Israel da Rocha

I-MIPS: Desenvolvimento de um processador baseado no MIPS monociclo com aplicação em FPGA

São José dos Campos - Brasil

Maio de 2022

Israel da Rocha

I-MIPS: Desenvolvimento de um processador baseado no MIPS monociclo com aplicação em FPGA

Relatório apresentado à Universidade Federal de São Paulo como parte dos requisitos para aprovação na disciplina de Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores.

Docente: Prof. Dr. Sergio Ronaldo Barros dos Santos

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Maio de 2022

Resumo

Com o desenvolvimento da tecnologia, até os dias de hoje os computadores passaram por uma série de evoluções em sua estrutura. O estado em que se encontra só foi possível graças ao conjunto de ideias que foram sendo agregadas, assim diferentes formas, arquiteturas, modelos e organizações foram criadas e podem servir de escolha para novos projetos. Com o passar do tempo, os principais elementos de um computador foram agrupados em um chip chamado processador, esse equipamento atualmente é o coração dos sistemas computacionais, capaz de interagir com elementos periféricos, com a memória e capaz de realizar uma série de tarefas específicas. Assim, para que o aprimoramento tecnológico nessas áreas continuem, deve-se entender o funcionamento da interconexão entre esses componentes e os seus princípios fundamentais. Esse trabalho tem como objetivo apresentar os principais conceitos sobre arquitetura e organização de computadores, e a partir disso, desenvolver passo a passo um processador chamado I-MIPS, tendo como referência o MIPS monociclo. Conforme o desenvolvimento, será mostrado como a estrutura criada suporta o processamento de cada instrução definida e seus modos de endereçamento. Ao final, é apresentado um resultado com interação com o usuário e discutido o estado final desenvolvido da cpu. Concluindo desse modo que os objetivos específicos e gerais foram alcançados, e portanto o *design* e o código desenvolvido são corretos e suficientes para o funcionamento da cpu.

Palavras-chaves: computador, processador, MIPS, arquitetura, organização.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

A história da criação de computadores até a chegada dos dias atuais remete aos tempos antigos, antes da era tecnológica, com ideias de automatização de trabalho. O que se espera de uma máquina é a realização de atividades específicas para facilitar a nossa vida. Ainda nessas epocas não exisitiam os conceitos tecnológicos atuais como processador, circuitos digitais, circuitos elétricos, lógica computacional, memória digital, etc. mas sim a ideia principal de reduzir o tempo gasto para realização de tarefas e a garantia de que uma mesma atividade produza o mesmo resultado. Ainda que antigas, essas e outras ideias podem ser vistas aplicadas nos dias atuais, (O tempo é uma das principais características que se busca em computadores, por isso, medidas temporais foram criadas, como MIPS (Microinstruções por segundo), frequência, latência, período são termos constantes utilizados nesse contexto), e a garantia que uma mesma atividade produza o mesmo resultado pode ser convertida atualmente como "Uma mesma combinação de entrada deverá produzir uma mesma combinação de saída", algo que é visto desde um nível alto de abstração como um usuário comum interagindo no computador (entrada) e vendo o resultado no monitor (Saída), para níveis cada vez mais baixo de abstração: funções de um software desenvolvido por um programador, resultados binários de um sistema lógico, etc.

Dito isso, o computador é um sistema complexo com vário componentes nos dias de hoje, e buscar as suas origens e precursores está intimamente ligado à definição de o que é e/ou compõe um computador. Se nos referirmos ao computador apenas como uma ferramenta capaz de auxiliar em cálculos, então o ábaco, uma ferramente para operações de soma e subtração desenvolvida em aproximadamente 3000 a.C (FILHO, 2007) poderia ser considerado o primeiro computador. Ou ainda se dissermos que é preciso ser uma ferramenta mecânica, então a máquina de pascal de 1642, considerada a primeira calculadora por Santos e Mariotto (2020) tomaria esse título. Ainda poderíamos citar a Máquina de diferenças e o engenho analítico de Babbage, com suas máquinas, nunca finalizadas, que seriam as primeiras programáveis para o uso de cálculos (CREPALDI; COSTA; ESCOBAL, 2017), ou ainda o Mark I por ser a primeira máquina eletromecânica automática (CHAVES et al., 2019), outros clamam que foi o ENIAC por ser o primeiro computador completamente digital eletrônico (CHAVES et al., 2019) e poderíamos continuar até as máquinas atuais. O importante é que com essa evolução, foram sendo agregados conhecimentos e técnicas e nomenclaturas e novos componentes que sofrem melhorias a cada ano, não é a toa que a lei de moore é tão famosa no ramo da tecnologia. Dessa forma é importante entendermos os fundamentos do computador atual para que possamos continuar aperfeiçoando essas tecnologias.

No estado atual, os principais elementos que compõe o computador e são capazes de realizar pequenas instruções foram condensados em um chip, o processador, o qual interage com a memória para realizar acesso e armazenamento de dados, por fim também há uma interação com elementos periféricos, que são de onde o computador irá receber valores de entrada e como irá apresentá-los na saída. Esses são os principais elementos modernos para se criar um computador. Durante trabalho iremos apresentar o desenvolvimentos desses elementos, especificar o que os compõe e de que forma é feito sua interação, ou seja, iremos desenvolver a arquitetura e organização do computador.

2 Objetivos

2.1 Gerais

Usar os conhecimentos adquiridos para o desenvolvimento de um processador chamado I-MIPS com arquitetura e organização bem definidos e capazes de realizar instruções suficientes para comandos básicos conhecidos no meio da programação. Ao final, um programa simples que aceita dados de entrada deverá estar funcional em um chip FPGA e apresentando saídas esperadas.

2.2 Específicos

1. Revisão bibliográfica
2. Definição do modelo e linha de arquitetura base
3. Escolha do sentido de interpretação dos dados (Endianness)
4. Definição do conjunto de instruções
5. Escolha do formato das instruções
6. Definir os modos de endereçamento
7. Caminho de dados
8. Posicionamento dos módulos de entrada e saída
9. Desenvolvimento da ULA
10. Desenvolvimento da Unidade de Processamento
11. Desenvolvimento da Unidade de Controle
12. Testes em módulos individuais e do sistema integrado
13. Validação prática com aplicação de um programa simples em FPGA

3 Fundamentação Teórica

3.1 Números binários

Desde muito cedo os humanos optaram por escolher uma base numérica para fazer contagens e posteriormente operações matemáticas. Essa escolha se baseou na quantidade de dedos nas mãos e assim é utilizado a base decimal para a maioria das operações e unidades de medidas cotidianas.

