PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

JARASper

Um Simulador de Arquiteturas Computacionais Flexível e Configurável ao Nível de Registradores

Philippe Jara de Mello Mesquita Martins

Projeto Final de Graduação

CENTRO TÉCNICO CIENTÍFICO - CTC

DEPARTAMENTO DE INFORMÁTICA

Curso de Graduação em Ciência da Computação

Rio de Janeiro, Junho de 2023

Philippe Jara de Mello Mesquita Martins

JARASper: Um Simulador de Arquiteturas Computacionais Flexível e Configurável ao Nível de Registradores

Relatório de Projeto Final, apresentado ao programa Ciência da Computação da PUC-Rio, como requisito parcial para a

obtenção do título de Bacharel em Ciência da Computação

Orientador: Edward Hermann Haeusler

Departamento de Informática – PUC-Rio

Rio de Janeiro

Junho 2023

Resumo

Jara, Philippe. Haeusler, Edward. JARAsper: Um Simulador de Arquiteturas Computacionais Flexível e Configurável ao Nível de Registradores. Rio de Janeiro, 2023. 60 p. Relatório de Projeto Final– Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

JARASper é um programa criado para servir como um ambiente de prototipação para arquiteturas computacionais no nível de registradores, barramentos e unidades lógicas, oferecendo construção de arquiteturas e opcodes customizáveis e ferramentas gráficas para sua simulação. Ferramentas como REPLs para opcodes e microcodes, assim como introspecção de memória.

Palavras-chave

registrador, opcode, microcode, simulador, unidade de controle

Abstract

Jara, Philippe. Haeusler, Edward. JARAsper: A Flexible and Configurable Simulator of Computer Architectures on the Register Level. Rio de Janeiro, 2023. 60 p. Relatório de Projeto Final–Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

JARASper is a program created to serve as a prototyping environment for computational architectures at the register, bus, and logic unit level, offering customizable architecture construction and opcodes, as well as graphical tools for simulation. It includes tools such as REPLs for opcodes and microcodes, as well as memory introspection.

Keywords

register, opcode, microcode, simulator, control unit

Sumário

Lista de Figuras

1. Introdução

O aprendizado de qualquer disciplina suficientemente complexa pode ser caracterizado como uma construção de conhecimento sobre camadas prévias. Desde o estudo da biologia até o de conceitos mais abstratos da matemática, tudo necessita de uma base para se sustentar. O conhecimento sem base nada mais é do que informação sem contexto, que não terá utilidade fora o uso mais imediato e literal, e na ciência da computação, o cenário não é diferente.

Assim como no conhecimento em geral, computadores são criados e configurados em camadas, começando desde o material mais primitivo até sua forma completa, em toda sua complexidade. Processadores, memórias e outros componentes hoje chegam a ser implementados a partir de bilhões de transistores contidos em uma área não maior do que a cabeça de um alfinete. Seguindo a ideia de estruturação em camadas, é possível reconhecer diversos níveis de abstração em que se pode trabalhar na criação e no entendimento de tais sistemas.

Um dos níveis mais relevantes para a análise dos sistemas computacionais hoje é o *nível dos registradores*. Registradores são, efetivamente, pequenos *recipientes* que contêm informação, e com base nestas informações e nos mecanismos de controle do fluxo, transformação e transporte dessas informações, é possível modelar praticamente todos os computadores comuns. Com mais alguns módulos básicos e primitivos (como barramentos, memória, unidades lógicas e de controle, entre outras), é possível descrever estruturas que representam a organização de qualquer máquina computacional que hoje se encontra no mercado ou na literatura.

A partir da utilização de registradores e outros componentes básicos, é possível imaginar ferramentas para prototipação focadas no aprendizado de sistemas computacionais, bem como em sua experimentação tendo em vista a criação de novas arquiteturas. Embora a ideia não seja completamente nova, as alternativas de ferramentas existentes geralmente recaem em pelo menos um de dois problemas:

1. Carecem de uma forma para livre criação de arquiteturas próprias, oferecendo somente uma já definida, como o MIPS (PATTERSON e HENNESSY, 2013), o JASPer (BURRELL, 2004) e o MMIX (KNUTH, 1999).
2. Não oferecem mecanismos para o desenvolvimento em níveis mais altos de abstração, levando, não raramente, ao uso de linguagens do tipo *Hardware Description Languages* (HDL), a exemplo da Verilog (VERILOG) e VHDL (IEEE COMPUTER SOCIETY, 2008), que exigem um nível de detalhamento alto, além de lidarem com conceitos de nível de abstração inferior ao desejado em uma prototipação rápida.

1.1. Introdução dos objetivos do projeto e metas atingidas

O desenvolvimento deste projeto é feito sobre o programa Jarasper "1.0" (JARA, 2018), um programa já existente.

O programa Jarasper 1.0 foi criado como um ambiente de prototipação para arquiteturas computacionais. Ele engloba os níveis de registradores e unidades lógicas. O propósito do Jarasper 1.0 é recriar e expandir a funcionalidade do Jasper. Especificamente, proporciona maior flexibilidade na construção de opcodes e microcodes para a arquitetura Jasper. No entanto o programa está limitado a trabalhar exclusivamente com uma única arquitetura.

Este projeto demonstra o processo de experimentação e aprimoração do programa Jarasper e a validação de suas novas funcionalidades, indo da versão "1.0" para a versão "2.0". Em sua conclusão as seguintes funcionalidades foram adicionadas ao programa:

- Suporte para arquiteturas configuráveis pelo usuário
- Movimentação de dados baseado em barramentos
- Construção de arquiteturas por meio de interface gráfica
- REPL de Microcódigos
- Construção de conjuntos de microcódigos para arquiteturas genéricas
- Modificação de memória pela interface gráfica

1.2. Sobre o ambiente computacional

O projeto foi desenvolvido com o objetivo de ser um programa nativo com suporte multiplataforma. Inicialmente ele foi construído em um notebook utilizando

Linux Mint(LEFÈBVRE, 2023) depois um desktop usando windows 7(MICROSOFT, 2020) e rodando Arch Linux(VINET, GRIFFIN e POLYÁK, 2023) por meio de uma máquina virtual VMware(VMWARE. INC., 2023) e finalmente um desktop usando Windows 10 (MICROSOFT, 2022)com uma máquina virtual rodando Arch Linux. Tanto o programa quanto o build system mantém compatibilidade com todos estes sistemas operacionais.

Em sua forma final o desenvolvimento está escrito completamente na linguagem C++ 14(C++ FOUNDATION, 2020), com exceção dos arquivos de configuração do build system e especificação da interface gráfica (GUI). Para a construção da GUI foi utilizado Qt5(QT PROJECT, 2023) e o seu build system qmake(QT PROJECT, 2023).

Existem tecnologias que não aparecem no projeto final, mas que foram em algum momento integradas no projeto e/ou foram estudadas a fundo para que no futuro haja integração. Estas tecnologias são Embedded Common Lisp(KOCHMANSKI, 2022) e o backend do llvm(LATTNER, 2022) e sua linguagem intermediária(IR).

1.3. Sobre a adequação do trabalho como projeto final

Este trabalho é uma culminação dos conceitos, teorias e habilidades práticas adquiridas durante os estudos. Ele demonstra uma aplicação direta dos tópicos discutidos durante o curso, principalmente nas áreas de construção de linguagens de programação, lógica, arquitetura de computadores, algoritmos e orientação a objetos; utilizando estes aprendizados como ponto de largada para efetuar minhas próprias pesquisar de aprofundamento. Em adição aos conceitos de organização de código e gerenciamento de projetos também abordados durante o curso.

A conclusão do projeto destaca a integração do conhecimento teórico com habilidades práticas, tanto em organização e construção de software, assim como a capacidade de aprofundamento independente em tópicos avançados.

2. Situação Atual

2.1.Descrição e avaliação de tecnologias e sistemas existentes

2.1.1. JASPer

Desenvolvido por Mark Burrel em 1995 JASPer é um programa que tem como objetivo ensinar fundamentos de processadores. Ele apresenta uma CPU hipotética simples como uma exibição animada de registradores e caminhos de dados associados, permitindo que o usuário examine as atividades do processador envolvidas na execução de instruções de máquina.

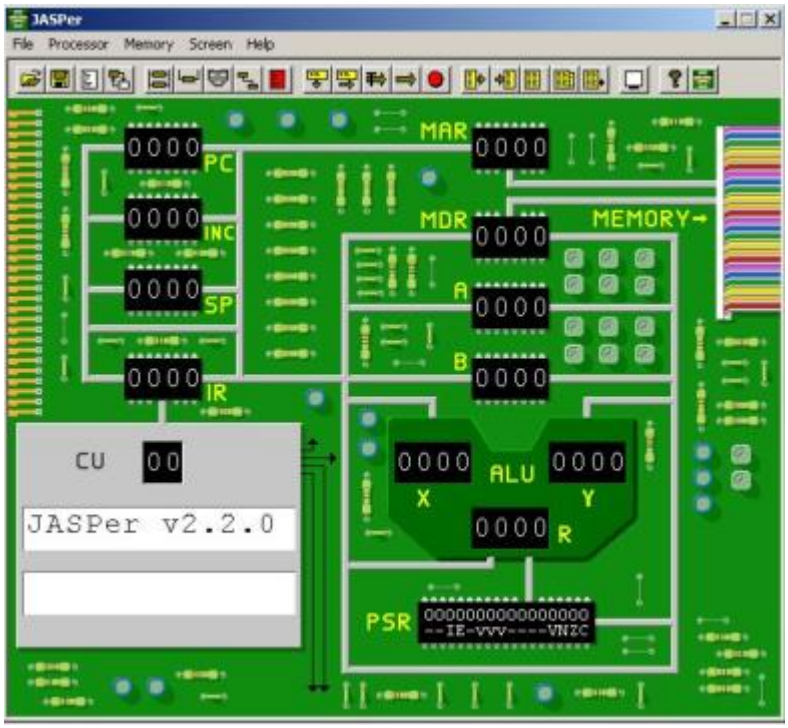


FIgura 1: Interface do JASPer

2.1.2. Verilog

Originalmente desenvolvido no início dos anos 1980 pela Gateway Design Automation Verilog é uma linguagem de descrição de hardware (HDL - Hardware Description Language) utilizada para projetar e modelar circuitos digitais. Ela

permite que os engenheiros de hardware descrevam o comportamento e a estrutura de um sistema digital, incluindo circuitos integrados, dispositivos eletrônicos e outros componentes digitais.

O Verilog, no entanto, é uma linguagem de descrição de hardware de baixo nível, complexa e com uma curva de aprendizado íngreme, o que a torna menos adequada para iniciantes e usuários procurando por prototipações rápidas e limitadas. Além disso, sua falta de visualização e abstração de nível mais baixo podem dificultar a compreensão dos conceitos básicos.

2.1.3. Jarasper 1.0

Jarasper 1.0 é um programa criado para servir como um ambiente de prototipação para arquiteturas computacionais no nível de registradores, barramentos e unidades lógicas. Ele foi desenvolvido para Linux na linguagem de programação C++ com Qt para sua interface gráfica. é a primeira versão do programa sendo desenvolvido neste projeto e a base em que o atual foi construído. Esta versão do programa tem como objetivo recriar e expandir a funcionalidade do Jasper, em particular permitir maior flexibilidade na construção de opcodes e microcodes para a arquitetura Jasper.

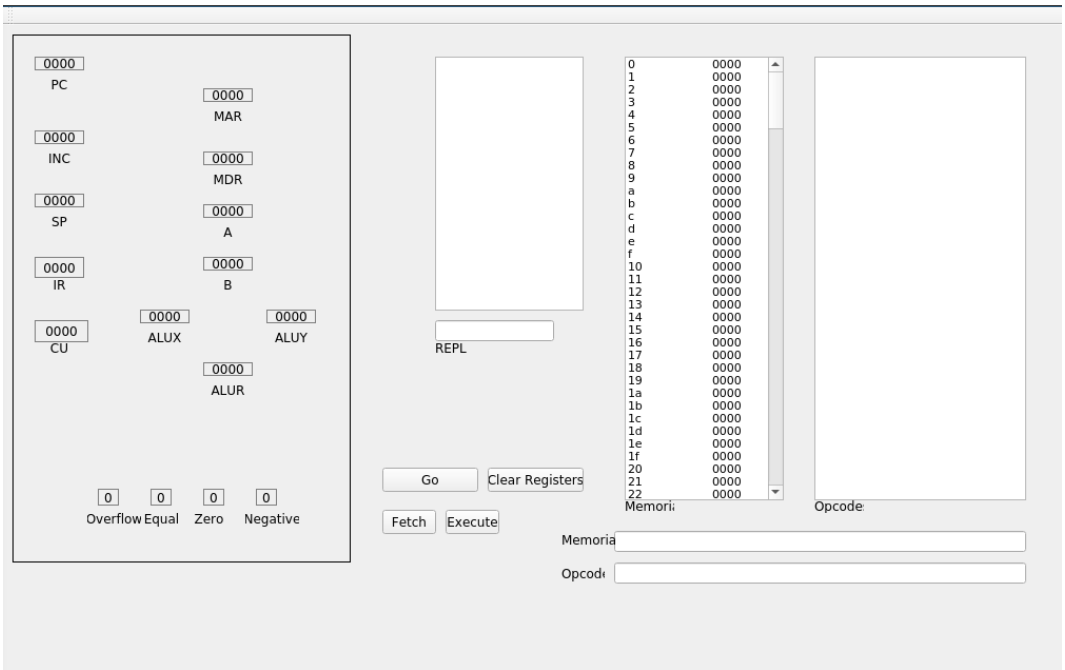


Figura 2: Jarasper 1.0 GUI ao iniciar

Os campos “memória” e "opcode"(figura1) recebem endereços que carregam arquivos de texto contendo o conteúdo de sua memória e a definição

dos opcodes. A listagem de opcodes e os valores em memória são carregados nas suas tabelas respectivas. O programa contido em memória é uma computação de 2^4 com o resultado sendo guardado na posição de memória 0x13.

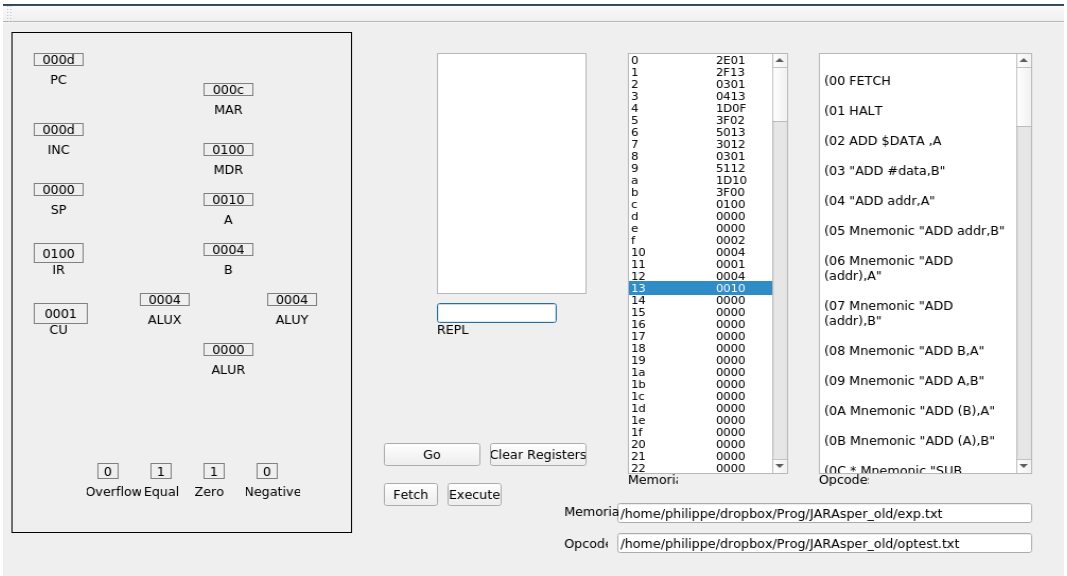


Figura 3: Jarasper 1.0 GUI com opcodes e memória carregados

2.1.3.1.- Análise dos componentes as serem expandidos

O jarasper 1.0 trabalha em sua simulação com os seguintes elementos principais: Componentes elementais; conjunto de microcódigos; Configuração de registradores; Interpretador; REPL;

2.1.3.1.1. Componentes elementais

Os componentes elementais do programa são os "tijolos" que permitem que todo o resto seja construído como combinações de tais componentes. Estes componentes são: Registrador; ALU (Unidade de lógica aritmética); Unidade de Controle e a Memória. Estes subcomponentes são atualmente definidos como subclasses de um componente.

2.1.3.1.1.1. Componente básico

```
class Component {
 public:
  unsigned int value;
  unsigned short bits;
  unsigned short type;
  MainWindow *window;
  QLabel *display;
  Component(){}
  Component(unsigned short, unsigned short);
  void assign(Component);
  void assign(Component, int);
  void assign(unsigned int val);
};
```

Figura 4: Declaração do componente básico em Jarasper 1.0

A classe que todos os outros componentes vão derivar seu funcionamento básico. bits se referem ao tamanho de bits do elemento, type define o tipo do subcomponente. Os valores dos registradores são unsigned para permitir o uso de complemento a dois nas simulações. Nota-se que Component nunca será usado por si só, somente suas sub-classes.

