# Apresentação

Boa Noite Membros da Banca: Professor Marcelo, Professor Wagner, ao meu orientador Professor Danilo, e ao demais presentes.

Meu nome é Judah Holanda e venho lhes apresentar meu TCC: Proposta de Arquitetura de Assembler Online Único Para Diversos Processadores.

# Contexto

Atualmente, é possível perceber que *softwares,* programas que rodavam em ambiente *off-line,* estão se tornando serviços *web* [1,2]. Possibilitando seu uso em qualquer lugar que possua conexão com a *internet*, e facilitando trabalhos colaborativos [3, 4, 5].

Para a criação de um projeto de *software*/*firmware* comercial, normalmente, é necessária a instalação de diversos programas, dentre eles IDE’s, programas de documentação, gerenciamento, controle de versão, entre outros. Cada um deles é direcionado para um sistema operacional específico e, algumas vezes, é necessário utilizar dois sistemas operacionais distintos, pois um ou mais programas podem não estar disponíveis para a plataforma utilizada no projeto. Outra importante questão é o tempo gasto para instalação desses softwares necessários para a elaboração e execução do projeto.

Depois desta etapa ainda, existe a pesquisa de *frameworks* e códigos para serem reutilizados neste projeto, pois não existe um local em que todos os códigos do mundo estejam armazenados e de fácil acesso. Finalmente, há a integração destes códigos com o projeto; isso implica em fazer tanto a integração com a IDE quanto com a plataforma utilizada. A integração pode resultar em: modificação, encapsulamento e criação de trechos de códigos para integração. Isto implica em ler um código que, geralmente, não está seguindo o mesmo padrão de codificação e documentação.

Uma IDE *online,* além de tornar desnecessária a instalação, permite o uso em qualquer lugar em que a *internet* esteja disponível, independente de plataforma ou sistema operacional. E esse é um dos motivos pelos quais os serviços *Web* têm alcançado tamanho sucesso e aumentado tanto ao longo dos anos. Programas que antes precisavam ser instalados na máquina, agora, se tornam *online*, sendo necessário apenas um *browser*. Tornando-se desnecessário se preocupar até mesmo com a atualização de *software*.

A integração de todas essas ferramentas e códigos em um mesmo serviço torna a criação de projetos muito mais rápida e prática, além de aumentar significativamente o reuso de código. Como a ferramenta está hospedada na nuvem, a colaboração entre programadores fica facilitada. Pois tanto a edição e como compartilhamento do código podem ser feitos a partir de qualquer dispositivo que possua um browser, entrada para texto e conexão com a internet.

Contudo, mesmo fazendo tudo isso, cada família de processador tem sua própria linguagem de máquina e, consequentemente, um *Assembly* próprio, o que dificulta muito a migração de uma plataforma para outra [6].

Este estudo propõe uma nova abordagem, criando um *Assembly* unificado que utilize apenas um conjunto restrito de instruções, facilitando assim a migração de plataformas.

Outra questão relevante é que ao migrar softwares para diferentes plataformas é necessário aprender algo novo, e normalmente, o mais difícil de assimilar é a terminologia. Então, mesmo que a ideia fundamental seja simples, quando é descrita com palavras não familiares pode dificultar bastante o entendimento [7]. Como é o caso de uma linguagem nova, e no caso do Assembly este é um problema bem comum, principalmente com seus mnemônicos.

Como o *Assembly* utiliza mnemônicos, isso demanda o estudo e aprendizado de cada instrução, tornando a linguagem bem mais complexa que as de alto nível, como C/C++ e Java, cujas funções são escritas de forma extensa, diferentemente de uma abreviação/sigla, como no caso dos mnemônicos. Por isso, usualmente, se aprende primeiro linguagens de alto nível, pois elas estão mais próximas da linguagem humana [6,7]. Então se propõe que as instruções sejam escritas de forma extensa, facilitando a leitura do código e o aprendizado da linguagem.

Como em vários países existe o hábito de se programar na própria língua oficial, cria-se outro problema, visto que se impede que pessoas que não falem tal língua consigam utilizar o seu código escrito. Então, para contornar este problema, o sistema terá como regra a codificação em inglês, devido a esta ser uma das línguas mais faladas no mundo, uma das mais fáceis de se aprender e por ter se tornado o padrão na maioria dos projetos.

