

# Entendendo a Necessidade de uma ISA Aberta e Implementação de um Núcleo RISC-V

João Vitor Rafael Chrisóstomo

**Resumo**—Este documento descreve um estudo sobre arquiteturas abertas, dedicadas e de domínio específico, a necessidade de uma ISA aberta e a implementação de um núcleo de processamento RISC-V, que é um conjunto de instruções modular, ampliável através de extensões, *open-source*, *royalty-free*, originalmente desenvolvido pela UC Berkeley e adequado para uso acadêmico e comercial. O núcleo foi inteiramente implementado na linguagem de descrição de hardware VHDL. Possui arquitetura Harvard, *Load-Store*, *In-Order* e uma pipeline clássica RISC de 5 estágios. A implementação foi testada contra o simulador online de RISC-V Venus. O núcleo foi primariamente projetado com intuios acadêmicos e de aprendizagem, existem alternativas mais completas disponíveis no mercado.

**Palavras Chave**— RISC-V, Processador, Núcleo, Pipeline, *In-Order*, Arquitetura Harvard, *Load-Store*, *Little-Endian*, VHDL, FPGA, Altera, RTL design, Open-Source Hardware.

## I. INTRODUÇÃO

COM o avanço da tecnologia de semicondutores e o aperfeiçoamento dos processos de fabricação, a indústria de eletrônicos vivenciou constante progresso em diversas métricas como, por exemplo, performance, consumo de potência, área do *die* e frequência máxima de operação. As Figuras 1.2 e 1.3 demonstram esta evolução ao longo de 45 anos.

Consequentemente, os custos destes dispositivos vêm gradativamente se tornando mais acessíveis e hoje se vivencia uma situação em que não somente computadores de grande porte como também sistemas embarcados estão presentes na sociedade de diversas maneiras. A Figura 1.1 demonstra o crescimento do número de linhas de telefones celulares.

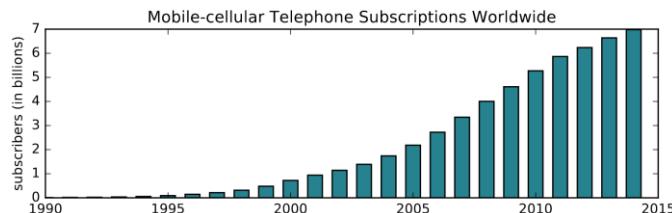


Figura 1.1 – Número de linhas de celular ao longo dos anos.

Fonte: [1].

Se acompanhamos a trajetória da evolução dos semicondutores, é possível observar que recentemente a quantidade de tempo entre litografias está gradativamente aumentando. A Intel utilizou seu processo de 14 nm durante 5 anos (de 2014 a 2019), enquanto na década de 90 e nos anos 2000 os processos eram substituídos, em média, a cada 2 anos.

Para que seja possível contornar o problema e dar continuidade ao legado, além de frentes de pesquisa que focam em avanços nos processos de fabricação como a utilização de litografias EUV (*Extreme Ultraviolet*) [2], [3], hoje também existem estudos paralelos em diversas outras áreas almejando melhorias em performance, densidade e preço.

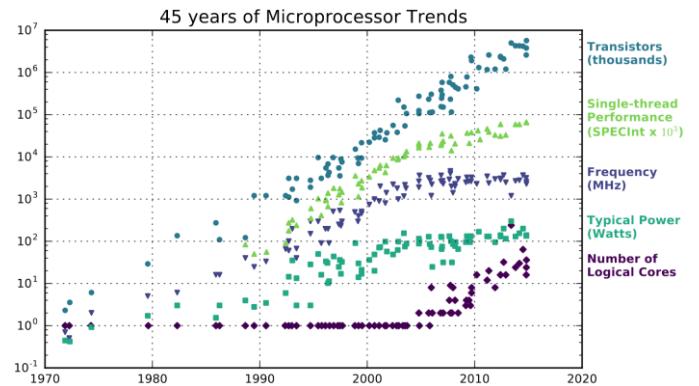


Figura 1.2 – Tendências na Indústria de Microprocessadores.

Fonte: [4], [5].

Dentre os diversos estudos, tem-se o empilhamento vertical de circuitos integrados interconectados através de *Through Silicon Vias* [6] e barramentos de interconexão para múltiplos circuitos integrados, processamento distribuído (AMD Infinity Fabric, por exemplo) [7] e a volta dos sistemas digitais altamente especializados de domínio específico [8], [9].

Empresas como TSMC (Taiwan Semiconductors), GlobalFoundries, Intel e Samsung, todas fabricantes de semicondutores, estão em constante colaboração com empresas projetistas de sistemas digitais como por exemplo ARM, Qualcomm, Apple e IBM. A inovação hoje não mais está atrelada majoritariamente à fabricação, mas sim também às arquiteturas dedicadas.

A dificuldade de se conseguir mais performance em processadores de uso geral, como pode ser visto na Figura 1.2, levou a indústria a abordar os problemas com paralelismo em mente. O número médio de núcleos de processamento em computadores pessoais passou de 1 a 4, e nos aparelhos celulares muitas vezes possuímos *clusters* distintos para núcleos de alta eficiência e núcleos de alta performance. O System on Chip Samsung Exynos 9820, por exemplo, possui 2 núcleos Moongose 4 (IP da própria Samsung), 2 núcleos ARM Cortex-A75 e 2 núcleos ARM Cortex-A55 [10].

# SPECint2006 Processor Performance vs VAX-11/780

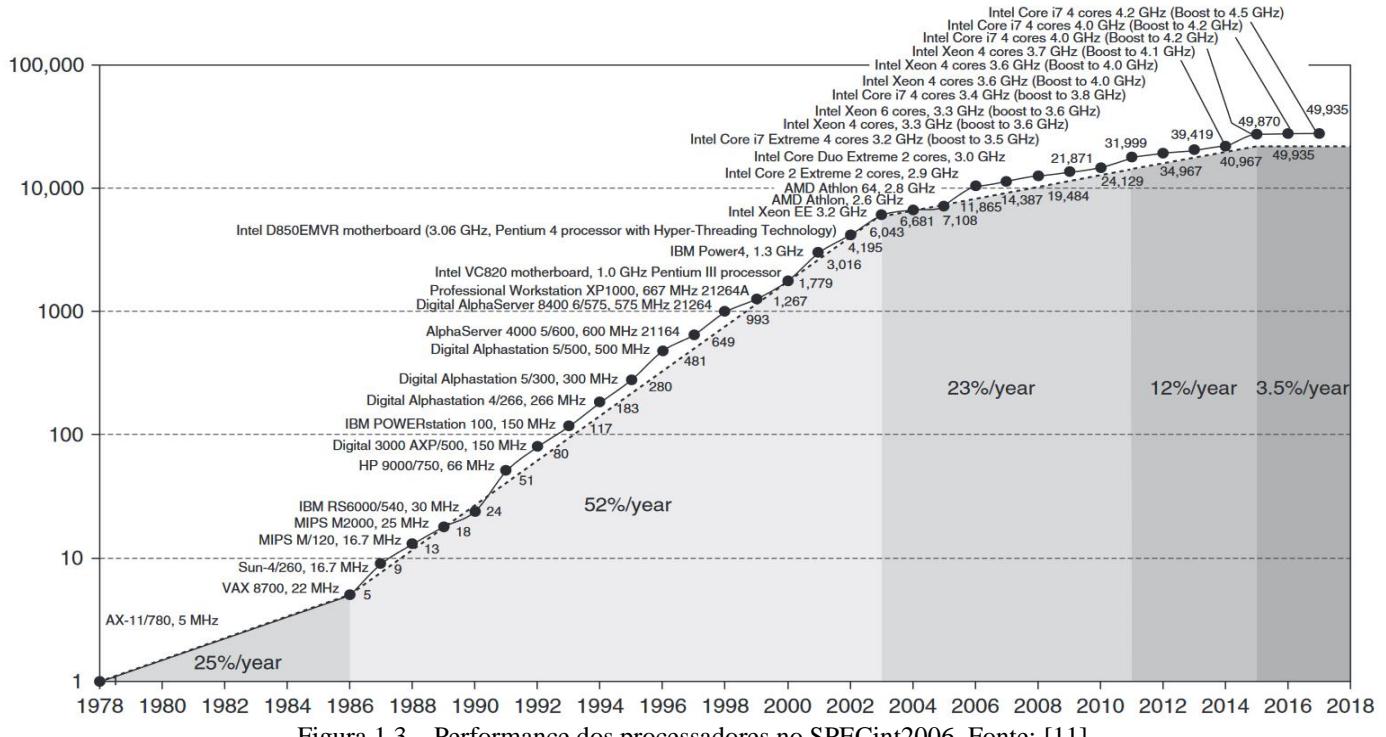


Figura 1.3 – Performance dos processadores no SPECint2006. Fonte: [11].

A utilização de GPUs ou FPGAs como aceleradores também se tornou popular. A grande maioria dos supercomputadores modernos possuem sistemas híbridos compostos por CPUs e GPUs, incluindo os dois primeiros colocados do ranking TOP500 [12], que possuem CPUs IBM POWER 9 e GPUs Nvidia Tesla V100 [13], [14]. A Figura 1.4 demonstra um *node* do supercomputador Summit.

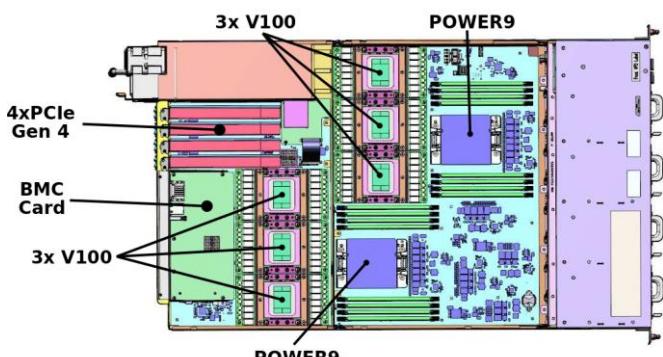


Figura 1.4 - Fonte: IBM.

Diante desta realidade, dispositivos eletrônicos extremamente especializados em uma única tarefa, denominados de “domínio específico”, surgem com o intuito de promover uma melhora significativa em eficiência computacional e energética.

Empresas de grande escala optaram por desenvolver seus próprios aceleradores de uso específico. O Google, por

exemplo, desenvolveu TPUs (Tensor Processing Units) direcionadas a aplicações de inteligência artificial, que demonstram resultados satisfatoriamente positivos quando comparadas à GPUs [15], isto pode ser observado na Figura 1.5.

## ResNet-50 Training Cost



Figura 1.5 - Fonte: [15].

Entretanto, é possível observar que ao longo dos anos as novas linguagens de programação tenderam a promover abstrações para agilizar o tempo de desenvolvimento e facilitar a vida dos programadores. Infelizmente, abstrações e generalizações costumam andar de mãos dadas com perda de performance e má utilização do hardware disponível. Este fenômeno é claramente observável quando comparamos a velocidade de execução de programas implementados em linguagens de extremo alto nível como Python com equivalentes em C ou C++, e em alguns casos é possível ver

ganhos significativos em uma implementação escrita manualmente em Assembly, especialmente quando a mesma utiliza instruções de operações SIMD (*Single Instruction Multiple Data*, AVX para x86 e NEON para ARM).

A Figura 1.6 demonstra a diferença de performance em SoCs ARM executando código escrito em C e seu equivalente em Assembly utilizando instruções NEON. A Figura 1.7, por sua vez, demonstra como é possível adquirir mais performance com os processadores atualmente no mercado caso seus recursos sejam bem utilizados.

Dav1d AV1 Decoder - ARM performance C vs NEON

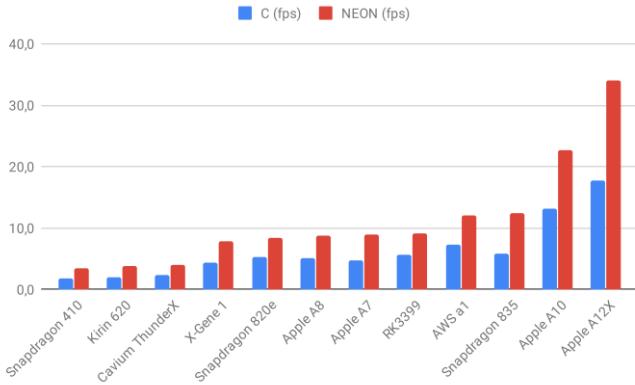


Figura 1.6 - Performance do dav1d, código C e NEON.

Fonte: [16].

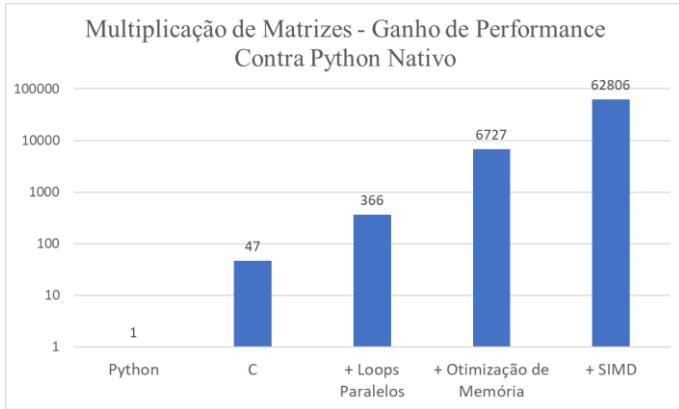


Figura 1.7 – Performance em multiplicação de matrizes.

Fonte: Adaptado de [17].

A padronização das diversas interfaces de comunicação, protocolos, codificações e conexão de componentes proporciona um ambiente de desenvolvimento em que projetistas podem utilizar diferentes IPs como *building blocks*, de forma que projetos complexos possam ser realizados mais rapidamente e com custo reduzido.

A colaboração dentro da indústria de eletrônicos deu origem a diversos padrões adotados pela grande maioria dos fabricantes. Dentre eles, temos exemplos de protocolos de comunicação como PCI-Express, AXI (AMBA), I2C, Wishbone, NVMe, SATA, etc. A padronização é de interesse das empresas pois aumenta o alcance de seus produtos e torna-

os adequados a serem utilizados em uma gama de sistemas distintos muitas vezes mantendo a compatibilidade com qualquer fabricante arbitrário contanto que ambos utilizem os mesmos protocolos.

Infelizmente, quando se trata de processadores e seus conjuntos de instruções, a colaboração no mercado é um acontecimento recente [18]. O mercado é largamente preso a ISAs (*Instruction Set Architecture*) proprietárias fechadas, que estão sobre o controle de suas respectivas empresas e não proporcionam maleabilidade, liberdade de *design* ou garantia de segurança. As duas ISAs mais presentes hoje no mercado são sem sombra de dúvida ARM e x86, e nenhuma das duas permite implementações *royalty-free*.

As Figuras 1.8 e 1.9 mostram como as ISAs ARM e x86 cresceram no mercado ao longo das décadas.



Figura 1.8 - Crescimento dos produtos que utilizam CPUs ARM. Fonte: ARM 2018 Roadshow.

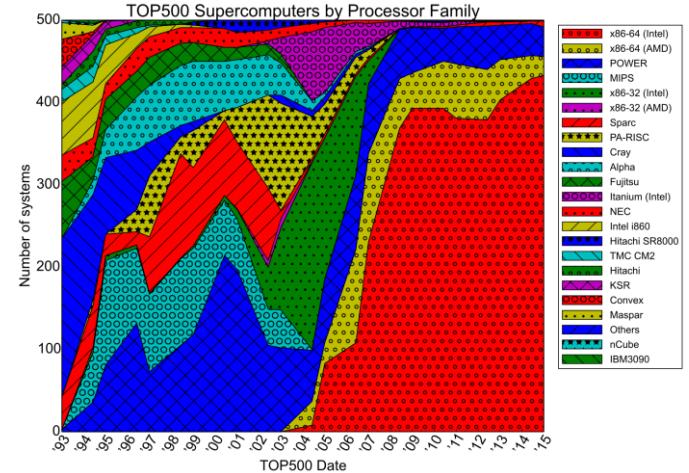


Figura 1.9 - Distribuição das ISAs em supercomputadores. Fonte: [12].

O plano de negócios da ARM, mostrado na Figura 1.10, se sustenta ao redor de *royalties* cobrados em cima da venda de dispositivos que utilizam propriedade intelectual da empresa.

Em casos raros, a ARM também licencia a própria ISA (atualmente ARMv8), permitindo que empresas parceiras desenvolvam seus próprios núcleos (Samsung, Apple, Marvell, Qualcomm, etc).

# CPU Engagement Models With ARM

## Cortex License

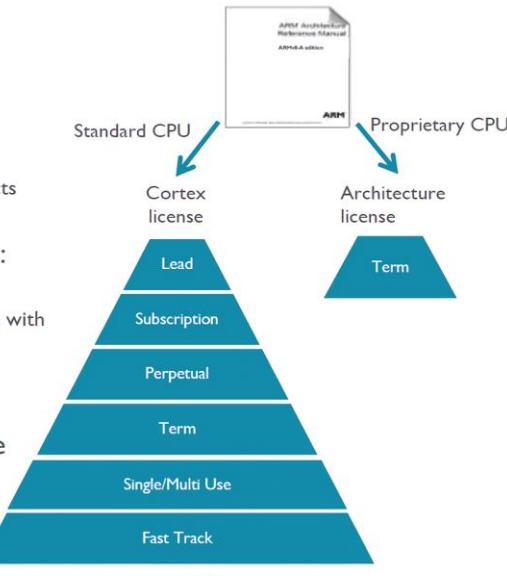
Partner licenses complete microarchitecture design

- Wide choices available
- Many different A, R & M products

CPU differentiation through:

- Flexible configuration options
- Wide implementation envelope with different process technologies

Range of licensing & engagement models possible



**ANANDTECH**  
© ARM 2016

## Architecture License

Partner designs complete CPU microarchitecture from scratch

- Clean room – no reference to Cortex designs

Freedom to develop any design

- Must conform to the rules & programmers model of a given architecture variant
- Must pass ARM architecture validation to preserve software compatibility

Long term strategic investment

**ARM**

Figura 1.10 - Fonte: [19].

A Intel e a AMD detêm total controle sobre a arquitetura x86 e devido a fatores históricos esta ISA se estabeleceu no mercado de computadores pessoais e servidores.

Em 2005, a Apple anunciou que adotaria processadores Intel x86 em sua linha de computadores pessoais, que até então utilizava processadores IBM PowerPC. A transição durou vários anos e somente foi inteiramente finalizada em 2013. Hoje, uma possível transição de x86 para outra ISA provavelmente ainda seria um processo árduo, mas a abundância de software *open-source* é indiscutível e em muitos casos a recompilação para outra arquitetura é relativamente trivial, contanto que os compiladores suportem a ISA.

Em 2018, diversas vulnerabilidades foram descobertas em processadores Intel, ARM e IBM, como Meltdown, Spectre e Zombieload. Diversos sistemas, mesmo que tenham sido projetados com segurança em mente, tornaram-se comprometidos [20]–[22]. Tamanha tragédia talvez pudesse ter sido evitada caso as especificações das ISAs assim como as descrições de hardware dos núcleos fossem abertamente publicadas e pudessem ser revisadas por toda a indústria mais rapidamente [18].

RISC-V é uma ISA moderna, modular, ampliável através de extensões, aberta e *royalty-free*. Sua especificação está inteiramente disponível e seu objetivo principal é servir de base para que empresas, instituições, ou qualquer outra entidade possa desenvolver não somente periféricos compatíveis mas também seus próprios núcleos, SoCs e aceleradores [18].

No início de 2019, a RISC-V Foundation já possuía mais de 275 membros filiados [23]. Empresas importantes na indústria como Nvidia e Western Digital já declararam a utilização de processadores RISC-V simples em seus produtos [24], [25]. Outras empresas como Google, Samsung, Thales, Qualcomm, Microchip, NXP, Micron, IBM, Huawei, Mellanox, Nokia,

Sony, SK Hynix e TSMC fazem parte da fundação e demonstram interesse em utilizar RISC-V [26].

O suporte de software já existe, compiladores essenciais como GCC e Clang (LLVM) já geram código RISC-V nativamente. Sistemas operacionais de uso geral como Linux e FreeBSD foram pioneiros em implementar as mudanças necessárias para suportar RISC-V, embora hoje sistemas operacionais em tempo real como FreeRTOS e Zephyr também o suportem [27].

Devido à natureza aberta, engenheiros possuem inteira maleabilidade em seus projetos e diferentes núcleos podem ser projetados com intuito de priorizar diferentes atributos. Núcleos projetados para um uso específico podem conter extensões customizadas, ou hardware adicional. No que diz respeito à segurança, abertura dos documentos e das descrições de hardware fazem com que falhas de segurança possam ser detectadas e corrigidas muito mais rapidamente.

A indústria já é habituada a operar ao redor de diferentes padronizações de maneira satisfatória, e este fenômeno é facilmente observável principalmente em software. A tabela 1.1 demonstra vários exemplos de implementações abertas e proprietárias que coexistem e atendem ao mesmo padrão.

Campo	Padrão	Implementação aberta	Implementação proprietária
Sistema Operacional	POSIX	Linux, FreeBSD	MacOS, AIX
Compilador	C	GCC, Clang	ICC, MSVC
Video Encoder	H.265	x265	NVENC
API Gráfica	Vulkan	Anvil, RADV	GeForceDriver

Tabela 1.1 - Padrões e Implementações. Fonte: Autor.

## II. PIONEIROS DO RISC-V

A SiFive, empresa fundada pelos autores da ISA RISC-V, já possuía produtos disponíveis em 2017. Em 2018 lançou a HiFive Unleashed, primeira placa de desenvolvimento capaz de rodar Linux, e a Hifive 1, uma placa que é compatível com os ambientes de desenvolvimento para Arduino. Devido à forte correlação entre o Berkeley Architecture Research e a SiFive, seus IPs são normalmente escritos em Chisel, uma linguagem de descrição de hardware de alto nível embarcada em Scala [28].

A Andes Technology Corporation, um dos membros fundadores da RISC-V Foundation, lançou seus primeiros núcleos no mercado em 2017. A empresa desenvolveu diversas extensões personalizadas e contribui ativamente no crescimento do ecossistema [29].

A Esperanto Technologies possui núcleos de uso específico em aplicações de *machine learning* [30].

A Syntacore, outra empresa fundadora da RISC-V Foundation, licencia sua propriedade intelectual relacionada a RISC-V desde 2015. Eles mantêm uma implementação *Open-Source* inteiramente escrita em System Verilog destinada a aplicações de IoT e sistemas embarcados [31].

A Western Digital, em 2019 anunciou o núcleo SweRV, que é *superscalar* (*2 way dispatch*) com uma *pipeline* de 9 estágios. O núcleo está disponível no GitHub e é escrito em System Verilog. A empresa irá utilizá-lo em seus controladores de memória *flash* e SSDs [32].

O Alibaba Group, em 2019 anunciou o processador XuanTie 910, que possui 16 núcleos RV64GC. Esta iniciativa demonstra o interesse Chinês, um gigante na indústria de eletrônicos, na ISA RISC-V, devido a sua característica de livre uso [33].

O MIT CSAIL CSG possui núcleos RISC-V escritos em Bluespec, uma linguagem de descrição de hardware de alto nível baseada em Haskell, desenvolvida por um professor também do MIT, Arvind Mithal [34].