2.1.3.1.1.2. Registrador Genérico

```
class GeneralPurposeRegister : public Component{
 public:
  GeneralPurposeRegister();
  GeneralPurposeRegister(unsigned short, unsigned short);

};
```

Figura 5: Declaração de registrador genérico em Jarasper 1.0

O registrador genérico, efetivamente igual ao componente básico

2.1.3.1.1.3. ALU (unidade de lógica aritmética)

ALU em si é composta por 3 registradores e 4 flags, no entanto somente são simulados 2 registradores (x,y), o registrador de resultado é tratado

internamente.

```
class ALU : public Component {
 public:
  GeneralPurposeRegister *x;
  GeneralPurposeRegister *y;
  int overflow;
  int zero;
  int negative;
  int equal;
  unsigned int r;

  ALU();
  ALU(GeneralPurposeRegister *, GeneralPurposeRegister *);
  void checkFlags(int,unsigned int) ;
  void add() ;
  void sub() ;
  void bitShiftL(unsigned int);
  void bitShiftR(unsigned int);
};
```

Figura 6: Declaração de alu em Jarasper 1.0

2.1.3.1.1.4. Unidade de Controle

```
class ControlUnit : public Component{
 public:
  ControlUnit();
  ControlUnit(unsigned short, unsigned short);
};
```

Figura 7: Declaração da unidade de controle em Jarasper 1.0

Similar ao registrador genérico, é igual ao componente básico.

2.1.3.1.1.5. MDR (memory data register) e MAR (memory adress register)

```
class MemoryAdressRegister : public Component{
 public:
  MemoryAdressRegister();
  MemoryAdressRegister(unsigned short, unsigned short);

  void assignToMemory( std::string[], unsigned short);
};
```

Figura 8: Declaração de MAR em Jarasper 1.0

```
class MemoryDataRegister : public Component{
 public:
  MemoryDataRegister();
  MemoryDataRegister(unsigned short, unsigned short);

  void assignFromMemory( std::string[], unsigned short);
};
```

Figura 9: Declaração de MDR em Jarasper 1.0

Ambos são similares ao registrador genérico, com a única diferença sendo a capacidade de ler e escrever na memória utilizando assignToMemory() e assignFromMemory().

2.1.3.1.1.6. Outros Subcomponentes

Jarasper 1.0 contém ainda mais subcomponentes, estes sendo: program conter, stack pointer e instruction register. As suas implementações, no entanto, são idênticas a um registrador, pelo motivo que eles não passam de registradores com funções específicas na arquitetura.

2.1.3.1.1.7. Classe mainWindow do Qt

Todo programa de utilizando Qt como interface gráfica tem uma classe referente à sua janela principal, e na versão estável do programa ela é quem contêm todos estes elementos, logo é relevante saber o seu formato.

```
class MainWindow : public QMainWindow
{
  friend struct ComponentList;
  friend class Component;
  Q_OBJECT
 public:
  std::string mem[MEMSIZE],REPLmem[MEMSIZE],
    opcode[MAXOPCODE][MAXMICROC],Dopcode[100][MAXMICROC];
  GeneralPurposeRegister A, B, ALUx, ALUy;
  MemoryAdressRegister MAR;
  MemoryDataRegister MDR;
  StackPointer SP;
  ProgramCounter PC;
  Incrementer INC;
  InstructionRegister IR;
  ALU alu;
  ControlUnit CU;
  ComponentList cl;
  int opcodeCount, memCount;
  void clean_gui();
  void update_gui(std::string);
  void reset_all_comp_values();
  void update_all_components();
  void update_mem(unsigned short);
  void setDisplay(Component*);
  void setA(unsigned int);
  void setB(unsigned int);
  void setALUx(unsigned int);
  void setALUy(unsigned int);
  void setALUr(unsigned int);
  void setIR(unsigned int);
  void setPC(unsigned int);
  void setSP(unsigned int);
  void setCU(unsigned int);
  void setINC(unsigned int);
  void setMAR(unsigned int);
  void setMDR(unsigned int);
  void setOverflow(unsigned int);
  void setZero(unsigned int);
  void setEqual(unsigned int);
  void setNegative(unsigned int);
  explicit MainWindow(QWidget *parent = 0);
  ~MainWindow();
  private slots:
    void on_MEM_GO_clicked();
    void on_REPL_INPUT_returnPressed();
    void on_CLEAR_BUTTON_clicked();
    void on_MEM_LOCATION_returnPressed();
    void on_OPCODE_LOCATON_returnPressed();
    void on_MEM_FETCH_clicked();
    void on_MEM_EXECUTE_clicked();
    void on_MEMORY_VIEW_itemChanged(QListWidgetItem *item);
 private:
    Ui::MainWindow *ui;
```

Figura 10: Declaração de mainWindow em Jarasper 1.0

Esta classe é a responsável por orquestrar todo o movimento de dados entre os registradores simulados e a memória.

2.1.3.1.2. Conjunto de microcódigos e construção de opcodes

2.1.3.1.2.1. Conjunto de Microinstruções

O conjunto de microinstruções que permite o usuário gerar suas microcódigos, e consequentemente juntar tais microinstruções em um opcode. Os

microcódigos oferecidos são: "Assignment", "Add", "Sub", "Read from Memory Adress", "Write to Memory Adress", "Shift Left", "Shift Right", "Halt", “flag check”.

As microinstruções são representadas desta forma:

Assignment	<-
Shift Left	<<
Shift Right	>>
Add	+
Subtract	-
Read from Memory Adress*	MDR<-/MAR
Write to Memory Adress*	/MAR<-MDR
Flag check**	If PSRx==1

*: estes são específicos para MAR e MDR.

**para este, o valor de x pode ser qualquer uma das 4 flags da ALU, caso a flag em questão não esteja zerada o opcode termina a execução.

2.1.3.1.2.2. Conjunto de opcodes

Uma vez com os microcódigos prontos, podemos construir os opcodes. Eles são formados com “(“ seguido de qualquer coisa até fim da linha, normalmente o valor de seu código e um mnemônico. Após a primeira linha seguem as microinstruções, uma por linha até terminar em uma linha que tem somente “)”. Linhas podem ser comentadas utilizando “;”. Exemplos abaixo:

```
(00 FETCH
MAR<-PC
INC<-PC
PC<-INC
MDR<-/MAR
IR<-MDR
CU<-/IR
)
```

```
(05 Mnemonic "ADD addr,B"
;Description "Add to reg. B from a direct addr."
MAR<-'IR
MDR<-/MAR
ALUy<-MDR
ALUx<-B
ALUx+ALUy
B<-ALUr
)
```

```
(05 Mnemonic "ADD addr,B"
;Description "Add to reg. B from a direct addr."
MAR<-'IR
MDR<-/MAR
ALUy<-MDR
ALUx<-B
ALUx+ALUy
B<-ALUr
)
```

FIgura 11: Definição de opcodes para Jarasper 1.0

2.1.3.1.2.3. Interpretador:

Responsável por interpretar os opcodes criados pelo usuário e executar suas microinstruções sobre o sistema, atrelado à Unidade de Controle.

2.1.3.1.2.4. REPL (read-evaluate-print-loop):

Uma ferramenta que permite colocar opcodes diretamente na unidade de controle do programa e executar, efetivamente pulando o passo de fetch em um fetch-execute loop.

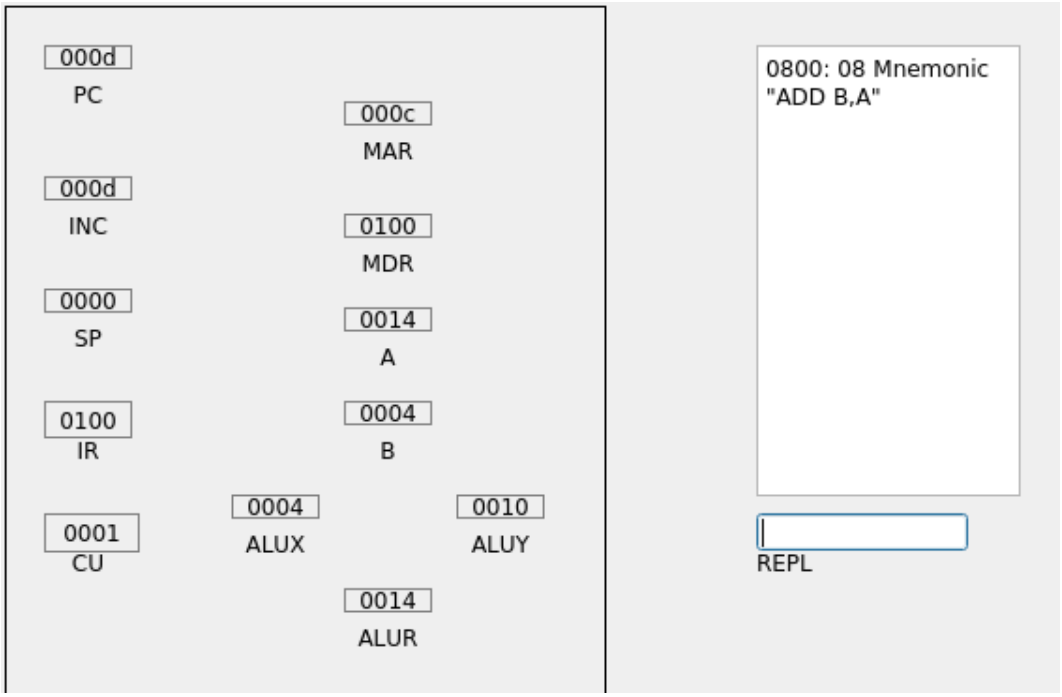


FIgura 12: REPL fazendo operação ADD B,A

3. Objetivos

3.1. Objetivos do trabalho

Em seu estado atual o JARAsper está funcional, mas para poder dividir espaço com os produtos que sua proposta inicial almejava (MIPS (PATTERSON e HENNESSY, 2013), Jasper (BURRELL, 2004), MMIX (KNUTH, 1999)) são necessárias e propostas aprimorações. Estas aprimorações são divididas em cinco campos: Correção de bugs; Suporte Multiplataforma; Aprimoração da Sintaxe e ferramentas linguísticas; Aprimoração da GUI e Implementação de operações por meio de barramentos.

3.1.1.Verificação e Correção de Bugs:

O projeto ainda tem certos problemas que precisam ser evitados manualmente, será necessário testar seu código para encontrar estes erros e áreas mal definidas para garantir completo funcionamento.

3.1.2. Suporte Multiplataforma:

Atualmente JARASper tem suporte para Linux 32bits e 64bits, suas versões em Microsoft Windows funcionam, mas tem bugs adicionais e certos problemas específicos relacionados à GUI, O mesmo ocorre com sua versão de FreeBSD, no entanto em uma escala maior.

3.1.3. Aprimoração da Sintaxe e Ferramentas Linguísticas:

Atualmente a sintaxe que o programa utiliza para construir os OPCODES por meio de microcodes é uma modificação da utilizada em Jasper com algumas mudanças e adições. Uma adição em particular é o encadeamento de operações em uma única linha, que no momento funciona somente para "assignment".

Em relação às ferramentas linguísticas é proposta a adição de mais camadas de abstração ao usuário. No momento o nível mais alta de programação é a utilização do REPL (read-evaluate-print-loop) para colocar opcodes diretamente na Unidade de controle, pulando assim o FETCH cycle, e a forma mais abstrata da representação de código é o opcode em si. Mais camadas de abstração possíveis sobre tal opcodes são desejadas, em primeiro momento permitir que opcodes possam ser utilizados no REPL por meio de *aliases* definidos junto com o opcode, e permitir a construção e utilização no REPL de "funções" compostas por um conjunto de opcodes.

3.1.4. Revisão da lingugagem de microcódigos para funcionamento genérico:

Jarasper 1.0 tem uma arquitetura pré-definida, para que a generalização das operações seja possível é necessário generalizar a linguagem dos microcódigos.

3.1.5. Aprimoraçao da GUI:

3.1.5.1. Parelelismo:

A GUI do JARASper no momento não tem suporte para diversos threads ou nenhuma outra forma de paralelismo. Isto é um grande problema pois o programa utiliza de animações para demonstrar os dados se movendo entre as áreas do sistema, impedindo o usuário de interagir com a janela, e em programas que tem loops infinitos tem somente a opção de fechar o programa para retomar controle.

3.1.5.2. Edição da Memória:

Uma vez carregada no programa, toda edição da memória externa só pode ser feita por meio de operações simuladas com opcodes, passando pelo MAR (Memory Access Register). Isto não é ideal pois para prototipação e ensino é conveniente poder modificar diretamente a memória externa sem tocar no estado do resto da máquina.

3.1.6. Implementação de operações por meio de barramentos:

A versão atual do programa não leva em conta o conceito de barramentos, limitando o escopo de simulação possível pelo programa, com isso em mente a implementação deste conceito em sua construção é um dos objetivos.

A ideia por trás do funcionamento da lógica do sistema é em construir sistemas usando somente os componentes básicos, componentes "básicos” são caracterizados como componentes que oferecem operações "exclusivas". Estes, no entanto podem necessitar de outros componentes para serem criados, como no caso da unidade de controle necessitando de um registrador interno.

O funcionamento do sistema tem como modelo teórico o seguinte:

A unidade de controle somente pode controlar as flags IN e OUT dos registradores, a transferência de dados "fluiria" naturalmente para os buses conectados e para os registradores que estiverem conectados pelo bus e com suas flags IN ligadas. Isto ocorre da mesma forma nos componentes básicos compostos por registradores (ALU e unidades de controle) e nas memórias externas (usando os buses conectados aos MAR's e MDR's).

Neste modelo existe um fator importante: A unidade de controle, os registradores e os barramentos não precisam ter conhecimento da "direção" dos dados que estão sendo movimentados pelo sistema, cada um faz seu papel de forma completamente independente, respeitando somente os clocks. Isto permite uma capacidade de composição excelente. Não é possível implementar algo 100% desta forma, no entanto manter estas propriedades é o foco principal do código de lógica dos componentes e o modo com que isso é feito será detalhado nas implementações.

4. Atividades Realizadas

4.1. Estudos Preliminares

Antes de iniciar o projeto final, já tinha um bom conhecimento sobre programação em C++, e experiência básica com Qt. Eu já havia completado a versão inicial do JARAsper para o PIBITI então existia alguma familiaridade. Conhecimento de common lisp por meio do Steel Bank Common Lisp(NEWMAN, 2023) também já era pré-existente.

4.2. Testes e protótipos para aprendizado e demonstração

Os principais protótipos em relação à features foram os relacionados à integração de um interpretador de common lisp no REPL por meio do ECL (KOCHMANSKI, 2022)à geração de código independente de plataforma e introspecção de métodos de otimização por meio da IR do LLVM, à construção da arquitetura de forma dinâmica na própria GUI, e a utilização do modelo de fluxo de dados por barramentos.

4.2.1. Integração de Embedded Common Lisp(ECL) no REPL:

ECL é uma implementação de Common Lisp para sistemas embarcados. Ele oferece várias formas de compilar o seu interpretador (executáveis, shared libraries, static libraries, .o, e arquivos .fasl carregados pelo interpretador) e oferece bindings para acessar tal interpretador por meio do código de C++.

Uma versão mais avançada do Jarasper 1.0 já tinha o ECL no REPL, no entanto ele não interagia com o programa, funcionando de forma isolada, recebendo código para eval e retornando o output no console do REPL. O objetivo principal deste protótipo era conseguir utilizar o interpretador para fazer o parsing da linguagem, facilitando assim a manipulação de código e programas externos, algo particularmente importante para a interação com o IR da LLVM.

Segue abaixo seu setup e funcionamento:

```lisp
;;;; parser.lisp
(defpackage :parser
  (:use :cl :cl-ppcre :alexandria)
  (:export :load-notice :looker))
(in-package :parser)
(print "FASL loaded")
(defun looker (look)
  (describe look)
  (princ look))
(defun load-notice ()
    (princ (ppcre:all-matches-as-strings ".+" "PPCRE"))
    )
(defvar path "~//projects//jarasper//llvmir.ll")

(defun parse-llvm-IR (path)
  (with-open-file (stream path)
    (cl-ppcre:all-matches-as-strings "define.+?{[.+?]}" stream)))

(defun read-all-lines-to-string (path)
  (with-open-file (stream path)
    (let (string final-string)
      (loop for temp-string = (read-line stream :eof-error-p nil)
         :while temp-string
         :do (append final-string temp-string )))))
```

FIgura 13: Código de ECL para ser compilado embarcado

```cpp
extern "C"{
//extern void init_lib_RUNTIME__ALL_SYSTEMS(cl_object);
extern void init_lib_RUNTIME__ALL_SYSTEMS(cl_object);
}
void inject_ecl(int argc,char** argv){
/* Initialize ECL */
cl_boot(argc, argv);

/* Initialize the library we linked in. Each library
 * has to be initialized. It is best if all libraries
 * are joined using ASDF:MAKE-BUILD.
 */

ecl_init_module(NULL, init_lib_RUNTIME__ALL_SYSTEMS);
cl_load(1, c_string_to_object("\"/home/philippe/projects//jarasper/lisp_files/parser/parser--all-systems.fasb\""));
cl_eval(c_string_to_object("(princ (+ 1 1 ))"));
atexit(cl_shutdown);

}
```

FIgura 14: Carregando o código da Figura 13 em C++

O propósito principal desta ferramenta seria facilitar a interação dinâmica com o IR do LLVM.