Ainda assim é necessário delimitar um padrão para nomenclatura de variáveis e métodos, pois mesmo tendo uma linguagem padrão isso não garante que os códigos serão de fácil leitura. Então, um padrão de nomenclatura também deve ser adotado, bem como de chaveamento e padrões de projetos.

Este projeto tem como proposta abordar estas questões levantadas através de uma arquitetura que permita a edição e armazenamento, em um servidor na nuvem, de um Assembly comum a diversas plataformas. Para a validação foi feita uma implementação simplificada, onde foca apenas em duas plataformas: o microcontrolador da Intel 8051 e o microprocessador z80 da Zilog. O Assembly genérico proposto, que recebeu o nome Easembly, é um Assembly de nível um pouco mais alto que consiste no conjunto de instruções comum a ambas as plataformas, mas que mesmo que reduzido, ainda seja possível realizar todas as operações básicas.

# 4 OBJETIVOS

## 4.1 Geral

Potencializar a reutilização de códigos com a criação de um *Assembly* unificado.

## 4.2 Específicos

1. Criar conjunto de instruções;
2. Determinar padrão de nomenclatura a ser seguido pelos seus usuários;
3. Utilizar armazenamento no próprio sistema; e
4. Criar sistema básico (*Easembly*).

# 5 JUSTIFICATIVA

Mesmo já existindo linguagens multiplataforma em alto nível, não é possível utilizá-las em processadores com pouco poder de processamento, como vários microcontroladores de 8 *bits*.

Mnemônicos dificultam a leitura e o entendimento dos códigos.

Conjuntos de instruções diferentes para cada família de processadores dificultam a migração de um processador para outro.

A utilização de uma plataforma *web* permite ao programador codificar, em qualquer lugar que possua *internet,* uma infinidade de dispositivos, sem a necessidade de instalar diversas ferramentas.

# 6 DISCUSSÃO TEÓRICA

A ideia deste conjunto reduzido de instruções é inspirada na arquitetura RISC dos processadores modernos. Para tornar este conjunto de instruções multiarquitetura é necessário reduzir o número de instruções ao máximo para que se tenha certeza de que ele vai estar presente em qualquer processador.

Mas para que o usuário possa codificar é necessário um ambiente de desenvolvimento, que no caso desta arquitetura é online. Um ambiente que integra todas as ferramentas necessárias para que o programador desenvolva, monte e depure erros de código, é chamado de IDE, ou Integrated Development Environment, embora nem sempre venha a ter todas estas funcionalidades. Um exemplo deste tipo de ferramenta pode ser visto na Figura 1. Na implementação realizada apenas uma parte deste tipo de programa foi codificada, deixando de lado o depurador de erros.

Na literatura cientifica é possível encontrar diversos trabalhos publicados, nos quais a ideia de uma IDE *Online* pode ser percebida, conforme mostrado na Figura 1. Embora não tenham sido encontrados trabalhos que exploram a ideia de *Assembler* unificado, foi possível encontrar trabalhos que utilizam uma linguagem independente de arquitetura, como Java [8,9,10,11,12,13]. Também foram encontrados *Assemblers* para processadores reconfiguráveis [14], embora não seja o foco deste estudo, algumas instruções são mantidas ao reconfigurar processadores e isto foi de grande utilidade ao projetar o conjunto de instruções. E finalmente um gerador *Assembler* [15], que desempenha um papel fundamental na ideia de utilizar um tradutor de Assemblies.

## 6.1 Editor

Existem diversos tipos de editores, dentre eles se destacam dois: os que editam texto e código-fonte.

Editores de texto são aqueles que permitem a edição de arquivos de texto, em pelo menos um formato. Entretanto, o editor de código-fonte é um editor de texto com funcionalidades mais específicas. Este tipo de programa é desenhado especificamente para editar código-fonte de *Software/Firmware*. Facilitando a programação. Editores como este podem ser uma aplicação, por si só. Como também podem fazer parte de outra, no caso, como de uma IDE.

## 6.2 Interpretador

O interpretador é um programa de computador que executa instruções escritas em uma determinada linguagem de programação.