O Indian Institute of Technology Madras desenvolveu diversos núcleos RISC-V escritos em Bluespec para aplicações que variam de IoT a servidores, seu projeto é chamado Shakti e eles buscam desenvolver também todas as ferramentas necessárias para o workflow VLSI (Very Large Scale Integration) [35].

A Nvidia, planeja utilizar núcleos RISC-V produzidos internamente na empresa para substituir os processadores Falcon, que controlam a sinalização e comunicação de diversos componentes em suas placas de vídeo [24].

A LowRISC é um projeto sem fins lucrativos que busca unicamente catapultar o movimento de *open-source hardware*, introduzindo um SoC inteiramente aberto com componentes de livre uso [36].

A universidade de Cambridge, Reino Unido, juntamente aos desenvolvedores do FreeBSD, trabalharam no pente do sistema operacional a plataforma RISC-V, visando criar um ecossistema hardware-software inteiramente aberto para ser utilizado academicamente em suas pesquisas [37], [38].

A ETH Zurich e a universidade de Bologna desenvolveram juntos o PULP, projeto que engloba diferentes núcleos orientados a IoT com consumo de potência extremamente baixo [39].

A SpinalHDL, como forma de promover sua linguagem de descrição de hardware de alto nível com o mesmo nome, desenvolveu um núcleo RISC-V próprio com foco em FPGAs. O núcleo venceu a competição de *softcores* RISC-V [40] e é utilizado no tutorial de RISC-V “Getting Started Guide”, ao lado dos sistemas operacionais Linux e Zephyr [41].

## III. OUTROS CONJUNTOS ABERTOS E A ESCOLHA DE RISC-V

A ISA RISC-V começou a ser desenvolvida pela UC Berkeley em 2010 como uma ISA interna, com intuito de ser utilizada em disciplinas, projetos de pesquisa e laboratórios da própria universidade. Após as especificações iniciais serem publicadas, a universidade começou a receber questionamentos externos a respeito de modificações semestrais na ISA, que deixou claro o interesse externo.

O nome RISC-V foi escolhido para representar a quinta iteração de ISAs da UC Berkeley, sendo RISC-I, RISC-II, SOAR e SPUR as outras quatro.

A ISA está altamente relacionada aos professores doutores Krste Asanović e David Patterson, ambos extremamente renomeados academicamente, o que facilitou catapultar o projeto e a adoção comercial.

RISC-V, entretanto, não foi a primeira tentativa de se estabelecer uma ISA aberta e livre para uso. Os autores avaliaram as opções existentes no mercado e sistematicamente desenvolveram RISC-V de maneira com que boa parte dos problemas enfrentados anteriormente pudessem ser evitados [8]. Dentre estes problemas, tem-se:

- Deixar muita coisa de fora. A ISA “Alpha” não possuía instruções de acesso a memória para transações de 8 ou 16 bits. MIPS não possuía instruções de acesso a memória explícitas para Floating Point [8].
- Incluir coisa demais. ARM possui diversas opções diferentes redundantes de *shift* e SPARC possui janelas para visibilidade de registradores [8].
- Deixar que os designs atuais de implementação influenciem a ISA. Como por exemplo atraso de desvio presente nas ISAs MIPS, SPARC e OpenRISC ou *traps* em operações de ponto flutuante na ISA Alpha [8].

Além disso, MIPS possuía diversos problemas relacionados a propriedade intelectual e marca registrada, possibilitando que implementações fossem possíveis contanto que não utilizassem o nome “MIPS”. OpenRISC é uma alternativa mais moderna, entretanto possui uma licença não permissiva (GPL) que não é atrativa para o mercado devido ao fato de que muitas empresas preferem manter suas descrições fechadas, de forma com que possam lucrar a partir do IP.

ISAs geralmente levam muito tempo para ser adotadas e ainda mais tempo para caírem em desuso caso alcancem relevância comercial. Logo, é preciso levar em consideração quais áreas estão emergindo e quais características são importantes para garantir que a mesma não venha a se tornar obsoleta.

Krste Asanović e David Patterson levaram em consideração vários critérios partindo do princípio de que três áreas principais seriam o foco da computação em um futuro próximo, sendo elas o mercado de IoT e sistemas embarcados, o mercado de

aparelhos móveis como celulares e tablets, e o mercado de servidores com processadores de alto desempenho e aceleradores de domínio específico [8].

Para atender plenamente todos os mercados citados acima com uma mesma ISA, os autores definiram os seguintes critérios.

- ISA base com extensões opcionais.
- Codificação de instrução compacta.
- Ponto flutuante de quadrupla precisão.
- Endereçamento em 32, 64 e 128 bits.

ISA	Base+Ext	Compact Code	Quad FP	Address	GCC	LLVM	Linux	QEMU
				32-bit	64-bit	128-bit		
SPARC V8	✓	✓		✓	✓	✓	✓	✓
OpenRISC			✓	✓	✓	✓	✓	✓
RISC-V	✓	✓	✓	✓	✓	✓	✓	✓

Figura 3.1 - Comparação das ISAs abertas. Fonte [8].

A Figura 3.1 faz um comparativo entre RISC-V e suas possíveis concorrentes em 2014. É importante salientar que a situação atual em 2019 é ligeiramente diferente, com MIPS inteiramente aberta e a possibilidade de se utilizar IBM Power em designs mais complexos.

Em agosto de 2019, a IBM anunciou que a Power ISA se tornaria *royalty-free* e que a OpenPOWER Foundation, uma associação de empresas com objetivo de catapultar o mercado de HPC (*High Performance Computing*) utilizando tecnologias abertas ao redor de processadores Power, se tornaria parte da Linux Foundation [42].

Os documentos de especificação da ISA já estavam disponíveis ao público, mas implementações eram bloqueadas por patentes e propriedades intelectuais da IBM [43]. Até então a IBM licenciava a utilização da ISA á parceiros dentro da fundação OpenPOWER.

Processadores IBM Power no passado foram utilizados, por exemplo, pela Apple, pela Sony em seu console Playstation 3 e pela Nintendo no Nintendo Wii. O foco da IBM atualmente encontra-se em servidores de alto desempenho e supercomputadores. A empresa, em parceria com a Nvidia e a Mellanox, ambas membros chave da fundação OpenPOWER, são responsáveis pelos supercomputadores Summit [13] e Sierra [14], atualmente os dois primeiros colocados no ranking Top500 [6].

A mudança de mentalidade busca trazer o sucesso da adoção rápida de RISC-V para o atual ecossistema Power, que engloba diversas tecnologias abertas relacionadas ao suporte de aceleradores externos como OpenCAPI (Open Coherent Accelerator Processor Interface) e OMI (Open Memory Interface).

A IBM, também em 2019, completou a aquisição da empresa Red Hat, responsável por diversos projetos *open-source* e possivelmente uma das maiores contribuidoras individuais do Linux, tornando-a hoje capaz de oferecer uma solução completamente aberta do hardware ao software.

Desta forma, a ISA Power pode se tornar uma competidora direta de RISC-V no que diz respeito abertura e liberdade de uso das especificações. Entretanto, RISC-V é uma ISA mais compacta, possui design modular e o subconjunto de base é perfeitamente adequado para diversas aplicações de baixa potência como sistemas embarcados.

A ISA RISC-V causou diversos outros impactos no mercado além da abertura da ISA Power, dentre eles é possível citar exemplos como a abertura da ARM para extensões customizadas nos núcleos Cortex-M [44], a declaração do governo indiano de que RISC-V será a ISA nacional de seus projetos [45], o forte investimento da China na ISA [33] e o estudo feito pela agência espacial europeia avaliando a viabilidade de substituir a ISA SPARC, utilizada em seus processadores LEON, por soluções baseadas em RISC-V [46].

A Figura 3.2 mostra os processadores e os respectivos conjuntos de instruções projetados para missões espaciais.

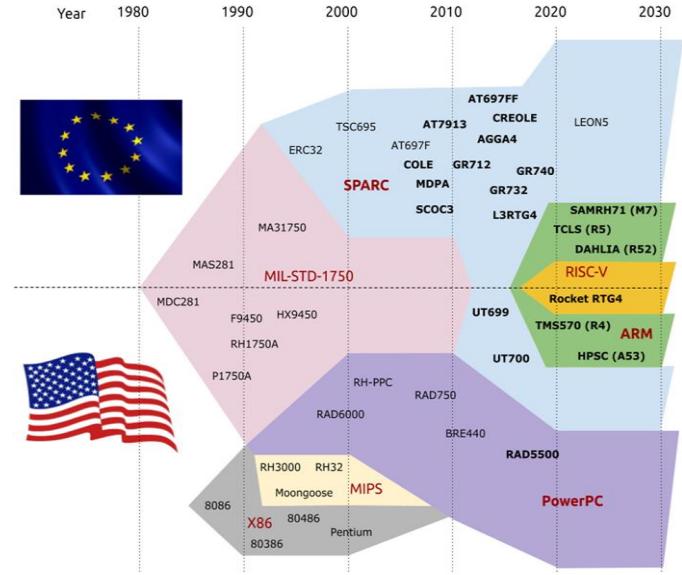


Figura 3.2 Processadores projetados para missões espaciais ao longo dos anos. Fonte: [46].

A Microsemi é uma empresa que disponibiliza FPGAs com as devidas proteções contra radiação e seus produtos são utilizados em missões espaciais com frequência. O ecossistema Mi-V [47] é a solução da Microsemi para incorporar núcleos RISC-V em seu ambiente de desenvolvimento, o LiberoSoC, de forma com que projetistas possam embarcar processadores RISC-V e depurar o software naturalmente com o mesmo *workflow* que eles possuem com alternativas ARM ou SPARC. A Agência Espacial Europeia escolheu utilizar SPARC nos anos 90 pelo fato de que a ISA era aberta e *royalty-free*, entretanto hoje já existe um ecossistema mais saudável em torno de RISC-V e estudos referentes a uma possível transição estão sendo realizados [46].

## IV. O CONJUNTO RISC-V

### A. Modularidade

Uma arquitetura de conjunto de instruções define os registradores internos, o funcionamento individual de cada instrução ao nível de máquina e o comportamento esperado do ponto de vista do programador.

A ISA RISC-V é dividida em níveis de privilégio, que servem para facilitar a implementação de sistemas operacionais e *hypervisors*, para máquinas virtuais.

A especificação formal da ISA está inteiramente contida nos documentos [48] e [49].

Conjuntos base de 32, 64 e 128 bits são especificados e funcionalidades adicionais podem ser implementadas através de extensões. Extensões padronizadas existem para as tarefas mais comuns esperadas de um processador moderno, entretanto nada impede que possíveis implementações possuam extensões customizadas inteiramente projetadas para um domínio específico.

A modularidade da ISA faz possível que implementações não assumam o compromisso de suportar todas as possíveis instruções, que por ventura habilita a existência de núcleos menores, mais baratos e mais especializados.

Os conjuntos base encorpam as principais operações com inteiros. Dentre as extensões padronizadas, temos instruções opcionais para operações com multiplicação e divisão, Atomics, Single Precision Floating-Point, Double Precision Floating-Point, acesso aos registradores de controle e status, Instruction-Fetch Fence, Quad-Precision Floating Point, Decimal Floating Point, instruções de 16 bits, manipulação de bit, Dynamic Languages, Transactional Memory, Packed-SIMD, Vectors, User Level Interrupts, Misaligned Atomics, Total Store Ordering, Supervisor-Level ISA, Hypervisor-Level ISA e Machine-Level ISA. A Figura 4.1 apresenta as extensões e suas respectivas siglas.

Para aplicações simples e sistemas embarcados, o conjunto base RV32E “Embedded” existe como alternativa ao RV32I convencional, onde a diferença está no número de registradores internos de uso geral, 16 em vez de 32.

Microcontroladores destinados a rodar programas escritos diretamente para RISC-V sem a necessidade da existência de um sistema operacional para administrar diferentes tarefas e recursos podendo também optar por conjuntos compactos como RV32EMC, onde a extensão M está presente para incluir instruções de multiplicação e divisão, enquanto a extensão C diminui o tamanho absoluto do programa através de instruções com codificação de 16 bits.

Processadores de alto desempenho e de uso geral designados a rodar kernels mais complexos como Linux, provavelmente optarão pelas extensões M, A, F, D, Zicsr, Zifencei e opcionalmente a extensão C, resultando assim em por exemplo RV64GC, onde o “G” significa “General Purpose”.

O núcleo descrito ao longo deste trabalho apenas implementa o conjunto base RV32I.

Subset	Name	Implies
Base ISA		
Integer Reduced Integer	I E	
Standard Unprivileged Extensions		
Integer Multiplication and Division	M	
Atomics	A	
Single-Precision Floating-Point	F	Zicsr
Double-Precision Floating-Point	D	F
Control and Status Register Access	Zicsr	
Instruction-Fetch Fence	Zifencei	
General	G	IMADZifencei
Quad-Precision Floating-Point	Q	D
Decimal Floating-Point	L	
16-bit Compressed Instructions	C	
Bit Manipulation	B	
Dynamic Languages	J	
Transactional Memory	T	
Packed-SIMD Extensions	P	
Vector Extensions	V	
User-Level Interrupts	N	
Misaligned Atomics	Zam	A
Total Store Ordering	Ztso	
Standard Supervisor-Level Extensions		
Supervisor-level extension “def”	Sdef	
Standard Hypervisor-Level Extensions		
Hypervisor-level extension “ghi”	Hghi	
Standard Machine-Level Extensions		
Machine-level extension “jkl”	Zxmjkl	
Non-Standard Extensions		
Non-standard extension “mno”	Xmno	

Figura 4.1 - Modularidade da ISA. Fonte [21].

### B. Registradores

RV32I, RV64I e RV128I definem 32 registradores internos para uso em operações com inteiros. RV32E, RV64E e RV128E são alternativas que diminuem este número para apenas 16. Caso a extensão F (*Single-Precision Floating Point*) seja implementada, a ISA define outros 32 registradores para serem utilizados em operações de ponto flutuante.

O registrador x0, denominado “zero”, sempre possui valor zero, e este não pode ser modificado. Esta característica é intencional e possibilita que diversas operações, como por exemplo, *jumps* não relativos ao PC e a modificação dos bits menos significativos em algum registrador sejam executadas sem a inserção de instruções complementares.

O manual do programador para RISC-V apresenta todas as denominações, mostrado na Figura 4.2, e descreve em que ocasiões cada registrador interno deve ser utilizado. Este atributo, entretanto, não é restritivo, apenas uma recomendação ou *guideline* para que códigos sejam mais facilmente entendidos por outros desenvolvedores de forma que sejam mais fácil de serem reutilizados e melhorados.

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

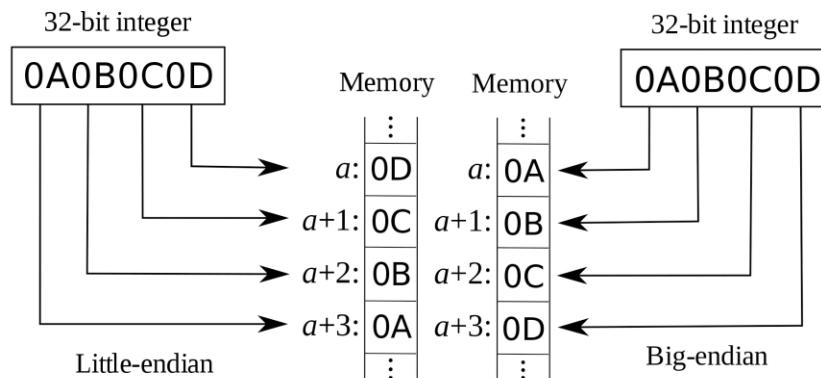
Figura 4.2 - Registradores Internos. Fonte [21].

### C. Endianess

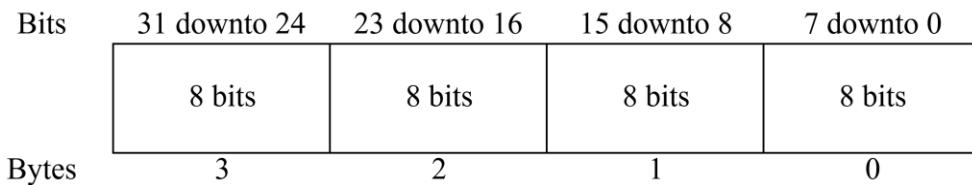
Embora ISAs RISC como POWER e MIPS tenham historicamente adotado arquiteturas *Big-Endian*, RISC-V optou por seguir o sucesso de x86 e é por padrão definida como *Little-Endian*, onde bytes menos significativos possuem os mais

baixos endereços na memória. A ISA, entretanto, não restringe uma possível implementação *Big-Endian* ou *Bi-Endian*.

A diferença entre *Big-Endian* e *Little-Endian* pode ser visto na Figura 4.3. A Figura 4.4 mostra como a ordenação dos bits e bytes é feita em uma palavra de 32 bits.

Figura 4.3 - *Endianness*. Fonte: R. S. Shaw.

### 32 Bits Little-Endian Word

Figura 4.4 – *Little-Endian Word*. Fonte: Autor.

#### D. Codificação das Instruções

A ISA define 6 tipos de codificações denominados Register/Register, Immediate, Store, Branch, Upper Immediate e Jump. Isto é necessário pois o conjunto base possui tamanho de instrução fixo de 32 bits.

Como pode ser visto na Figura 4.5, o registrador de destino, rd, assim como os registradores de origem, rs1 e rs2, sempre aparecem no mesmo lugar quando presentes. Instruções são agrupadas em diferentes tipos de codificação, como mostrado na Figura 4.6, dependendo de quais argumentos são necessários para o tipo de operação.

31	30	25 24	21	20	19	15 14	12 11	8	7	6	0	
		funct7		rs2		rs1	funct3		rd		opcode	R-type
		imm[11:0]				rs1	funct3		rd		opcode	I-type
		imm[11:5]		rs2		rs1	funct3	imm[4:0]			opcode	S-type
		imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]		opcode	B-type
		imm[31:12]							rd		opcode	U-type
		imm[20]	imm[10:1]	imm[11]	imm[19:12]				rd		opcode	J-type

Figura 4.5 - Codificação das instruções. Fonte [21].

inst[4:2]	000	001	010	011	100	101	110	111 (> 32b)
inst[6:5]								
00	LOAD	LOAD-FP	<i>custom-0</i>	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	<i>custom-1</i>	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	<i>reserved</i>	<i>custom-2/rv128</i>	48b
11	BRANCH	JALR	<i>reserved</i>	JAL	SYSTEM	<i>reserved</i>	<i>custom-3/rv128</i>	$\geq 80b$

Figura 4.6 - Subgrupos de Instruções. Fonte [21].

#### E. Imediatos

RISC-V, assim como SPARC, possui instruções com *offsets* de 12 bits e instruções com *upper immediates* de 20 bits. Desta forma é possível manter todas as instruções com tamanho fixo e operações que precisam modificar todos os 32 bits de um registrador são realizadas em uma série de duas instruções onde a primeira modifica os 20 bits mais significativos e a segunda os 12 menos significativos.

O exemplo mais comum para este tipo de operação é dado por LUI (*Load Upper Immediate*) e ADDI (*Add Immediate*) utilizando x0 como registrador de origem rs1, o que significa dizer que o valor do imediato será somado a 0 e então salvo em rd.

#### F. Inteiros

Instruções para computação com inteiros são codificadas como “R-type” quando trabalham apenas com registradores internos, ou “I-type” caso utilizem imediatos.

A ISA define operações lógicas através das instruções AND, OR e XOR, que executam operações *bitwise*.

As operações aritméticas disponíveis são ADD e SUB, e ambas sempre assumem que os valores utilizados são

sinalizados em complemento de 2.

Há também instruções para *shifts* lógicos e aritméticos SLL, SRL e SRA. O primeiro “S” denomina *Shift*, a segunda letra define a direção e pode ser “L”, *Left* ou “R”, *Right*. A terceira letra define se o *shift* será lógico “L” ou aritmético “R”.

*Shifts* aritméticos respeitam o sinal do imediato, que é codificado em complemento de 2.

*Shifts* lógicos simplesmente estendem o valor com zeros.

SLT e SLTU são instruções de comparação que modificam o valor do registrador de destino, rd, para 1 caso rs1 seja menor que rs2/imm.

O modificador U, presente no final de algumas instruções, indica que estas devem ser executadas interpretando os valores como *unsigned*.

O modificador I, também presente no final de algumas instruções, indica que estas efetuam operações entre rs1 e um imediato. Uma instrução de subtração com imediato, possivelmente SUBI, não é necessária pois os imediatos são codificados em complemento de 2 e podem ser números negativos.

A Figura 4.7 apresenta a listagem de todas as instruções lógicas e aritméticas.

imm[11:0]	rs1	000	rd	0010011	ADDI
imm[11:0]	rs1	010	rd	0010011	SLTI
imm[11:0]	rs1	011	rd	0010011	SLTIU
imm[11:0]	rs1	100	rd	0010011	XORI
imm[11:0]	rs1	110	rd	0010011	ORI
imm[11:0]	rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	SLLI
0000000	shamt	rs1	101	rd	SRLI
0100000	shamt	rs1	101	rd	SRAI
0000000	rs2	rs1	000	rd	ADD
0100000	rs2	rs1	000	rd	SUB
0000000	rs2	rs1	001	rd	SLL
0000000	rs2	rs1	010	rd	SLT
0000000	rs2	rs1	011	rd	SLTU
0000000	rs2	rs1	100	rd	XOR
0000000	rs2	rs1	101	rd	SRL
0100000	rs2	rs1	101	rd	SRA
0000000	rs2	rs1	110	rd	OR
0000000	rs2	rs1	111	rd	AND

Figura 4.7 - Operações Lógicas e Aritméticas. Fonte [21].

RISC-V fornece dois tipos de instruções de transferência de controle, *jumps* não condicionais e *branches* condicionais.

A instrução JAL, *Jump and Link*, efetua um *jump* não condicional relativo ao PC e possui um imediato superior de 20 bits. JAL guarda o endereço de retorno, PC+4, num registrador de destino, normalmente x1.

JALR, *Jump and Link Register*, efetua um *jump* não condicional relativo a um registrador de origem, e possui um imediato inferior de 12 bits. Assim como em JAL, JALR guarda o endereço de retorno em um registrador de destino.

Caso um *jump* absoluto seja necessário, o registrador x0 pode ser utilizado como argumento de JALR.

Qualquer endereço existente em 32 bits é alcançável a partir

de uma combinação de AUIPC ou LUI com JALR, onde a primeira instrução modifica os 20 bits mais significativos do destino do PC e a segunda os 12 restantes.

Instruções de *branch* possuem imediatos de 12 bits relativos ao PC, e então efetuam a transferência de controle caso a condição testada seja verdadeira. As possíveis condições são “igual a”, “não igual a”, “menor que”, “maior ou igual a”, e podem ser feitas interpretando os números como *signed* ou *unsigned*, denominadas BEQ, BNE, BLT, BGE, BLTU e BGEU respectivamente.