Já na área computacional, desde que Von Neumann apresentou o EDVAC em 1945 (RANDELL; BABBAGE, 1973), os computadores começaram a implementar sistemas digitais utilizando base binária. Se para os humanos, a base decimal representa os 10 dedos da mão, para um computador, a base binária pode ser relacionada a passagem (1) ou não de corrente (0).

Ter uma base diferente significa que mesmas quantidades poderão ter valor numérico diferente. Por exemplo, o número 10 em binário representa 2 em decimal, e o número 10 em decimal representa 1010 em binário. Para que não haja confusão sobre qual base um valor está, o número anterior pode ser representado como 10_2 para binário e 10_{10} para decimal.

Em sistemas computacionais é muito comum números binários servirem de indexador ou mapeamento de ações. Nesse sentido, a Eq. 3.1 relaciona o número de bits à quantidade de indexadores binários possíveis.

$$f(n) = 2^n \tag{3.1}$$

Onde f é a quantidade em decimal que pode ser representados pelos n bits.

3.2 RISC x CISC

A intel, em 1971, deu ínicio a história dos processadores com o lançamento do intel 4004 (FERLIN, 2004), onde agora esse único chip continha os principais componentes de um computador, e junto com isso, foi preciso que houvesse uma estrutura de como os dados seriam interpretados e como iriam fluir internamente à esse *chip*, ou seja, que houvesse uma arquitetura. Dessa forma, dois principais modelos surgiram e foram à base para as arquiteturas mais modernas e que continuam sendo classificadas em uma dessas duas linhas: a arquitetura RISC e arquitetura CISC. Cada uma possui sua própria filosofia e características (OLIVEIRA, 2017).

A arquitetura CISC (*Complex Instruction Set Computer*) como o nome já diz, possuí um conjunto de instruções complexa, ela surgiu em um contexto onde os recursos físicos eram muito limitados e caro. Com isso em mente, um dos seus objetivos era não sobrecarregar o *hardware* e passar o máximo possível de problemas para o *software*. Assim uma de suas memoráveis características é de que o conjunto de instruções possuía tamanho variado e realizava ações mais complexas com uma única instrução, o que muitas vezes levava a múltiplos ciclos de clock para se completar.

A arquitetura RISC (*Reduced Instruction Set Computer*) diferentemente da CISC, buscou uma abordagem de diminuição da complexidade, as instruções se tornaram mais simples e cada uma seria realizada em um único ciclo de clock. A abordagem RISC foi se tornando mais chamativa conforme o avanço da tecnologia, com as memórias ficando mais rápidas e com preços acessíveis. Logo a arquitetura fazia maior uso dos recursos de *hardware* do que de *software*.

3.3 Harvard vs Von Neumann

Uma outra forma de classificação de arquitetura é entre o modelo Harvard e Von Neumann. Esses modelos concretizaram uma ideia que já estava sendo vagamente difundida no momento: a de se carregar um programa em memória. A máquina proposta pelo modelo de Von Neumann já previa o uso dos principais componentes de um computador: a unidade lógica e aritmética (ULA), a unidade de controle (UC), a unidade central de processamento (CPU), que contém os 3 anteriores, a memória e um módulo de entrada e saída de dados (KOWALTOWSKI, 1996). O que é interessante é o fato de existir apenas dois barramentos de dados entre a CPU e a memória: um barramento para envio de dados e instruções e um barramento para endereçamento.

Para exemplificar a passagem da instrução de ínicio à fim, foi seguido o fluxo: leitura →decodificação →requisição de dados →ULA →armazenamento. Inicialmente um programa é carregado no módulo de memória, o passo seguinte é recuperar a próxima instrução armazenada, o que requer 1 acesso na memória durante a passagem de *clock* (Podendo ser na subida, ou descida do *clock*), essa instrução então é decodificada. A terceira etapa é buscar de onde esses dados virão, se é da memória, ou diretamente dos registradores, em seguida uma operação é realizada pela ULA e por fim esse resultado será salvo ou na memória ou em um registrador. Note que o acesso à memória pode ser realizado mais de uma vez por instrução (dependendo se os operandos virão da memória ou não) e que o modelo possuí um barramento para o envio de dados e instruções, logo seriam precisos mais do que 1 ciclo de *clock* para se completar a tarefa.

O modelo de Harvard é bem similar ao anterior, ele também possui os mesmos componentes principais, exceto que nesse caso a memória é dividida entre memória de

dados e memória de instruções. Essa alteração faz com que seja necessário mais dois barramentos, dessa forma seriam ao todo: dois barramentos entre a CPU e a memória de dados (Dados e endereçamentos), dois barramentos entre a CPU e a memória de instrução (Instruções e endereçamento) e um barramento entre a CPU e entrada/saída. As etapas que uma instrução percorre para a realização continuam as mesmas, a diferença nesse é que a instrução e os dados virão de memórias e barramentos separados, permitindo que esses processos ocorram simultaneamente e em uma quantidade reduzida de números de *clock*. Em geral 1 ciclo de *clock* é suficiente, mas essa determinação depende do modo como cada arquitetura é planejada, bem como seu conjunto de instruções.

3.4 Unidade Lógica e Aritmética (ULA)

A unidade lógica e aritmética é um dos principas elementos dentro de um processador, ela é responsável por calcular as operações matemáticas mais básicas e por realizar operações lógicas bit a bit. A quantidade de operações que ela é capaz de realizar ou a quantidade de bits de entrada e saída ficam a cargo de quem irá implementar essa unidade. Apesar disso, as operações mais comuns são: add, sub, AND, OR, XOR, shift esquerdo, shift direito.

Para a entrada, a ULA aceita 2 campos de dados com a mesma quantidade de bits. Ela também tem entradas para o fluxo de controle que vai determinar qual das possíveis operações será realizada, ou se não irá realizar nenhum cáclulo. Por último, a principal porta de saída é o resultado final com a mesma quantidade de bits dos elementos de entrada, e ela também pode apresentar um bit de saída sinalizador sobre Carry-out, Zero, negativo ou *overflow*.