## 6.3 Tradutor

Um Tradutor de linguagem tem como objetivo, assim como um tradutor de línguas, traduzir uma linguagem *A* para uma linguagem *B*.

## 6.4 Compilador

Um Compilador é aquele que traduz um programa de uma linguagem de programação em *Assembly*, que é um conjunto de instruções que serão mapeadas para seu respectivo conjunto de comandos da linguagem de máquina, ou seja, é uma forma simbólica de como a máquina entende.

## 6.5 Assembler (Montador)

O Montador é o programa responsável por transcrever o programa da linguagem Assembly para linguagem de máquina.

## 6.6 Linker (Link-Editor)

A função do *linker*, ou ligador, é coletar procedimentos já traduzidos em linguagem de máquina, separadamente, e ligá-los uns aos outros para que eles possam executar como uma unidade chamada programa binário executável.

# 7 ARQUITETURAS RELACIONADAS

## 7.1 *Java*

*Java* é uma linguagem de programação de computadores de propósito geral, a qual é concorrente, orientada a objeto[25] e desenhada para ter o mínimo possível de dependências de implementação. Tem como objetivo ser uma linguagem que, uma vez codificada, possa ser executada em qualquer plataforma[26]. No caso, uma vez que o programa em *Java* foi compilado, ele pode ser executado em todas as plataformas que usam *Java*, sem a necessidade de recompilar ou de reescrever o código.[27]

As aplicações *Java* são, geralmente, compiladas em um *bytecode,* que pode rodar em qualquer *Java Virtual* *Machine* (JVM), independente da arquitetura do computador, um exemplo de seu funcionamento pode ser visto na Figura 10. Apesar desse nível a mais em sua execução, *Java* continua sendo umas das linguagens de programação mais populares em uso.

### 7.1.1 Java Virtual Machine (JVM)

A Java Virtual Machine (JVM) é uma máquina virtual que permite que um computador execute um programa *Java*.

### 7.1.2 Java Bytecode

O *Java Bytecode* é a linguagem de máquina da *Java Virtual Machine*.

(~1%) estão destinadas a nunca serem implementadas.[32,33,34,35]

## 7.2 .NET

O .NET Framework é composto pelo Common Language Runtime e pela biblioteca de classes do .NET Framework.

### 7.2.1 Common Language Runtime (CLR)

O .NET Framework fornece um ambiente de tempo de execução chamado Common Language Runtime, que executa o código e fornece serviços que tornam o processo de desenvolvimento mais fácil.

### 7.2.2 Common Language Infrastructure (CLI)

O Common Language Infrastructure (CLI) é uma especificação aberta desenvolvida pela Microsoft e padronizado pela ISO e ECMA que descreve o código executável e um ambiente de tempo de execução que permitem que várias linguagens de alto nível serem usadas em diferentes plataformas de computador sem ser reescrito para arquiteturas específicas.

### 7.2.3 Common Intermediate Language (CIL)

Common Intermediate Language (CIL, pronunciado ou "sil" ou "kil") (anteriormente chamado Microsoft Intermediate Language ou MSIL) é a linguagem de programação do nível mais baixo ainda legível definido pela especificação Common Language Infrastructure (CLI) e é usada pelo .NET Framework e Mono.

# 8 ARQUITETURA

O sistema consiste em um tradutor de *Assembly genérico*, por meio do qual serão feitas as traduções para o *Assembly* específico de cada arquitetura. De forma semelhante ao *Java Bytecode*, porém, ao invés de executar numa máquina virtual ou interpretar o *Assembly*, ele será traduzido para o *Assembly* nativo da máquina em que será executado.

O sistema consiste em 3 níveis: editor de *Assembly* intermediário, tradutor do *Assembly i*ntermediário para nativo e montador nativo.

## 8.1 Editor

Trata-se de um editor de código, em que a linguagem para a qual se direciona é um *Assembly* Intermediário, neste caso, o *Easembly*. Este editor deve possuir *Intelligent Code Completion* para facilitar a implementação do código e fazer uma interpretação básica da linguagem para checagem de erros de sintaxe.

Para que o Editor possa fazer o Intelligent Code Completion ele necessitará da sintaxe da linguagem para qual se deseja fazer o uso da ferramenta. Então é necessário que o projeto tenha um arquivo de sintaxe, o qual pode ser implementado como um dicionário dentro do código ou num arquivo de configurações a parte.