A Figura 4.8 apresenta a listagem de todas as instruções de desvio de controle de fluxo de programa.

imm[20 10:1 11 19:12]	rd	1101111	JAL			
imm[11:0]	rs1	000	JALR			
imm[31:12]	rd	0110111	LUI			
imm[31:12]	rd	0010111	AUIPC			
imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	BEQ
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011	BNE
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011	BLT
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011	BGE
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	1100011	BLTU
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	1100011	BGEU

Figura 4.8 - Instruções de Desvio de Controle. Fonte [21].

#### G. Acesso a Memória

RISC-V é uma ISA de arquitetura *Load-Store*, que por natureza significa que apenas algumas instruções podem acessar memória, e qualquer outra operação deve ser executada entre registradores internos.

Dentre estas instruções que podem acessar memória, há

operações de *Load*, que carregam um registrador interno com algum valor com origem na memória e instruções de *Store*, que guardam na memória algum valor com origem em algum registrador interno.

Operações com memória podem ser executadas em diversos formatos, definidos como *Byte* (8 bits), *Half-Word* (16 bits), *Word* (32 bits) ou *Double-Word* (64 bits). Entretanto, todas as

operações de memória são endereçadas a nível de Byte.

Instruções de *Load*, quando estão movendo dados de tamanho menor que seus registradores, como por exemplo LB, *Load Byte*, carregam os 8 bits menos significativos de rd com o valor originado na memória, e então completa os bits restantes com 0.

Instruções de *Store* sempre guardam os bits menos significativos provenientes dos registradores internos, ignorando aqueles que estão além do escopo da instrução. SH, *Store Halfword*, por exemplo, guarda os 16 bits menos significativos do registrador de origem.

A ISA recomenda que apenas operações alinhadas sejam efetuadas. Operações não alinhadas, como por exemplo escrever uma *Half-Word* no centro de uma *Word*, não são proibidas, porém requerem implementações mais complexas e potencialmente mais lentas.

Intuitivamente, todas as instruções de acesso a memória possuem nomes que iniciam com *Load* ou *Store*. Este então é seguido do formato a ser utilizado, resultando nas instruções LB, LBU, LH, LHU, LW, SB, SH, SW para RV32I/E.

A Figura 4.9 apresenta a listagem das instruções de acesso a memória.

imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU
imm[11:5]	rs2	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	010	imm[4:0]	0100011	SW

Figura 4.9 - Instruções de Acesso à Memória. Fonte [21].

## V. ORGANIZAÇÃO DE REFERÊNCIA

A implementação descrita neste trabalho toma como inspiração o modelo apresentado por David A. Patterson e John L. Hennessy em [50], onde os autores didaticamente evoluem de um *datapath* de ciclo único a um *datapath* com *pipeline* de 5 estágios.

A Figura 5.1 mostra o modelo de [50], que por motivos didáticos não implementa todo o conjunto base de instruções, apenas um conjunto mínimo para demonstração de como a arquitetura pode ser organizada.

As instruções incluídas no conjunto para demonstração são: LD, SD, ADD, SUB, AND, OR e BEQ.

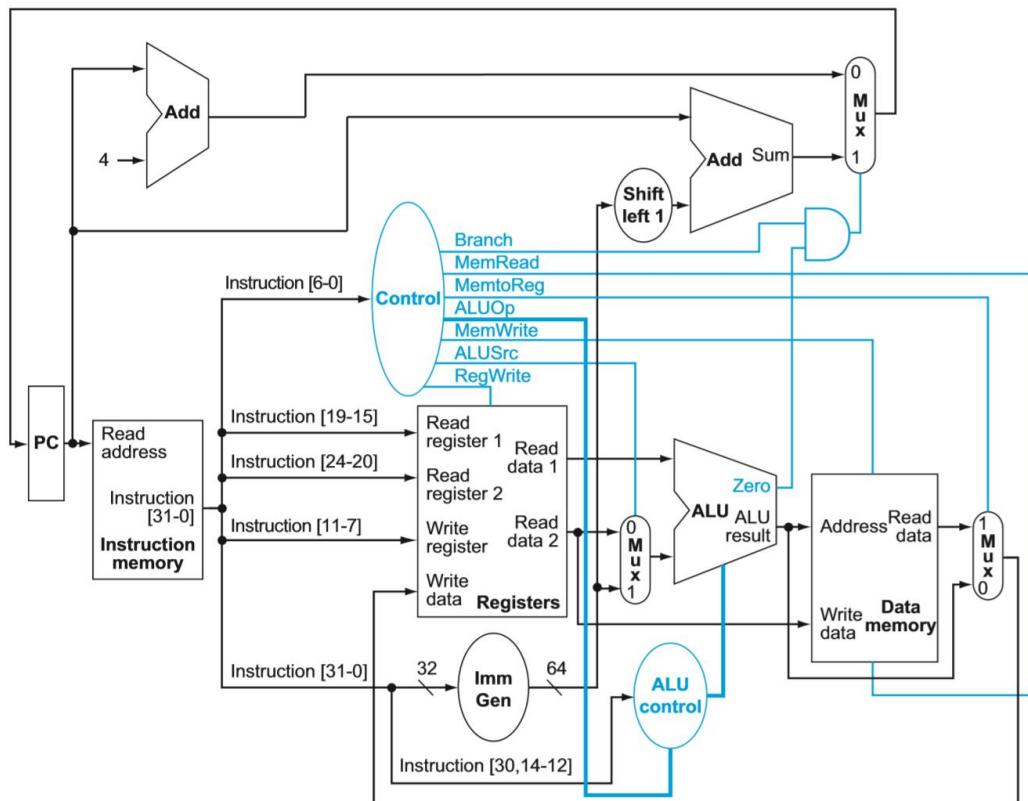


Figura 5.1 - Datapath para ciclo único. Fonte [50].

Com este *datapath*, a unidade de controle nada mais é que um decodificador e ela não possui máquinas de estado, apenas lógica combinatória.

O contador de programa (PC) recebe o endereço da próxima instrução (PC+4) na execução de qualquer instrução com exceção de BEQ, que retorna PC+4 apenas caso sua condição de teste (igualdade) não seja satisfeita. Caso a condição seja satisfeita, o sinal denominado “zero” aciona a porta AND junto do sinal “branch”, e então um multiplexador altera a entrada do PC para o endereço de destino do desvio, que é calculado na execução da própria instrução.

A memória de programa recebe do PC o endereço da instrução atual, acarretando a mudança de sua saída para a respectiva instrução, ainda codificada.

Esta instrução, então, é repassada ao Register File, ao Immediate Generator e ao controlador, orquestrando a execução da instrução em questão.

Como os valores rd, rs1, rs2, funct3, funct7 e opcode estão sempre codificados nos mesmos bits quando presentes em uma instrução, apenas o Immediate Generator precisa receber a instrução inteira, visto que imediatos podem ser codificados de diversas maneiras dependendo de qual instrução esteja sendo executada.

O controlador utiliza os 7 primeiros bits da instrução para gerar os sinais de controle restantes, sendo eles Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc e Regwrite.

Como já mencionado, Branch serve para escrever o endereço de destino de BEQ no PC.

MemRead controla a leitura da memória de dados.

MemtoReg controla o multiplexador de escrita ao Register File, alternando entre a saída da ALU e a saída da memória de dados.

ALUOp define qual operação a ALU executará.

MemWrite faz com que os dados na entrada da memória de dados sejam escritos no endereço apontado pela ALU.

ALUSrc controla o multiplexador que alterna uma das entradas da ALU entre rs2 e o imediato.

RegWrite controla a escrita ao Register File.

Com isso temos um processador com um fluxo de dados simples, capaz de executar o subconjunto determinado pelos autores como suficiente para uma demonstração puramente didática. Entretanto, é possível realizar diversas mudanças no projeto para que o mesmo seja capaz de finalizar instruções mais rapidamente.

O *datapath* de ciclo único limita a frequência máxima do *clock* à instrução que utiliza o maior número de componentes lógicos em série, neste caso LD. Implementações de ciclo único possuem as vantagens de ocuparem pouco espaço em um possível FPGA ou ASIC e de serem relativamente simples de se implementar. Entretanto, o caminho crítico torna-se longo e consequentemente os ciclos de *clock* também devem ser longos.

A solução é dividir o fluxo de dados em estágios, onde cada estágio é responsável por uma determinada operação, e ao término de todos os estágios uma instrução tenha sido inteiramente finalizada. Esta técnica é chamada de *instruction pipelining* e é mostrada na Figura 5.2. Atualmente, todos os processadores comerciais de alto desempenho são projetados desta forma.

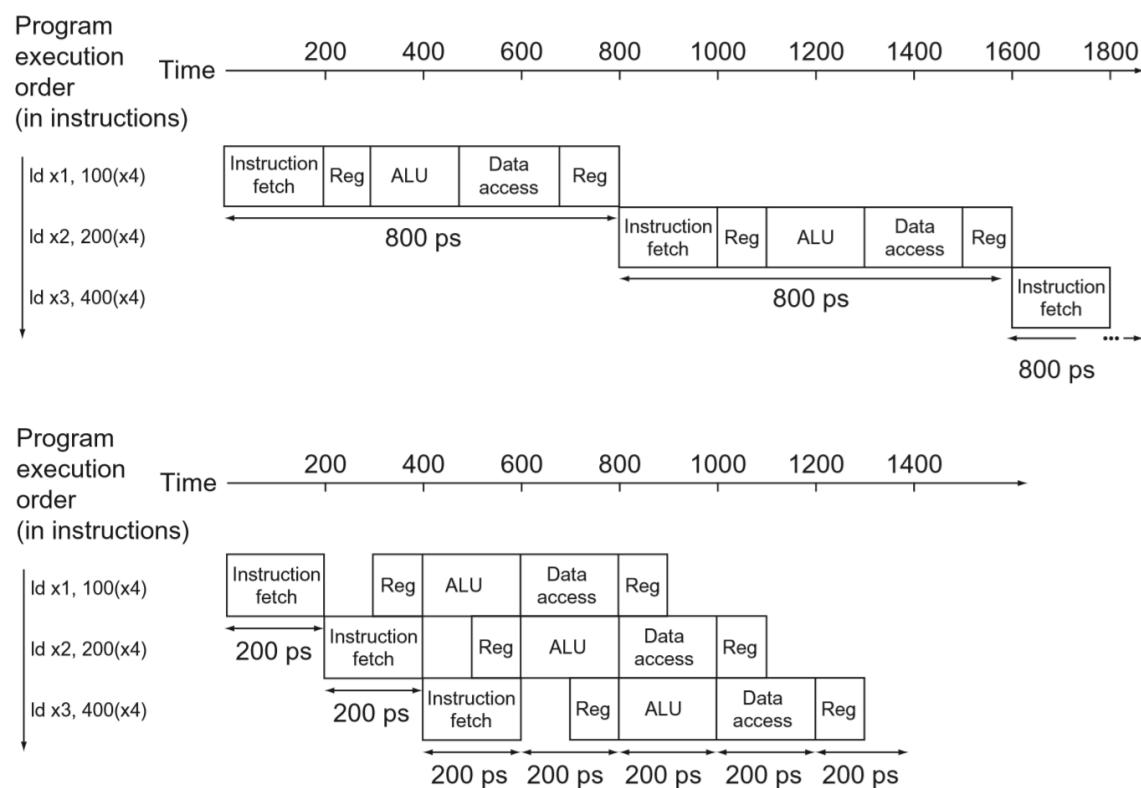


Figura 5.2 - Vantagem em Tempo de Execução com *Pipeline*. Fonte [50].

O objetivo é diminuir o tempo total de execução incluindo registradores entre os componentes lógicos do fluxo de dados, desta forma, diferentes instruções estarão em andamento ao mesmo tempo, cada uma em um estágio, onde há uma instrução

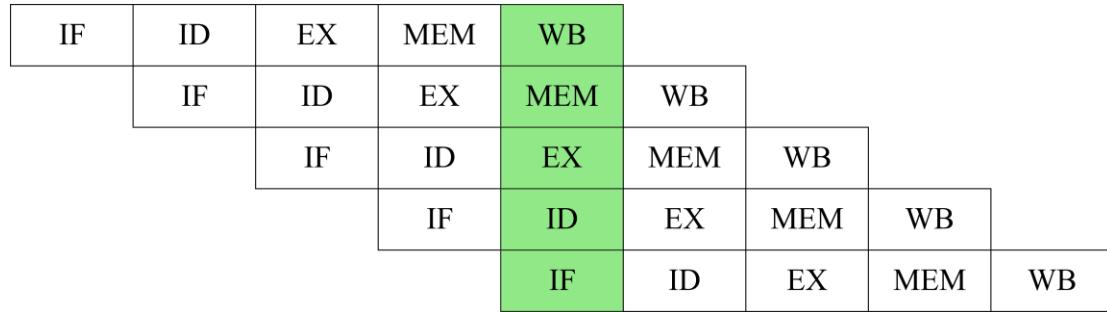


Figura 5.3 - Pipeline filling. Fonte: Autor.

É possível ver que, após a inicialização do processador, a *pipeline* estará em estado de *filling*, onde a mesma está a cada ciclo de *clock* preenchendo os correspondentes estágios com diferentes instruções. Quando a primeira instrução alcança o estágio de Write Back, todos os outros estágios já se encontram com instruções posteriores em andamento.

sendo finalizada a cada ciclo de *clock*. A Figura 5.3 demonstra uma *pipeline* clássica de 5 estágios.

David A. Patterson e John L. Hennessy apresentam os possíveis estágios no *datapath* de ciclo único conforme mostrado na Figura 5.4.

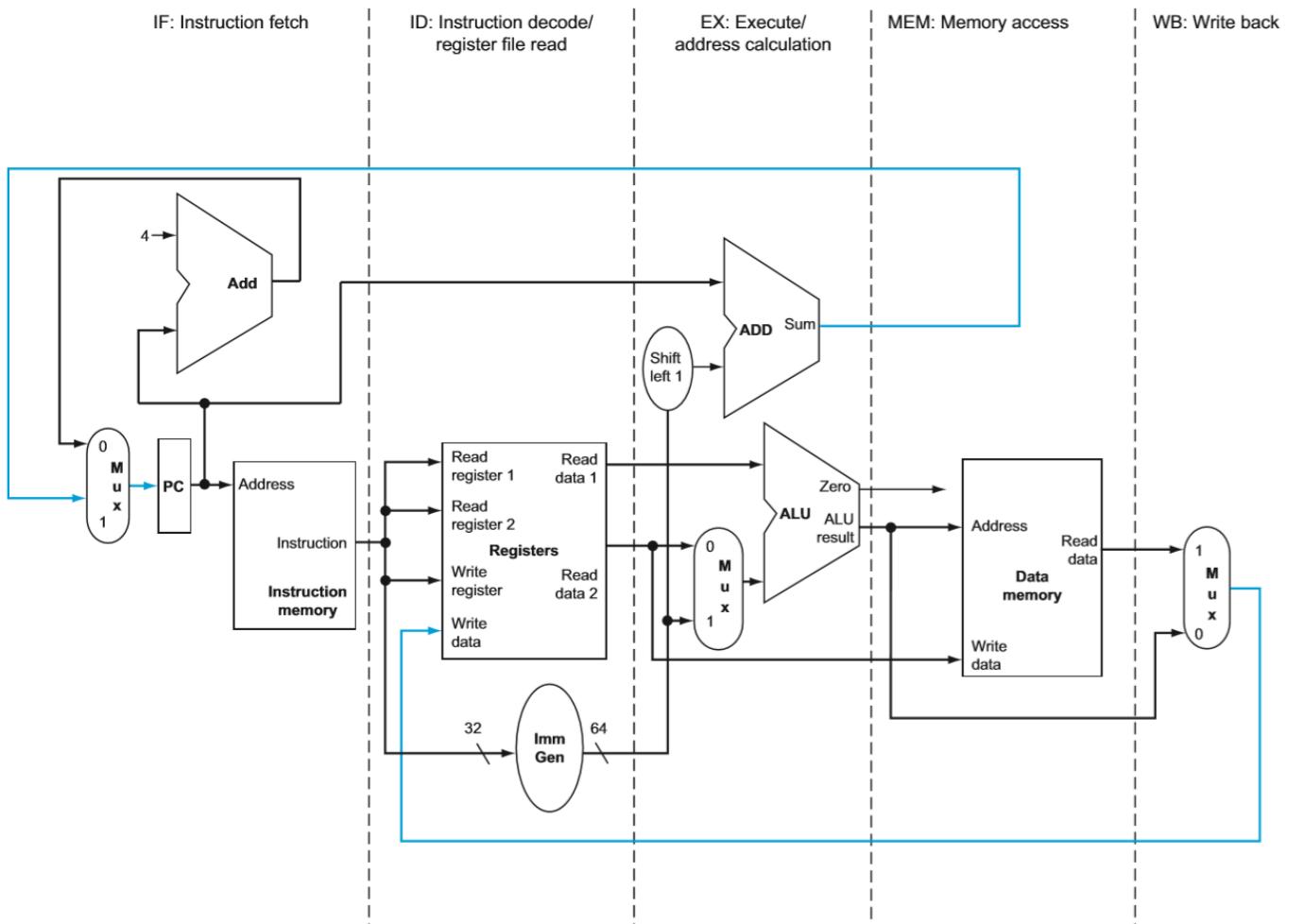


Figura 5.4 - Sugestão para divisão do datapath. Fonte [50].

O modelo proposto com 5 estágios, denominados de Instruction Fetch, Instruction Decode, Execute, Memory Access e Write Back é conhecido como *Classic RISC Pipeline* e foi utilizado por diversos processadores incluindo implementações das ISAs MIPS e SPARC.

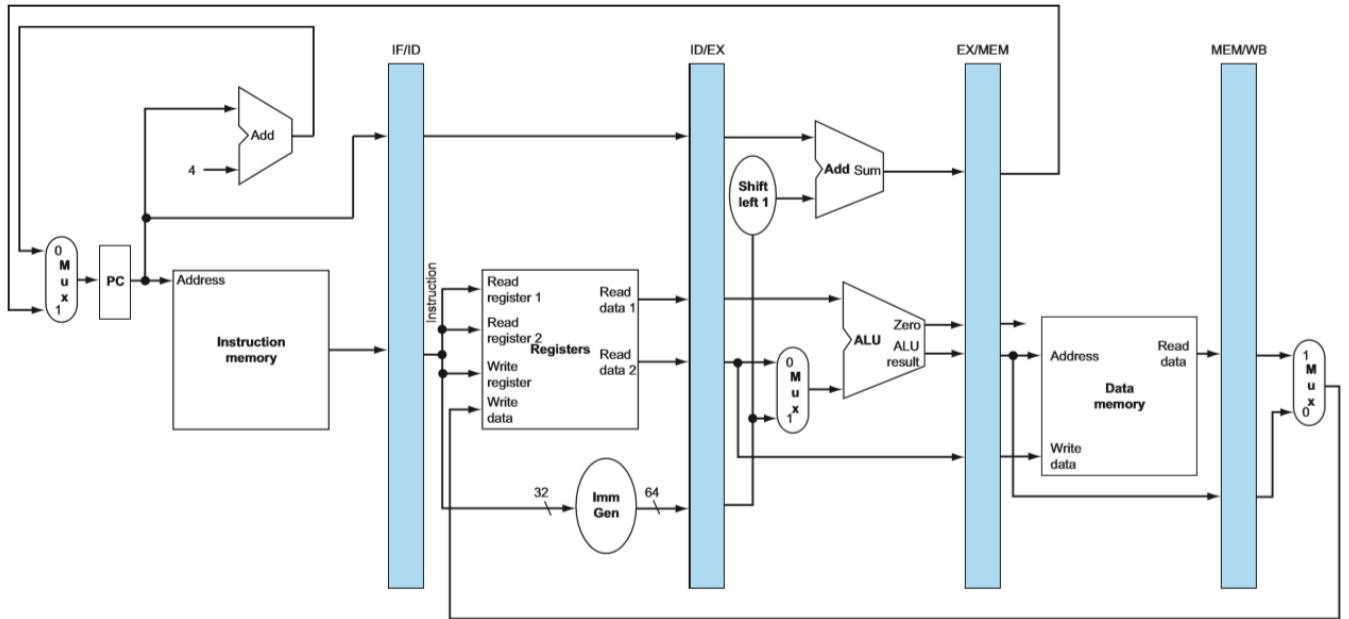


Figura 5.5 - *Datapath* preliminar com registradores de *pipeline*. Fonte [50].

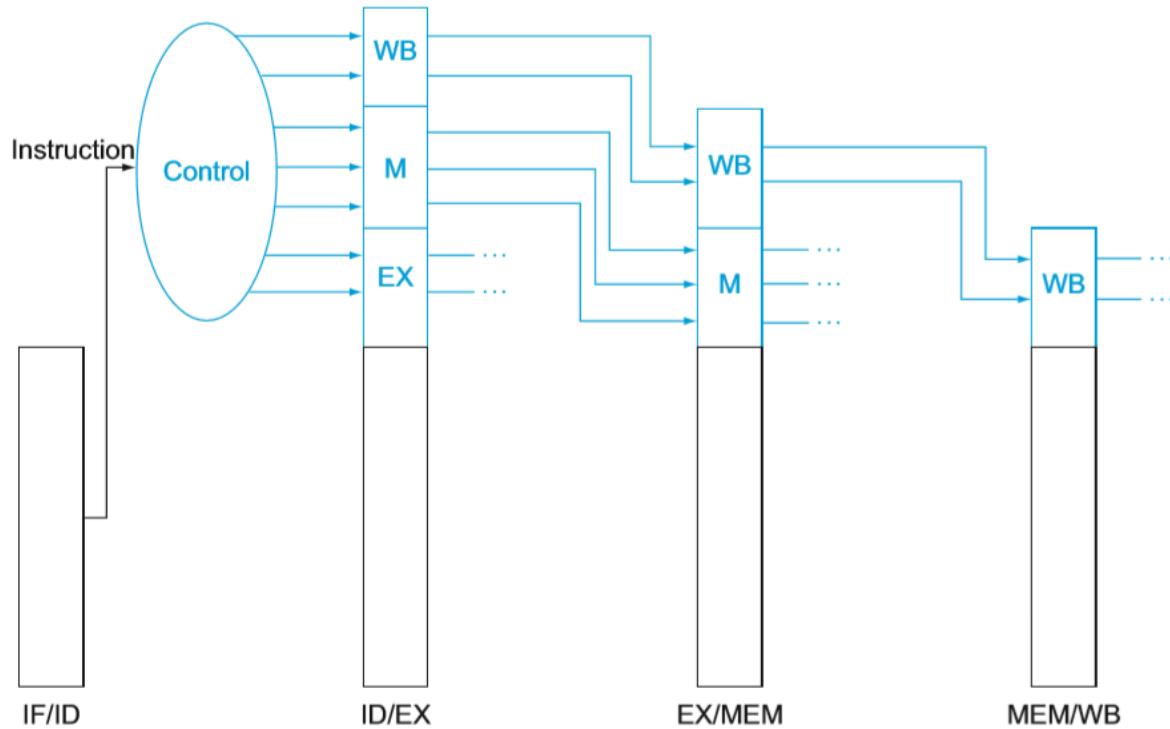


Figura 5.6 - Sinais de controle na *pipeline*. Fonte [50].

A *pipeline* pode ser representada como mostrado na Figura 5.5, com a inclusão dos registradores entre os estágios.

Para que seja possível manter o controlador como uma unidade lógica simples sem máquinas de estado, os sinais de controle também navegam o *datapath* através de registradores de *pipeline*, como mostrado na Figura 5.6.