3.5 Kit FPGA (Arranjo de Portas Programáveis em Campo)

O FPGA é um conjunto de circuitos lógicos, como o nome já diz, possíveis de serem programados. Mesmo após a finalização de um *design* carregado em FPGA, é possível alterar a sua programação. Além disso, a interação entre o desenvolvedor e o kit se dá nos limites entre *hardware* e *software*. É possível utilizar linguagens de descrição de *hardware* para o desenvolvimento de projetos. Essa linguagem é sintetizada dentro dos elementos lógicos do FPGA.

3.6 Verilog

É uma das linguagens de descrição de *hardware* mais populares devido a sua flexibilidade e robustez. O objetivo dessas lingugagens é facilitar a interação entre um projetista

e o *hardware*. Pessoas que já tiveram algum contato com linguagens de programação, como C/C++, devem reconhecer a semelhança entre as linguagens. Entretanto, diferentemente de uma linguagem de programação, a linguagem de descrição de *hardware* apenas indica o comportamento desejado de um circuito e posteriormente o circuito será sintetizado por outros meios. Dessa forma, Verilog ajuda na implementação de modelos para testes e simulações.

3.7 MIPS

O MIPS (*Microprocessor without interlocked pipeline stages*) é um nome dado a uma arquitetura de microprocessadores de propósito geral (FREITAS et al., 2010), é popularmente conhecida e usada em meios acadêmicos por seu caráter didático. Ela é voltada para se ter alto desempenho, o que envolve fazer grande uso do *hardware*, ter instruções simples e realizadas em um único ciclo de *clock*, dessa forma ela é do tipo RISC.

Visando a redução do tempo de processamento, as únicas operações que acessam a memória de dados (O que é custoso), são as operações de *load* e *store*, por isso é chamada de arquitetura *load-store*, enquanto as outras operações envolvem apenas valores imediatos e registradores (Os quais tem um tempo de acesso mais rápido).

A simplicidade e boa divisão do conjunto de instruções e das etapas de processamento permitiram que essa arquitetura desenvolvesse 3 modelos:

- Monociclo: Modo mais fácil de entender e desenvolver e possuí uma unidade de controle simples. Todo o processamento da instrução se dá em um ciclo de *clock*, porém o tempo do *clock* fica sujeito ao tempo necessário para se realizar a instrução mais lenta.

- Multiciclo: Nesse caso o processamento é dividido em 5 etapas onde cada etapa é realizada sequencialmente em cada ciclo de *clock*. Logo, cada instrução leva 5 ciclos de *clock* para terminar, porém o tempo do *clock* fica sujeito a etapa mais demorada.

- Pipeline: Modo mais complexo, faz grande uso do *hardware* pois pode chegar a realizar diferentes operações de até 5 instruções simultâneamente. O tempo de *clock* é o mesmo que no modo multiciclo, porém, em situações ideais, uma instrução é finalizada a cada ciclo.

4 Desenvolvimento

A ideia principal para o desenvolvimento foi a de se utilizar a arquitetura MIPS monociclo como base. Assim como o MIPS, a arquitetura segue a linha RISC no que se diz respeito a manter um formato de instruções de tamanho fixo e a respeito da ideia principal da maioria das instruções. Com relação à forma como a memória é dividida e acessada é de acordo com o modelo Harvard, mantendo dois módulos de memória separados, um para instruções e um para dados.

4.1 Aspectos gerais

- Arquitetura com base no modelo RISC
- Arquitetura Harvard
- Little endian
- Monociclo

4.2 Registadores

Os registradores serão frequentemente utilizados em operações, principalmente por serem de rápido acesso e fácil manipulação. Os registradores podem fazer operações entre si, entre imediato, e armazenam ou enviam dados para a memória principal. O banco de registradores é composto por 32 bits x 32 registradores de propósito geral. Outros registradores específicos também se destacam, como $PC(ProgramCounter)$, um registrador interno responsável por apontar os endereços de instruções.

4.3 ISA

Após a decisão inicial de se tomar a arquitetura MIPS como modelo inicial, foi definido o conjunto de instruções, as quais foram divididas em 5 diferentes classes. A separação nos diferentes grupos foi feito de modo a facilitar a compreensão do que acontecerá internamente ao processador já que para os humanos é mais fácil agrupar elementos semelhantes e entender suas características de conjunto do que compreender isoladamente o funcionamento de cada um, como um robô faz. Do ponto de vista do processador essa separação não tem valor lógico. Os grupos estão listados abaixo:

1. Aritmético: Usado em operações matemáticas simples
2. Comparação: Específico para fazer comparações entre dados
3. Movimentação: Acesso ou armazenamento de dados
4. Bitwise: Cálculos bit a bit
5. Desvio: Alteração no fluxo de leitura das próximas instruções

Todas as instruções que serão utilizadas no processador estão registradas nas Tabelas [1, 2, 3, 4, 5]. Cada tabela se refere a um dos conjuntos já citados anteriormente. O primeiro campo contém o nome de cada operação, o segundo se refere ao apelido que é usado para a programação. Já o terceiro campo contém o tipo de endereçamento usado. A operação mostra o que será realizado. O campo de *OpCode* é um valor binário para que a máquina possa fazer diferenciação entre as instruções. Por fim o campo de *funct* nas Tabelas [1,2,3] pode ser entendido como uma extensão do *OpCode* para diferenciação de subinstruções.

Tabela 1 – Instruções do tipo Aritmético

Nome	Mnemônico	Endereçamento	Operação	Funct OpCode
Adição	add	Registrador	r1 ← r1 + r2	00001 000001
Subtração	sub	Registrador	r1 ← r1 - r2	00010 000001
Adição Imediato	addi	Imediato	r1 ← r1 + Im	00011 000001
Subtração Imediato	subi	Imediato	r1 ← r1 - Im	00100 000001

Tabela 2 – Instruções do tipo Bitwise

Nome	Mnemônico	Endereçamento	Operação	Funct OpCode
AND	AND	Registrador	r1 ← r1 && r2	00001 000010
OR	OR	Registrador	r1 ← r1 | | r2	00010 000010
NOT	NOT	Registrador	r1 ← !r1	00011 000010
XOR	XOR	Registrador	r1 ← r1 ^r2	00100 000010
ANDi	ANDi	Registrador	r1 ← r1 && Im	00101 000010
ORi	ORi	Registrador	r1 ← r1 | | Im	00110 000010
NOTi	NOTi	Registrador	r1 ← !Im	00111 000010
XORi	XORi	Registrador	r1 ← r1 ^Im	01000 000010
shift left	shiftL	Imediato	r1 ← r1 << shamt	01001 000010
shift right	shiftR	Imediato	r1 ← r1 >> shamt	01010 000010