Para editar o código é necessário que o sistema permita a inclusão, exclusão e armazenamento do código.

## 8.2 Tradutor

Este tradutor tem como função traduzir o *Assembly* Intermediário para o *Assembly* Nativo, no qual a partir de um arquivo de configurações conseguirá fazer a tradução para uma ou mais linhas da linguagem para a qual se deseja traduzir.

Para que a tradução seja realizada, o sistema requer um arquivo dicionário para cada Assembly que se deseja traduzir. A fim de que não exista inconsistências nas implementações desta arquitetura, foi especificado que o arquivo dicionário será um arquivo no formato JSON.

## 8.3 Montador

Após a transcrição do código para o *Assembly* nativo, este deve ser montado para a linguagem de máquina relacionada ao *Assembly* transcrito. Para que isso aconteça de forma consistente é importante que o compilador seja o mesmo para todas as suas implementações. O compilador proposto para o sistema faz parte de um pacote Linux já existente no mercado: o SDCC, que possui vários montadores Assemblies, os quais serão utilizados para fazer a transcrição do código de Assembly gerado para o código de máquina da plataforma alvo.

O compilador SDCC pode ser utilizado por linha de comando, bem como seus montadores, possibilitando a sua utilização a partir de outro código, como o implementado neste trabalho. Além de mostrar erros de montagem, que são importantes para a validação do trabalho.

# 9 ESTUDO DE CASO

Este estudo de caso tem como objetivo fazer uma validação simples da arquitetura proposta, para isto foi implementado uma versão desta arquitetura e foram realizados alguns testes.

A arquitetura do sistema consiste em um servidor que utiliza PHP, tanto para realizar o processamento do *core* do sistema quanto para organizar e estruturar as páginas com o HTML5.No lado do cliente foi utilizado JavaScript, tanto para se comunicar com o servidor, quanto para realizar a parte mais complexa do sistema, o Editor de Código, incluindo também o roteamento dos módulos, que serão explicados com maior profundidade no decorrer deste capítulo, e tratamento de *cache* para acelerar o processamento/carregamento da página. E finalmente requererá que seus usuários sigam o padrão de nomenclatura “*Object Oriented Programming Languages Pattern”* descrito pelo projeto Open Source: “Pattern”[38].

## 9.1 *User Interface*

### 9.1.1 Single Page

O *design* HTML/CSS foi um dos maiores empecilhos para este sistema, pois a experiência necessária em *design* HTML/CSS, para fazer um sistema como este, era mínimo ou inexistente. E à medida que se avança no *design, ele* se torna maior e mais complexo, o que para alguém sem experiência dificulta imensamente o trabalho. Uma medida que facilitou esta implementação foi a modularização da página feita a partir dos includes do PHP, o qual pode se fundir ao HTML.

#### 9.1.1.1 Modularização da página

A única página acessada pelo usuário é a *index*. Esta página foi divida em módulos, que são arquivos no formato PHP/HTML, iniciais: *Head*, *Header*, *Body* e *Footer*, os quais, individualmente, carregarão outros módulos a partir de configurações enviadas para o servidor, a partir do *JavaScript* do cliente e/ou configurações padrão, *Cache*, *Storage*, *IP address*, localização e língua do navegador (como no caso do sistema de tradução).

O *Head* é responsável por armazenar e informar para o navegador informações como título, autor, *keywords*, descrição, ícone e afins. É responsável também pelos *links* de CSS utilizados pelas páginas, incluindo a seleção do CSS específico para cada tipo e tamanho de dispositivo. O *Header é* responsável pelo menu estático no topo da página, cujas atribuições são a navegação entre as subpáginas carregadas no *Body*, e o *login*/*logoff* .

O *Body* é o local onde são carregadas as páginas em que o usuário navegará pelo *site*, além de ter a possibilidade de carregar uma janela, por cima dele mesmo.