Desta forma, sinais que não pertencem a um determinado estágio são passados ao próximo estágio até que alcancem seu destino.

Tem-se então o processador com os registradores de *pipeline* e seu controlador, mostrado na Figura 5.7.

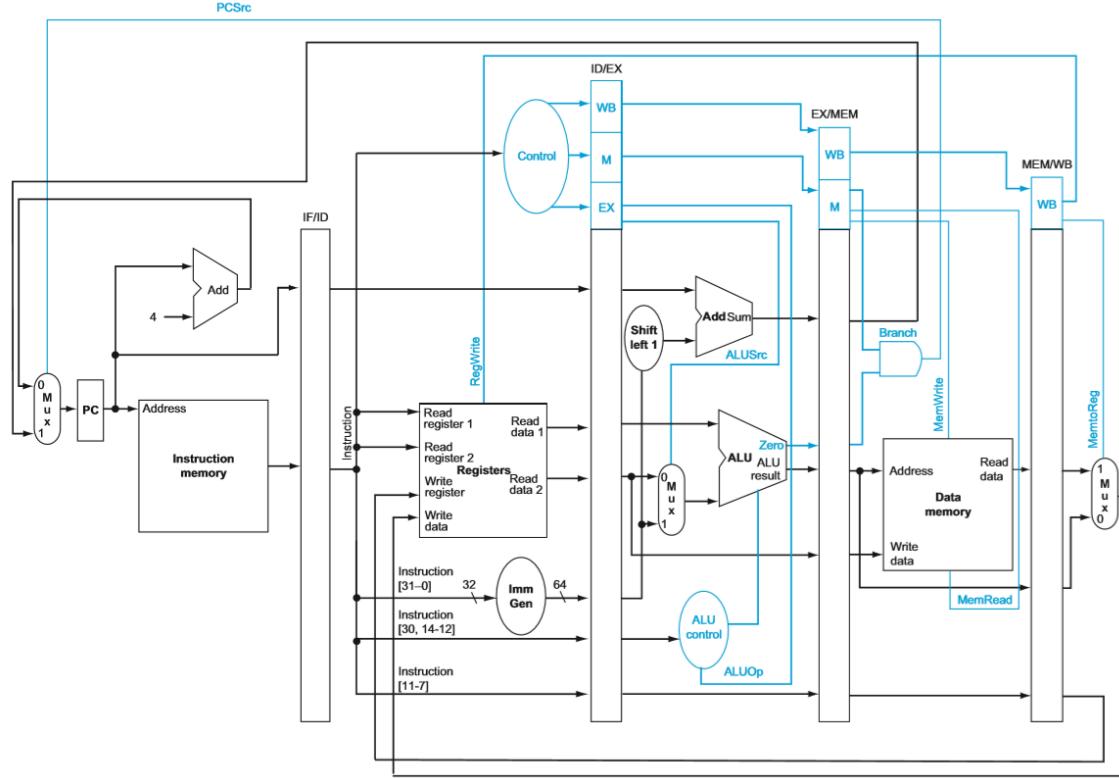


Figura 5.7 - *Datapath* com registradores de *pipeline* e sinais de controle. Fonte [50].

A inclusão da *pipeline* desta maneira, entretanto, causa diversos problemas denominados *pipeline hazards*, que podem ser divididos entre *data hazards*, *structural hazards* e *control hazards*.

*Data Hazards* ocorrem quando um determinado estágio precisa utilizar um valor que se encontra num estágio posterior da *pipeline*. Isto ocorre caso o valor rs1 ou rs2 de uma determinada instrução seja igual aos valores rd de instruções anteriores antes desta ter alcançado seu estágio de Write Back.

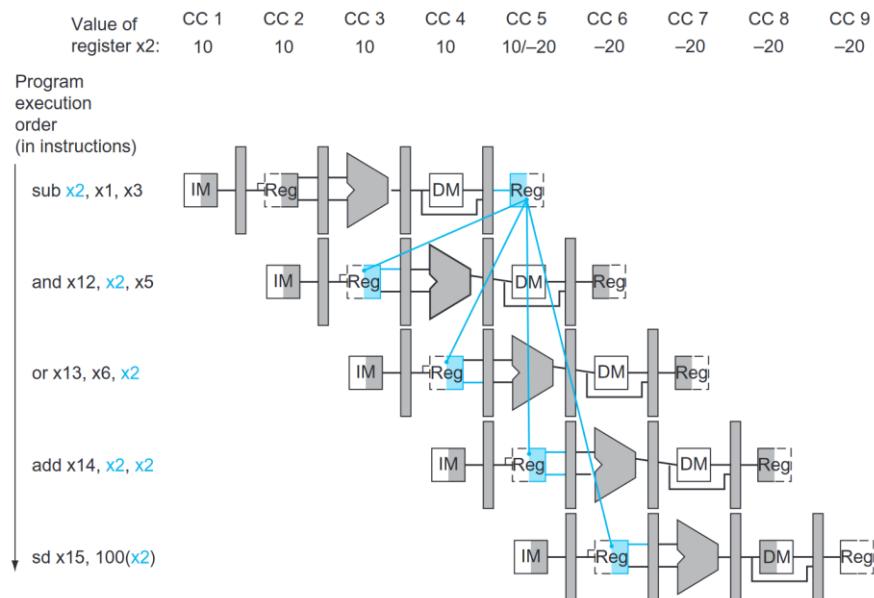


Figura 5.8 - *Data Hazards*. Fonte [50].

Na Figura 5.8, esta dependência é representada por setas que possuem sua direção apontando para a esquerda.

Uma possível solução seria inserir instruções NOPs entre as instruções com dependência, como mostrado na tabela 5.1, de forma que o estágio de Instruction Decode da instrução

dependente ocorra junto ou depois do estágio de Write Back da instrução que altera o valor do registrador. Esta solução, entretanto, diminui o desempenho do processador devido a inserção de operações nulas na *pipeline*.

IF	ID	EX	MEM	WB
SUB x2, x1, x3				
STALL	SUB x2, x1, x3			
STALL	STALL	SUB x2, x1, x3		
AND x12, x2, x5	STALL	STALL	SUB x2, x1, x3	
OR x13, x6, x2	AND x12, x2, x5	STALL	STALL	SUB x2, x1, x3

Tabela 5.1 - Pipeline stalls. Fonte: Autor.

Alternativamente, levando em consideração que os valores já foram computados e que é preciso apenas repassá-los ao estágio de Execute, onde eles de fato serão utilizados, é possível adicionar um elemento responsável por comparar os valores de

rs1 e rs2 com os valores de rd de estágios posteriores e multiplexadores para repassar os dados, como mostrado na Figura 5.9.

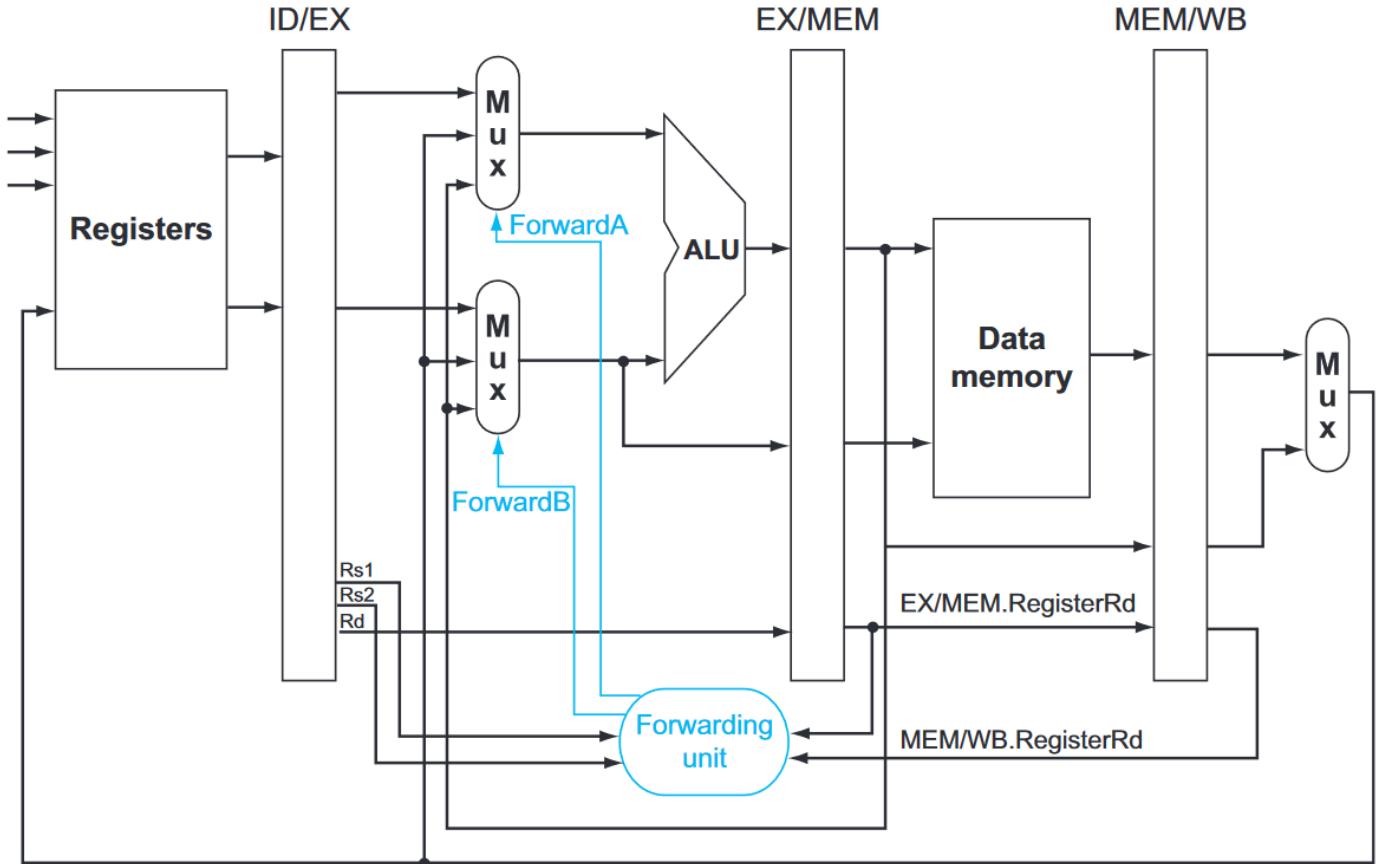


Figura 5.9 - Datapath com registradores de pipeline e Forwarding Unit. Fonte [50].

O datapath apresentado, entretanto, ainda não resolve *data hazards* relacionados a instruções de acesso à memória.

Os autores deixam a possível modificação como exercício para os leitores, e então adicionam uma unidade responsável por inserir NOPs caso haja *data hazard* em instruções de *Load*

ou *Store*. A solução, entretanto, diminui a performance do processador e apenas é recomendada quando não se pode garantir leitura assíncrona da memória de dados. Esta modificação é mostrada na Figura 5.10.

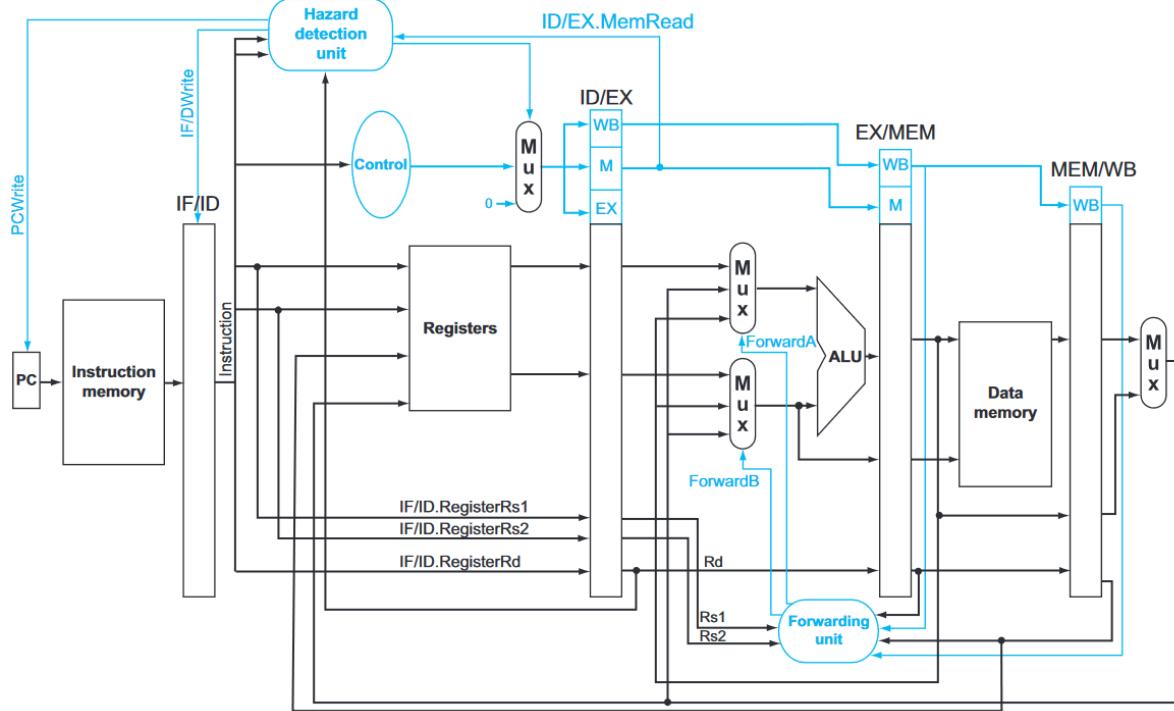


Figura 5.10 - Datapath com registradores de pipeline, Forwarding Unit e Hazard Detection Unit. Fonte [50].

Para finalizar, os autores modificam o *datapath* para que o mesmo possa realizar decisões de desvio de controle durante o estágio de Instruction Decode, diminuindo assim o número de instruções que devem ser descartadas caso haja um desvio. O comportamento padrão é assumir que as instruções de *branch* não são pegas, neste caso, há apenas BEQ e pode-se adicionar

um comparador que responde caso a diferença entre os valores seja nula, acionando um sinal de *flush* que limpa os registradores IF/ID e ID/EX, como mostrado na Figura 5.11.

O resultado alcançado é um processador *single-issue* com execução *in-order* e *pipeline* de 5 estágios.

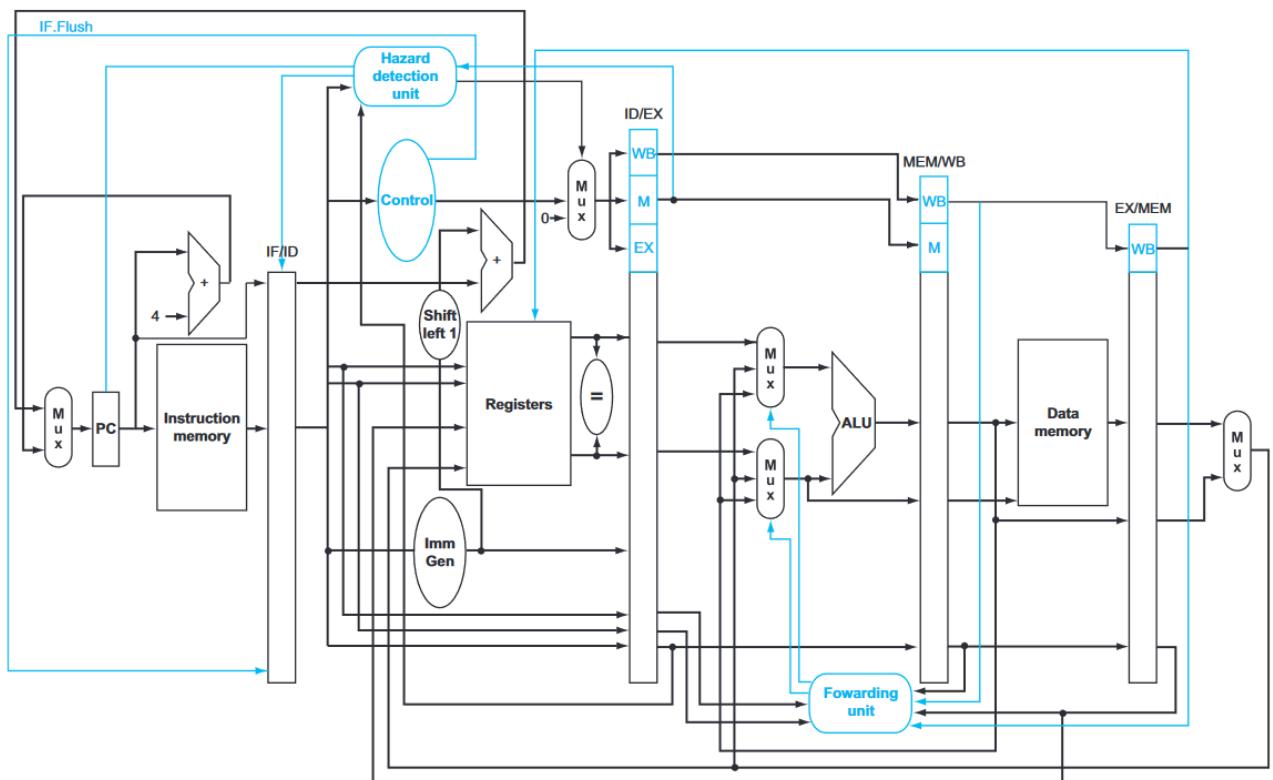


Figura 5.11 - Datapath final. Fonte [50].

## VI. POSSÍVEIS APRIMORAMENTOS E COMPARAÇÃO ENTRE NÚCLEOS

Ainda existem diversas modificações que podem ser feitas para melhorar o desempenho do projeto, como por exemplo a inclusão de paralelismo estrutural tornando possível que mais de uma instrução seja computada ao mesmo tempo em um

determinado estágio. Processadores que conseguem finalizar mais de uma instrução por ciclo de *clock* em um único núcleo são denominados *superscalars*, e essencialmente qualquer processador moderno de uso geral com alto desempenho implementa paralelismo de diversas maneiras diferentes.

A Figura 6.1 demonstra um diagrama de *superscalar pipeline* onde cada estágio funcional é dividido em duas etapas.

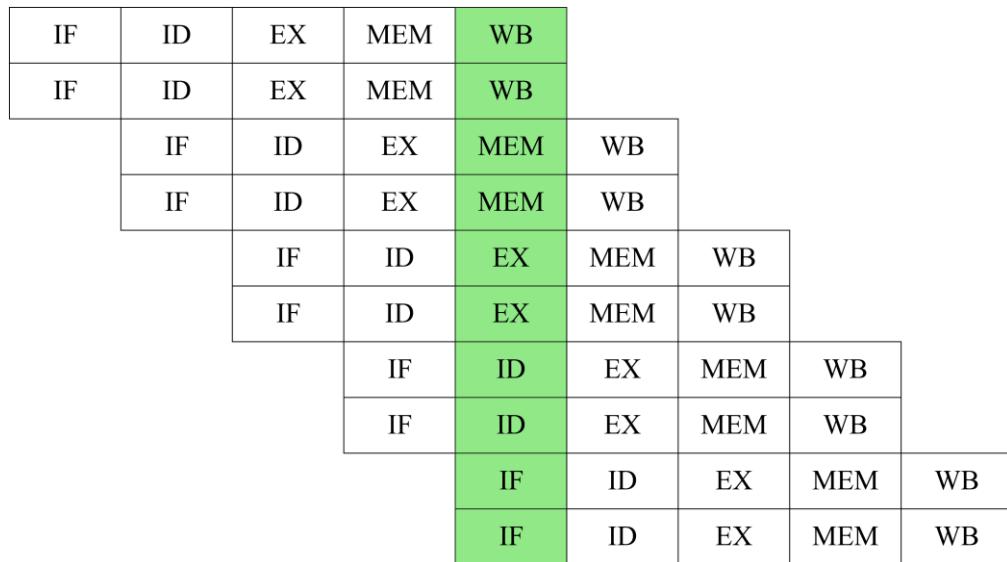


Figura 6.1 – Exemplo de *pipeline* em processador *superscalar*. Fonte: Autor.

Outro estilo de design que normalmente é implementado junto ao paralelismo de estruturas é a execução fora de ordem de instruções, ou *out of order execution*, que adiciona uma etapa de *register renaming* onde as instruções são reorganizadas de

maneira com que seja possível utilizar o maior número de elementos estruturais durante cada ciclo de execução, em outras palavras impedindo que *pipeline stalls* ocorram.

## Cortex-A77: Microarchitecture overview

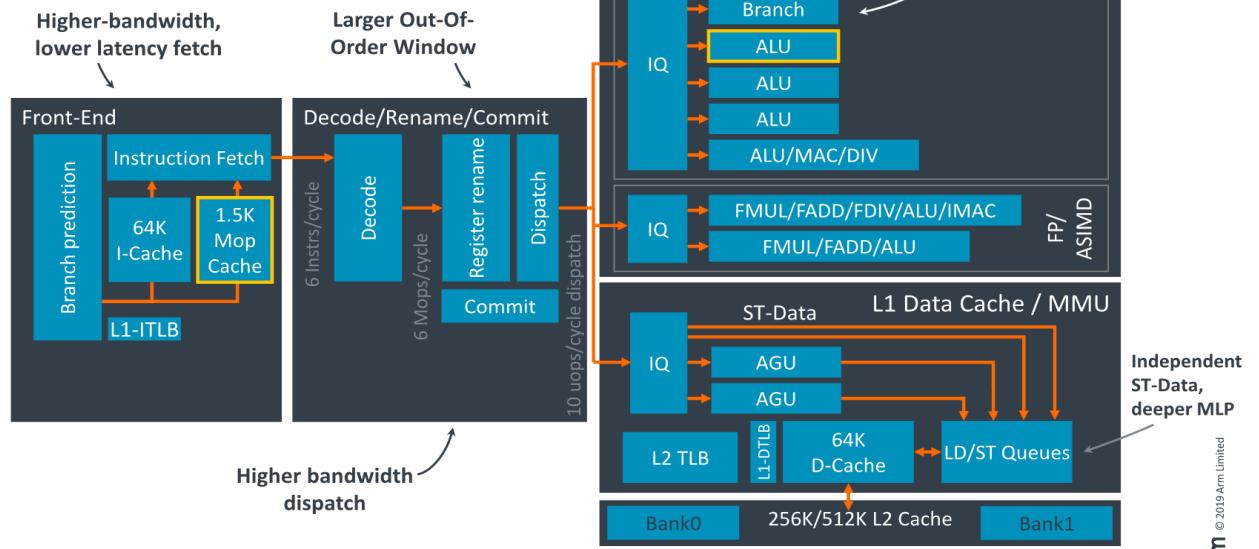


Figura 6.2 - Cortex-A77. Fonte: ARM Tech Day.

Na Figura 6.2 temos um esquemático de alto nível do ARM Cortex-A77, demonstrando uma organização que efetua o *fetch* de 6 instruções por ciclo de *clock* e que despacha 10 *micro-operations* às unidades de execução. Como ARMv8 é uma ISA com um elevado número de instruções, elas são divididas em operações menores denominadas *micro-operations*.