Tabela 3 – Instruções do tipo Comparação

Nome	Mnemônico	Endereçamento	Operação	Funct OpCode
Less	less	Registrador	rf $\leftarrow$ (r1 $<$ r2)?	00001 000011
Grand	grand	Registrador	rf $\leftarrow$ (r1 $>$ r2)?	00010 000011
Equal	eq	Registrador	rf $\leftarrow$ (r1 $=$ r2)?	00011 000011
Not Equal	neq	Registrador	rf $\leftarrow$ (r1 $\neq$ r2)?	00100 000011
Less Equal	leq	Registrador	rf $\leftarrow$ (r1 $\leq$ r2)?	00101 000011
Grand Equal	geq	Registrador	rf $\leftarrow$ (r1 $\geq$ r2)?	00110 000011
Less Immediate	lessi	Imediato	rf $\leftarrow$ (r1 $<$ Im)?	00111 000011
Grand Immediate	grandi	Imediato	rf $\leftarrow$ (r1 $>$ Im)?	01000 000011
Equal Immediate	eqi	Imediato	rf $\leftarrow$ (r1 $=$ Im)?	01001 000011
Not Equal Immediate	neqi	Imediato	rf $\leftarrow$ (r1 $\neq$ Im)?	01010 000011
Less Equal Immediate	leqi	Imediato	rf $\leftarrow$ (r1 $\leq$ Im)?	01011 000011
Grand Equal Immediate	geqi	Imediato	rf $\leftarrow$ (r1 $\geq$ Im)?	01100 000011

Tabela 4 – Instruções do tipo Movimentação

Nome	Mnemônico	Endereçamento	Operação	OpCode
Move	mv	Registrador	r1 = r2	000100
Move Immediate	mvi	Imediato	$r1_{15:0} = Imm_{16}$	000101
Store word	sw	Registrador Base	r2 $\rightarrow$ Mem(r1 + Desl)	000110
Load word	lw	Registrador Base	r1 $\leftarrow$ Mem(r1 + Desl)	000111
Load up	lup	Imediato	$r1_{31:16} \leftarrow Im_{16}$	001000
Load down	ldown	Imediato	$r1_{15:0} \leftarrow Im_{16}$	001001

Tabela 5 – Instruções do tipo Desvio

Nome	Mnemônico	Endereçamento	Operação	OpCode
Jump	jump	Indireto por reg	PC ← r1	001010
Jump and Link	jal	Indireto por reg	PC ← r1, ra←PC + 1	001011
Jump Conditional	jc	Indireto por reg	se rf = 1 então PC ← r1	001100
Branch	branch	Relativo ao PC	PC ← PC + Im + 1	001101
Branch and Link	bal	Relativo ao PC	PC ← PC + Im + 1, ra←PC + 1	001110
Branch Conditional	bc	Relativo ao PC	se rf = 1 então PC ←PC + Im + 1	001111
Input	get	—	Mem(r1 + Desl) <- in	010000
Output	print	—	Mem(r1 + Desl) -> out	010001
NOP	NOP	—	—	000000
STOP	STOP	—	—	111111

Tendo todas as instruções que serão utilizadas, o próximo passo é definir os formatos que serão necessários e suficientes para suportar as ações. Como a maioria das instruções seguem o modelo do MIPS, é comum que o formato das instruções também seja semelhante, embora não exatamente iguais.

Todos os formatos de instrução tem como primeiro campo o *OpCode*, seguido do endereço de um registrador *r1*, os próximos campos são diferentes de acordo com cada formato.

O formato I ainda possui um campo para o endereço de um segundo registrador *r2*, um campo chamado de *shamt* utilizado em instruções *shift* e o campo de *funct*, fora 6 bits que não foram utilizados. Fazem parte desse formato as instruções: *add*, *sub*, *AND*, *OR*, *NOT*, *XOR*, *less*, *grand*, *eq*, *neq*, *leq*, *geq*, *shiftL*, *shiftR*.

O formato II ainda possui um campo Imm_{16} representando um valor imediato de 16 bits e o campo *funct*. Fazem parte desse formato as instruções: *mvi*, *addi*, *subi*, *ANDi*, *ORi*, *NOTi*, *XORi*, *lessi*, *grandi*, *eqi*, *neqi*, *leqi*, *geqi*, *lup*, *ldown*.

O formato III possui também um segundo registrador *r2* e um campo de deslocamento com 16 bits. Fazem parte desse formato as instruções: *mv*, *jump*, *jal*, *jc*, *branch*, *bal*, *bc*, *sw*, *lw*, *get*, *print*, *NOP*, *STOP*.

No total serão implementadas 42 instruções.

Tabela 6 – Formato de instruções tipo I

6 bits	5bits	5 bits	5 bits	6 bits	5 bits
OpCode	$r1$	$r2$	*shamt*	—	*funct*

Tabela 7 – Formato de instruções tipo II

6 bits	5 bits	16 bits	5 bits
OpCode	$r1$	Imm_{16}	*funct*

Tabela 8 – Formato de instruções tipo III

6 bits	5 bits	5 bits	16 bits
OpCode	$r1$	$r2$	Desl

Ainda relacionado as instruções, foi dito anteriormente que para as intruções das Tabelas [1, 3, 2] o campo de *funct* e *OpCode* são combinados para formar instruções mais específicas internas à ULA. A combinação desses códigos é convertido em um sinal da ULA, chamado de *AluOp* e o mapeamento das ações com base nesse novo sinal é dado pela Tabela 9.

Tabela 9 – Mapeamento de operações do sinal da ULA

Nome	AluOp
add	00001
sub	00010
AND	00011
OR	00100
NOT	00101
XOR	00110
shiftL	00111
shiftR	01000
Less	01001
Grand	01010
Equal	01011
Not Equal	01100
Less Equal	01101
Grand Equal	01110
Load Up	01111

4.4 Caminho de dados

Nessa seção é mostrado gradativamente o caminho de dados desenvolvido, inicialmente o esquemático possui os módulos principais, com exceção do módulo de controle de

sinais (Unidade de Controle) e o módulo de entrada e saída. Em sequência são realizadas algumas instruções por etapas e os elementos faltantes no diagrama são adicionados conforme a necessidade.