E, finalmente, há o Footer, com um menu estático no fim dá página, no qual se encontram dois botões que abrem suas respectivas janelas, um para tradução e outro para informações de contato e do *website.*

### 9.1.2 Cache/Storage manipulation

Para acelerar o carregamento da página foram utilizados *Cache* e *Web Storage*, a fim de que não existam requisições desnecessárias ao servidor. Ao carregar um módulo PHP/HTML, que para o cliente chega apenas como um trecho HTML, ele é salvo no *Cache* e no *Storage*. Antes de fazer uma requisição ao servidor ele checa se a informação já existe no *Cache* ou no *Storage* e, se ela existir, simplesmente a carrega no seu trecho específico, impedindo a existência de requisições desnecessárias ao servidor, e assim carrega as páginas mais rapidamente.

### 9.1.3 Ausência de Imagens

O uso de imagens sempre foi um empecilho no tráfego de dados pela rede *WEB*, e elas aparentam ser imprescindíveis, à primeira vista, pois são os símbolos digitalizados que mais se aproximam do mundo real. Isso as torna importantes para a realização de uma *interface* intuitiva, afinal, os botões, por exemplo, são uma reprodução de um elemento físico real muito conhecido pela humanidade, o botão.

Mas as imagens apresentam uma característica que pode transformá-las em um estorvo, o tamanho. Inicialmente, eram apenas uma matriz de cores, agora existem diversos formatos, que incluem, inclusive, compressão, que, às vezes, resultam até em perda de qualidade. Mesmo com todas as inovações para que elas deixem de ser um obstáculo, ainda não são suficientes, em muitos casos. Há pouco tempo o *Google* realizou uma transformação do seu *logo*, sendo o tamanho uma de suas preocupações, que saltou, em torno, de 14 *kbytes* para 305 *bytes*, resultando em um carregamento mais rápido, o que é muito importante num *site* como ele, muito utilizado[39].

Mesmo assim, a imagem ainda tem um problema difícil de ser contornado, que é permanecer com uma resolução agradável a todos os dispositivos, uma vez que cada uma tem resolução e tamanho físico diferentes, não adiantando apenas dar um *Zoom* na imagem, pois esta perderá sua qualidade. Então, como resolver um problema de tamanho sem ter uma equipe de *design* especializada? Como resolver o problema de imagens diferentes, para diferentes dispositivos?

Existem, então, formas pouco utilizadas pelos sistemas da atualidade, as imagens vetorizadas, que não utilizam muito espaço em disco e não perdem qualidade ao serem redimensionadas, como SVG [40]. No entanto é difícil encontrar um editor que tenha esse tipo de suporte, e que possa ser reproduzido por *browsers* e sistemas. Então, foi seguido um caminho menos ortodoxo, utilizando-se fontes de texto. Os processadores de fontes são altamente performáticos, fontes ocupam um espaço mínimo em disco e possuem um grande conjunto de símbolos. Cada um destes símbolos pode substituir uma imagem, sem contar que não perdem qualidade ao serem redimensionados, e possuem um grande suporte.

A estratégia seguida para contornar esse gargalo foi utilizar o mínimo de imagens possível, e onde não era possível removê-las, foram substituídas por símbolos de uma fonte.

Enfim este recurso teve como objetivo diminuir o tráfego de bytes para servidor, permitindo que ele possa focar seus recursos computacionais em tarefas mais importantes como a tradução do Easembly e na compilação. Além de tornar o site mais leve, rápido de ser carregado e de utilizar seus recursos.

### 9.1.4 Sistema de tradução

No *design* HTML não existe texto puro, sempre é uma variável carregada a partir de uma função de uma classe responsável pela linguagem. Tal classe checa a linguagem, seja por configuração, *default*, localização, IP address, idioma do *browser* ou mesmo seleção do usuário, e busca no sistema de arquivo se existe o arquivo para aquele módulo e para aquela linguagem. Caso exista, ele abre o arquivo de linguagem e procura a variável requisitada e, por fim, retorna seu valor. Caso contrário, ele retorna para o idioma *default*, que é inglês.

O arquivo de idioma foi criado para ser o mais simples possível, para que um tradutor leigo possa fazer um novo arquivo para uma nova linguagem, sem precisar entender de *tags* complexas de um tipo de arquivo específico como XML ou JSON.