4.2.2. LLVM (IR):

LLVM (Low Level Virtual Machine) é uma infraestrutura de código aberto utilizada para desenvolver compiladores, otimizadores e outras ferramentas relacionadas à construção de software.

O LLVM IR (Intermediate Representation) é uma linguagem de programação de baixo nível e independente de plataforma. Ela atua como uma representação intermediária entre o código-fonte e o código de máquina final gerado pelo LLVM. A IR é projetada para capturar a semântica essencial do programa e facilitar otimizações e transformações.

Uma das características distintivas do LLVM é sua arquitetura modular, que inclui os chamados backends. Os backends do LLVM são responsáveis por gerar o código de máquina específico de uma determinada arquitetura de hardware ou plataforma de destino. Esses backends traduzem o LLVM IR otimizado em instruções de baixo nível que podem ser executadas diretamente pelo processador.

Cada backend é desenvolvido para uma arquitetura específica, como x86, ARM, MIPS, entre outras. Eles implementam as instruções e otimizações adequadas para aproveitar ao máximo o hardware subjacente. Dessa forma, o LLVM pode gerar código de máquina eficiente e otimizado para diferentes plataformas.

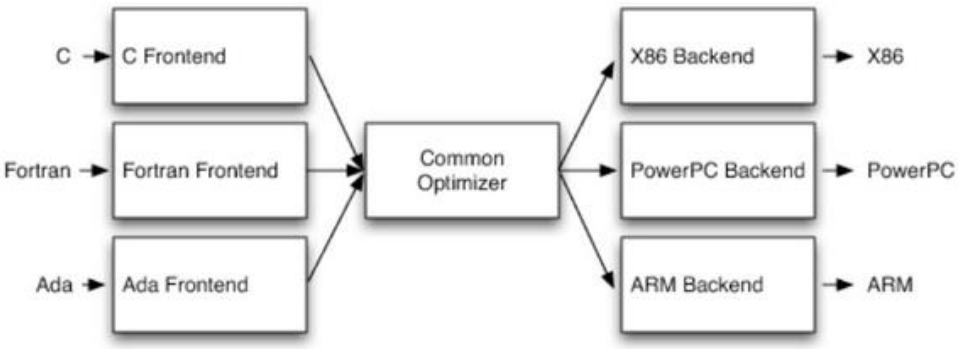


FIgura 15: Representação da estrutura do LLVM Core

Os backends do LLVM são essenciais para a portabilidade da IR. Com o mesmo conjunto de otimizações aplicadas ao LLVM IR, é possível gerar código

de máquina para uma variedade de arquiteturas, simplificando o suporte a diferentes sistemas e dispositivos.

Por meio desta ferramenta seria possível criar backends para as arquiteturas desejados no JARAsper e permitir que qualquer linguagem que tenha um frontend pronto para LLVM (estas incluem haskell, Javascript, Fortran, rust) criar código em já otimizado para sua arquitetura e na sua linguagem de máquina definida. Adicionalmente as otimizações são compostas por vários passos independentes, possibilitando a exploração de etapas de otimização intermediárias e como que elas impactam o código em questão.

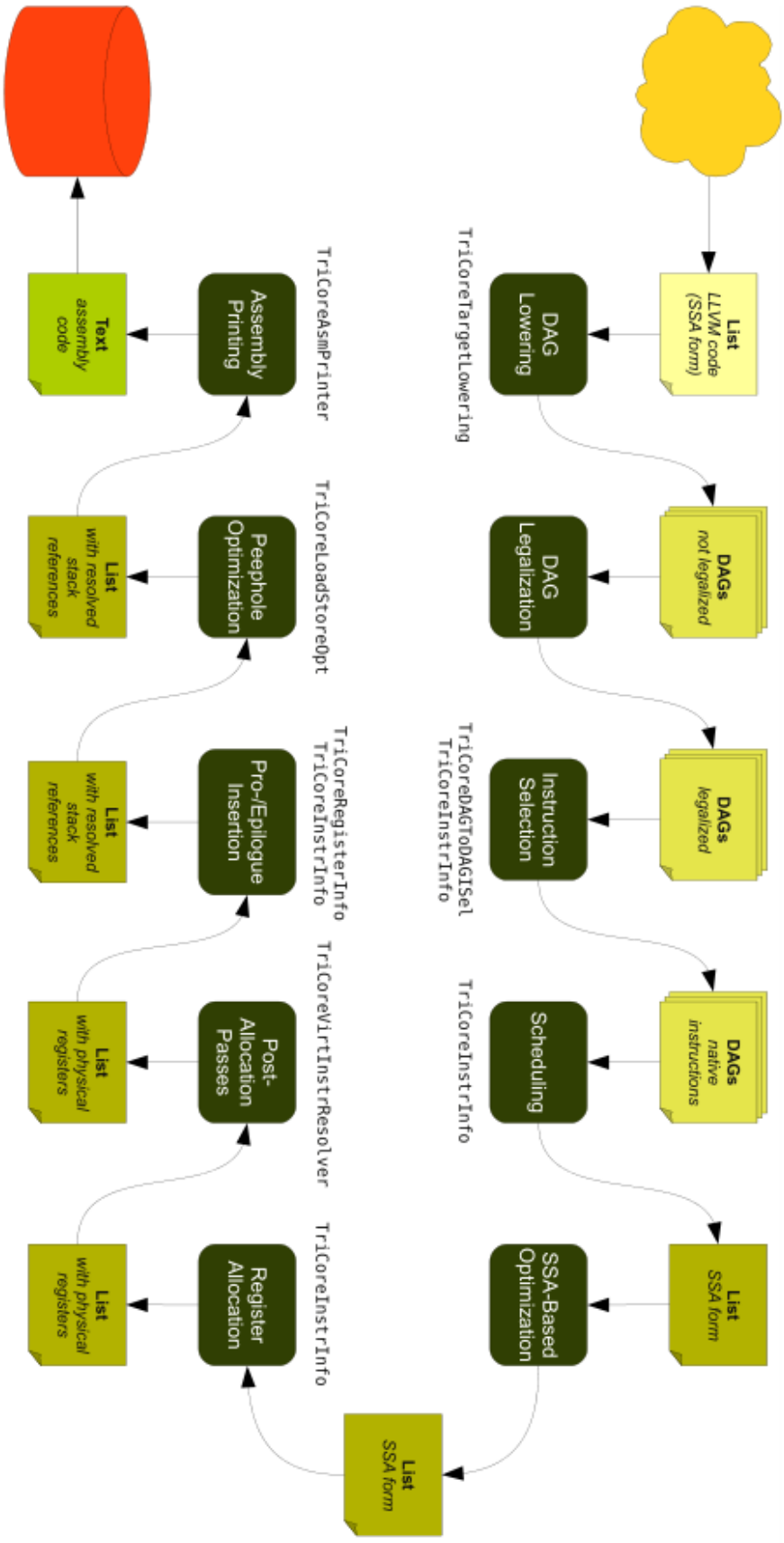


FIgura 16: Diagrama dos passos de otimização da IR (Erhardt, 2009)

O potencial desta ferramenta para o projeto é muito grande, no entanto acabou sendo muito complexo. Para conseguir fazer com que ela funcionasse para a arquitetura Jasper que o Jarasper 1.0 utiliza e para arquiteturas e linguagens genéricas definidas pelo usuário seria necessário mais tempo para implementação.

4.2.3.Construção Dinâmica de Arquitetura pela GUI:

Devido à generalização alcançada pelo conceito dos componentes durante o período de projeto final I foi explorada a ideia de o usuário conseguir criar suas próprias arquiteturas por meio da GUI, sem precisar recompilar o programa. Ao fim do período em questão o seguinte protótipo foi criado (com exceção da memória) pela interface gráfica:

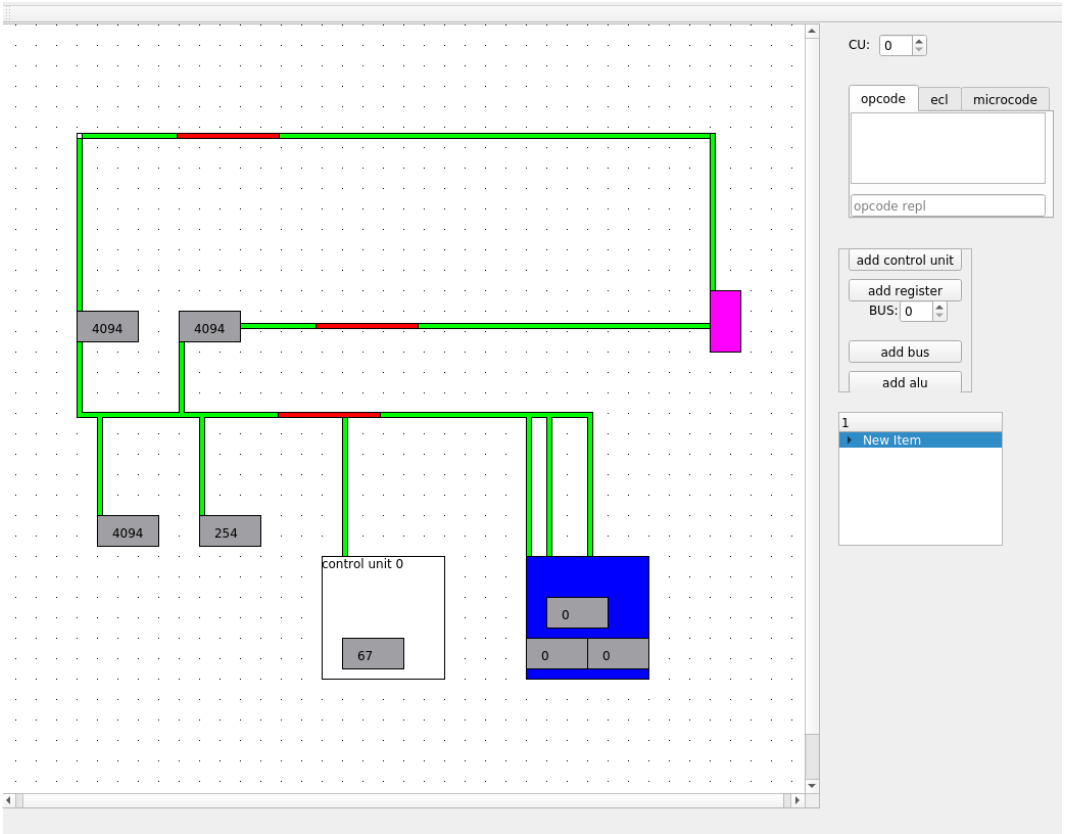


FIgura 17: Protótipo da GUI

Este protótipo contém os seguintes componentes:

- Unidade de controle(branco)

- Registrador interno da unidade de controle
- ALU(azul)
- 3 registradores internos da ALU, dois para argumentos e um para resultado
- Memória(ciano)
- MDR(memory adress register) e MAR(memory data register) atrelados à memória
- 2 barramentos exclusivos para conectar MAR e MDR à memória
- 2 registradores avulsos
- 1 barramentos que conecta todos os registradores entre si

Para adicionar um componente no protótipo era necessário escolher uma unidade de controle que iria fornecer o seu escopo, clock e conjunto de opcodes e um barramento para estar conectado inicialmente. A escolha de unidade de controle e barramento são feitas escolhendo o número deles nos menus CU e BUS. O trecho vermelho nos barramentos é uma âncora para mover os barramentos, um para cada barramento.

Os elementos gráficos custom criados pelo Qt por meio do QGraphicsPath para este protótipo foram dois, uma para um retângulo genérico e outro para o barramento que conecta os retângulos genéricos, assim como a lógica para movimentação dos barramentos:

```
class CustomRectItem : public QObject , public QGraphicsRectItem
{

Q_OBJECT
public:
  QGraphicsSimpleTextItem info;
  CustomRectItem(QGraphicsItem* parent = 0);
  CustomRectItem(const QRect& rect, QGraphicsItem* parent = 0);
  void setText(QString);

signals:
  void pos_change(CustomRectItem*);

protected:
    QVariant itemChange(GraphicsItemChange change,
                        const QVariant &value);
};
```

FIgura 18: Declaração do retângulo genérico

O método pos_change(CustomRectItem*) irá enviar o sinal para os barramentos conectados a este registrador se ajustarem caso o usuário o mova na tela.

```
class custom_bus_item : public QObject , public QGraphicsPathItem
{

  Q_OBJECT
public:
  QGraphicsRectItem base_bus;
  std::unordered_map<CustomRectItem *,
                     std::unique_ptr<QPainterPath>> linked_registers;
  custom_bus_item(QGraphicsItem* parent = 0, size_t wid = 5);
  void paint(QPainter *painter,
             const QStyleOptionGraphicsItem *,
             QWidget *);

  void link(CustomRectItem *);
  void remove_link(CustomRectItem *);

public slots:
  void update_path(CustomRectItem*);

private:
  size_t width;
};
```

FIgura 19: Declaração do barramento gráfico

Os métodos link(CustomRectItem*) e remove_link(CustomRectItem*) são os responsáveis por adicionar ou remover registradores do barramento, afetando o mapa "linked_registers". update_path é o slot que recebe os sinais de movimento dos registradores conectados e modificam o caminho do barramento em resposta.

4.2.4. Implementação inicial do modelo de fluxo de dados por barramentos:

Como previamente mencionado em objetivos, é necessário permitir que o fluxo de dados seja controlado por um clock global e simule a passagem de dados por barramentos, permitindo assim que erros e corrupções ocorram caso cuidado não seja tomado para como que os registradores estão abrindo e fechando seu acesso aos barramentos.

Para fazer com que isto funcionasse foi necessário refazer todos os componentes e grande parte da lógica.

4.2.4.1. Barramento:

```
class bus : public QObject{
Q_OBJECT
public:
  int bits;
  std::bitset <max_bits> info;
  void set (int arg);
  bus (int bits);
  bus (int inf, int bits);
}
```

FIgura 20: Declaração do protótipo do barramento

A variável bits representa o tamanho do barramento em bits, a variável "info" é a informação atualmente no barramento, set() é uma função para modificar o valor de "info". Ela tem dois construtores, um dando um valor inicial e outro não.

4.2.4.2. Registrador:

```
class regist : public QObject {
Q_OBJECT
public:
  size_t bits;
  size_t id;
  std::bitset<max_bits> info;
  std::vector<std::shared_ptr<bus>> in;
  std::vector<std::shared_ptr<bus>> out;
  mov_cnt<QLabel> * display;
  regist();
  regist(size_t bits,size_t id);
  regist(size_t bits,size_t id, QWidget *parent);
  void link_in(std::shared_ptr<bus> arg);
  void link_out(std::shared_ptr<bus> arg);
  void remove_link_in(std::shared_ptr<bus> arg);
  void remove_link_out(std::shared_ptr<bus> arg);
  void set(int arg);
  void set(std::bitset<max_bits> arg);

};
```

FIgura 21: Declaração do protótipo do registrador

"A", "B" e "Z" são ponteiros para os registradores da alu, sendo "Z" o registrador contendo o resultado.

Notar que:

1. As flags não foram implementadas

2. Os registradores da alu não precisam estar conectados entre si por um bus interno para simplificação, a alu faz as transferências por si só.