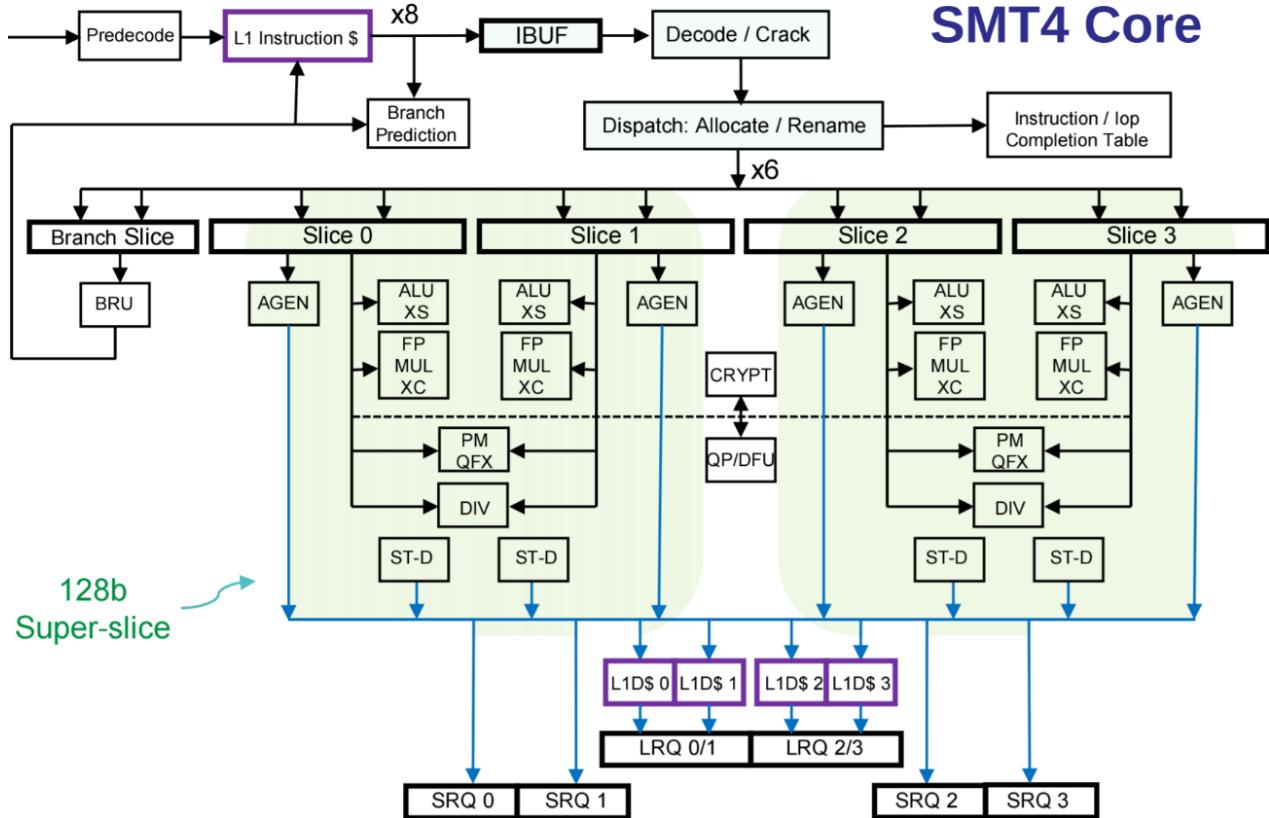


Figura 6.3 - IBM POWER9 SMT4. Fonte: IBM Power9 Features and Specifications.

Dentre os núcleos RISC-V desenvolvidos pela própria UC Berkeley, o Sodor, mostrado na Figura 6.6, é a opção simples e didática utilizada nas disciplinas e laboratórios de graduação relacionados a arquitetura de computadores [51].

O núcleo é escrito em Chisel, uma linguagem de descrição de hardware de alto nível embarcada em Scala, assim como Rocket e BOOM, os outros dois núcleos desenvolvidos pela mesma universidade.

O núcleo Rocket, mostrado na Figura 6.4, possui uma *pipeline* de 5 estágios, suporta a instruções com *floating point* e faz parte do Rocket Chip Generator, mostrado na Figura 6.5, plataforma também escrita em Chisel capaz de gerar SoCs RISC-V que possuem núcleos, interface para coprocessadores, memórias e controladores para interfaces AXI [52].

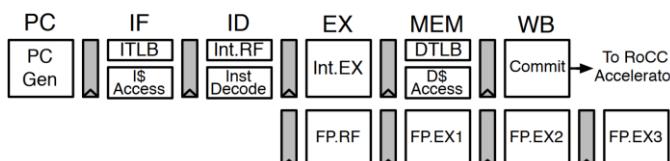


Figura 6.4 - Estágios de *Pipeline* do Núcleo Rocket. Fonte: [52].

É interessante observar que núcleos destinados a supercomputadores possuem organizações similares, como é o caso do POWER9, da IBM. Na Figura 6.3 é possível ver que ele efetua o *fetch* de 8 instruções por ciclo e decodifica 6, renomeando-as para que sejam distribuídas às unidades de execução.

## SMT4 Core

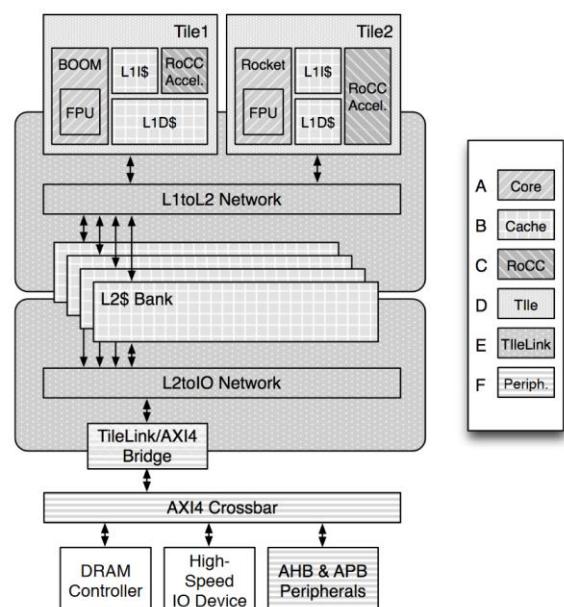


Figura 6.5 - Exemplo de SoC gerado pelo Rocket Chip Generator. Fonte: [52]

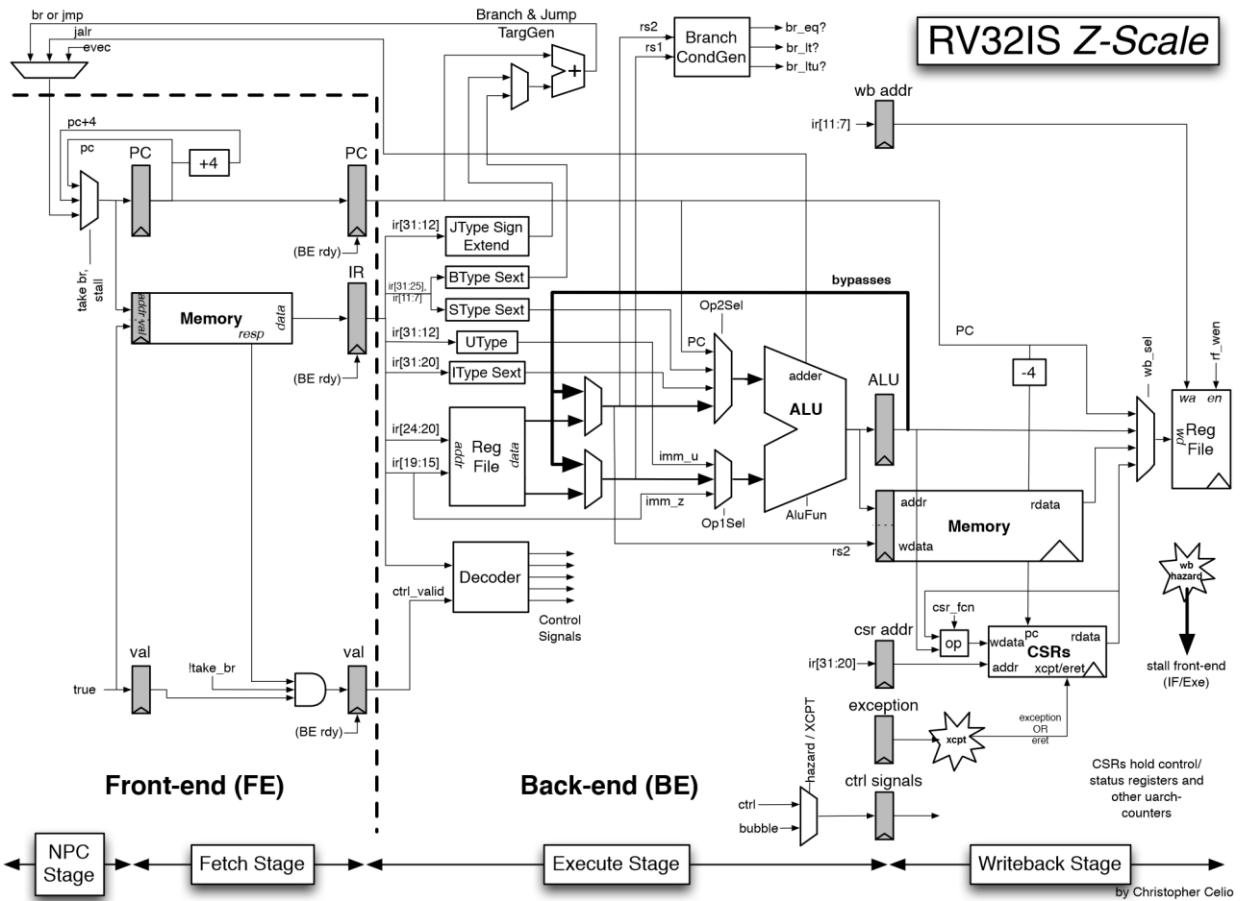


Figura 6.6 – Processador Sodor Z-scale. Fonte: [51]

O BOOM, que possui sua *pipeline* demonstrada na Figura 6.9, é o núcleo com execução fora de ordem da família, seu objetivo principal era servir como ecossistema de pesquisas relacionadas a processadores de alto desempenho e as devidas escolhas de design que tornam a execução fora de ordem possível. O núcleo possui suporte a diversas extensões, sendo um núcleo RV64G com memória virtual capaz de rodar Linux ou outros sistemas operacionais.

O projeto fez parte da tese de doutorado de Christopher Celio [52], aluno da própria UC Berkeley, e inicialmente o objetivo era conseguir uma performance equiparável com o ARM Cortex-A9.

As Figuras 6.7 e 6.8 apresentam uma comparação do Rocket e do BOOM com determinados núcleos comerciais.

Processor	Core Area (core+L1s)	Scaled Area (to 28 nm)
Intel Xeon E5 2687W (Sandy) <sup>†</sup>	≈18 mm <sup>2</sup> @ 32nm	14 mm <sup>2</sup>
Intel Xeon E5 2667 (Ivy)*	≈12 mm <sup>2</sup> @ 22nm	19 mm <sup>2</sup>
RV64 BOOMv1 four-wide	1.4 mm <sup>2</sup> @ 45nm	0.54 mm <sup>2</sup>
RV64 BOOMv1 two-wide	1.1 mm <sup>2</sup> @ 45nm	0.43 mm <sup>2</sup>
ARM Cortex-A15*	2.8 mm <sup>2</sup> @ 28nm	2.8 mm <sup>2</sup>
<b>RV64 BOOMv2 (BROOM chip)</b>	0.52 mm <sup>2</sup> @ 28nm	0.52 mm <sup>2</sup>
ARM Cortex-A9 (Kayla Tegra 3)*	≈2.5 mm <sup>2</sup> @ 40nm	1.2 mm <sup>2</sup>
MIPS 74K <sup>‡</sup>	2.5 mm <sup>2</sup> @ 65nm	0.46 mm <sup>2</sup>
RV64 Rocket*	0.5 mm <sup>2</sup> @ 45nm	0.19 mm <sup>2</sup>
ARM Cortex-A5 <sup>‡</sup>	0.5 mm <sup>2</sup> @ 40nm	0.25 mm <sup>2</sup>

Figura 6.7 - BOOMv2 contra processadores comerciais.  
Fonte: [5].

Processor	CoreMark/ MHz/Core	Freq (MHz)	CoreMark/ Core
Intel Xeon E5 2687W (Sandy) <sup>†</sup>	<b>7.36</b>	3,400	25,007
Intel Xeon E5 2667 (Ivy)*	<b>5.60</b>	3,300	18,474
RV64 BOOMv1 four-wide	<b>5.18</b>	n/a **	n/a **
RV64 BOOMv1 two-wide	<b>4.87</b>	n/a **	n/a **
ARM Cortex-A15*	<b>4.72</b>	2,116	9,977
<b>RV64 BOOMv2 (BROOM chip)</b>	<b>3.77</b>	1,250 §	4,713 §
ARM Cortex-A9 (Kayla Tegra 3)*	<b>3.71</b>	1,400	5,189
MIPS 74K <sup>‡</sup>	<b>2.50</b>	1,600	4,000
RV64 Rocket*	<b>2.32</b>	1,500 **	3,480 **
ARM Cortex-A5 <sup>‡</sup>	<b>2.13</b>	1,000	2,125

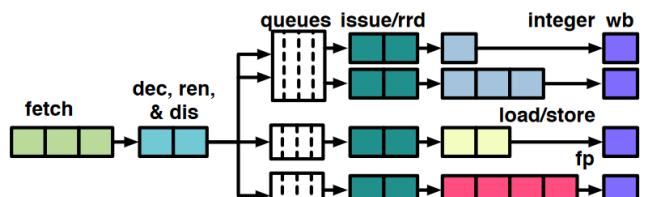
Figura 6.8 - BOOMv2 contra processadores comerciais.  
Fonte: [5].

Figura 6.9 - Esquemático simplificado da Pipeline do BOOMv2. Fonte: [5].

Quando comparado ao Cortex-A15, núcleo de alto desempenho da ARM de 2012 com *pipeline* esquematizada na Figura 6.10, observa-se que as diferenças estão na quantidade de estágios, e de unidades de execução.

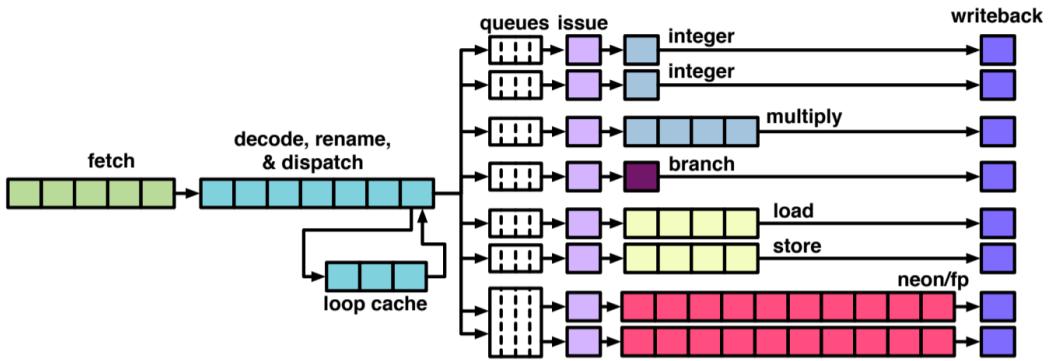


Figura 6.10 Pipeline do Cortex-A15. Fonte [53].

O núcleo BOOM, com esquemático mais detalhado mostrado na Figura 6.11, segue os mesmos princípios de *design* dos núcleos de alto desempenho atualmente no mercado e sua natureza *open-source* levou a empresa Esperanto Technologies a modificá-lo com objetivo de melhorar seu desempenho e torná-lo mais competitivo.

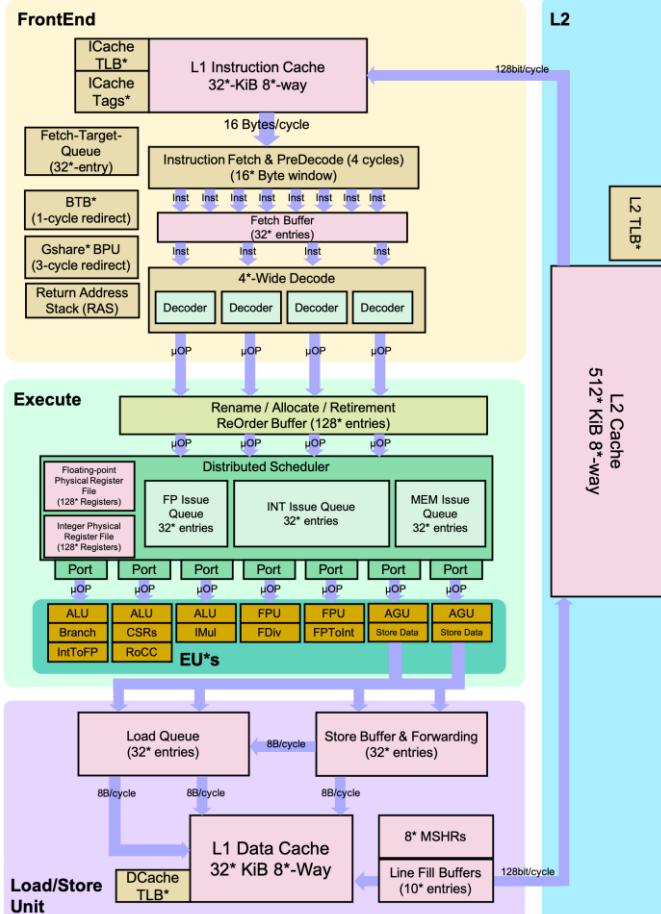


Figura 6.11 Esquemático do Núcleo BOOMv2. Fonte: [54].

O núcleo resultante, nomeado ET-Maxion e com esquemático demonstrado na Figura 6.12, apresenta 10 níveis de *pipeline* e um *clock* máximo esperado de 2.0 GHz em um processo de fabricação de 7 nm. Sua performance por GHz é

comparável ao ARM Cortex-A57, como mostrado na Figura 6.13.

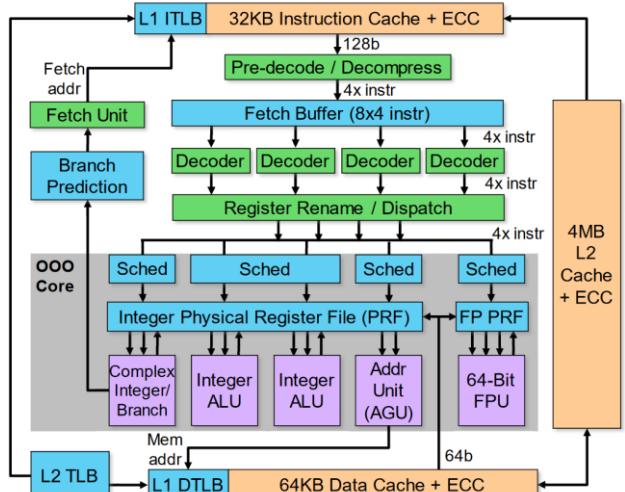
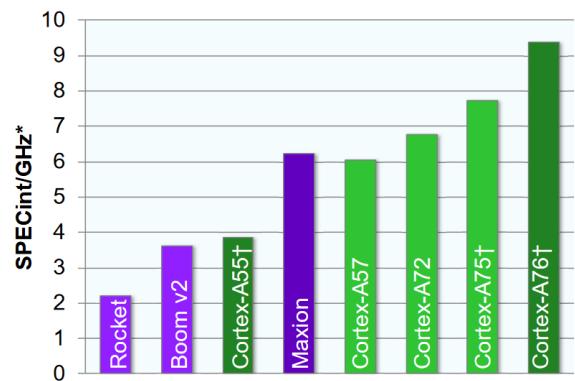


Figura 6.12 ET-Maxion. Fonte: [30].

Figura 6.13 Performance por *clock* do ET Maxion. Fonte: [30]

A diferença de performance entre o ET Maxion e o BOOM original demonstra que o ambiente de hardware aberto é extremamente benéficio para empresas que querem entrar no mercado rapidamente.

A SiFive anunciou em Outubro de 2019 seus novos núcleos com execução fora de ordem fabricados em um *node* de 7 nm com desempenho comparável ao ARM Cortex-M72, demonstrando grande interesse no mercado embarcado de alta performance [55].

## VII. PROJETO

O projeto realizado é um núcleo RISC-V que possui as principais características de uma implementação didática e capaz de executar todas as instruções contidas no conjunto base RV32I.

As diferenças entre a implementação em VHDL e o modelo proposto por David A. Patterson e John L. Hennessy em [23] originam-se da necessidade de implementar todas as instruções do conjunto base e não somente um subconjunto didático.

*Pipeline stalls* originados de *data hazards* com instruções de acesso a memória foram removidos utilizando *forwarding*, e as tomadas de decisão em instruções de desvio de controle são feitas pela ALU principal, apenas durante o estágio de Execute.

O núcleo possui a mesma *pipeline* clássica RISC de 5 estágios, sendo eles Instruction Fetch, Instruction Decode, Execute, Memory Access e Write Back, mostrada na Figura 7.1.

Devido ao fato de que o projeto não inclui mais de um núcleo de processamento, não possui múltiplos RISC-V *harts* (*threads* de execução distintos) presentes no espaço de memória e o único núcleo existente não possui características para finalizar mais de uma instrução por ciclo de *clock* (não é um núcleo *superscalar* com *out-of-order execution*), não existem motivos para implementar um modelo de consistência de memória complexo.

A escolha do núcleo de execução *in-order* e com apenas uma unidade de execução foi feita para manter o projeto relativamente simples.

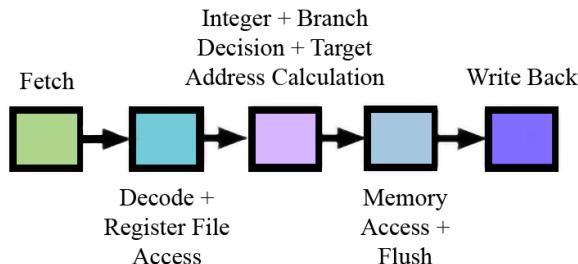


Figura 7.1 – Esquemático simplificado da *pipeline* do projeto. Fonte: Autor.

### A. Instruction Fetch

O estágio Instruction Fetch é o responsável por atualizar o valor do Program Counter e buscar a próxima instrução na memória de programa. Ele possui dois componentes, sendo eles Program Counter e Program Memory.

Program Counter é composto por um registrador, que guarda o valor da próxima instrução a ser buscada na Program Memory, um somador, que sempre incrementa o valor do registrador interno por um valor fixo de +4, de forma que possa apontar para a próxima instrução e um multiplexador, que alterna entre o valor incrementado pelo somador e um valor externo que vêm do estágio de Execute quando há desvio de controle.

O selecionador do multiplexador é controlado por uma porta lógica OR que recebe os sinais ALU\_branch\_Response e Jump\_Flag, onde o mesmo seleciona o sinal “PC+4” caso o nível lógico resultante seja 0, ou “Jump\_branch\_Destination”

caso seja 1. A Figura 7.2 demonstra o estágio.