### 9.1.5 Editor

O Editor do sistema implementado possui diversas ferramentas para facilitar a codificação, como pesquisa, refatoração de nome de variáveis, Intelligent Code Completion e várias outras ferramentas. Para fazer um editor com *Intelligent Code Completion* e um Interpretador básico para erros de sintaxe, foi utilizado um *embedded* editor, chamado *Ace*. Infelizmente, devido a algumas tecnologias usadas no projeto, e por utilizar linguagens não suportadas por este editor, foram necessárias algumas modificações em seu código-fonte, como inclusões e edições de arquivos *JavaScript* em pastas como *Model*, *Documents*, *Requires*, e outras mais. Além, é claro, de um CSS específico para se encaixar ao *Design* do *site*.

Neste caso, ele pode ser usado para Codificar para *Easembly,* *Assembly* do Z80 e *Assembly* do 8051.

### 9.1.6 Autenticação/Login

O *Login* e a Autenticação do usuário são feitas pela API do *Facebook*, eliminando assim a preocupação com o registro, armazenamento e segurança dos dados do usuário.

### 8.1.7 CRUD

A UI permite que o usuário crie, pesquise, remova e edite, como a Figura 27 mostra, arquivos *Assembly* de 8051, Z80 e o *Easembly.*

## 9.2 Core

### 9.2.1 Tradutor

O Tradutor de *Easembly* para outro *Assembly* funciona a partir do momento em que o usuário clica no botão de salvar ou compilar. Feito isso, é enviado o código feito pelo usuário para o servidor, este código é armazenado em arquivo texto no formato .easy e, então, lido. A fase seguinte é a tradução.

Para cada *Assembly* requisitado para a realização de uma tradução, é necessário um dicionário para dizer ao tradutor como fazer tal procedimento. Este dicionário utiliza um arquivo JSON, que contém a instrução, ou instruções, respectiva no *Assembly* destino para um comando do *Easembly*, assim como o mapeamento de registradores e identificadores de base numérica (prefixos e/ou sufixos que determinam a base de um valor numérico).

Com o dicionário é possível fazer a tradução e, então, armazenar o arquivo *Assembly* resultante da tradução.

### 9.2.2 Compilador

O sistema optou por utilizar o SDCC de código aberto, que pode ser utilizado via linha de comando, a qual é executada após a tradução ser realizada, se o usuário tiver clicado no botão para compilar.

Este compilador foi escolhido, pois pode fazer a compilação dos dois *Assemblies* selecionados para o projeto das arquiteturas dos processadores Z80 e 8051, além de diversos outras arquitetura que o Easembly possa no futuro traduzir.

# 10 ANÁLISE DOS RESULTADOS

A análise a seguir será feita de forma simplificada, pois o estudo apenas tem como objetivo mostrar a potencial possibilidade de funcionar para todos os *Assemblies*. Também devido ao prazo, não foi possível fazer estudos mais aprofundados, nem testes de performance com estatísticas.

O primeiro programa teste tem como objetivo testar o uso da instrução de movimentação de dados, envolvendo todas suas variações básicas, além de testar todas as possibilidades de usar um comando Easembly e testar um comando complexo, ou seja um comando que se traduz em mais de uma instrução no Assembly alvo. Este programa e o resultado de suas traduções podem ser visualizados na Figura 31. E foi possível notar que o resultado da tradução foi coerente e dentro do esperado. Ao final estes programas foram montados pelo SDCC, sem acusar nenhum erro de compilação, indicando que o programa transcrito é montável.

A imagem acima mostra um código *Easembly* e o resultado de sua tradução para os *Assemblies* dos processadores 8051 e Z80, respectivamente. É possível notar que o *Easembly* ficará sempre limitado pelo processador mais “simples”. No exemplo da imagem, é possível notar que como o Z80 tem uma instrução de *delay* de 4 ciclos de *clock*, e o 8051, de apenas um ciclo, o sistema só suportará *delay* múltiplos de 4, perdendo, então, todas as possibilidades da instrução mais flexível. No caso, um código feito em 8051 poderia ter um *delay* de 5 ciclos, enquanto um feito em Z80 ou gerado a partir o *Easembly* atual não poderia.

O segundo programa tem como objetivo gerar um programa de delay na ordem de 40 ciclos de clock, verificar a variação de ciclos gerado entre os códigos gerados além de testar loop. Os ciclos de cada instrução podem ser vistos na Figura 32, abaixo. Foi possível notar que estes programas transcritos permanecem coerentes depois da tradução.

