;
```

Código A.6: Código utilizado para descrever a memória de programa para o caso de teste escolhido.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL ;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity prog_mem is


  port (
    addr : in std_logic_vector(15 downto 0) ;
    data : out std_logic_vector(15 downto 0)
  );


end prog_mem;


architecture arch of prog_mem is
begin
  process (addr)
  begin
    case to_integer(unsigned(addr)) is
      when 0 =>  data <= "0111100000000000" ; --MOV R0, 0
      when 2 =>  data <= "0111100000000001" ; --MOV R1, 0
      when 4 =>  data <= "0111100000011010" ; --MOV R2, 3
      when 6 =>  data <= "1111100000000000" ; --NOP (Espera WB de R0)
      when 8 =>  data <= "1111100000000000" ; --NOP
      when 10 =>   data <= "0000001000000101" ; --MOV DPTR_L, R0
      when 12 =>   data <= "0000001000000110" ; --MOV DPTR_H, R0
      when 14 =>   data <= "1111100000000000" ; --NOP  (Espera WB de DPTR_H)
      when 16 =>   data <= "1111100000000000" ; --NOP
      when 18 =>   data <= "1111100000000000" ; --NOP
```

```
      when 20 =>   data <= "1111100000000000" ; --NOP
      when 22 =>   data <= "1101011111001000" ; --LOOP: STR R1, R0
      when 24 =>   data <= "0001100000001000" ; --ADD R0, 1
      when 26 =>   data <= "0001100000010001" ; --ADD R1, 2
      when 28 =>   data <= "1111100000000000" ; --NOP  (Espera WB de R0)
      when 30 =>   data <= "1111100000000000" ; --NOP
      when 32 =>   data <= "1111100000000000" ; --NOP
      when 34 =>   data <= "0010011000010011" ; --CMP R0, R2, R3
      when 36 =>   data <= "1010011111111000" ; --BLT LOOP
      when 38 =>   data <= "1111100000000000" ; --NOP  (Espera a decisao do salto)
      when 40 =>   data <= "1111100000000000" ; --NOP
      when 42 =>   data <= "1111100000000000" ; --NOP
      when 44 =>   data <= "1111100000000000" ; --NOP
      when 46 =>   data <= "1000011111111111" ; --HERE: JMP HERE

            when others => data <= "1111100000000000" ; --NOP
    end case ;
  end process;
end;
```

Código A.7: Código utilizado para testar o processador com a memória de programa do código A.6.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL ;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

library work;
use work.all;

entity teste_mem is

PORT (
    pc : out std_logic_vector(15 downto 0) := "0000000000000000";
    inst : out std_logic_vector(15 downto 0) ;
    reset : in std_logic ;
    dptr : out std_logic_vector(15 downto 0) ;
    data : inout std_logic_vector(7 downto 0) ;
    wr_data_mem : out std_logic ;
    clk : in std_logic
  ) ;


end teste_mem;


architecture arch of teste_mem is
  component datapath is

    PORT (
      pc : out std_logic_vector(15 downto 0) := "0000000000000000";
      inst : in std_logic_vector(15 downto 0) ;
      reset : in std_logic ;
      dptr : out std_logic_vector(15 downto 0) ;
      data : inout std_logic_vector(7 downto 0) ;
      wr_data_mem : out std_logic ;
      clk : in std_logic
```

```
    ) ;
  end component;

  component prog_mem is
    port (
      addr : in std_logic_vector(15 downto 0) ;
      data : out std_logic_vector(15 downto 0)
    );
  end component;

  signal inst_sig : std_logic_vector(15 downto 0) ;
  signal addr : std_logic_vector(15 downto 0) ;

begin
  processador : datapath port map (pc => addr,
                    inst => inst_sig,
                    reset => reset,
                    dptr => dptr ,
                    data => data,
                    wr_data_mem => wr_data_mem,
                    clk => clk);

  memoria : prog_mem port map (addr => addr,
                  data => inst_sig) ;

  pc <= addr ;
  inst <= inst_sig ;
end;
```