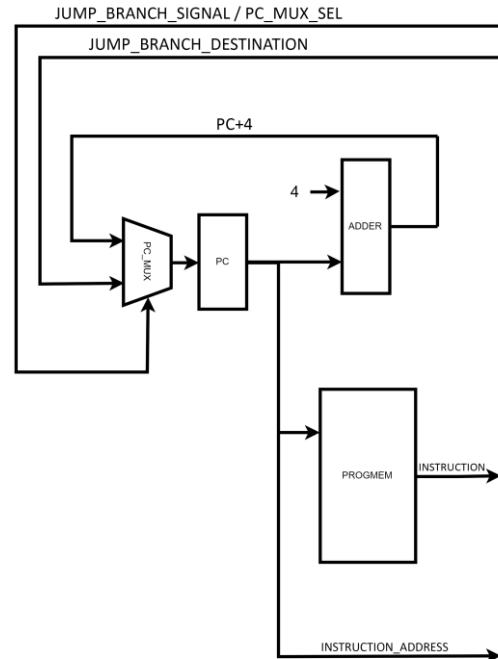


Figura 7.2 - Contador de programa. Fonte: Autor.

Com intuito de proporcionar conveniência na modificação do conteúdo de memória quando utilizando o Quartus, a Program Memory é implementada utilizando o *IP Core* da Intel “ALTSYNCRAM”, que permite inicialização de memória utilizando arquivos HEX ou MIF. A Figura 7.3 apresenta o gerenciador de memória do Quartus Prime.

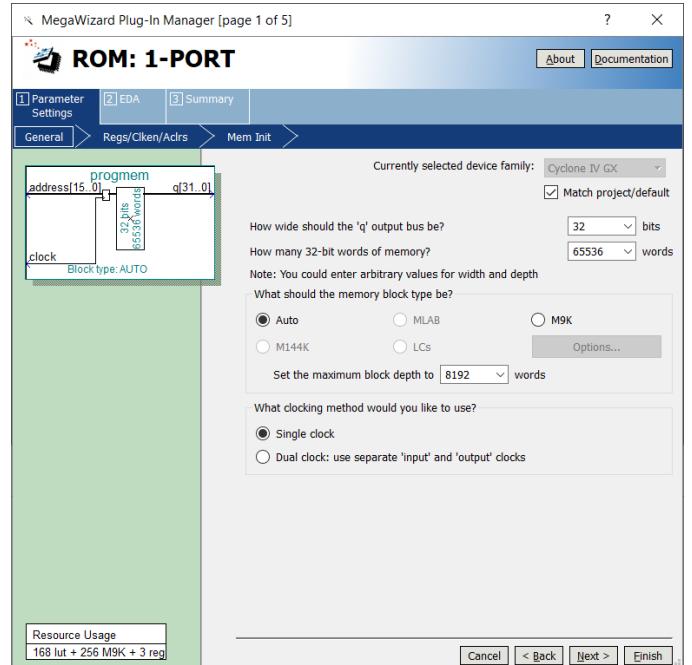


Figura 7.3 - MegaWizard Plug-In Manager. Fonte: Autor.

Devido ao fato de que a Intel não permite a utilização do “ALTSYNCRAM” sem entrada registrada em FPGAs que não

sejam da linha Stratix, o registrador de entrada da Program Memory deve possuir sempre valor igual ao registrador interno do PC. Levando isso em consideração, o PC possui duas possíveis saídas, Address e Next\_Address, onde o primeiro deve ser utilizado caso o projeto tenha que ser portado para um FPGA de outra empresa como Xilinx ou Microsemi e a memória utilizada não possua registrador de entrada.

A utilização de memórias *Read-Through*, que possuem endereço de leitura registrado, com a saída Address do PC consequentemente resultará em um atraso de 1 ciclo de *clock* na chegada da instrução ao registrador de *pipeline* IF/ID, impossibilitando que a operação de *flushing* (substituição das instruções em IF/ID e ID/EX por NOPs quando *branches* são pegas) seja executada apropriadamente.

Os registradores de *pipeline* IF/ID recebem a instrução a ser decodificada durante o próximo estágio e o seu endereço de memória, que é utilizado em desvios de controle para calcular o endereço de destino relativo ao PC.

#### B. Instruction Decode

O estágio de Instruction Decode é o responsável por mandar a instrução ao controlador, que na verdade é um decodificador simples que gera os sinais de controle dependendo da instrução de entrada. Além disso, o Register File é acessado atualizando seus valores de saída para rs1 e rs2.

Este estágio também é relativamente simples e possui apenas os dois componentes já citados, o controlador e o Register File.

Como já mencionado, o controlador é o elemento responsável por fazer a decodificação da instrução. Isto é, ordenar os bits do imediato e gerar 14 sinais de controle que viajarão ao longo da *pipeline* para realizar a respectiva instrução.

No próprio estágio ID, os sinais Regfile\_Read\_Address\_0 e Regfile\_Read\_Address\_1 são enviados ao Register File para que tenhamos os valores de rs1 e rs2 em Regfile\_Out\_0 e Regfile\_Out\_1.

Os demais sinais de controle gerados pelo controlador são denominados Jump Target Mux Sel, Mux\_1\_Sel, ALU\_Operation, ALU\_branch, ALU\_branch\_Control, Jump\_Flag, Data\_Format, Datamem\_Write\_Enable, Write Back\_Mux\_Sel, Regfile\_Write\_Address e Regfile\_Write\_Enable.

A nomenclatura utilizada é definida da seguinte forma, sinais de controle sem sufixos são os sinais que estão atualmente sendo gerados pelo controlador. Sinais de controle com um determinado sufixo, como por exemplo “EX\_MEM”, são sinais de controle que foram gerados anteriormente e atualmente se encontram nos registradores da *pipeline* EX/MEM.

O Register File possui apenas os 32 registradores de uso geral destinados a operações com inteiros. O registrador x0, assim como definido, não pode ter seu valor alterado. As saídas do Register File são implementadas utilizando dois multiplexadores assíncronos. A escrita ao Register File, entretanto, é síncrona e é feita durante a borda de descida.

Os sinais Regfile\_Write\_Enable\_MEMORY\_WB, Regfile\_Write\_Address\_MEMORY\_WB e Regfile\_Write\_Data vêm do estágio de Write Back, último estágio da *pipeline*.

É importante notar que caso um valor do Register File seja modificado durante uma operação de leitura, o valor mais recente irá para o próximo estágio.

Além de receber as duas saídas do Register File e todos os sinais de controle gerados que não são utilizados no estágio de ID, os registradores de *Pipeline* ID/EX também recebem o endereço da instrução, visto que o mesmo é utilizado durante EX por instruções de desvio de controle.

#### C. Execute

Em Execute, operações aritméticas são efetuadas, decisões de *branch* são tomadas e endereços de destino são calculados. Um sinal de controle é enviado à IF para que o PC seja corretamente atualizado. Este estágio é composto por 6 componentes, sendo eles Jump\_Target\_Unit, Forwarding\_Unit, Forward\_Mux\_0, Forward\_Mux\_1, Immediate\_Mux e Arithmetic\_Light\_Unit.

Jump\_Target\_Unit é o componente responsável por calcular endereços de destino durante instruções de desvio de controle, e possui como entradas Regfile\_Out\_0\_ID\_EX, Immediate\_ID\_EX, Instruction\_Address\_ID\_EX e JTU\_Mux\_Sel\_ID\_EX. Assim como dito anteriormente, o sufixo “ID\_EX” implica que estes sinais estão vindo dos registradores de *pipeline* entre os estágios de Instruction Decode e Execute.

JTU\_Mux\_Sel\_ID\_EX define se o cálculo do endereço de destino será efetuado utilizando o endereço da instrução, que é o caso nas instruções AUIPC, JAL, BEQ, BNE, BLT, BGE, BLTU e BGEU, ou Regfile\_Out\_0\_ID\_EX, que é utilizado apenas para JALR visto que a mesma é a única instrução de desvio de controle que não é relativa ao valor do PC.

Forwarding\_Unit é o componente responsável pelo redirecionamento de valores computados durante instruções passadas, que se encontram em um futuro estágio da *pipeline* mas ainda não foram escritos no Register File durante a borda de descida do estágio de Write Back. O componente controla dois multiplexadores, denominados Forward\_Mux\_0 e Forward\_Mux\_1, que substituem as entradas da ALU caso necessário.

O Forwarding\_Unit, com seus possíveis valores de saída mostrados na tabela 7.1 e sua simulação mostrada na Figura 7.4, compara os endereços rs1 e rs2 da atual instrução em Execute com os endereços rd das instruções que se encontram durante os estágios de Memory Access e Write Back. O componente também checa se as instruções passadas são instruções aritméticas ou de Load, através do valor do sinal de controle Write\_Back\_Mux\_Sel.

Origem dos dados	Forward_mux_0/1_control
Register File	000
ALU_Output_EX/MEM	001
Datamem_Output(MEM)	010
ALU_Output_MEMORY_WB	011
Datamem_Output_MEMORY_WB	100

Tabela 7.1 - Controle de forward. Fonte: Autor.

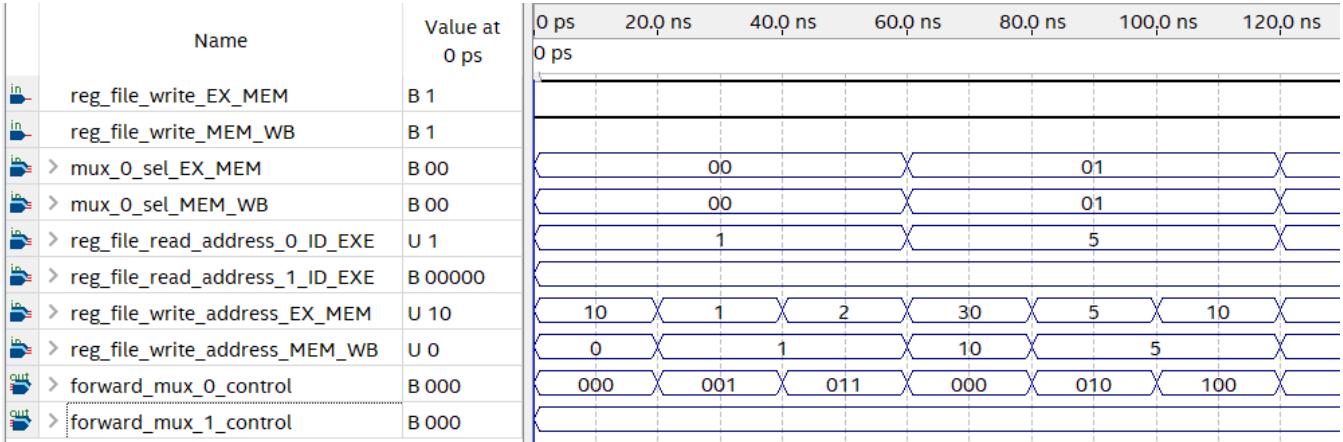


Figura 7.4 - Simulação waveform do Forwarding Unit. Fonte: Autor.

Na imagem podemos avaliar o funcionamento do Forwarding\_Unit. Quando o endereço de destino rd dos estágios posteriores (Memory Access e Write Back) são diferentes dos endereços de origem rs1 e rs2 em Execute, vemos que o sinal de controle *forward\_mux\_0\_control* possui seu valor mantido em “000”, que é a situação onde nenhum *forward* é necessário.

Quando os endereços rd de ambos Memory Access e Write Back são iguais a rs1 ou rs2 do estágio de Execute, a prioridade é do estágio mais próximo pois ele possui dados mais atualizados. Vemos então que a saída *forward\_mux\_0\_control* é alterada para “001”, indicando *forward* dos dados calculados pela ALU no ciclo anterior (que se encontram em Memory Access), isso se dá pelo fato de que *mux\_0\_sel\_EX\_MEM* possui seu valor em “00” (operação de Write Back deve ser feita com o resultado da ALU).

Mantendo as condições iguais ao caso anterior, quando rd de Write Back é igual a rs1 ou rs2, e rd de Memory Access é diferente, a operação de *forward* é feita com os dados calculados pela ALU a dois ciclos atrás, e isso é refletido na saída *forward\_mux\_0\_control*, que muda seu valor para “011”.

Caso *mux\_0\_sel\_EX\_MEM* ou *mux\_0\_sel\_MEM\_WB* possuam valor “01”, a instrução destes valores são instruções de *load* e a operação de *forward* deve ser feita com valores de saída da memória de dados, resultando nos valores “010” e “100” em *forward\_mux\_0\_control*.

Resumindo, operações de *forward* podem ser efetuadas para trazer ao estágio de Execute dados das saídas da ALU ou da memória de dados que se encontram em Memory Access ou Write Back.

O componente *Forward\_Mux\_0* é um multiplexador de 5

para 1 e possui as entradas *Register\_File\_Output\_0\_ID\_EX*, *Datamem\_Output*, *ALU\_Output\_EX\_MEM*, *Datamem\_Output\_MEM\_WB*. O componente *Forward\_Mux\_0* possui as mesmas entradas com exceção de *Register\_File\_Output\_0\_ID\_EX*, onde a mesma é substituída por *Register\_File\_Output\_1\_ID\_EX*. Os respectivos sinais de controle *Forward\_Mux\_0\_Control* e *Forward\_Mux\_1\_Control* são providos pelo *Forwarding\_Unit*.

*Immediate\_Mux (Mux\_1)* é o componente encarregado de substituir a saída do *Forward\_Mux\_1* por *Immediate\_ID\_EX* durante instruções aritméticas que utilizam imediatos, e possui seu sinal de controle *Immediate\_Mux\_Sel\_ID\_EX* presente nos registradores de *pipeline ID/EX*, sinal este que foi originado pelo controlador durante o estágio de ID.

O *Arithmetic\_Language\_Unit*, ou *ALU*, é o componente principal do estágio de Execute. Ele é o encarregado de efetuar operações lógicas, aritméticas, cálculo de endereços de memória para acessar memória de dados, faz comparações de *branch* e gera o sinal *ALU\_branch\_Response*, que em conjunto com o sinal *Jump\_Flag*, através de uma porta lógica OR, controla o Program Counter para que o mesmo tenha seu valor atualizado para o destino de um *Jump* ou *branch*.

O *ALU* é controlado por três sinais de controle, *ALU\_Operation\_ID\_EX*, *ALU\_branch\_ID\_EX* e *ALU\_branch\_Control\_ID\_EX*.

*ALU\_Operation\_ID\_EX* é utilizado por qualquer operação que utilize a *ALU* e não seja uma instrução de *branch*. Ela define se a *ALU* executará ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND ou suas equivalentes que utilizam imediatos.

Na Figura 7.5 podemos ver uma simulação da *ALU* onde ela executa todas as possíveis operações.

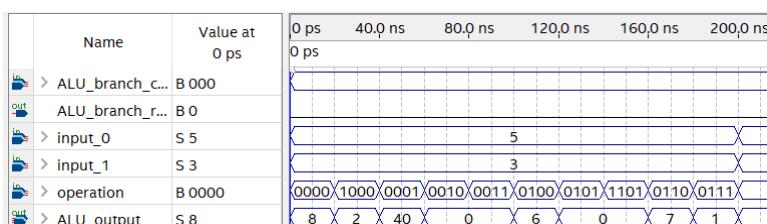


Figura 7.5 - Simulação da ALU. Fonte: Autor.

Operação	Codificação
ADD	0000
SUB	1000
SLL	0001
SLT	0010
SLTU	0011
XOR	0100
SRL	0101
SRA	1101
OR	0110
AND	0111

Tabela 7.2 Controle da ALU. Fonte: Autor.

Os 4 bits de controle são originados da própria codificação das instruções. Os 3 bits menos significativos de ALU\_Operation são os bits 14, 13 e 12 das instruções codificadas em tipo R ou tipo I. O bit mais significativo, que é 1 apenas para SUB ou SRA/SRAI, é codificado na posição 30 da instrução. A tabela 7.2 mostra como o ALU é controlado para operações lógicas e aritméticas.

ALU\_branch\_ID\_EX configura a ALU em modo de *branch*, e ALU\_branch\_Control\_ID\_EX controla qual condição deve ser testada, de forma com que qualquer instrução de *branch*, BEQ, BNE, BLT, BGE, BLTU ou BGEU, possa ser executada. Assim como ALU\_Operation, ALU\_branch\_Control também está codificado nas próprias instruções de *branch* no campo funct3. A tabela 7.3 mostra como o ALU é controlado em caso de *branch*.

Branch	Sinal de controle
BEQ	000
BNE	001
BLT	100
BGE	101
BLTU	110
BGEU	111

Tabela 7.3 Controle da ALU para *branches*. Fonte: Autor.

Os registradores de *pipeline* EX/MEM recebem os sinais Jump\_Flag, ALU\_branch\_Response, ALU\_Output, Forward\_Mux\_1\_Output, Instruction\_Address\_ID\_EX e todos os sinais de controle restantes.

O funcionamento correto da operação de *forwarding* em conjunto aa ALU pode ser testado através de uma série de Fibonacci, mostrada na tabela 7.4.

Ordem	Endereço	Instrução
1 <sup>a</sup>	00	ADDI x2, x0, 1
2 <sup>a</sup>	04	ADD x31, x0, x2
3 <sup>a</sup>	08	ADD x1, x1, x2
4 <sup>a</sup>	12	ADD x31, x0, x1
5 <sup>a</sup>	16	ADD x2, x2, x1
6 <sup>a</sup>	20	ADD x31, x0, x2
7 <sup>a</sup>	24	JALR x0, x0, 8

Tabela 7.4 Série de Fibonacci em RISC-V. Fonte: Autor.

O programa gera uma série de Fibonacci no registrador x31, utilizando x1 e x2 para guardar os valores enquanto os mesmos são calculados. É possível reparar que há diversos problemas de *data hazard*, visto que as instruções aritméticas irão preencher a *pipeline* e precisar de valores resultantes de instruções anteriores antes que os mesmos sejam escritos no Register File (ou seja, antes deles alcançarem seus estágios de Write Back).

A instrução no endereço 04 (ADD x31, x0, x2) alcança o estágio de Execute quando a instrução anterior (ADDI x2, x0, 1) encontra-se no estágio de Memory Access. Como a instrução em 04 possui seu rs2 igual o rd da instrução em 00, a Forwarding Unit “repassa” este valor já calculado, que se encontra no estágio seguinte, ao estágio de Execute.

A instrução no endereço 08, entretanto, também precisa do valor calculado pela primeira instrução. Quando a instrução em 12 (ADD x1, x1, x2) chega ao estágio de Execute, a instrução em 04 já encontra-se no estágio de Write Back, o que significa dizer que o estágio de Instruction Decode da instrução em 08 ocorreu antes do valor de x2 ter sido atualizado.

Neste caso, a Forwarding Unit realiza uma operação de *forward* repassando valores com origem no estágio de Write Back, visto que rs2 da instrução 08, x2, é igual ao rd da instrução 00.

O mesmo comportamento pode ser verificado ao longo do programa entre as demais instruções.

A Figura 7.6 mostra a série de Fibonacci sendo gerada em uma simulação de *waveform* no Quartus II.

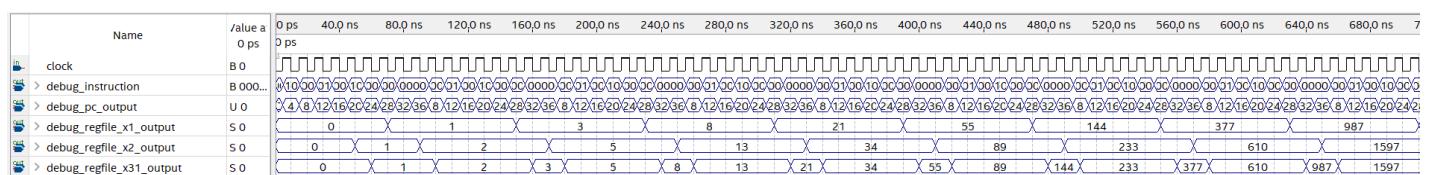


Figura 7.6 - Série de Fibonacci. Fonte: Autor.

Podemos ver que o valor do controle do Forwarding Mux 1 varia entre “000” (quando o mesmo pode utilizar os dados providos pelo Register File no estágio anterior), “001” (quando

ALU\_Output\_EX\_MEM precisa ser repassado a ALU) e “011” (quando ALU\_Output\_MEM\_WB precisa ser repassado ao ALU).

Para testar o critério de prioridade da operação de *forwarding*, um programa bem mais simples pode ser utilizado.

A tabela 7.5 mostra um programa simples que gera *data hazards*.

Ordem	Endereço	Instrução
1 <sup>a</sup>	00	ADDI x2, x2, 1
2 <sup>a</sup>	04	ADDI x2, x2, 1
3 <sup>a</sup>	08	ADDI x2, x2, 1

Tabela 7.5 *Forwarding*. Fonte: Autor.

Neste caso, a segunda instrução precisa resgatar o valor calculado na primeira instrução e durante a terceira há *data hazard* entre ela e ambas as instruções anteriores.

A primeira instrução entra em execução normalmente, e quando a mesma alcança o estágio MEM a segunda instrução encontra-se em EX. O Forwarding Unit, então, controla o Forward Mux de maneira com que ALU\_Output\_EX/MEM entre como argumento da ALU.

Quando a terceira instrução chega em EX, seu rs1 (x2) é igual ao rd de ambas instruções anteriores (que se encontram em MEM e WB), neste caso, o *forward* é feito do estado de MEM. A operação de *forward* é demonstrada na Figura 7.7.

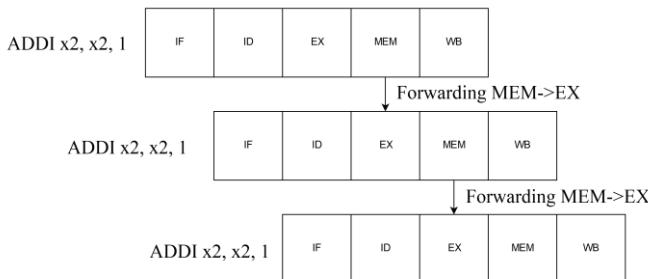


Figura 7.7 - *Forwarding*. Fonte: Autor.

#### D. Memory Access

Em Memory Access, a memória de dados é acessada e os registradores de *pipeline* IF/ID e ID/EX são limpos caso uma instrução de transferência de controle tenha sido executada com êxito no estágio anterior. Ele também apenas possui 2 componentes, sendo eles Datamem e Flushing Unit.

A memória de dados é dividida entre 4 componentes com barramento de entrada e saída de 8 bits. Este design possibilita que operações de memória possam ser feitas em todos os tamanhos suportados pela ISA, sendo eles Byte (8 bits), Half-Word (16 bits) e Word (32 bits) para RV32I.

Uma Word, por exemplo, possui seu primeiro endereço de byte em um número múltiplo de 4, e os 3 endereços posteriores, que pertencem aos outros bytes que compõem a mesma Word,

preenchem os espaços até o próximo endereço múltiplo de 4, que pertence à próxima Word.

Desta forma, cada elemento de memória encarrega-se de guardar um quarto de uma Word. O componente denominado Datamem Interface, que engloba Datamem, é o responsável por direcionar os dados aos respectivos componentes de memória e de fazer a conversão de endereços.

Abaixo podemos ver a topologia e como as memórias internas são endereçadas.