O terceiro programa é um programa para testar operações lógico-aritméticas, no caso foi feito um programa para multiplicar dois valores. E em seguida foram feitos programas manualmente para verificar a perda de performance do código gerado. O código feito em Z80 manualmente não houve muita diferença em relação ao código gerado, então foi omitido, mas o código feito para 8051 teve uma diferença considerável. O código manual, como permite utilizar todas as instruções, utiliza apenas 6 ciclos enquanto o gerado pelo sistema é da ordem de n, n sendo o número de vezes que o número será somado repetidas vezes. A razão pela qual o programa foi feito desta maneira, ocorreu porque o sistema ignora a endianness dos processadores. Mesmo a complexidade deste programa sendo mais complexa o resultado de sua tradução também demonstrou-se correta.

O quarto programa foi um programa de divisão de um valor por outro, seguindo estratégia semelhante ao programa de multiplicação. E novamente tem-se uma diferença considerável entre o programa manual e o gerado pelo Easembly. O resultado da tradução pode ser visualizado na figura 34 abaixo. Este programa de forma semelhante aos outros acima, também obteve sucesso ao traduzir para as plataformas alvo de forma correta.

Como o *Easembly* tem um conjunto de instruções menor que o conjunto das plataformas que gera, ele perde no quesito performance, pois existem instruções em ambas as arquiteturas que podem fazer em apenas um comando, o que o *Easembly* faz utilizando várias instruções. O *Easembly* poderia ter um conjunto maior que as arquiteturas destino atuais, incluir todas as instruções específicas e, assim, sanar o problema de desempenho. Caso isso fosse feito, outro problema apareceria. Como instruções específicas podem não existir em uma ou mais das plataformas alvo, isso implicaria em uma sequência de comandos para executar a operação desejada. Isto significa que uma instrução exclusiva de uma plataforma, nas demais arquiteturas, deverá ser composta por uma sequência de instruções, para ter o mesmo resultado. Esta sequência de comandos poderá implicar no uso de registradores. Para implementar esta ideia, existem algumas possibilidades, na hipótese de não existir nenhum registrador a mais que o *Easembly*, o sistema teria que, de alguma forma, saber se existe algum registrador que não tenha sido usado ou usar a pilha, caso não esteja cheia. Na possibilidade de ter algum registrador a mais, ou que sejam definidos um ou mais registradores exclusivos, é possível utilizá-los para contornar este problema.

Outro problema encontrado foi o número de registradores, o qual precisa ser reduzido para, no máximo, a quantidade de registradores da arquitetura que tem menos. Isso implica que o sistema restringirá o programador pelo número do processador com menos registradores.

Enfim, o *Easembly* sempre ficará restringido, de alguma forma, pelos processadores mais “simples”. Mas é importante ressaltar que o sistema obteve sucesso em traduzir todos os testes feitos com a lógica correta, além de suas transcrições serem livres de erros de montagem, pelo menos para os montadores do pacote SDCC.

# 11 CONCLUSÃO

A partir da análise dos resultados obtidos foi possível perceber que a arquitetura permite a união dos Assemblies de diversos processadores. No entanto ficará restringido, de alguma forma, pelos processadores mais básicos.

Então, a arquitetura proposta mostrou-se viável, o que o torna de alguma forma útil, mas, provavelmente, haverá uma queda na performance de seus programas, à medida que mais processadores forem se integrando à plataforma. Para solucionar este problema, sugere-se um trabalho futuro, cujo objetivo é ter uma arquitetura mais flexível para que as plataformas superiores não sejam tão mal aproveitadas devido ao rebaixamento das inferiores.

O trabalho futuro aqui proposto aproveitará a ideia do Easembly, no entanto o programador selecionará para quais plataformas deseja programar, e o sistema criará um set de instruções para aquele conjunto de processadores. Dessa forma as plataformas serão melhor aproveitadas, já que o conjunto de instruções semelhantes será, provavelmente maior, em virtude do conjunto de processadores selecionados pelo usuário ser reduzido. Inclusive, o número de registradores seria, presumivelmente similar, pois o programador selecionaria processadores compatíveis com o projeto que ele desejasse realizar, indicando uma provável semelhança.