3. As operações de SHR e SHL operam acima do registrador "A" somente, o outro registrador teoricamente tendo a quantidade de bits a serem "shiftados"

4.2.4.3. Unidade de Controle:

```
class control_unit : public QObject{
Q_OBJECT
public:
  std::shared_ptr<regist> cu_reg;
  std::map <size_t,
        std::shared_ptr<bus>> buses;
  std::map <size_t,
        std::tuple<std::shared_ptr<regist>,
                   bool,
                   bool>> regists_in_out;
  std::map <size_t,
        std::shared_ptr<alu>> alus;
  std::map <size_t,
        std::vector<size_t>> opcodes;
  size_t map_reg_counter;
  size_t map_bus_counter;
  size_t map_alu_counter;
  size_t map_mar_counter;
  size_t map_mdr_counter;
  size_t operator_size;
  size_t operand_size;
  size_t operand_amnt;
  std::map<size_t, size_t> mdrs_id;
  std::map<size_t, size_t> mars_id;
  mov_cnt<QLabel> *display;
  control_unit(size_t cu_reg_s,
               size_t operator_s,
               size_t operand_s,
               size_t operand_amnt);

  control_unit(size_t cu_reg_s,
               size_t operator_s,
               size_t operand_s,
               size_t operand_amnt,
               QWidget *parent);
    control_unit(size_t arg);

  size_t make_bus(int bits);
  size_t make_regist(int bits);
  size_t make_internal_regist(int bits, QWidget *parent);
  size_t make_mdr(int bits, const std::shared_ptr<memory> &mem);
  size_t make_mar(const int bits, const std::shared_ptr<memory> &mem);
  size_t make_alu(std::shared_ptr<regist> A,
                  std::shared_ptr<regist> B,
                  std::shared_ptr<regist> Z);
```

FIgura 22: Declaração do protótipo da unidade de controle ½

```
std::shared_ptr<regist> get_register(size_t id);
std::shared_ptr<regist> get_mar(size_t id);
std::shared_ptr<regist> get_mdr(size_t id);
bool get_register_in(size_t id);
bool get_register_out(size_t id);
std::shared_ptr<bus> get_bus(size_t id);
std::shared_ptr<alu> get_alu(size_t id);
void set_in(size_t id);
void set_out(size_t id);

void assignment(size_t id_reg1, size_t id_reg2);
void add(size_t id);
void sub(size_t id);
void SHR(size_t id_alu, size_t amnt);
void SHL(size_t id_alu, size_t amnt);

void read(const std::shared_ptr<regist> &mar,
      const std::shared_ptr<regist> &mdr,
      const std::vector<std::shared_ptr<memory>> &memories);
void write(const std::shared_ptr<regist> &mar,
       const std::shared_ptr<regist> &mdr,
       const std::vector<std::shared_ptr<memory>> &memories);
  void execute(const std::vector<std::shared_ptr<memory>> &memories);
void opcode_execute(const std::vector<std::shared_ptr<memory>> &);
void interpret_minst(size_t,const std::vector<std::shared_ptr<memory>> &);
void reg_out();
void reg_in();
void sync_bus();
```

FIgura 23: Declaração do protótipo da unidade de controle 2/2

A unidade de controle agora é responsável pela criação e controle dos componentes sob seu escopo tomando o papel de manipulação anteriormente feito pela mainWindow do Qt. Fora isto ela agora controla o fluxo de dados por meio de sync_bus(), que computa como que os dados naquele clock cycle iriam se comportar com os devidos valores em barramentos e devidos registradores abertos/fechados. Notar que com isto também foi adicionado suporte para mais de uma memória externa.

4.2.4.4. Memória:

```
class memory : public QObject{
Q_OBJECT
public:
  size_t const len;
  std::vector<size_t> body;
  std::shared_ptr<bus> addr_bus;
  std::shared_ptr<bus> data_bus;
  memory (size_t mem_size, size_t len);
  memory (size_t mem_size, size_t mem_block_len, size_t abus_len, size_t dbus_len);
}
```

FIgura 24: Declaração do protótipo da memória

Notar que os registradores conectados com o elemento addr_bus representam os MAR e os conectados com data_bus MDR.

4.3.Projeto e especificação do sistema

4.3.1. Funcionalidade

O sistema desenvolvido oferece ao usuário um pacote que permite a construção de arquiteturas de computadores utilizando os blocos fundamentais da unidade de controle, registradores genéricos, memória, unidade de lógica aritmética (ALU) e barramentos. Junto da nova estrutura ele oferece a capacidade de construir seus próprios opcodes por meio de um conjunto de microinstruções e carregamento de memória arbitrária para simulação de programas e algoritmos completos, memória que pode ser editada diretamente na GUI caso desejado, sem necessitar de operações de leitura. Também é disponível um repl para os opcodes definidos pelo usuário e microinstruções para exploração imediata sem necessitar modificar a memória.

O funcionamento básico da arquitetura é em volta do ciclo de fetch-decode-execute, há botões dedicados para a simulação deste ciclo na GUI e consequentemente a capacidade de rodar programas em usos sucessivos. Isto é necessário pois o REPL não afeta os registradores de instrução.

Todo o fluxo de dados entre os registradores é feito indiretamente populando barramentos com valores de registradores "abertos" e populando registradores "fechados" naquele ciclo com os valores no barramento.

O programa hoje tem como padrão tamanho de word 16 bits, registradores com este word size e opcodes de 16 bits, 8 bits de operando e 8 bits de operador, permitindo sem nenhuma modificação até 255 opcodes criados pelo usuário, assim como uma memória com barramentos de endereçamento de 16 bits.

A arquitetura padrão que será carregada ao iniciar o programa tem a seguinte configuração:

- Unidade de controle
- Registrador interno da unidade de controle
- ALU
- 3 registradores internos da ALU, dois para argumentos e um para resultado
- Memória

- MDR(memory adress register) e MAR(memory data register) atrelados à memória
- 2 barramentos exclusivos para conectar MAR e MDR à memória
- 2 registradores avulsos
- 1 barramento que conecta todos os registradores entre si

O arquivo contendo um conjunto de opcodes oferecido com o programa é feito para esta arquitetura.

A GUI hoje tem limitações em particular no quesito de construção das arquiteturas que o código por trás não tem, em particular as seguintes operações precisam necessariamente ser feitas no código em si:

- Construção de arquiteturas com mais de uma unidade de controle:

Isto ocorre pois não há mecanismo ainda pela GUI para conectar uma unidade de controle nova em um barramento gerado pela interface gráfica. Ademais a GUI não irá carregar os opcodes específicos desta unidade de controle.

- Conexão de registradores em mais de um barramento:

Similar ao problema acima, hoje não é possível por meio da interface gráfica conectar registradores em mais de um barramento, pois por meio dela a conexão é feita no momento da construção do elemento.

- Modificação do tamanho da word da unidade de controle e tamanho dos registradores:

Todos os registradores criados pela interface gráfica irão seguir o tamanho da word. Para modificar o tamanho da word, é necessário modificar uma flag no código e recompilar. Notar que ao criar registradores pelo código eles podem ter tamanho diferentes que a word.

- Modificação do tamanho de operadores e operandos de opcodes:

Também é necessário ser feito pelo código, em particular na construção das unidades de controle que desejam ter esta mudança. Um breve exemplo e explicação de como fazer a construção pelo código está disponível no apêndice B.

4.3.2.Estrutura do sistema

A estrutura do sistema é composta de 3 camadas: mainWindow, overseer, unidade de controle.

1. mainWindow

   É a camada mais alta do projeto, ela contém os atributos específicos de Qt da interface gráfica como sinais e slots e uma única instância do overseer.

2. overseer

   É a camada que encapsula o funcionamento da simulação e seus componentes, sendo responsável por dois conjuntos, um de unidades de controle e um de memórias. Ele é necessário pois memórias não são exclusivas de uma unidade de controle e potencialmente múltiplas unidades de controle precisam ser manejadas.

3. control_unit

   É a camada que encapsula o funcionamento interno de uma unidade de controle e todos os elementos que estão em seu escopo, sendo responsável pela construção e manipulação dos elementos sob seu escopo. Esta unidade de controle irá ter sob o seu escopo suas instâncias de registradores, barramentos, alus e opcodes. Ela também contém referências para as memórias, mesmo que as memórias não estejam sob seu escopo, permitindo que suas instruções a modifiquem ao decorrer de um programa.

4.3.2.1.Elementos encapsulados por control_unit

Dentre da unidade de controle os elementos que compõem a arquitetura - com exceção da memória- são localizados.

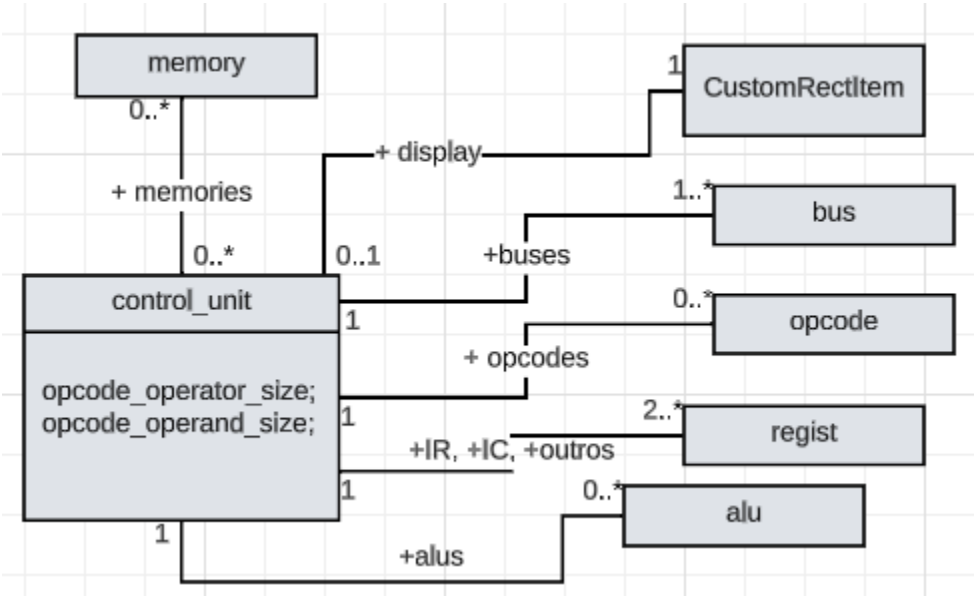


FIgura 25: Diagrama de classe simplificado de control_unit

4.3.2.1.1. regist

O elemento referente ao registrador. É uma classe que contém as suas informações de tamanho em bits, valor atual, id, barramentos que está conectado e uma referência para seu elemento gráfico.

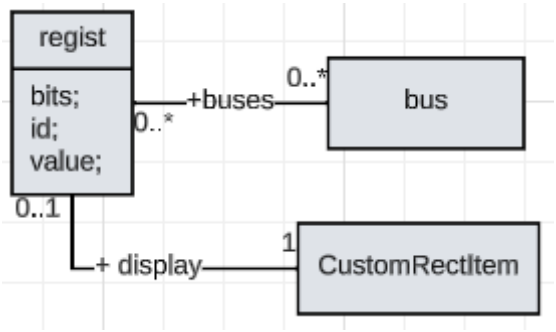


FIgura 26: Diagrama de classe simplificado de regist

4.3.2.1.2.bus

O elemento referente ao barramento. É uma classe que contém as suas informações de tamanho em bits e valor atual e uma referência para seu elemento gráfico.

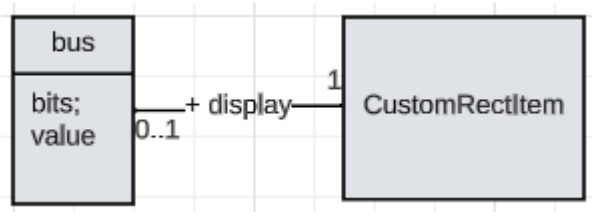


FIgura 27: Diagrama de classe simplificado de bus

4.3.2.1.3.alu

O elemento referente à unidade lógica aritmética. É uma classe que contém 3 registradores, referentes à ALUa, ALUb e ALUz, 5 flags referentes à overflow, negative, equal, zero e carry e uma referência para seu elemento gráfico.

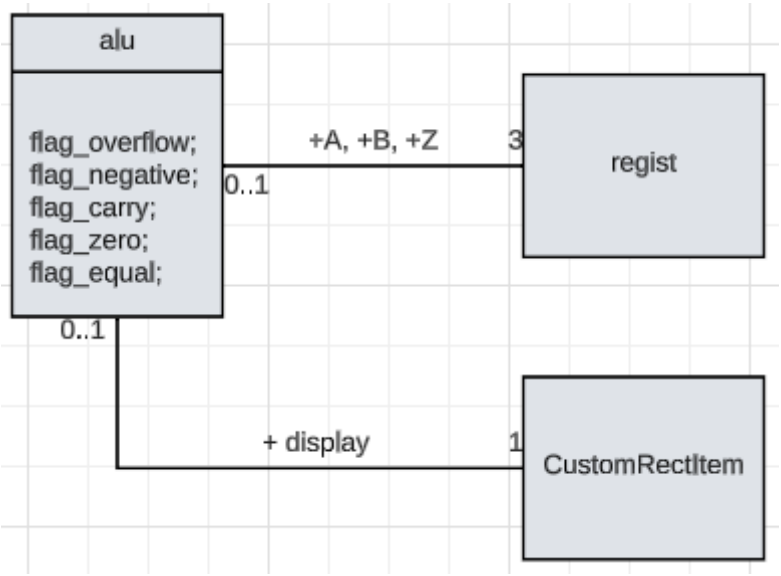


FIgura 28: Diagrama de classe simplificade de alu

4.3.2.1.4.opcode

O elemento referente a um opcode. É uma classe que contém o seu identificador literal, o seu operando e um conjunto de microcódigos. A categorização do operador é feita pela ordem de carregamento no arquivo de

definição. O opcode com operador zero sempre deve ser a operação de FETCH.

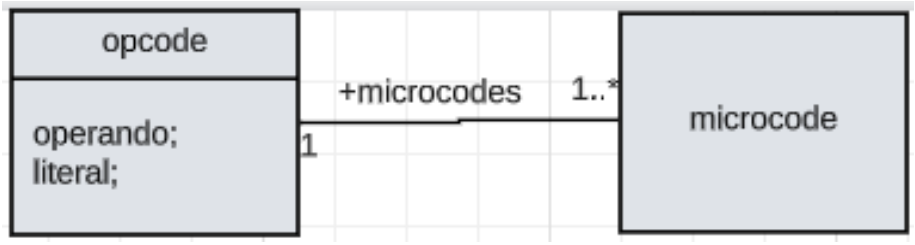


FIgura 29: Diagrama de classe simplificade de opcode

4.3.3. Construção de microcódigos

As microinstruções oferecidas são: "Assignment", “Assignment from IR operand “, “Increment program counter “, “Halt “, "Add", "Sub", "Read from Memory Adress", "Write to Memory Adress", "Shift Left", "Shift Right", "Halt", “Flag check”.

As microinstruções são representadas desta forma:

Descrição	Operador File Input	Operador Código
Assignment	<-	0x1
Assignment from IR operand	<-‘0	0xC
Increment program counter	INC	0xD
Halt	HALT	0x0
Shift Left	SHL	0x6
Shift Right	SHR	0x7
Add	ADD	0x2
Subtract	SUB	0x5
Read from Memory Adress	READ	0x3
Write to Memory Adress	WRITE	0x4
Flag check*	If PSRx==1	0x8-0xB

*para este, o valor de x pode ser qualquer uma das 4 flags da ALU(equal, zero, carry,negative), caso a flag em questão não esteja zerada o opcode termina a execução.

4.3.4.Definição de opcodes

Opcodes são carregados no programa por meio de um arquivo de texto construído pelo usuário para se adequar à sua linguagem. Um conjunto de opcodes para a arquitetura padrão já foi definido A estrutura deste arquivo é igual à estrutura usada no Jarasper 1.0.

Os opcodes são compostos por um parêntese "(" seguido de qualquer conteúdo até o final da linha, que normalmente inclui o valor do código e um mnemônico. Após a primeira linha, as microinstruções são adicionadas, uma por linha, até alcançar uma linha que contenha apenas um parêntese ")". É possível adicionar comentários às linhas utilizando o ponto e vírgula ";".

Como esta versão não tem uma arquitetura única, é necessário se referir aos registradores por seu número de id, decidido pela ordem que o registrador foi construído na arquitetura.

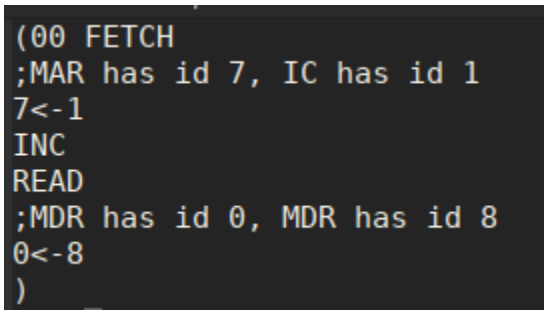
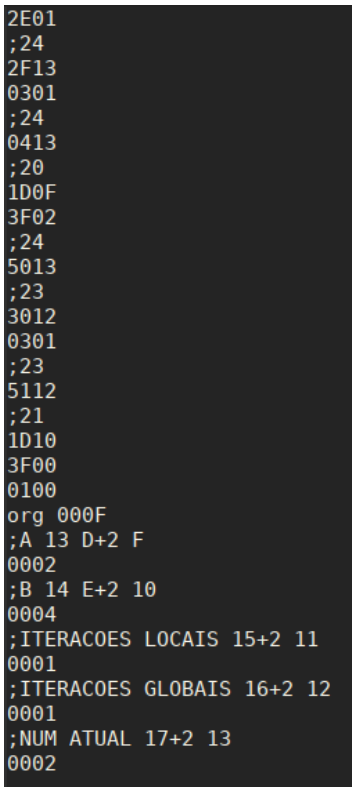


FIgura 30: Exemplo de FETCH para arquitetura padrão

O operador do opcode é definido pela ordem no arquivo, e o opcode de número 0 deve sempre ser a implementação de FETCH para a sua arquitetura.