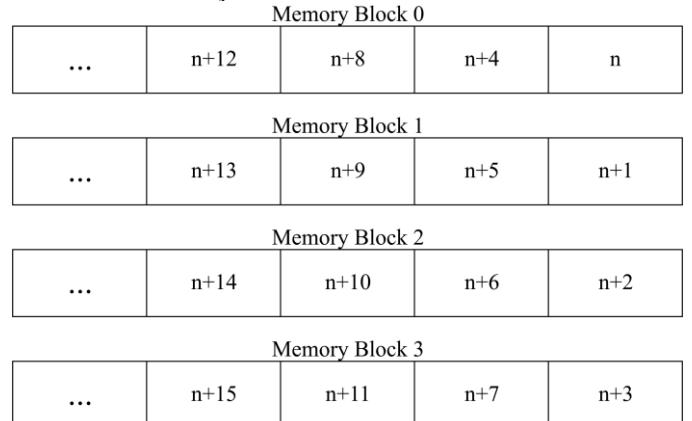


Figura 7.8 - Topologia de memória. Fonte: Autor.

Na Figura 7.8, 4 possíveis *words* podem ser acessadas, e cada uma delas só possui um único byte em cada bloco de memória. O componente Datamem Interface primeiramente divide o endereço de entrada visível à arquitetura por quatro, de forma com que possua o endereço equivalente ao primeiro bloco de memória. Os blocos de memória posteriores possuem seus endereços mapeados com offsets de +1 a +3.

Em uma instrução de SB, por exemplo, em que os 8 bits menos significativos de rs2 são salvos na memória de dados, Datamem Interface direciona os dados ao respectivo bloco de memória e não modifica os valores contidos nos demais.

Em qualquer instrução de Load, Datamem Interface formata os dados de saída dos 4 blocos e reconstrói o array de 32 bits apropriadamente.

O componente Flushing Unit é necessário devido ao fato de que o PC por padrão sempre é incrementado em +4, somente recebendo o endereço de destino de um possível *jump* ou *branch* durante o estágio de Execute destas instruções. Visto que o PC precisa de um pulso de *clock* para atualizar seu valor, o mesmo só chega a possuir sua saída apontando ao endereço de destino em caso de *jump* ou *branch* no estágio de Memory Access.

A saída de *flush*, mostrada na Figura 7.9, é acionada por uma porta lógica OR que possui ALU\_branch\_Response\_EX\_MEM e Jump\_Flag\_EX\_MEM como entradas. O sinal Flush\_Signal é mantido em nível lógico 1 apenas durante o pulso positivo do *clock*, de forma com que o registrador de *pipeline* IF/ID possa ser atualizado pelo próximo endereço apontado por PC.

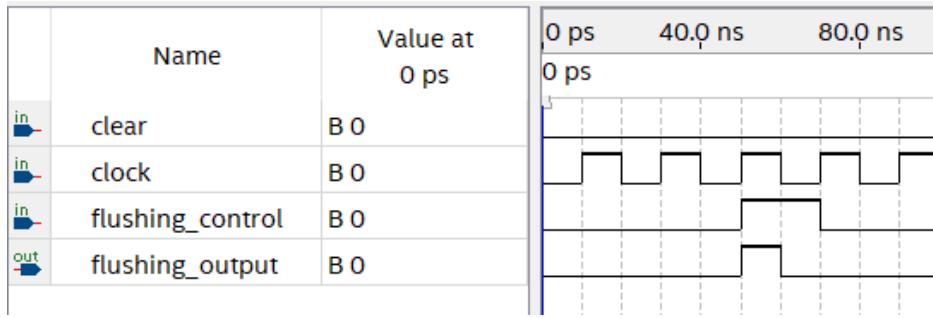


Figura 7.9 - Simulação de waveform do Flushing Unit. Fonte: Autor.

Na tabela 7.6 é possível ver o progresso de uma possível instrução de *branch* quando a condição é verdadeira e há uma mudança no endereço do PC.

PC	IF/ID	ID/EX	EX/MEM	MEM/WB
BRANCH				
BRANCH+4	BRANCH			
BRANCH+8	BRANCH+4	BRANCH		
DESTINATION	NOP (FLUSHED)	NOP (FLUSHED)	BRANCH (PEGA)	

Tabela 7.6 - Branches pegas. Fonte: Autor.

O comportamento é definido de maneira que as instruções contidas nos endereços posteriores (denominadas “BRANCH+N”, na imagem) começem a ser computadas antes mesmo da condição de *branch* ser testada. Caso não seja necessário alternar o endereço do PC para “destination”, a operação de *flush* não é realizada e nenhuma outra ação precisa ser tomada visto que o núcleo já se encontra na situação em que ele possui as próximas instruções em andamento na *pipeline*.

O comportamento descrito pode ser visto na tabela 7.7.

PC	IF/ID	ID/EX	EX/MEM	MEM/WB
BRANCH				
BRANCH+4	BRANCH			
BRANCH+8	BRANCH+4	BRANCH		
BRANCH+12	BRANCH+8	BRANCH+4	BRANCH (NÃO PEGA)	

Tabela 7.7 - Branches não pegas. Fonte: Autor.

No caso das instruções JAL, JALR e AUIPC, que não possuem condições a serem testadas, o *flush* durante Memory Access sempre acontece.

### E. Write Back

O estágio de Write Back possui apenas 2 componentes, um somador que incrementa o endereço da instrução em +4 antes do mesmo ser salvo em rd durante as instruções JAL e JALR, e um multiplexador denominado Write Back\_Mux (Mux\_0) que seleciona qual dado deve ser escrito no Register File.

Neste estágio os sinais restantes, Register\_File\_Write\_Enable\_MEM\_WB, Register\_File\_Write\_Address\_MEM\_WB e Write Back\_Mux\_Sel\_MEM\_WB são utilizados.

A escrita ao Register File acontece durante a descida do *clock*. Como a leitura do mesmo acontece de maneira assíncrona, uma instrução que se encontre no estado de Instruction Decode sempre irá mandar os valores “novos” ao

estado de Execute.

## VIII. TESTES

Devido à rápida aceitação da ISA RISC-V, o ecossistema de software vem crescendo saudavelmente e existem diversas ferramentas que podem ser utilizadas para checar se o funcionamento de uma determinada implementação é compatível com as especificações descritas em [21] e [22].

Ferramentas como QEMU e rv8, Figura 8.1, simulam o comportamento de uma possível implementação através da conversão das instruções RISC-V em instruções x86, que podem ser executadas nativamente em computadores pessoais.

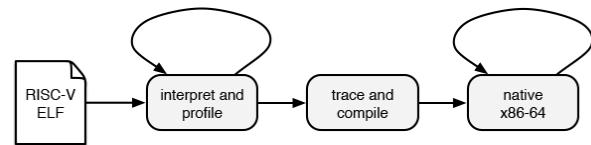


Figura 8.1 - Funcionamento do rv8. Fonte: rv8 Github Repository.

Os simuladores Spike e Western Digital ISS possuem pior performance, entretanto apresentam resultados para cada instrução individualmente de forma com que seja possível checar todo o fluxo do programa.

Além destes, existem também simuladores de *runtime* suficientemente adequados para testes de programas, como o Jupiter [56] e Venus [57], que possui uma versão online disponível e foi utilizado como referência comportamental.

## IX. COMPILAÇÃO DE PROGRAMAS C/C++ EM RISC-V

O núcleo foi testado utilizando código de máquina escrito manualmente. A compilação de programas escritos em C/C++, entretanto, pode ser realizada em sistemas operacionais UNIX utilizando a RISC-V GNU Toolchain. A *toolchain* suporta 2 modos de operação, a geração de um arquivo ELF genérico utilizando newlib, e a geração de um arquivo ELF que utiliza glibc para Linux. A compilação também pode ser feita utilizando Clang com RISC-V LLVM.

A Sifive disponibiliza um programa chamado elf2hex, que faz a conversão dos arquivos ELF gerados em arquivos HEX, que podem ser utilizados como inicialização de memória em FPGAs.

## X. RESULTADOS E CONCLUSÕES

O objetivo original da escolha deste tema era estudar e se familiarizar com a ISA RISC-V, entretanto diversos outros frutos foram colhidos ao longo desta jornada.

Dentre eles estão presentes conhecimentos sobre diversas práticas de design de processadores, como execução fora de ordem, *superscalar datapaths*, linguagens de sintetização de hardware de alto nível (Chisel, SpinalHDL, Bluespec), utilização de ferramentas para verificação (QEMU, Verilator, etc).

O projeto abriu portas para possíveis trabalhos futuros, como a implementação da ISA privilegiada, a introdução de uma extensão customizada destinada a acelerar alguma tarefa específica, a migração para um *datapath* com execução fora de ordem, a inserção de hierarquia de memória e a adição de algum barramento de interconexão como AXI ou AXI-Lite.

O núcleo desenvolvido executa com sucesso qualquer instrução contida no conjunto base RV32I, o projeto possui 2315 linhas de código VHDL, estando estas separadas em 29 arquivos distintos para diferentes componentes, relacionados na tabela 10.1.

Componente	Linhas VHDL
Adder	23
ALU	129
Controller	494
Datamem_interface	225
Datamem	25
Datapath	175
EX_MEM_DIV	140
Flushing_unit	38
Forwarding_unit	73
ID_EX_DIV	217
IF_ID_DIV	53
Jump_target_unit	21
MEM_WB_DIV	98
Microcontroller	68
Mux_2_1	18
Mux_3_1	20
Mux_5_1	22
Mux_32_1	49
Progmem_interface	26
Program_counter	25
Reg1b	28
Reg2b	28
Reg3b	28
Reg4b	28
Reg5b	28
Reg32b	28
Reg32b_falling_edge	28
Register_file	180
TOTAL	2315

Tabela 10.1 – Linhas VHDL/Componente. Fonte: Autor.

A sintetização completa efetuada pelo Quartus Prime 18.1 com o FPGA Cyclone V como alvo resultou em uma utilização de recursos folgada, como pode ser visto nas Figuras 10.1 e

10.2. Isto já era esperado visto que o núcleo não possui lógica para efetuar comunicação com periféricos externos implementada. Em sistemas simples, onde protocolos mais elaborados (como os definidos pelo AMBA) não são necessários, o usuário pode optar por externalizar alguns endereços de memória nos pinos do FPGA, transformando-os em GPIOs.

Analysis & Synthesis Resource Usage Summary		
	Resource	Usage
1	Estimate of Logic utilization (ALMs needed)	1704
2		
3	Combinational ALUT usage for logic	2116
1	-- 7 input functions	32
2	-- 6 input functions	1094
3	-- 5 input functions	366
4	-- 4 input functions	167
5	-- <=3 input functions	457
4		
5	Dedicated logic registers	1514
6		
7	I/O pins	523
8	Total MLAB memory bits	0
9	Total block memory bits	4194304
10		
11	Total DSP Blocks	0
12		
13	Maximum fan-out node	clock~input
14	Maximum fan-out	2027
15	Total fan-out	28755
16	Average fan-out	5.54

Figura 10.1 – Utilização de recursos. Fonte: Autor

Flow Summary	
	<<Filter>>
Flow Status	Successful - Sat Sep 14 22:20:16 2019
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	riscv_microcontroller
Top-level Entity Name	microcontroller
Family	Cyclone V
Device	5CGXFC9E7F35C8
Timing Models	Final
Logic utilization (in ALMs)	1,632 / 113,560 ( 1 % )
Total registers	1649
Total pins	523 / 616 ( 85 % )
Total virtual pins	0
Total block memory bits	4,194,304 / 12,492,800 ( 34 % )
Total DSP Blocks	0 / 342 ( 0 % )
Total HSSI RX PCSS	0 / 12 ( 0 % )
Total HSSI PMA RX Deserializers	0 / 12 ( 0 % )
Total HSSI TX PCSS	0 / 12 ( 0 % )
Total HSSI PMA TX Serializers	0 / 12 ( 0 % )
Total PLLs	0 / 20 ( 0 % )
Total DLLs	0 / 4 ( 0 % )

Figura 10.2 – Sumário de utilização do FPGA.

Fonte: Autor.

A análise de temporização do Quartus Prime 18.1 concluiu que o design conseguiria manter sua funcionalidade recebendo um *clock* de entrada de até 51.98 MHz como visto na Figura 10.3. Isto é excelente, as placas de desenvolvimento de baixo custo como a Terasic DE2 costumam possuir osciladores internos de 27 MHz ou 50 MHz.

Slow 1100mV OC Model Fmax Summary			
	Fmax	Restricted Fmax	
1	51.98 MHz	51.98 MHz	clock

Figura 10.3 – *Clock* máximo. Fonte: Autor.

É possível calcular a quantidade média de instruções executadas por segundo. Instruções de desvio, sejam estas *jumps* ou *branches*, diminuem a performance média do núcleo pelo motivo de que elas limpam os registradores de *pipeline* IF/ID e ID/EX através do *flush*.

A distribuição de instruções nos programas que compõem o *benchmark* SPECint2006, na tabela 10.2, demonstra que em média aproximadamente 20% das instruções executadas são instruções de desvio e que a maioria delas são instruções de *branch*. Levando isso em consideração, a performance média, no pior caso, é de aproximadamente 1.4 ciclos por instrução, e com *clock* de 50 MHz isto resulta em 35.71 MIPS.

Program	Loads	Stores	Branches	Jumps	ALU operations
astar	28%	6%	18%	2%	46%
bzip	20%	7%	11%	1%	54%
gcc	17%	23%	20%	4%	36%
gobmk	21%	12%	14%	2%	50%
h264ref	33%	14%	5%	2%	45%
hmmer	28%	9%	17%	0%	46%
libquantum	16%	6%	29%	0%	48%
mcf	35%	11%	24%	1%	29%
omnetpp	23%	15%	17%	7%	31%
perlbench	25%	14%	15%	7%	39%
sjeng	19%	7%	15%	3%	56%
xalancbmk	30%	8%	27%	3%	31%

Tabela 10.2 – Distribuição das instruções RISC-V em SPECint2006. Fonte: [11]

Uma taxa de execução de instruções, entretanto, não é uma medida absoluta de desempenho. O número de instruções necessárias para se executar um determinado programa varia entre diferentes ISAs. Embora atualmente haja quantidades relativamente altas de memória em nossos computadores pessoais, o tamanho do programa ainda é relevante, principalmente em sistemas embarcados e dispositivos de IoT.

Temos que a performance final de um possível núcleo pode ser quantificada como:

$$\text{Performance} = \frac{\text{Ciclos}}{\text{Instrução}} \cdot \frac{\text{Segundos}}{\text{Ciclo}} \cdot \frac{\text{Instruções}}{\text{Programa}}$$

Felizmente, RISC-V é uma ISA competitiva no quesito instruções por programa, também demonstrando tamanho de programa comparável ou melhor que x86 ou ARM quando a extensão C está disponível [58]. A tabela 10.3 apresenta uma comparação do número de instruções entre diversas ISAs diferentes, enquanto a tabela 10.4 apresenta uma comparação de tamanho de programa.

benchmark	x86-64 micro-ops	x86-64	IA-32	ARMv7	ARMv8	RV64G	RV64GC+ fusion
400.perlbench	1.13	1.00	1.04	1.16	1.07	1.17	1.14
401.bzip2	1.12	1.00	1.05	1.04	1.03	1.33	1.08
403.gcc	1.19	1.00	1.03	1.29	0.97	1.36	1.34
429.mcf	1.04	1.00	1.07	1.19	1.02	0.94	0.93
445.gobmk	1.11	1.00	1.00	1.19	1.10	1.18	1.11
456.hammer	1.47	1.00	1.19	1.45	1.21	1.16	1.16
458.sjeng	1.07	1.00	1.06	1.22	1.12	1.29	1.16
462.libquantum	0.88	1.00	1.62	1.38	0.95	0.83	0.83
464.h264ref	1.47	1.00	1.03	1.17	1.14	1.64	1.46
471.omnetpp	1.24	1.00	1.20	1.08	0.98	1.05	1.03
473.astar	1.04	1.00	1.11	1.17	1.05	0.99	0.89
483.xalancbmk	1.07	1.00	1.10	1.18	1.05	1.15	1.14
geomean	1.14	1.00	1.12	1.21	1.06	1.16	1.09

Tabela 10.3 – Número de instruções no SPECint2006 relativo à x86. Fonte: [58].

benchmark	x86-64	ARMv7	ARMv8	RV64G	RV64GC
400.perlbench	1.00	1.21	1.11	1.22	0.92
401.bzip2	1.00	1.07	1.07	1.38	1.06
403.gcc	1.00	1.40	1.05	1.47	1.03
429.mcf	1.00	1.40	1.20	1.11	0.83
445.gobmk	1.00	1.18	1.09	1.17	0.87
456.hammer	1.00	1.41	1.18	1.13	0.90
458.sjeng	1.00	1.19	1.09	1.25	0.92
462.libquantum	1.00	1.90	1.30	1.14	0.82
464.h264ref	1.00	1.14	1.12	1.61	1.28
471.omnetpp	1.00	1.17	1.06	1.13	0.79
473.astar	1.00	1.22	1.10	1.03	0.82
483.xalancbmk	1.00	1.28	1.14	1.24	0.91
geomean	1.00	1.28	1.12	1.23	0.92

Tabela 10.4 – Tamanho de programa no SPECint2006 relativo à x86. Fonte: [58].

Um dos objetivos do núcleo era ser acessível e de fácil entendimento, podendo ser utilizado no aprendizado sobre RISC-V ou arquitetura de computadores em geral. Quando comparado a outros núcleos abertos, é possível notar que as alternativas costumam possuir mais linhas de código e muitas vezes são projetos escritos em HDLs menos difundidas como visto na tabela 10.5.

Núcleo	HDL Primária	Linhas HDL
Rocket	Chisel	51276
BOOM	Chisel	21461
Riscy	Bluespec	11946
RiscyOO	Bluespec	43341
Lizard	PyMTL	37081
VexRiscv	SpinalHDL	89576

Ariane	SystemVerilog	72051
RI5CY	SystemVerilog	7529
SwerV	SystemVerilog	55312
Ibex	SystemVerilog	40602
Picorv32	Verilog	5834
Serv	Verilog	17023
HummingBird	Verilog	1122652
ORCA	VHDL	566195
ReonV	VHDL	1549266

Tabela 10.5 – Linhas de HDL de outros núcleos.  
Fonte: Autor.

O crescimento rápido da ISA RISC-V é animador, diversas universidades anunciaram a utilização da ISA não somente em projetos de pesquisa como também nas próprias disciplinas de graduação.

No Brasil, o laboratório de arquiteturas dedicadas da Universidade Federal de Campina Grande (UFCG) atualmente desenvolve um SoC com processador RISC-V dedicado a comunicação, criptografia e controle [59]. A Universidade Federal do Rio Grande (FURG) é membra da RISC-V Foundation e recentemente traduziu o livro “The RISC-V Reader: An Open Architecture Atlas” [60]. Em Novembro de 2018, aconteceu na Escola Politécnica da Universidade de São Paulo (USP) o primeiro “RISC-V Conference Day” no Brasil, contemplando diversas palestras [61]. A Universidade Federal do Rio Grande do Norte (UFRN) é listada como usuária acadêmica da plataforma PULP [62].

Diversos trabalhos podem ser encontrados nas bibliotecas virtuais de outras universidades brasileiras e citar todos é inviável, mas isso apenas mostra como uma arquitetura aberta, livre de *royalties* e com suporte comercial pode ajudar na movimentação do mercado de semicondutores e sistemas embarcados do país.

Para finalizar, o núcleo desenvolvido está disponível em <https://github.com/Artoriuz/maestro> com uma licença *open-source* MIT, que permite que seja utilizado e modificado abertamente.