Para simplificação, alguns barramentos não são conectados no começo. Ainda sobre *bus*, a quantidade total de bits que são transferidos por algum módulo pode ser verificada logo após a porta de saída, ao lado de um risco diagonal no barramento. Enquanto que próximo às portas de entrada podem ser conferidos os índices dos bits no formato $x : y$, com x indicando o bit mais significativo e y o bit menos significativo.

Os *mux's* são componentes que aparecem constantemente devido sua propriedade de escolha. Para diferenciação eles serão reconhecidos pelos seus sinais de controle.

O PC (*Program Counter*) é um registrador responsável por apontar o endereço das intruções. Nos esquemáticos que se seguem, PC representa o endereço da instrução que está sendo processada naquele ciclo de *clock*, enquanto PC' representa o endereço da instrução do ciclo de *clock* seguinte. A Figura 1 mostra que PC é incrementado em 1 para obter o próximo endereço, ou seja, nesse caso $PC' = PC + 1$.

4.4.1 Operações Registrador - Registrador e Registrador - Imediato

O primeiro módulo principal é a memória de instrução, ela possui apenas uma entrada I (*Instruction*) representando o endereço de instrução alimentado por PC. Internamente a instrução é buscada no enderço especificado, esse dados de instrução representado por ID (*Instruction Data*) é quebrado em partes menores de saída e alimentam outros módulos.

Tomando a instrução *add* como caso teste, relembrando da Tabela 6, a operação envolve dois registradores $r1$ e $r2$ que são recuperados pelos endereços passados em $Instruction_{25:21}$ e $Instruction_{20:16}$ repectivamente, no banco de registradores pelas entradas $RA1$ (*Register Address 1*) e $RA2$ (*Register Address 2*). O banco de registradores ainda possui duas portas de saída $DR1$ (*Data Register 1*) e $DR2$ (*Data Register 2*) que repassam os dados recuperados no endereço previamente recebido. A próxima etapa envolve a ULA, que por sua vez recebe os dois operandos e um sinal de controle *AluOp* para diferenciar qual operação interna irá realizar. Por fim o resultado da operação deverá ser retornado para o banco de registradores pela porta $DR1'$ a fim de substituir o valor inicial $DR1$. Note que o banco de registradores recebe um sinal $RegW$ na porta EW(*Enable Write*) para, nesse caso, permitir a escrita.

Com essa estrutura é possível realizar funções entre registradores dos tipos aritméticos e booleanos. Para extender essas funções para suportar operandos imediatos também, o *mux DataOp* serve de seleção entre imediato e registrador, além disso um extensor de sinal é necessário uma vez que o valor imediato incial tem 16 bits e as operações ocorrem

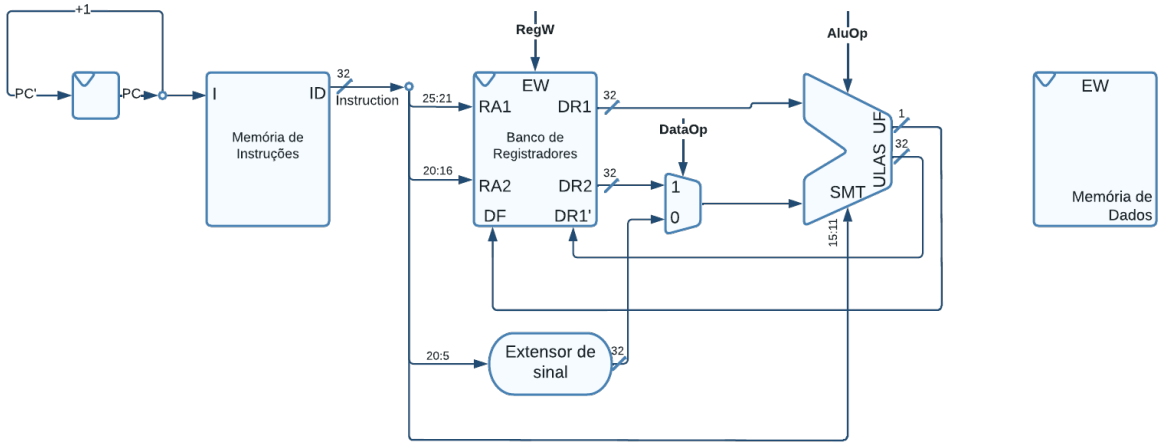
sobre 32 bits.

Para *shiftL* e *shiftR* a ULA ainda recebe um valor na porta SMT indicando a quantidade de bits a serem movidos para a esquerda ou para a direita.

Para finalizar o caminho da Figura 1, a ULA também realiza as operações de comparação, para isso, é feito uma subtração entre os operandos. O resultado da operação em combinação com o tipo de comparação feita (menor, menor igual, igual, maior, maior igual) gera um valor binário de 1 bit emitido na porta de saída UF (*ULA Flag*) e armazena no banco de registradores na porta DF (*Data Flag*).

Com essas conexões é possível realizar todas as instruções do tipo Aritmético, Bitwise, Comparação e as instruções *mv*, *lup* e *ldown*.

Figura 1 – Caminho de dados com principais módulos e com conexões iniciais para primeiras instruções



Fonte: Autor

4.4.2 Acesso e salvamento na memória de dados

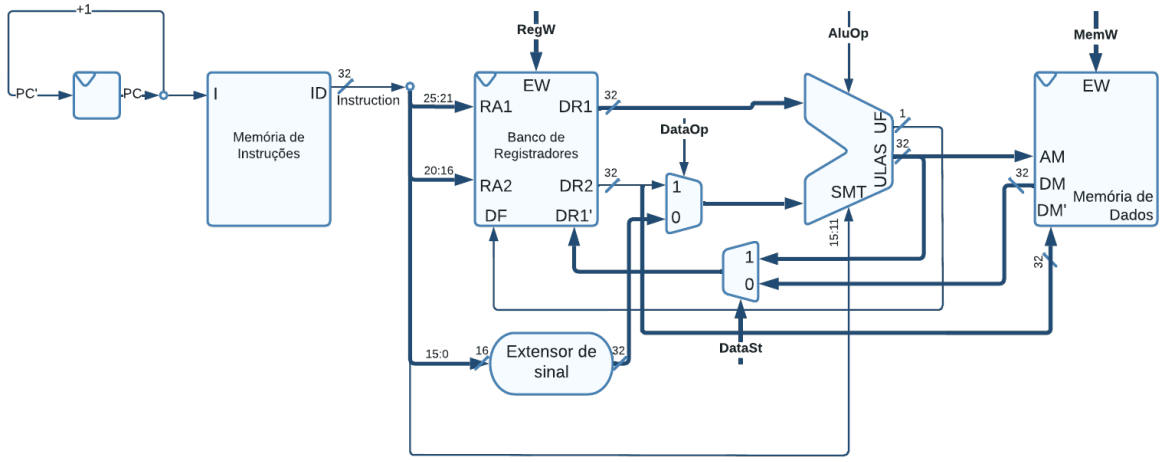
Até então o acesso e salvamento de dados foi feito somente usando registradores e valores imediatos, será abrangido agora também os dados em memória, o que envolve basicamente as instruções *lw* e *sw*. A Figura 2 auxilia no entendimento dos próximos passos.