4.3.5.Definição da memória

A memória é carregada por meio de um arquivo de texto, ou inicializada zerada. Este arquivo de texto é organizado com uma linha por instrução em hexadecimal, cada linha representando sequencialmente uma posição em memória. Comentários podem ser feitos entre as linhas e devem ser prefixados por “;” . É possível inserir padding entre linhas utilizando a seguinte sintaxe: org [número da linha para pular]. Por exemplo escrevendo em uma linha org 000F irá fazer com que a próxima linha seja o valor no endereço 000F, independentemente da posição antes do comando.



```
2E01
;24
2F13
0301
;24
0413
;20
1D0F
3F02
;24
5013
;23
3012
0301
;23
5112
;21
1D10
3F00
0100
org 000F
;A 13 D+2 F
0002
;B 14 E+2 10
0004
;ITERACOES LOCAIS 15+2 11
0001
;ITERACOES GLOBAIS 16+2 12
0001
;NUM ATUAL 17+2 13
0002
```

FIgura 31: Exemplo de memória que computa 2^4 na arquitetura padrão

4.3.6.Apresentação da interface

Notar que todo registrador tem o seguinte formato em seu corpo: texto1:texto2. texto1 representa o número de identificação do registrador no escopo de sua unidade de controle, texto2 representa o seu valor.

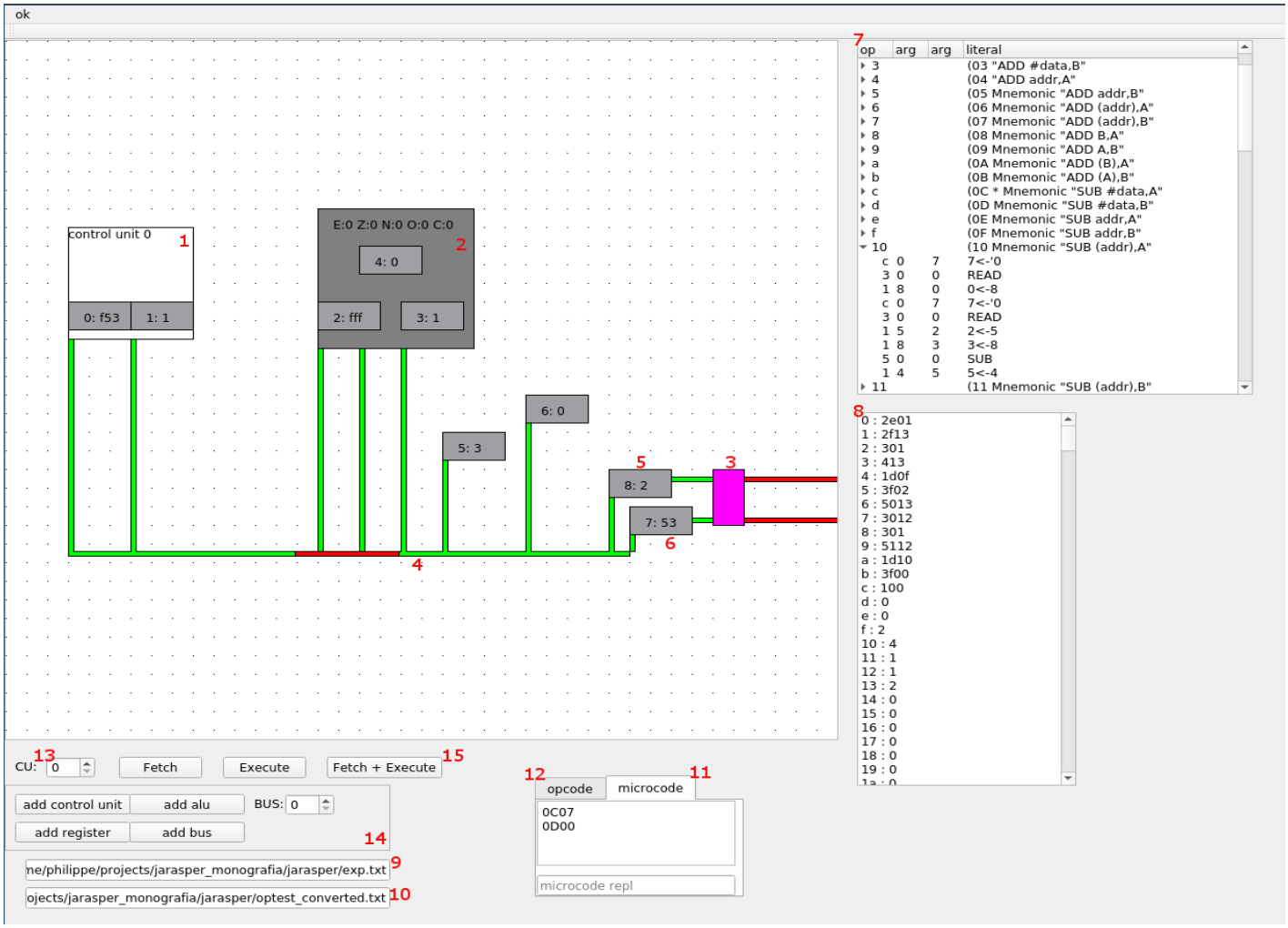


FIgura 32: Interface gráfica com marcadores para explicação

Explicação dos marcadores numerados:

1. Uma unidade de controle, ela contém 2 registradores que sempre vão ocupar os números 0 e 1, eles são o registrador de instruções e o contador de instruções respectivamente.
2. Uma ALU, ela contém 3 registradores que sempre vão ser construídos e dados números de identificação nesta ordem: ALUa, ALUb e ALUz. No exemplo(fig.32) estes se referem aos números 2,3 e 4 respectivamente.

3. Um bloco de memória, ele é o único elemento da arquitetura que não é restrito ao escopo da unidade de controle, podendo ser em teoria acessado em diversas unidades de controle diferentes. No entanto essa capacidade não é suportada pela GUI no momento.
4. Um barramento, no caso desta arquitetura o único barramento não especializado. O trecho vermelho é o ponto que pode ser usado para mover o caminho do barramento sem mover nenhum dos registradores conectados. Notar que há dois barramentos especializados saindo da memória (marcador 3), estes barramentos são independentes ao barramento apontado no marcador 4.
5. MDR, um registrador especializado para receber e inserir dados em uma memória a qual ele está conectado por um barramento especial.
6. MAR, um registrador especializado para apontar o endereço de memória a pegar e inserir dados, ele está conectado por um barramento especial à sua memória.
7. Listagem de opcodes, uma lista de todos os opcodes carregados e seus mnemônicos. Ao expandir um item é possível ver a sequência de microinstruções que compõe tal opcode, tanto em uma linguagem humana quanto no código real da microinstrução separado em operador/argumento/argumento
8. Listagem de memória, o conteúdo da memória. Uma linha é equivalente a um endereço de memória e pode ser editado com duplo clique. Notar que no código cada conjunto de memória é exclusivo à uma memória, no entanto múltiplas memórias não são implementadas ainda na GUI logo é inacessível.
9. Endereço de arquivo que contém a memória
10. Endereço de arquivo que contém as definições de opcode
11. REPL de microinstruções, por aqui é possível escrever microinstruções para serem rodada na unidade de controle especificada (marcador 13). Utilização do REPL não irá modificar o conteúdo do registrador de instrução (IR) nem do contador de instrução (IC).
12. REPL de opcodes, por aqui é possível escrever opcodes para serem rodados na unidade de controle especificada (marcador 13). Utilização do REPL não irá modificar o conteúdo do contador de instrução (IC).
13. O número da unidade de controle sobre a qual as operações devem ocorrer

14. Controle de adição de elementos. Adição de registrador e ALU dependem do valor do bus e do valor da CU (marcador 13) para decidir sob qual unidade de controle ele está e qual barramento eles se conectam. Adição de barramento depende somente da unidade de controle para decidir o escopo de seu barramento. Notar que a GUI ainda não oferece suporte completo para múltiplas unidades de controle.
15. Este botão irá iniciar um ciclo de fetch+execute na unidade de controle selecionada, efetivamente ele roda o opcode 0x0000(FETCH sempre é o primeiro opcode) seguido do opcode que foi carregado da memória ao IR no fim do fetch. Diferentemente do REPL, esta rotina afeta os registradores da unidade de controle do programa, e pode ser usado sucessivamente para rodar um programa em memória.

5. Implementação

5.1.Planejamento e execução de testes funcionais

5.1.1.Planejamento

O planejamento descrito na proposta de projeto final estava com as datas assumindo que Projeto Final 2 seria feito no semestre de 2022.2, no entanto ele foi feito no semestre de 2023.1. Os meses foram ajustados para refletir esta mudança.

Fevereiro-Março: Finalização da mudança do fluxo para ser baseado em clocks e barramentos.

Março-Abril: Implementação das aprimorações da GUI sugeridas e seus testes.

Abril-Maio: Implementação das aprimorações da sintaxe e ferramentas linguísticas, e seus testes.

Maio-Junho: Implementação do suporte multiplataforma

Junho: Finalização da documentação do projeto final.

Algumas coisas não se adequaram ao tempo do projeto, a finalização do fluxo previsto para fevereiro-março foi mais complexa que o esperado, atrasando um pouco as aprimorações da GUI.

O mês de abril e parte de maio não foram produtivos por problemas fora do meu controle. Isto impactou as aprimorações de sintaxe, não conseguindo implementar a tempo de forma satisfatória a construção de opcodes como conjunto de outros opcodes pelo REPL. Também causou um impacto no suporte para openBSD, não consegui resolver os seus problemas a tempo de forma satisfatória.

5.1.2.Execução de testes funcionais

Os testes foram todos realizados utilizando a interface gráfica e a configuração padrão da arquitetura. A sua execução e método em maior detalhe está descrita no apêndice A e suas imagens demonstrativas no apêndice B.

5.1.2.1.Microcodes

Testes sobre o funcionamento do conjunto de microcódigos. Microcódigos de controle de fluxo serão testados com os opcodes:

- 0x01: assign (<-)
- 0x02: ALU sum(ADD)
- 0x03: memory read (READ)
- 0x04: memory write (WRITE)
- 0x05: ALU sub(SUB)
- 0x06: SHIFT LEFT (SHL)
- 0x07: SHIFT RIGHT (SHR)
- 0x0C: Literal Assignment from IR (<-'0)
- 0x0D: Increment Instruction Counter (INC)

5.1.2.2.Opcodes

Testes sobre a construção de opcodes e utilização de microcódigos de controle de fluxo por meio da flag equal:

- Opcode com controle de fluxo(if PSRe==0, Branch if not equal)
- Opcode com controle de fluxo(if PSRe==1, Branch if equal)

5.1.2.3.Carregamento de opcodes pelo arquivo

Testes sobre o carregamento do arquivo de opcodes criado pelo usuário:

- Carregamento de arquivo inexistente ou com caminho inválido
- Carregamento de arquivo com comentários

5.1.2.4. Carregamento de memória por arquivo

Testes sobre carregamento do arquivo de memória criado pelo usuário:

- Carregamento de arquivo inexistente ou com caminho inválido
- Carregamento de arquivo com “org”, comentários e código
- Carregamento de arquivo endereçando posições maiores que a memória disponível

5.2.Comentários sobre a implementação

A implementação não ofereceu muitos problemas específicos, grande parte foram resolvidos durante o planejamento ou foram dados menor prioridade por questão de tempo.

Dois problemas, no entanto, valem notar:

- Sincronização de clocks entre unidades de controle diferentes:

Este foi o primeiro problema que precisou de uma mudança estrutural no código, a classe overseer foi criada inicialmente com o único propósito de ser o ponto de entrada para chamar um cycle() em todas as unidades de controle.

- Responsabilidade por simular movimento por barramentos:

Com a mudança de simples atribuição estática entre registradores pré-definidos para movimentação de dados em barramentos fica a dúvida de quem é responsável por essa simulação e como posso minimizar a quantidade de informação redundante ao falar em relação entre componentes. Inicialmente foi feita uma classe separada que varria cada unidade de controle, fazia a relação entre os barramentos e registradores e efetuava a modificação; esta classe, no entanto ficava muito complexa e não se aproveitava das relações já existentes entre os componentes. A forma atual delega essa responsabilidade à cada unidade de controle, que varre sua lista de registradores para verificar quais estão conectados com qual barramento e quais estão abertos, colocando o valor dos registradores abertos nos barramentos apropriados e em seguida procura os registradores abertos para receber os valores do barramento. Desta forma o único componente que tem a informação de qual registrador está conectado com qual barramento é o registrador, o barramento é somente uma "caixa" com um valor.

6. Considerações Finais

6.1. Contribuições deste trabalho

Este projeto oferece para educadores uma possível alternativa mais abrangente e flexível ao JASPer para o ensino de arquiteturas computacionais ao nível de registradores, assim como uma ferramenta de exploração e prototipação para entusiastas sem precisar adentrar nas complexidades de uma linguagem de descrição de hardware.

6.2. O que faria diferente em retrospecto

De uma parte de organização o projeto sofreu muito com falta de foco, tanto no período do projeto final quanto no período anterior no desenvolvimento da versão 1.0. Enquanto o cronograma foi seguido na medida do possível muito tempo foi gasto explorando alternativas como a integração do LLVM e IR, que por mais que fossem muito promissoras necessitavam de um conhecimento maior que o tempo de projeto oferecia. Este tempo gasto poderia ter sido usado para melhorar a integração entre backend e interface gráfica para uma melhor experiência padrão, em particular criando uma forma de salvar as suas arquiteturas criadas que por sua vez abriria muitas possibilidades para expansão.

De uma parte técnica a troca do QtWidgets(QT PROJECT, 2020)para o QML(QT PROJECT, 2020) iria facilitar o desenvolvimento da interface gráfica, e a utilização de bison(THE FREE SOFTWARE FOUNDATION, 2014)e flex(PAXSON, 2017) para a análise dos opcodes e microcodes ao invés de um parser criado por mim do zero.

6.3.Trabalho Futuro

O projeto tem diversos caminhos possíveis para avançar, tanto em relação a adição de funcionalidade quanto à expansão de escopo. Um caso de expansão de escopo que pretendo implementar é suporte para simulação de máquinas de ilimitados registradores(BRAINERD, 1974) e máquinas de único registrador(BRAINERD, 1974).

Outros caminhos para avançar com o projeto em certa ordem de complexidade:

- Completar a integração de GUI com funcionalidade de código.
- Implementar uma animação ao rodar um programa similar ao jarasper 1.0.
- Permitir que o usuário salve suas arquiteturas sem precisar mexer no código.
- Adicionar uma forma do usuário renomear os seus registradores e ser capaz de usar estes nomes ao invés dos números de id em seu microcódigo.
- Adicionar um gerador de opcode para gerar opcodes de operações básicas (assign, add, sub,shr,shl) para cada elemento novo que for adicionado à arquitetura.
- Implementar a integração do backend do LLVM, construindo automaticamente um backend para cada arquitetura que o usuário construa e criando uma paridade entre as microinstruções e o IR do llvm.
- Uma vez com o backend do LLVM implementado implementar ferramentas de introspecção de passos de otimização intermediários pelo backend.

7. Referências

BRAINERD. The Single-Register Machine. In: BEAINERD, W. S. **Theory of Computation**. 1st. ed. Illinois: Wiley, 1974. Cap. 4, p. 99-102.

BRAINERD. The Unlimited Register Machine. In: BRAINERD, W. S. **Theory of Computation**. 1st. ed. Illinois: Wiley, 1974. Cap. 4, p. 91-95.

BURRELL, M. **Fundamentals of Computer Architecture**. [S.l.]: Palgrave, 2004.

C++ FOUNDATION. C++14 reference. **CPP Reference**, 2020. Disponivel em: <https://en.cppreference.com/w/cpp/14>. Acesso em: 23 jun 2021.

ERHARDT, C. Design and Implementation of a TriCore Backend for the LLVM Compiler Framework, p. 19, 2009.

IEEE COMPUTER SOCIETY. **IEEE Standard 1076: VHDL Language Reference Manual**. [S.l.]. 2008.

JARA, P. **Jarasper: Um Simulador de Arquiteturas Computacionais Flexível e Configurável ao Nível de Registradores**. [S.l.]. 2018.

KNUTH, D. E. **MMIXware:** A RISC Computer for the Third Millennium. Heidelberg: Springer-Verlag, 1999.

KOCHMANSKI, D. **Embeddable Common-Lisp**, 2022. Disponivel em: <https://ecl.common-lisp.dev/>. Acesso em: 30 jan 2023.

LATTNER, C. The LLVM Target-Independent Code Generator. **LLVM**, 2022. Disponivel em: <https://llvm.org/docs/CodeGenerator.html>. Acesso em: 12 jun 2022.

LEFÈBVRE, C. **Linux Mint**, 2023. Disponivel em: <https://linuxmint.com/>. Acesso em: 23 jan 2023.

MICROSOFT. Windows 7. **Microsoft Products**, 2020. Disponivel em: <https://learn.microsoft.com/en-us/lifecycle/products/windows-7>. Acesso em: 3 jun. 2023.

MICROSOFT. Windows 10. **Microsoft Software**, 2022. Disponivel em: <https://www.microsoft.com/en-us/software-download/windows10>. Acesso em: 3 jun 2023.

NEWMAN, W. **Steel Bank Common Lisp**, 2023. Disponivel em: <http://www.sbcl.org/>. Acesso em: 7 fev 2023.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design:** The Hardware/Software Interface. 5th. ed. [S.l.]: Morgan Kaufmann, 2013.