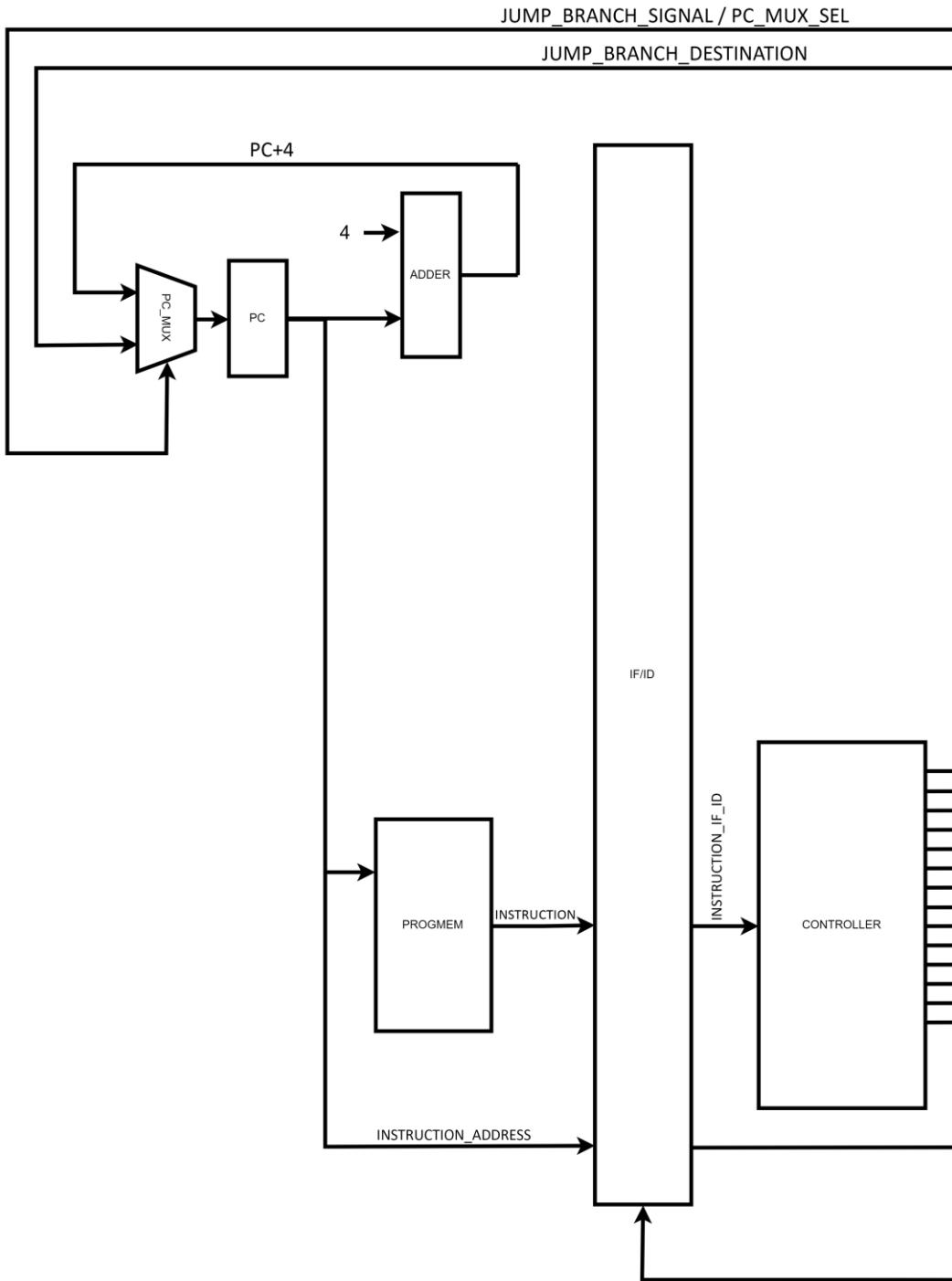
## REFERÉNCIAS

- [1] ‘UNdata | record view | Mobile-cellular telephone subscriptions’. [Online]. Available: <http://data.un.org/Data.aspx?d=ITU&f=ind1Code%3aI271>. [Accessed: 19-Sep-2019].
- [2] C. Wagner and N. Harned, ‘EUV lithography: Lithography gets extreme’, *Nat. Photonics*, vol. 4, pp. 24–26, Jan. 2010.
- [3] R. Xie *et al.*, ‘A 7nm FinFET technology featuring EUV patterning and dual strained high mobility channels’, in *2016 IEEE International Electron Devices Meeting (IEDM)*, 2016, pp. 2.7.1-2.7.4.
- [4] K. Rupp, ‘40 Years of Microprocessor Trend Data | Karl Rupp’.
- [5] C. P. Celio, ‘A Highly Productive Implementation of an Out-of-Order Processor Generator’, p. 157.
- [6] I. J. Cutress, ‘Intel’s Interconnected Future: Combining Chiplets, EMIB, and Foveros’. [Online]. Available: <https://www.anandtech.com/show/14211/intels-interconnected-future-chiplets-emib-foveros>. [Accessed: 20-Jul-2019].
- [7] N. Beck, S. White, M. Paraschou, and S. Naffziger, “‘Zeppelin’: An SoC for multichip architectures”, in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, 2018, pp. 40–42.
- [8] D. A. Patterson and J. L. Hennessy, ‘A New Golden Age for Computer Architecture: Domain-Specific Hardware/Software Co-Design, Enhanced Security, Open Instruction Sets and Agile Chip Development.’, 04-Jun-2018. [Online]. Available: <https://iscaconf.org/isca2018/docs/HennessyPattersonTuringLectureISCA4June2018.pdf>. [Accessed: 01-Sep-2019].
- [9] N. P. Jouppi *et al.*, ‘In-datacenter performance analysis of a tensor processing unit’, in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 1–12.
- [10] ‘Exynos 9820 - Samsung - WikiChip’. [Online]. Available: <https://en.wikichip.org/wiki/samsung/exynos/9820>. [Accessed: 18-Sep-2019].
- [11] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier, 2012.
- [12] ‘June 2019 | TOP500 Supercomputer Sites’. [Online]. Available: <https://www.top500.org/lists/2019/06/>. [Accessed: 20-Aug-2019].
- [13] ‘Summit’, *Oak Ridge Leadership Computing Facility*.
- [14] ‘Sierra’, *Computing*. [Online]. Available: <https://computing.llnl.gov/computers/sierra>. [Accessed: 24-Aug-2019].
- [15] ‘Cloud TPU’, *Google Cloud*. [Online]. Available: <https://cloud.google.com/tpu/>. [Accessed: 20-Aug-2019].
- [16] E. ter Hoeven, ‘dav1d 0.1.0 release: The first benchmarks’, *Medium*, 11-Dec-2018. [Online]. Available: <https://medium.com/@ewoutterhoeven/dav1d-0-1-0-release-the-first-benchmarks-5404360e44e3>. [Accessed: 02-Sep-2019].
- [17] Charles E. Leiserson *et al.*, ‘There’s plenty of room at the top: What will drive growth in computer performance after Moore’s Law ends?’, 2019.
- [18] D. A. Patterson and K. Asanović, ‘Instruction Sets Should Be Free: The Case For RISC-V | EECS at UC Berkeley’. [Online]. Available: <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/E ECS-2014-146.html>. [Accessed: 20-Jul-2019].
- [19] A. Frumusanu, ‘ARM Details Built on ARM Cortex Technology License’. [Online]. Available: <https://www.anandtech.com/show/10366/arm-built-on-cortex-license>. [Accessed: 01-Sep-2019].
- [20] M. Lipp *et al.*, ‘Meltdown’, *ArXiv180101207 Cs*, Jan. 2018.
- [21] P. Kocher *et al.*, ‘Spectre Attacks: Exploiting Speculative Execution’, *ArXiv180101203 Cs*, Jan. 2018.

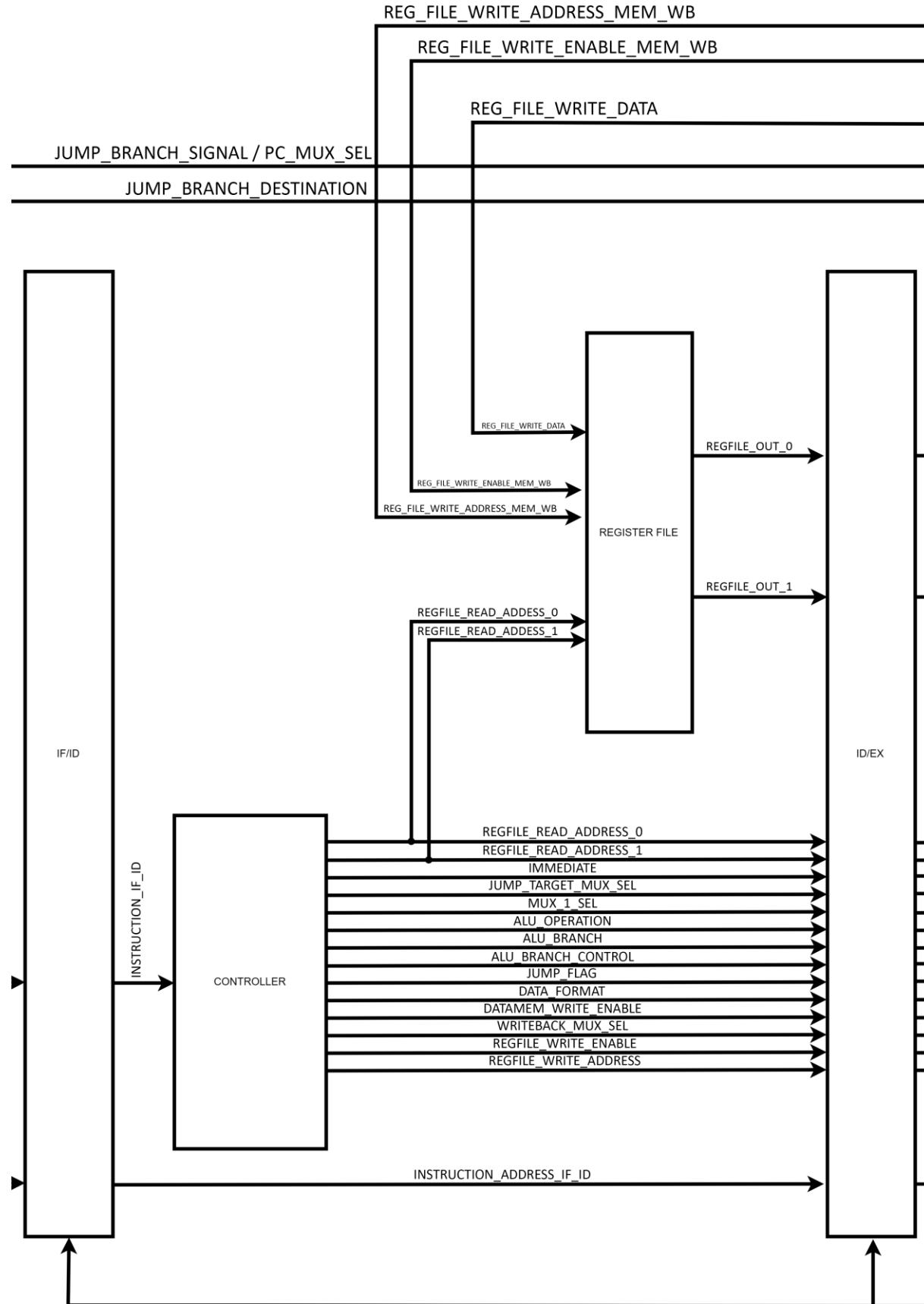
- [22] M. Schwarz *et al.*, ‘ZombieLoad: Cross-Privilege-Boundary Data Sampling’, *ArXiv190505726 Cs*, May 2019.
- [23] ‘Why RISC-V?’, *RISC-V Foundation*. [Online]. Available: <https://riscv.org/why-risc-v/>. [Accessed: 20-Jul-2019].
- [24] J. Xie, ‘NVIDIA RISC-V Story’, Jul-2016. [Online]. Available: [https://riscv.org/wp-content/uploads/2016/07/Tue1100\\_Nvidia\\_RISCV\\_Stor\\_y\\_V2.pdf](https://riscv.org/wp-content/uploads/2016/07/Tue1100_Nvidia_RISCV_Stor_y_V2.pdf). [Accessed: 20-Jul-2019].
- [25] Z. Z. Bandic, R. Golla, and D. Vucinic, ‘CPU Project in Western Digital: From Embedded Cores for Flash Controllers to Vision of Datacenter Processors with Open Interfaces’, p. 24.
- [26] ‘Members at a Glance’, *RISC-V Foundation*. [Online]. Available: <https://riscv.org/members-at-a-glance/>. [Accessed: 20-Jul-2019].
- [27] ‘Software Status’, *RISC-V Foundation*. [Online]. Available: <https://riscv.org/software-status/>. [Accessed: 20-Jul-2019].
- [28] ‘Boards & Software - SiFive’, <https://www.sifive.com/share.png>. [Online]. Available: <https://www.sifive.com/boards>. [Accessed: 13-Nov-2019].
- [29] ‘RISC-V@Andes’, *Andes Technology*..
- [30] L. Gwennap, ‘ESPERANTO MAXES OUT RISC-V’, p. 5, 2018.
- [31] ‘Syntacore | custom cores and tools’. [Online]. Available: <https://syntacore.com/>. [Accessed: 13-Nov-2019].
- [32] ‘RISC-V | Western Digital’. [Online]. Available: <https://www.westerndigital.com/company/innovations/risc-v>. [Accessed: 24-Oct-2019].
- [33] K. Q. 27 J. 2019 at 01:10, ‘Alibaba sketches world’s “fastest” “open-source” RISC-V processor yet: 16 cores, 64-bit, 2.5GHz, 12nm, out-of-order exec’. [Online]. Available: [https://www.theregister.co.uk/2019/07/27/alibaba\\_risc\\_v\\_chip/](https://www.theregister.co.uk/2019/07/27/alibaba_risc_v_chip/). [Accessed: 24-Oct-2019].
- [34] ‘Riscy Expedition | MIT’s riscy expedition.’ [Online]. Available: <http://csg.csail.mit.edu/riscy-e/>. [Accessed: 13-Nov-2019].
- [35] ‘Shakti Processor’. [Online]. Available: <https://shakti.org.in/>. [Accessed: 13-Nov-2019].
- [36] ‘lowRISC: Collaborative open silicon engineering · lowRISC: Collaborative open silicon engineering’. [Online]. Available: <https://www.lowrisc.org/>. [Accessed: 13-Nov-2019].
- [37] R. Bukin, ‘FreeBSD/RISC-V’, University of Cambridge Computer Laboratory, 01-May-2016.
- [38] ‘riscv - FreeBSD Wiki’. [Online]. Available: <https://wiki.freebsd.org/riscv>. [Accessed: 13-Nov-2019].
- [39] ‘PULP platform’. [Online]. Available: <https://www.pulp-platform.org/>. [Accessed: 13-Nov-2019].
- [40] ‘Announcing the Winners of the RISC-V Soft CPU Contest’, *RISC-V Foundation*, 01-Oct-2019. [Online]. Available: <https://riscv.org/2019/10/announcing-the-winners-of-the-risc-v-soft-cpu-contest/>. [Accessed: 13-Nov-2019].
- [41] ‘RISC-V - Getting Started Guide’, p. 30.
- [42] ‘Embracing and expanding the open hardware ecosystem’, *IBM IT Infrastructure Blog*, 21-Aug-2019. [Online]. Available: <https://www.ibm.com/blogs/systems/embracing-and-expanding-the-open-hardware-ecosystem/>. [Accessed: 24-Aug-2019].
- [43] IBM, ‘Power ISA™ Version 3.0 B’. 29-Mar-2017.
- [44] ‘Instruction Sets | Arm Custom Instructions – Arm Developer’. [Online]. Available: <https://developer.arm.com/architectures/instruction-sets/custom-instructions>. [Accessed: 24-Oct-2019].
- [45] ‘Krsnte Asanović : RISC-V Momentum is Massive in India - EE Times India’. [Online]. Available: <https://www.eetindia.co.in/news/article/20190104NT01-RISC-V-momentum-is-massive-in-India>. [Accessed: 24-Oct-2019].
- [46] S. Di Mascio, A. Menicucci, E. Gill, G. Furano, and C. Monteleone, ‘Leveraging the Openness and Modularity of RISC-V in Space’, *J. Aerosp. Inf. Syst.*, vol. 0, no. 0, pp. 1–19, 0.
- [47] ‘Mi-V RISC-V Ecosystem | Microsemi’. [Online]. Available: <https://www.microsemi.com/product-directory/fpga-soc/5210-mi-v-embedded-ecosystem#overview>. [Accessed: 13-Nov-2019].
- [48] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, ‘The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Document Version 20190608-Base-Ratified’, CS Division, EECS Department, University of California, Berkely, Jun. 2019.
- [49] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, ‘The RISC-V Instruction Set Manual. Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified’, CS Division, EECS Department, University of California, Berkely, Jun. 2019.
- [50] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2017.
- [51] ‘ucb-bar/riscv-sodor’, *GitHub*. [Online]. Available: <https://github.com/ucb-bar/riscv-sodor>. [Accessed: 09-Sep-2019].
- [52] K. Asanovic *et al.*, ‘The Rocket Chip Generator’, p. 11.
- [53] ‘Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7’, *EETimes*. [Online]. Available: [https://www.eetimes.com/document.asp?doc\\_id=1279167](https://www.eetimes.com/document.asp?doc_id=1279167). [Accessed: 09-Sep-2019].
- [54] ‘The BOOM Pipeline — RISCV-BOOM documentation’. [Online]. Available: <https://docs.boom-core.org/en/latest/sections/intro-overview/boom-pipeline.html#boom-pipeline>. [Accessed: 24-Oct-2019].
- [55] ‘Incredibly Scalable High-Performance RISC-V Core IP - SiFive’, <https://www.sifive.com/share.png>. [Online]. Available: <https://www.sifive.com/blog/incredibly-scalable-high-performance-risc-v-core-ip>. [Accessed: 27-Oct-2019].
- [56] A. Castellanos, *andrescv/Jupiter*. 2019.
- [57] S. K., *ThaumicMekanism/venus*. 2019.

- [58] C. Celio, P. Dabbelt, D. A. Patterson, and K. Asanović, ‘The Renewed Case for the Reduced Instruction Set Computer: Avoiding ISA Bloat with Macro-Op Fusion for RISC-V’, *ArXiv160702318 Cs*, Jul. 2016.
- [59] ‘LAD - Laboratório de Arquiteturas Dedicadas’. [Online]. Available: <http://lad.dsc.ufcg.edu.br/lad/pmwiki.php?n=Lad.Projetos>. [Accessed: 17-Sep-2019].
- [60] ‘The RISC-V Reader: An Open Architecture Atlas’. [Online]. Available: <http://www.riscbook.com/portuguese/>. [Accessed: 17-Sep-2019].
- [61] ‘RISC-V Conference Day’. [Online]. Available: <http://www.lsi.usp.br/risc-v/>. [Accessed: 17-Sep-2019].
- [62] ‘PULP users’. [Online]. Available: [https://pulp-platform.org/pulp\\_users.html](https://pulp-platform.org/pulp_users.html). [Accessed: 27-Oct-2019].

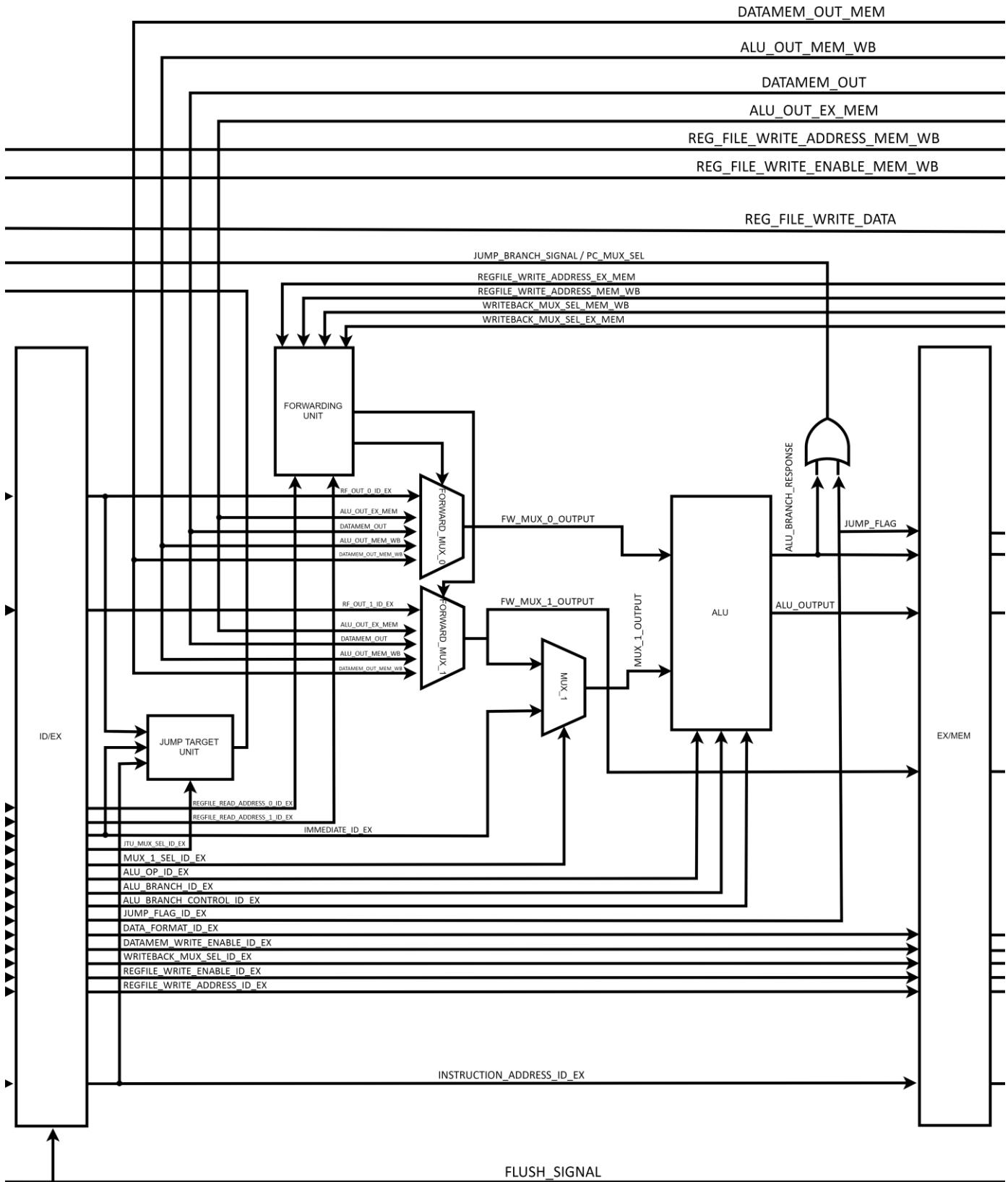
## Anexo 1 – Instruction Fetch



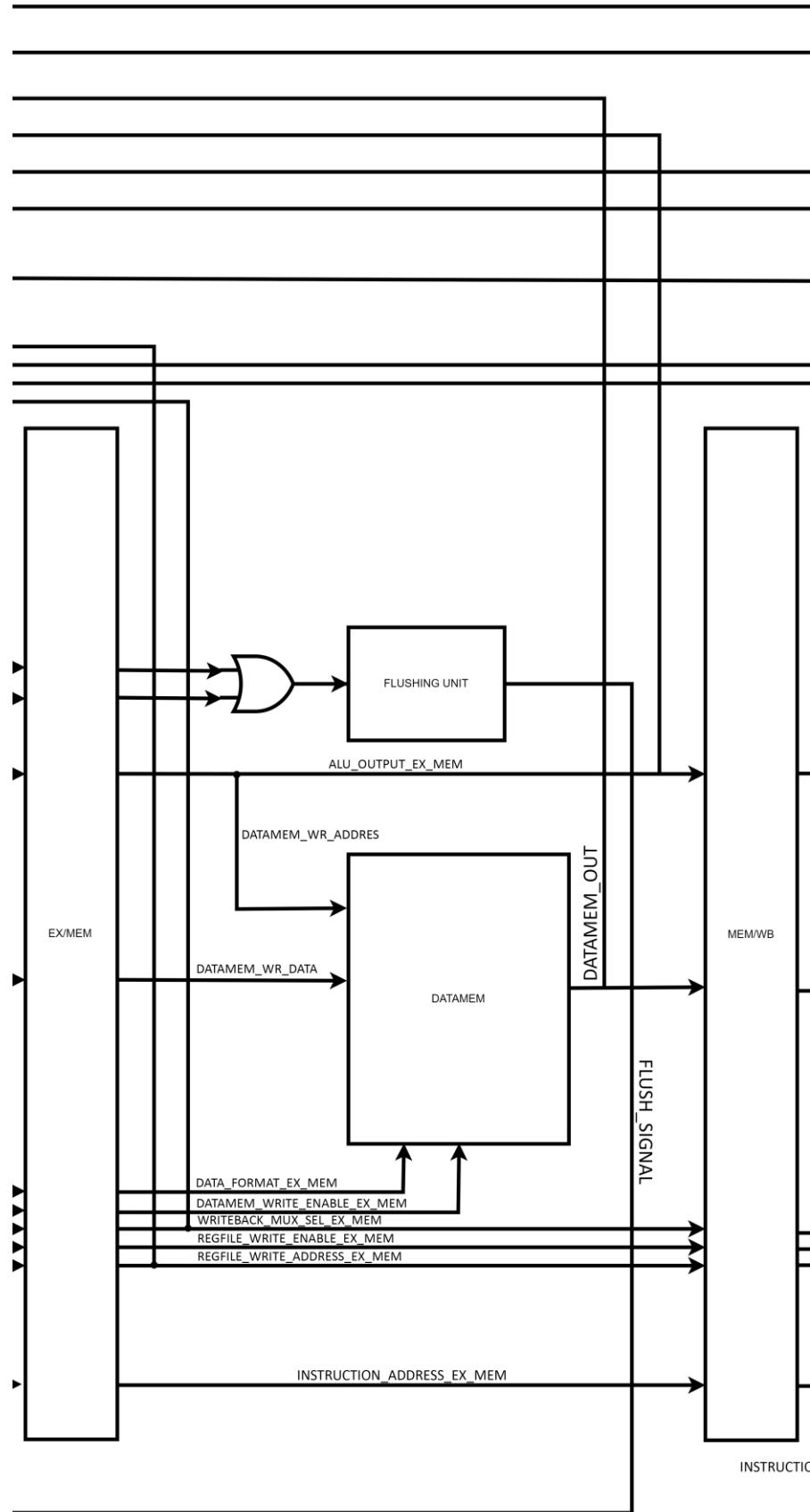
## Anexo 2 – Instruction Decode



## Anexo 3 – Execute



## Anexo 4 – Memory Access



## Anexo 5 – Write Back