Nesse caso, se trata de um endereçamento por registrador base, logo o valor do registrador $r1$ é somado a um valor imediato de deslocamento (Note que $DataOp = 0$ para escolha do imediato) na ULA produzindo o endereço final. Esse resultado é passado para a porta de entrada AM (*Address Memory*) da memória de dados. O próximo passo depende se irá ser armazenazado ou recuperado um valor desse módulo.

No caso de *lw*, o dado deve ser armazenado em um registrador. Dessa forma, a porta de saída DM ($DataMemory$) envia o valor. Como $r1$ pode ser escrito tanto por um valor vindo da memória quanto como resultado da ULA, o *mux* $DataSt$ faz o controle de qual a origem do dado, nessa situação $DataSt = 0$ e o dado virá da memória.

No caso de *sw*, o dado do registrador $r2$ será armazenado no módulo de memória. Aqui o dado é passado para a porta de entrada DM' e o sinal $MemW$ ($Memory\ Write$) será ativado.

Figura 2 – Novas conexões e destaque nos principais sinais para suportar instruções *lw* e *sw*



Fonte: Autor

4.4.3 Desvios Incondicionais

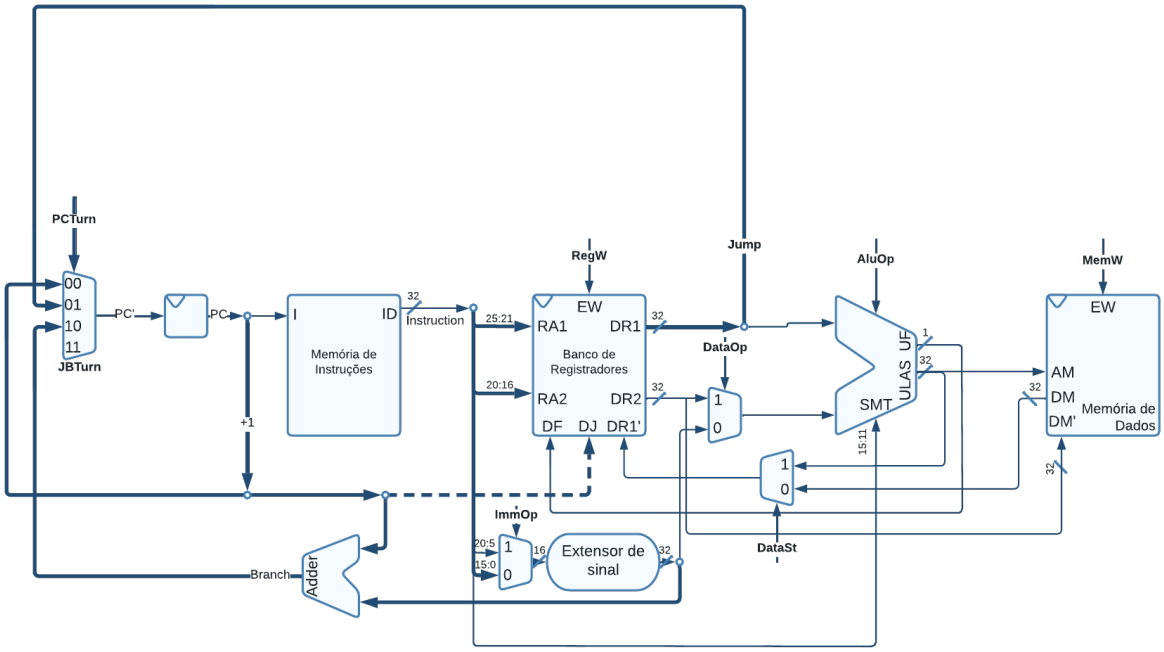
O restante das instruções principais são do tipo de desvio, o que implica em dizer que a partir de então nem sempre $PC' = PC + 1$. Além disso, não há nenhum tipo de operação lógica ou aritmética e também não há acesso a memória de dados.

Relembrando, a instrução de *jump* toma um desvio baseado no endereço armazenado em um registrador (endereçamento indireto por registrador), dessa forma o valor de $r1$ simplesmente é passado para o *mux* $PCTurn$ e selecionado como opção de PC' como mostra a Figura 3.

Já para a tomada do *branch*, o modo de endereçamento é relativo ao PC. Primeiramente é importante notar que instruções aritméticas e de desvios fazem o uso de imediato com endereços de campo distintos, por isso o *mux* $ImmOp = 0$ seleciona o campo do imediato de deslocamento. Em seguida o imediato com 16 bits é extendido para os 32 bits, em sequência ele é somado juntamente com o valor de $PC + 1$, por fim o resultado é selecionado pelo *mux* $PCTurn$ e se torna PC'.

Com poucas alterações é possível extender essa implementação para as instruções *jal* e *bal*, basta acresentar um caminho (Linha em pontilhado na Figura 3) que será responsável por transferir o valor do endereço $PC + 1$ para a porta DJ no banco de registradores, onde será armazenado em um registrador específico.