PAXSON, V. flex. **Github**, 2017. Disponivel em: <https://github.com/westes/flex>. Acesso em: 1 jun. 2023.

QT PROJECT. QML Reference. **Qt 5.15 Documentation**, 2020. Disponivel em: <https://doc.qt.io/qt-5/qtqml-index.html>. Acesso em: 23 mar 2023.

QT PROJECT. QtWidgets Class Reference. **Qt 5.15 Documentation**, 2020. Disponivel em: <https://doc.qt.io/qt-5/qtwidgets-module.html>. Acesso em: 03 dec 2022.

QT PROJECT. **Qt git index**, 2023. Disponivel em: <https://code.qt.io/cgit/qt/qtbase.git/>. Acesso em: 24 jan 2023.

THE FREE SOFTWARE FOUNDATION. GNU Bison. **GNU**, 2014. Disponivel em: <https://www.gnu.org/software/bison/>. Acesso em: 20 fev 2023.

VERILOG. Verilog. Disponivel em: <http://www.verilog.com/>. Acesso em: 27 abr. 2017.

VINET, J.; GRIFFIN, A.; POLYÁK, L. **Arch Linux**, 2023. Disponivel em: <https://archlinux.org/>. Acesso em: 16 jan 2023.

VMWARE. INC. VMware Workstation Player. **VMware**, 2023. Disponivel em: <https://www.vmware.com/content/vmware/vmware-published-sites/us/products/workstation-player.html.html>. Acesso em: 15 fev 2023.

8. Apêndice

8.1.Apêndice A- Casos de Teste Funcionais

8.1.1.Microcodes

Todos os microcodes serão testados utilizando o microcode repl, que aceita seus códigos diretamente e utilizando a configuração padrão de arquitetura com:

- Unidade de controle
- Registrador interno de instruções (IR) da unidade de controle (0)
- Registrador interno de contagem (IC) da unidade de controle (1)
- ALU
- 3 registradores internos da ALU, dois para argumentos (2, 3) e um para resultado (4)
- Memória
- MDR (8) e MAR (7) atrelados à memória
- 2 barramentos exclusivos para conectar MAR e MDR à memória
- 2 registradores avulsos (5,6)
- 1 barramento que conecta todos os registradores entre si

Notar que todas as imagens demonstrativas dos testes estão disponíveis no apêndice A.

0x01: assign (<-)

Atribuição de um registrador para outro

- Valores Iniciais: Registrador 5 com valor 3, Registrador 6 com valor 0
- Comando: 0x0156 (equivalente a 6<-5)
- Resultado esperado: Registrador 5 com valor 3, Registrador 6 com valor 3
- Resultado obtido: Registrador 5 com valor 3, Registrador 6 com valor 3

0x02: ALU sum(ADD)

Soma dos registradores internos da ALU número zero

- Valores Iniciais: Registrador 2(ALUa) com valor 0xfff, Registrador 3(ALUb) com valor 1, registrador 4(ALUz) com valor 0
- Comando: 0x0200 (equivalente a SUM)
- Resultado esperado: Registrador 2 com valor 0xfff, Registrador 3 com valor 2, Registrador 4 com valor 0x1001
- Resultado obtido: Registrador 2 com valor 0xfff, Registrador 3 com valor 2, Registrador 4 com valor 0x1001

0x03: memory read (READ)

Leitura de memória utilizando MDR e MAR de número zero

- Valores Iniciais: Registrador 7(MAR) com valor 2, Registrador 8(MDR) com valor 2, endereço de memória "2" com valor 0x0301
- Comando: 0x0300 (equivalente a READ)
- Resultado esperado: Registrador 2 com valor 2, Registrador 8 com valor 0x0301
- Resultado obtido: Registrador 2 com valor 2, Registrador 8 com valor 0x0301

0x04: memory write (WRITE)

Escrita na memória utilizando MDR e MAR de número zero

- Valores Iniciais: Registrador 7(MAR) com valor 2, Registrador 8(MDR) com valor 0x301, endereço de memória "2" com valor 0
- Comando: 0x0400 (equivalente a WRITE)
- Resultado esperado: endereço de memória "2" com valor 0x0301
- Resultado obtido: endereço de memória "2" com valor 0x0301

0x05: ALU sub(SUB)

Subtração dos registradores internos da ALU número zero

- Valores Iniciais: Registrador 2(ALUa) com valor 0xfff, Registrador 3(ALUb) com valor 2, registrador 4(ALUz) com valor 0
- Comando: 0x0500 (equivalente a SUM)

- Resultado esperado: Registrador 2 com valor 0xfff, Registrador 3 com valor 2, Registrador 4 com valor 0xffd
- Resultado obtido: Registrador 2 com valor 0xfff, Registrador 3 com valor 2, Registrador 4 com valor 0xffd

0x06: SHIFT LEFT (SHL)

Shift left do valor em ALUa pela quantidade de bits em ALUb, na ALU número zero.

- Valores Iniciais: Registrador 2(ALUa) com valor 0xfff, Registrador 3(ALUb) com valor 2, registrador 4(ALUz) com valor 0
- Comando: 0x0600 (equivalente a SHL)
- Resultado esperado: Registrador 2 com valor 0xfff, Registrador 3 com valor 2, Registrador 4 com valor 0x3ffc
- Resultado obtido: Registrador 2 com valor 0xfff, Registrador 3 com valor 2, Registrador 4 com valor 0x3ffc

0x07: SHIFT RIGHT (SHR)

Shift right do valor em ALUa pela quantidade de bits em ALUb, na ALU número zero.

- Valores Iniciais: Registrador 2(ALUa) com valor 0x0fff, Registrador 3(ALUb) com valor 1, registrador 4(ALUz) com valor 0
- Comando: 0x0700 (equivalente a SHL)
- Resultado esperado: Registrador 2 com valor 0x0fff, Registrador 3 com valor 1, Registrador 4 com valor 0x07ff
- Resultado obtido: Registrador 2 com valor 0x0fff, Registrador 3 com valor 1, Registrador 4 com valor 0x07ff

0x08-0x0B: Branch microcodes, precisam ser testados usando opcodes

0x0C: Literal Assignment from IR (<-'0)

Assign operando dentro do registrador valor 0(instruction register)

- Valores Iniciais: Registrador 0(IR) com valor 0x0f53, Registrador 7(MAR) com valor 2
- Comando: 0x0C07 (equivalente a 7<-'0)
- Resultado esperado: Registrador 0(IR) com valor 0x0f53, Registrador 7(MAR) com valor 0x0053
- Resultado obtido: Registrador 0(IR) com valor 0x0f53, Registrador 7(MAR) com valor 0x0053

0x0D: Increment Instruction Counter (INC)

incrementa o registrador 1(IC)

- Valores Iniciais: Registrador 1(IC) com valor 0x0000
- Comando: 0x0D00 (equivalente a INC)
- Resultado esperado: Registrador 1(IC) com valor 0x0001
- Resultado obtido: Registrador 1(IC) com valor 0x0001

8.1.2. Opcodes

Todos os opcodes serão testados utilizando o opcode repl, que aceita seus códigos diretamente e utilizando a configuração padrão de arquitetura. O único caso casos de opcodes que não podem ser testados somente testando microcodes são os que utilizam controle de fluxo. Será testado abaixo o controle de fluxo por meio da flag “equal”.

1. Opcode com controle de fluxo(E=0):

   Irá incrementar o contador de instrução caso ALU com id:0 tenha a flag E ativada.

   E = 0

- Valores Iniciais: Registrador 1(IC) com valor 0x0000, registrador 0(IR) com valor 0x0000, ALU 0 com flag E = 0
- Comando no repl: 0x0100
- Resultado esperado: Registrador 1(IC) com valor 0x0000, registrador 0(IR) com valor 0x0100

- Resultado obtido: Registrador 1(IC) com valor 0x0000, registrador 0(IR) com valor 0x0100

E = 1

- Valores Iniciais: Registrador 1(IC) com valor 0x0000, registrador 0(IR) com valor 0x0000, ALU 0 com flag E = 1
- Comando no repl: 0x0100
- Resultado esperado: Registrador 1(IC) com valor 0x0001, registrador 0(IR) com valor 0x0100
- Resultado obtido: Registrador 1(IC) com valor 0x0001, registrador 0(IR) com valor 0x0100

8.1.3. Carregamento de opcodes pelo arquivo

Os testes de carregamento de opcode serão feitos utilizando a GUI para carregar arquivos de opcodes no programa.

1. Arquivo inexistente:

   Caminho vazio para carregar a memória.

   - Estado inicial da memória: Nenhum opcode carregado
   - Estado esperado: Nenhum opcode carregado
   - Estado obtido: Nenhum opcode carregado

2. Opcodes com comentários:

   Arquivo com comentários.

```
(00 FETCH
;MAR has id 7, IC has id 1
7<-1
INC
READ
;MDR has id 0, MDR has id 8
0<-8
)
```

FIgura 33: opcode para teste de carregamento

- Estado inicial da memória: Nenhum opcode carregado
- Estado esperado: FETCH definido com microcodes 7<1, INC, READ e 0<-8.
- Estado obtido: FETCH definido com microcodes 7<1, INC, READ e 0<-8.

8.1.4. Carregamento de memória por arquivo

Os testes de carregamento de memória serão feitos utilizando a GUI para carregar arquivos de memória no programa. A memória da arquitetura sendo usada para o teste tem 5000 posições e uma word de 16 bits.

1. Arquivo inexistente:

   Caminho vazio para carregar a memória.

   - Estado inicial da memória: Todos os valores 0x0
   - Estado esperado: Todos os valores 0x0
   - Estado obtido: Todos os valores 0x0
2. Arquivo com “org”, comentário e código:

   Carregamento do seguinte arquivo para testar org, comentários e valores comuns:

```
2E01
;24
2F13
org 000F
FF00
```

FIgura 34: Memória para teste de carregamento

   - Estado inicial da memória: Todos os valores 0x0
   - Estado esperado: Posição 0x0 de memória com valor 0x2E01, posição 0x1 com valor 0x2F13, posições 0x2-0xE com valor 0x0, posição 0xF com valor 0xFF00, todas as posições subsequentes vazias.

- Estado obtido: Posição 0x0 de memória com valor 0x2E01, posição 0x1 com valor 0x2F13, posições 0x2-0xE com valor 0x0, posição 0xF com valor 0xFF00, todas as posições subsequentes vazias.

3. Arquivo endereçando posições maiores que a memória disponível

Utilizando uma memória com 5000 posições de memória, carregamento do seguinte arquivo:

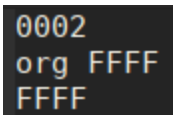


FIgura 35: Memória para teste de carregamento excedendo limite

- Estado inicial da memória: Todos os valores 0x0
- Estado esperado: Posição 0x0 de memória com valor 0x2, todas as posições subsequentes vazias.
- Estado obtido: Posição 0x0 de memória com valor 0x2, todas as posições subsequentes vazias.

8.2.Apêndice B- Imagens dos Casos de Teste Funcionais

8.2.1.Microcodes:

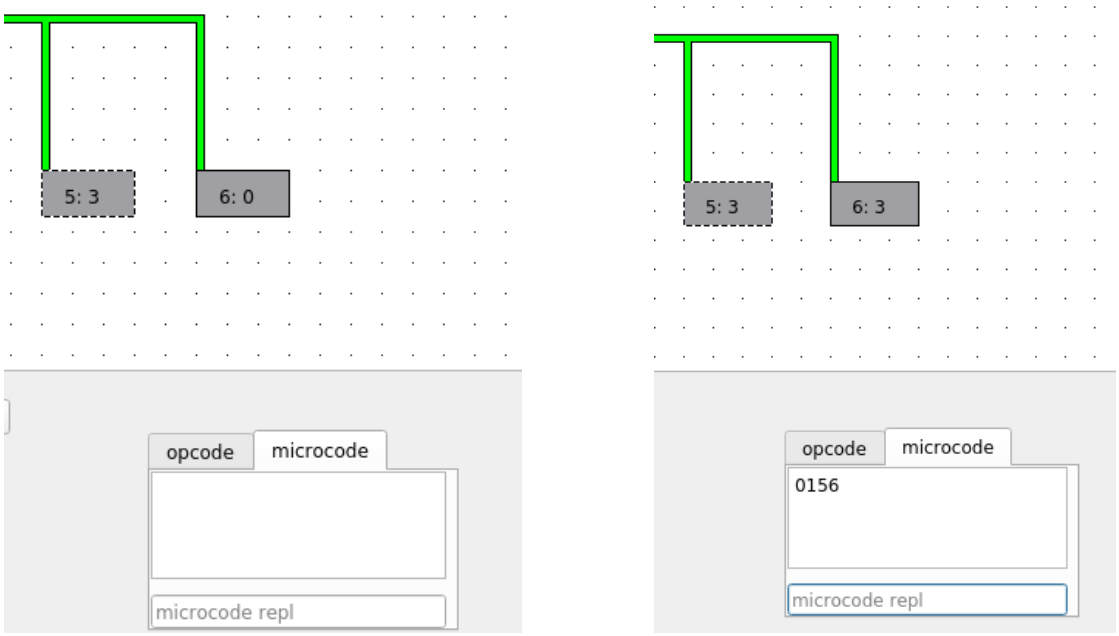


FIgura 36: Antes e depois do teste de microinstrução “assign”(0x1)

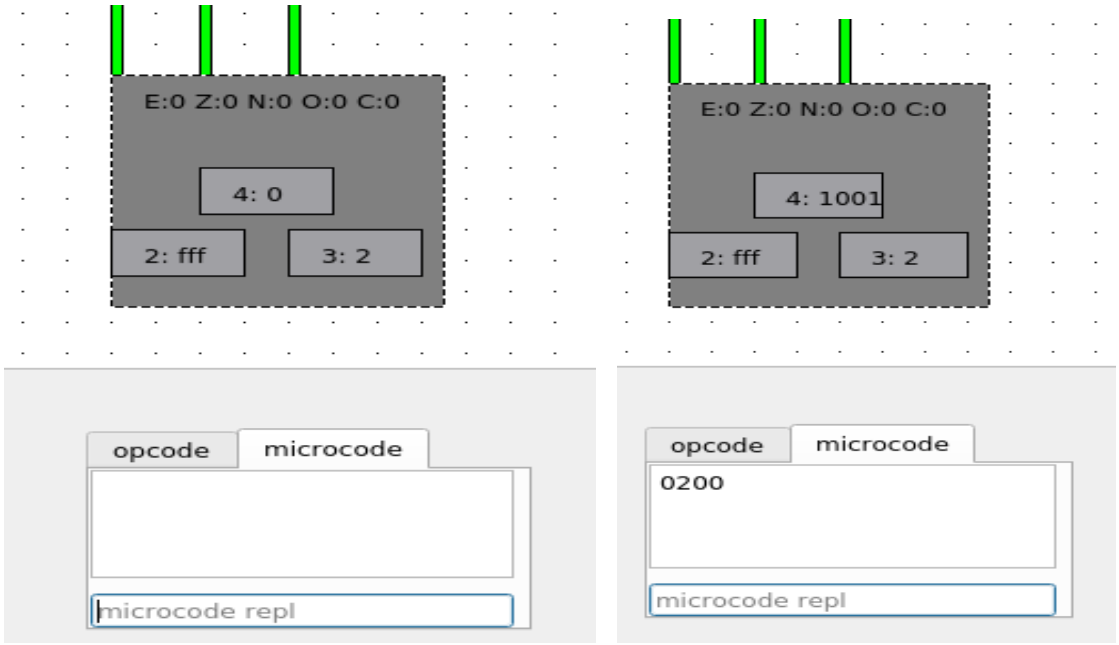


FIgura 37: Antes e depois do teste de microinstrução ADD(0x2)

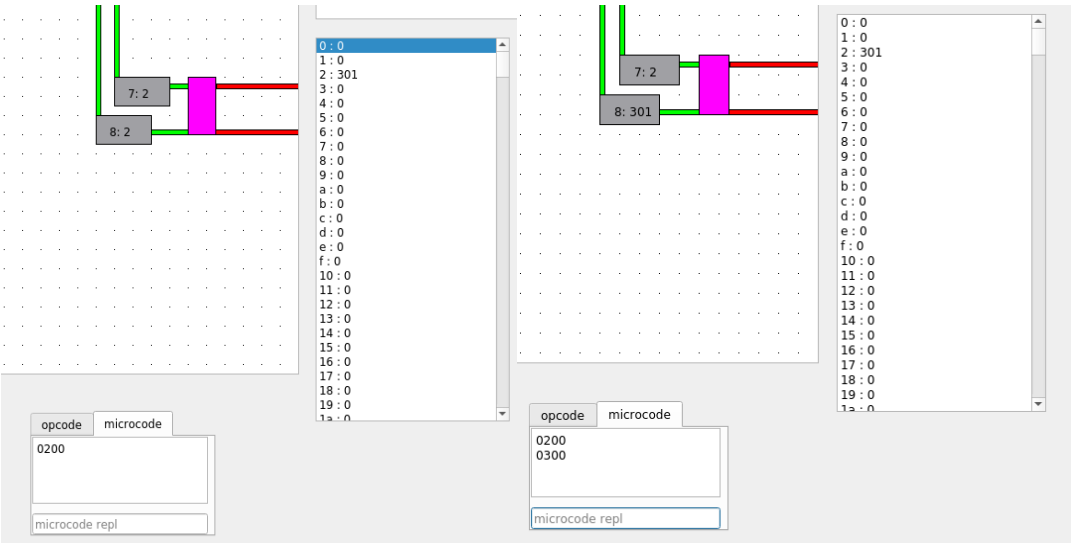


FIgura 38: Antes e depois do teste de READ(0x3)