Figura 3 – Caminho de dados para instruções *jump*, *jal*, *branch*, *bal*



Fonte: Autor

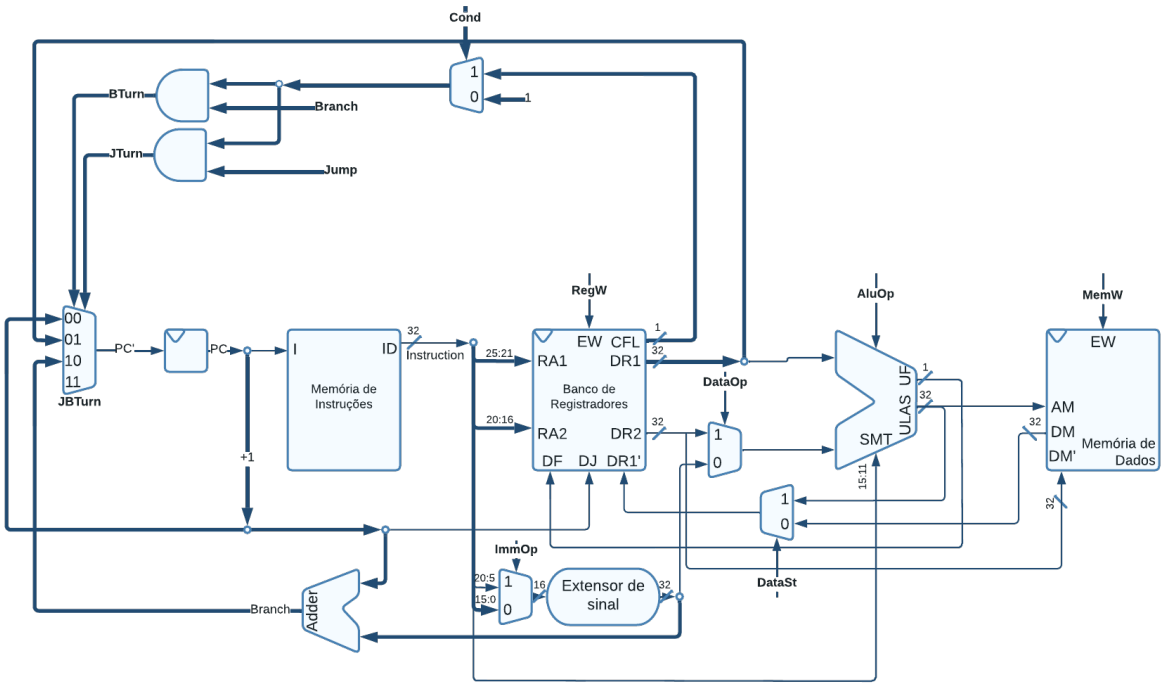
4.4.4 Desvios Condicionais

Por fim sobraram as instruções *jc* e *bc*. Para entender como foi alterada a estrutura para incrementar essas instruções, primeiro deve-se melhorar a ideia por trás do sinal $PCTurn$ (O qual seleciona quem será PC'). Como ele está representando dois bits, o sinal de controle foi separado em 2 entradas distintas: $JTurn$ e $BTurn$ e o *mux* em questão terá o nome $JBTurn$ como mostra a Figura 4. Dessa forma sempre que a unidade de processamento for tomar um desvio de *jump*, o sinal $JTurn$ estará ativo, enquanto $BTurn$ estará em baixo, e sempre que for tomado um desvio de *branch* os sinais se invertem, além disso, quando os dois sinais tiverem valor 0 então segue-se o fluxo sequencial onde $PC' = PC + 1$, e é esperado que ambos sinais nunca estejam em 1, se não estará em um estado inválido.

Combinados com esse *mux*, destacam-se 3 componentes da arquitetura: 2 portas lógicas AND e o *mux Cond* (*Conditional*). As duas portas AND's recebem em uma de suas entradas um sinal de desvio *Jump*/*Branch* e um valor do *mux Cond* em outra. Note que se o *mux* receber um sinal de controle 0, indicando que não é uma operação condicional,

então o valor 1 é repassado adiante e assim o resultado das portas AND's depende somente dos sinais *Jump* e *Branch*, e a arquitetura funcionará exatamente como estava funcionando com as funções de desvios incondicionais já demonstradas. Por outro lado, se o valor de controle do *mux* for igual a 1, então estará selecionando desvios condicionais. Assim, uma *flag* condicional vinda do banco de registradores pela porta de saída CFL (*Conditional Flag*) irá servir de entrada para as portas *AND's*. Logo um desvio condicional ocorrerá se e somente se a *flag* de condição for verdadeira e uma operação de desvio estiver selecionada, caso contrário $JBTurn = 00$ e o fluxo de instrução padrão $PC' = PC + 1$ será tomado.

Figura 4 – Caminho de dados para todas as instruções com destaque para desvios condicionais