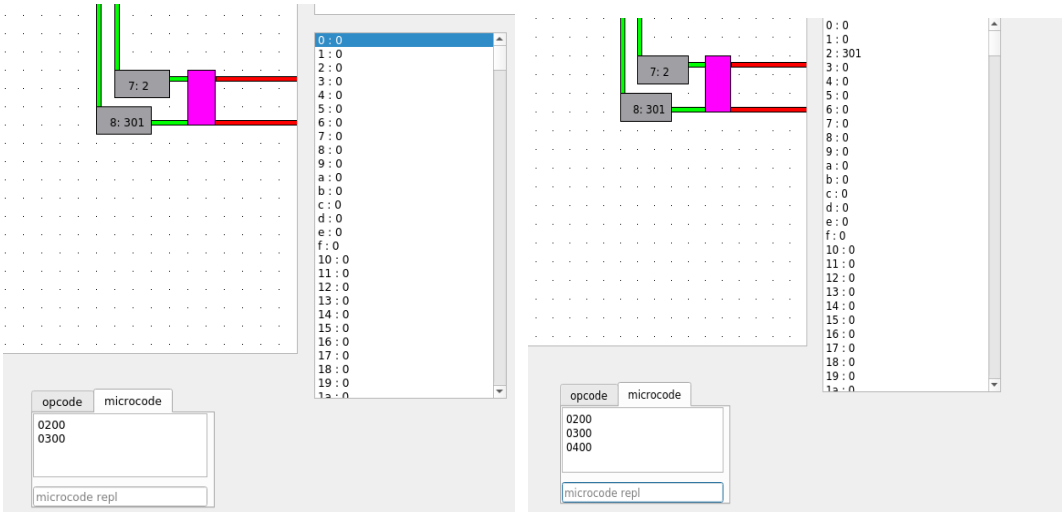


FIgura 39: Antes e depois do teste de WRITE(0x4)

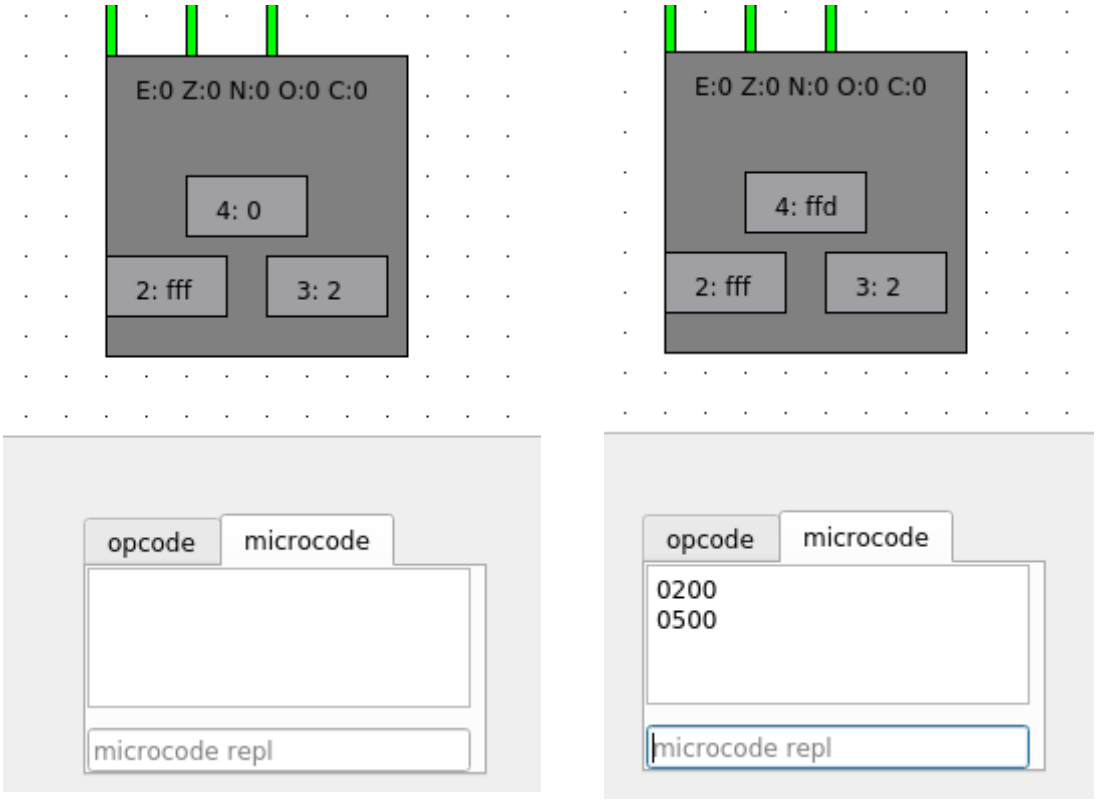


FIgura 40: Antes e depois do teste de SUB(0x3)

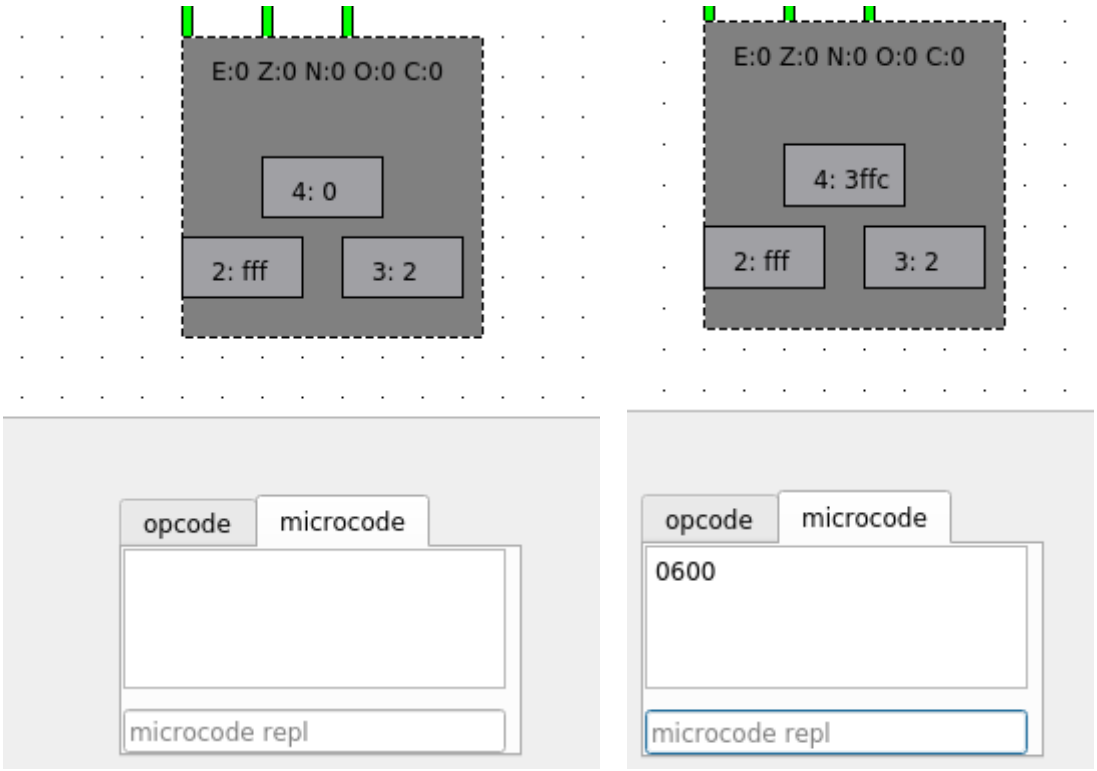


FIgura 41: Antes e depois do teste de SHL(0x6)

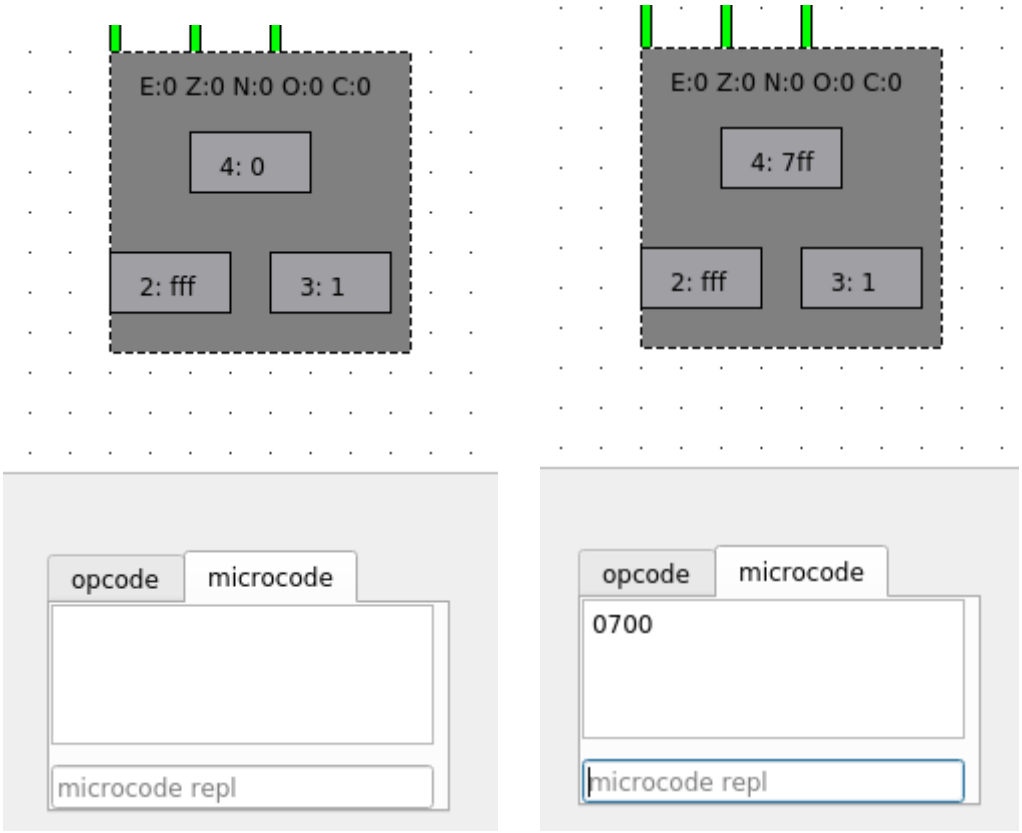


FIgura 42: Antes e depois do teste de SHR(0x7)

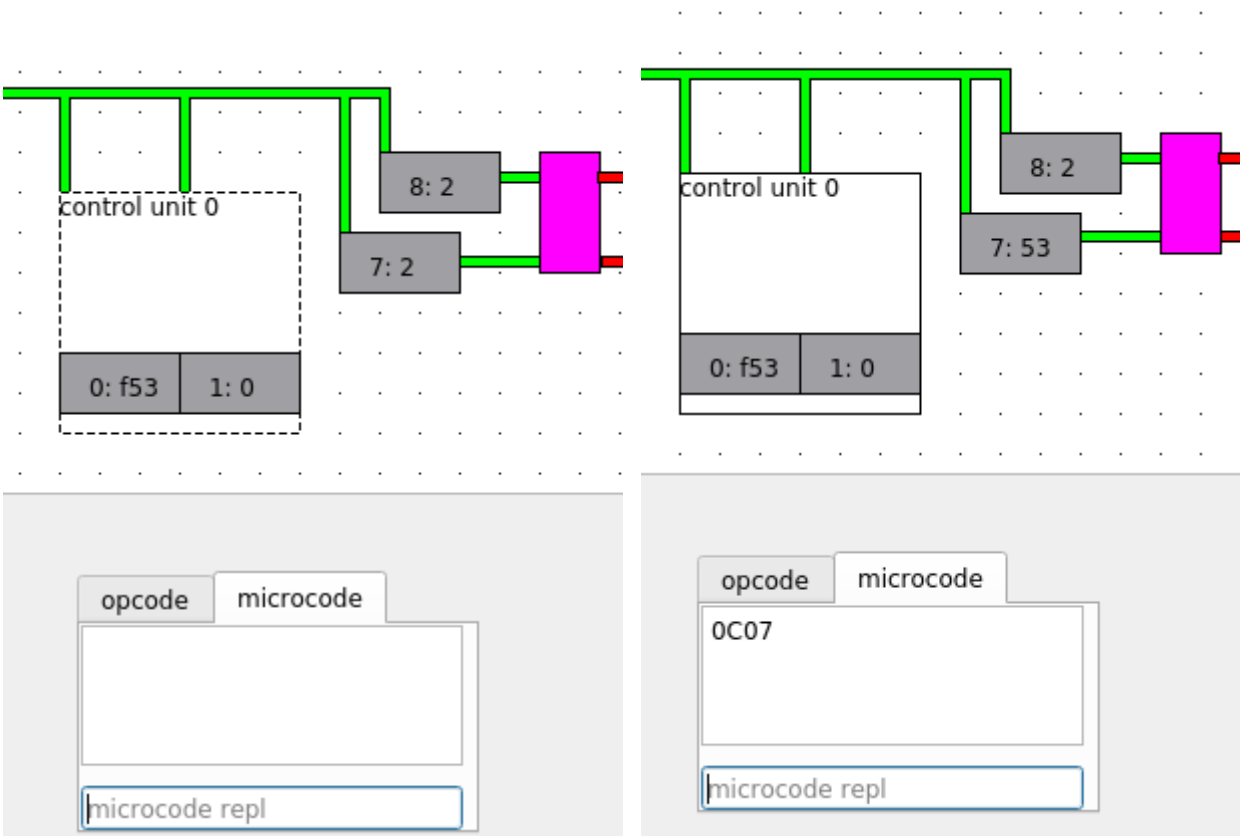


FIgura 43: Antes e depois do teste de "assignment literal"(0xC)

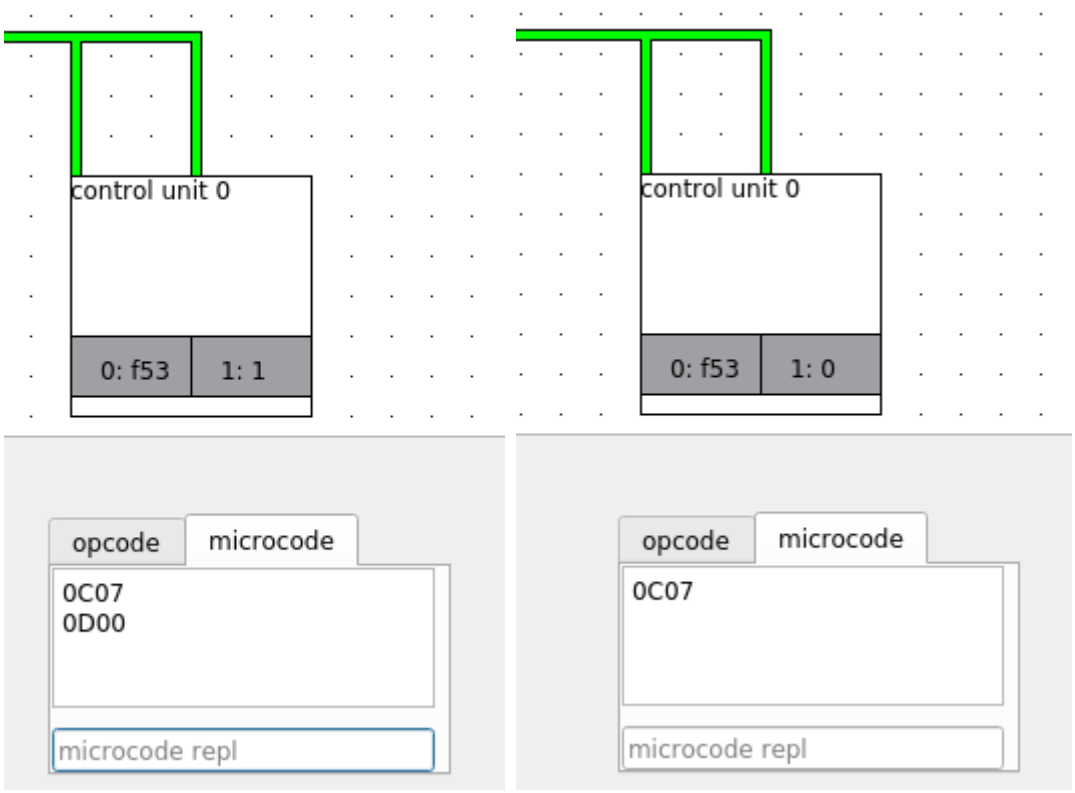


FIgura 44: Antes e depois do teste de INC(0xD)

8.2.2.Opcode

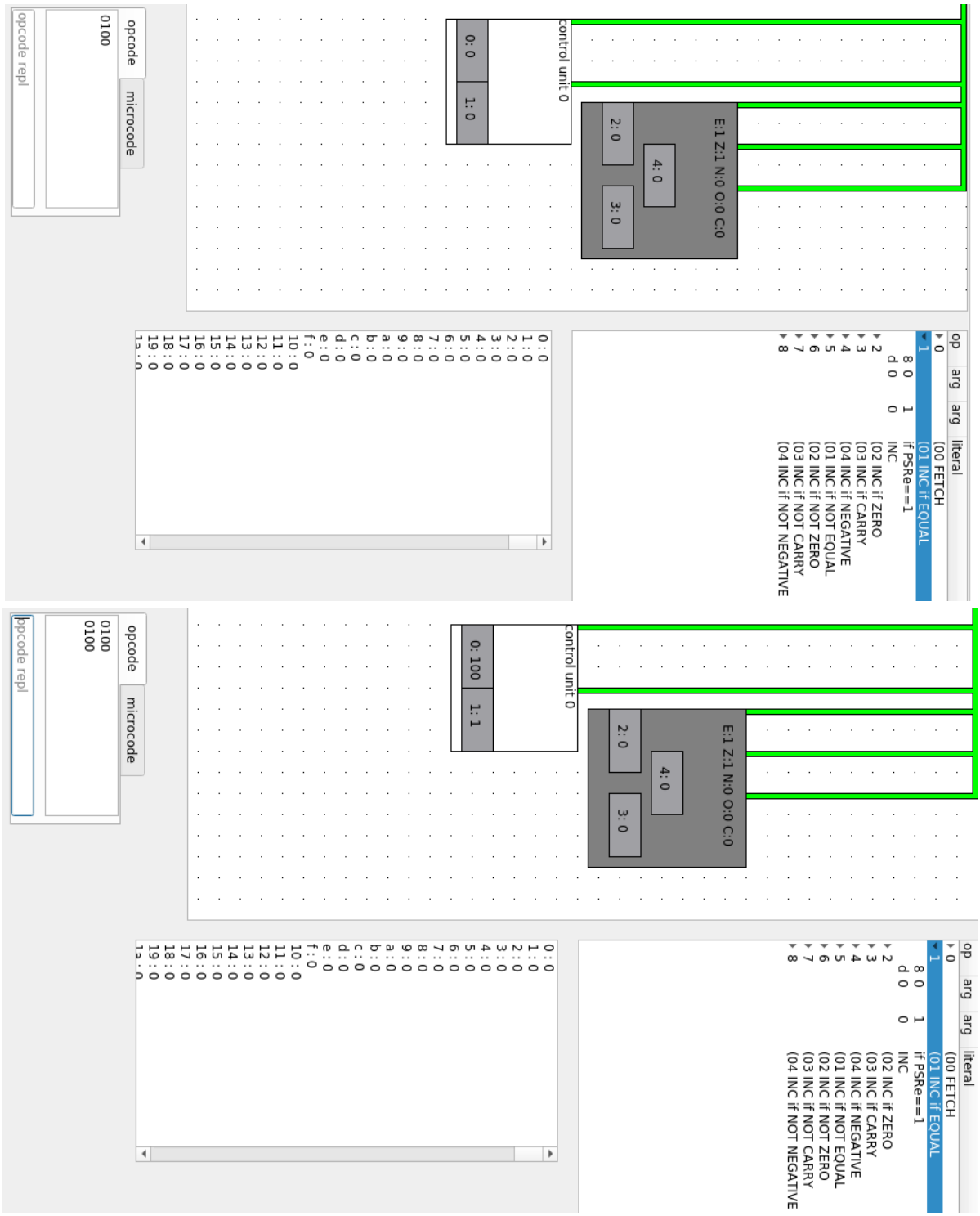


FIgura 45: Antes e depois do teste de opcode com controle de fluxo(E=1)

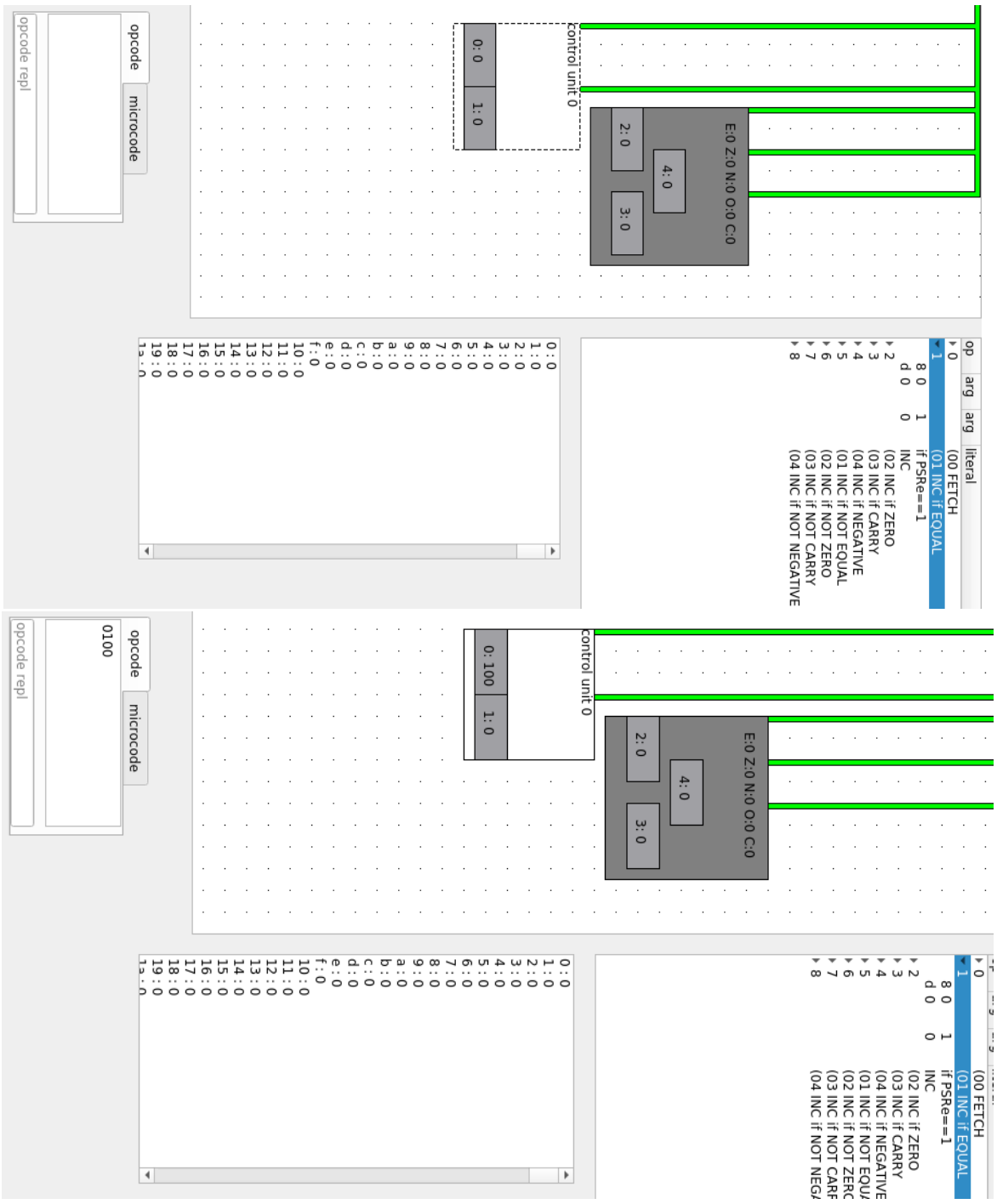


FIgura 46: Antes e depois do teste de opcode com controle de fluxo(E=0)

8.2.3.Carregamento de opcodes

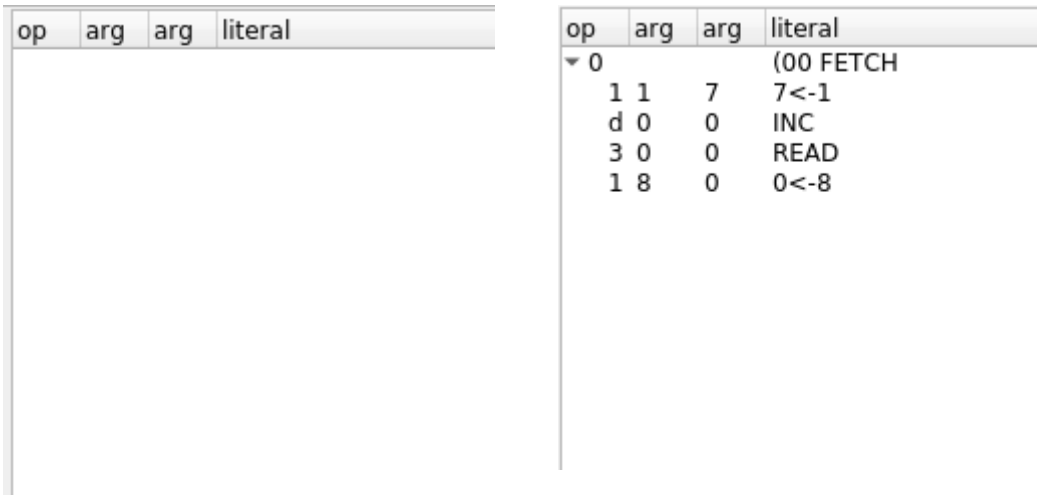


FIgura 47: Antes e depois do teste de carregamento de opcodes

8.2.4.Carregamento da memória

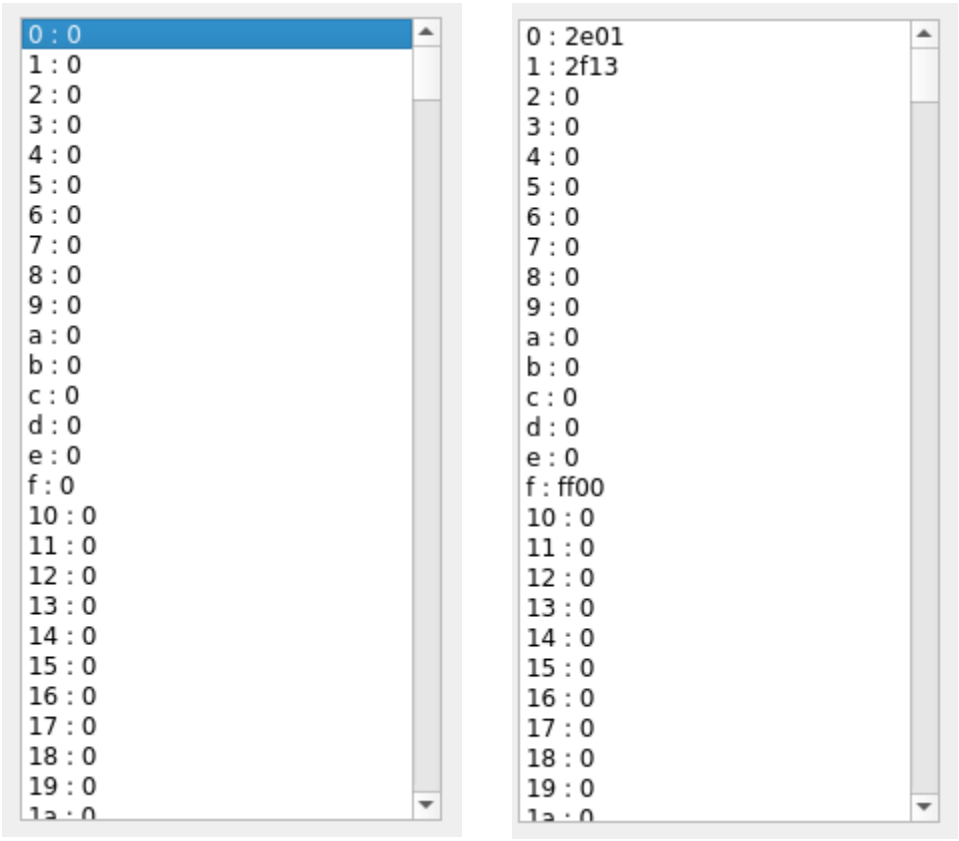


FIgura 48: Antes e depois do teste de carregamento de memória

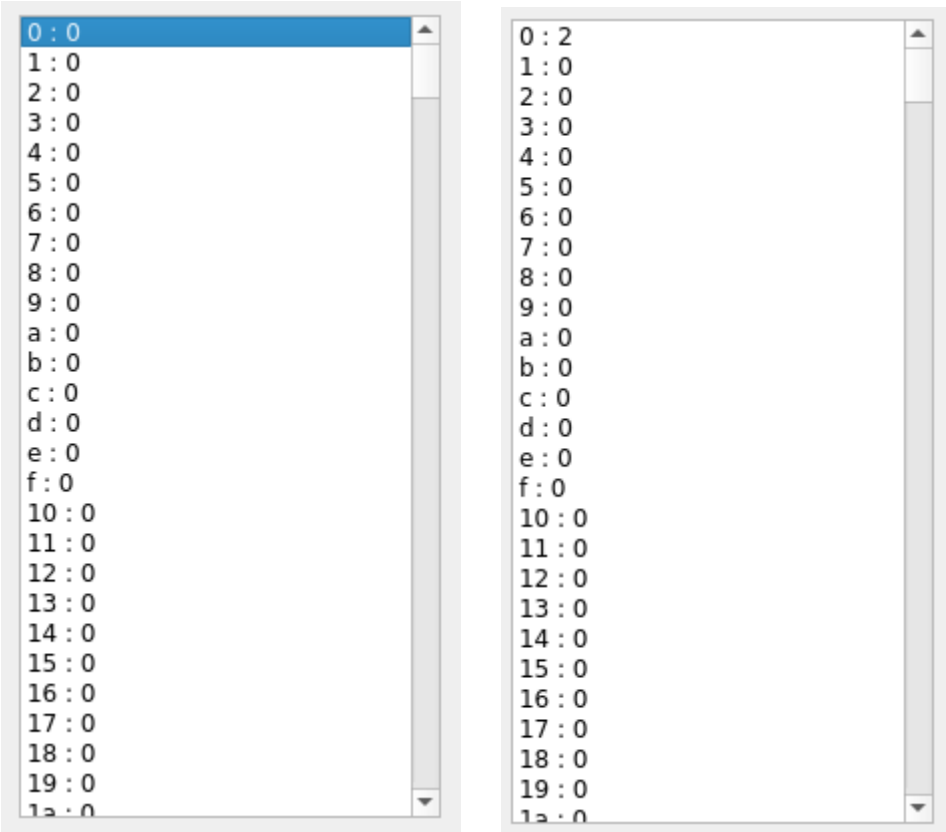


FIgura 49: Antes e depois do teste de carregamento de memória maior que limite

8.3.Apêndice C- Construção de arquitetura por código

Para construir sua arquitetura pelo código diretamente é necessário modificar a primeira função em mwin.cpp (void mwin::load_default_architecture()):

```
void mwin::load_default_architecture(){
  /*creates a control unit with REGISTER_SIZE IR and IC size,
    opcode with 8 bits for operator and 8 for operand*/
    ov.make_cu(REGISTER_SIZE,8,8,1)->display->info.setText("control unit 0");
  /*creates a memory with MEMSIZE adresses of REGISTER_SIZE word size,
    REGISTER_SIZE width mdr bus and REGISTER_SIZE width mar bus*/
    ov.memories.push_back(make_shared<memory>(MEMSIZE, REGISTER_SIZE, REGISTER_SIZE, REGISTER_SIZE));
    auto cu = ov.control_units[0];
    auto mem = ov.memories[0];
    //creates an ALU with REGISTER_SIZE ALUa, ALUb and ALUz
    cu->make_alu(REGISTER_SIZE);
    cu->make_regist(REGISTER_SIZE);
    cu->make_regist(REGISTER_SIZE);
    //creates a bus with REGISTER_SIZE width and connects it to the control units registers
    auto bu = cu->get_bus(cu->make_bus(REGISTER_SIZE));
    for(auto& item:cu->regists_in_out){
      (get<0>(item.second))->link_in(bu);
      (get<0>(item.second))->link_out(bu);
    }
    /*creates the memory adress register and the memory data register
      and links them with the previously defined bus*/
    auto mar_id = cu->make_mar(REGISTER_SIZE, mem);
    auto mdr_id = cu->make_mdr(REGISTER_SIZE, mem);
    auto mdr = cu->get_mdr(mdr_id);
    mdr->link_in(bu);
    mdr->link_out(bu);
    auto mar = cu->get_mar(mar_id);
    mar->link_in(bu);
    mar->link_out(bu);
    //setting the initial value of the register 5 to 0x3
    cu->get_register(5)->set(0x003);

}
```

FIgura 50: Exemplo de implementaçao por meio de código

É possível modificar os #defines de REGISTER_SIZE e MEMSIZE em global_macros.h, no entanto não é necessário caso queira colocar diretamente os valores na criação dos elementos.