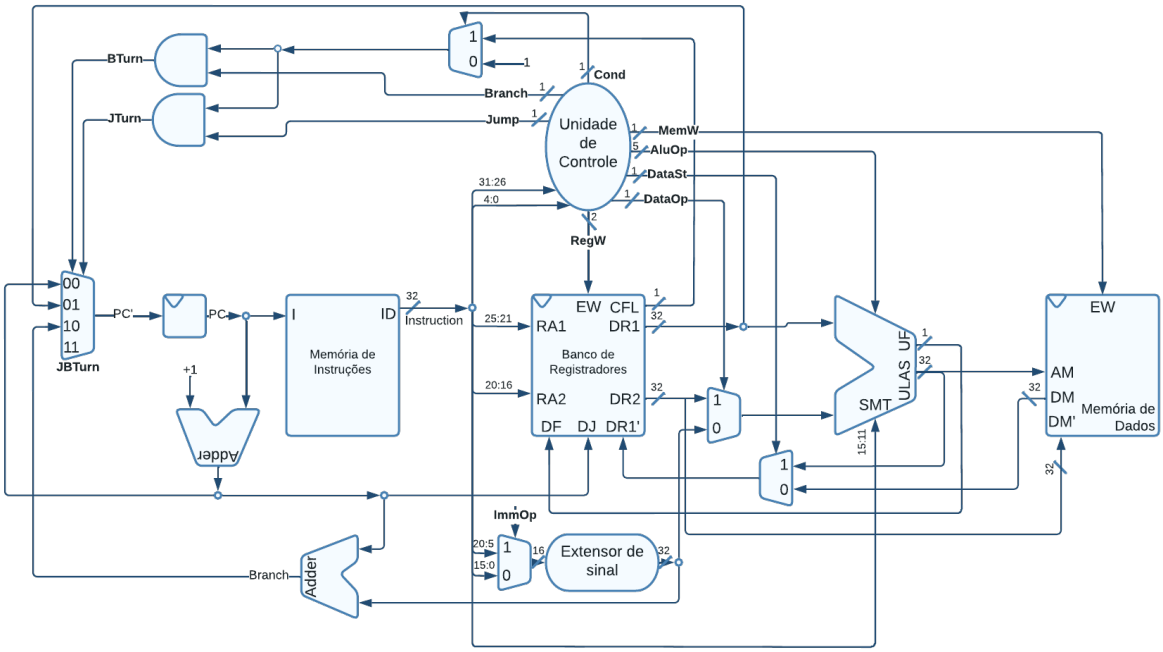
Fonte: Autor

4.4.5 CPU

A Figura 5 mostra o driagrama da cpu (ainda sem o módulo de entrada e saída) suas conexões. A Unidade de Controle e todos os sinais e caminhos vindos dela são exibidos, não há destaque em nenhum caminho específico e está explicitado o componente *Adder* para o incremento de 1 no endereço de instrução de PC.

As únicas duas portas de entrada da UC são alimentadas pelos campos de $Instruction_{31:26}$ e $Instruction_{4:0}$ que correspondem ao *OpCode* e *funct*.

Figura 5 – Caminho de dados completo sem módulo de entrada e saída



Fonte: Autor

4.4.6 Temporização e entrada e saída de dados

Para se ter o controle do momento correto em que os dados são acessados, lidos e escritos, é importante que o sistema seja síncrono, ou seja, atualize seus dados com base no *clock*. Para esse projeto, foi reservado a subida do *clock* para leitura, enquanto a borda de descida serve para fazer a escrita dos dados.

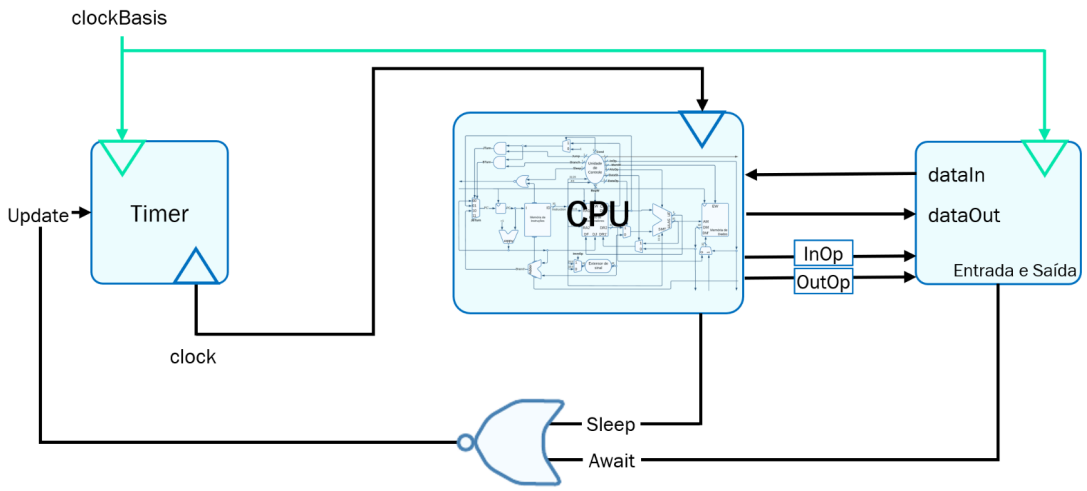
Até então, todas as instruções desenvolvidas são instruções que realizam operações entre seus próprios dados já armazenados e sem interação externa. Para que o *design* final esteja completo, é preciso que instruções de interação com o usuário também sejam suportadas: *get*(*input*) e *print*(*output*)

A Figura 6 mostra como o módulo I/O, a cpu e interrupções do sistema se interligam.

Nessa etapa é importante notar que há dois *clocks* distintos: o *clockBasis*, como o nome já diz, serve como base principal, ele é o *clock* mais rápido e rege o sincronismo do módulo de I/O. O segundo temporizador é o *clock* da cpu, ele é gerado dentro do módulo *Timer*, a partir do *clockBasis* e, para esse projeto, a sua frequência é a metade da frequência do *clockBasis*. Esses fatos são interessantes pois nos diz que a entrada e a saída de dados no nível do usuário ocorrem mais rápido do que a troca de instruções (*clock* $<$ *clockBasis*), o que é algo desejado, pois é mais confortável ver os resultados em uma tela/display antes que outras instruções ocorram, evitando o *delay*.

Apesar da simplicidade do módulo *Timer*, ele permite uma funcionalidade impor-

Figura 6 – Temporizador e entrada e saída



Fonte: Autor

tante. Dado o valor do sinal *update* = 0, o módulo mantém o estado do *clock* "congelado", enquanto o *clockBasis* continua normalmente; ou para *update* = 1, ambos os *clocks* funcionarão normalmente. Essas características permitem o uso fácil de interrupções.

As interrupções podem ocorrer em duas situações: quando a própria cpu envia a instrução *STOP* a fim de parar seu processamento definitivamente; ou quando a instrução *get* é utilizada e está esperando por um *input* do usuário. Nesse último a interrupção é temporária. A porta NOR expressa a interrupção dos dois casos.

Além dos dois *clocks*, a cpu se conecta com o módulo de entrada e saída por 4 conexões: 2 para sinais de controle (*InOp* e *OutOp*) e 2 para envio e recebimento de dados (*dataIn* e *dataOut*).

Baseado nas instruções, temos que essas conexões funcionam de tal modo:

- *print*: Sinal *OutOp* = 1, e módulo I/O recebe *dataOut*. Após o recebimento, *OutOp* = 0
- *get*: Sinal *InOp* = 1, *await* = 1 causando interrupção na cpu até que o usuário envie os dados através do *dataIn*. Após o envio, *await* = 0 e *InOp* = 0
- *STOP*: Sinal *sleep* = 1, causando interrupção definitiva do sistema

4.5 Códigos principais

4.5.1 Memória de Instrução

Código da memória de instrução

```
module MI(
        // IN
        input[31:0] I,  // Endereco da instrucao

        // OUT
        output[31:0] ID // Instrucao

        );

        reg[31:0] mem_i[31:0];  // Memoria de fato

        initial begin
        /*
        * Put codes in here
        */
        end

        assign ID = mem_i[I];   // Repassa dado do endereco I

endmodule
```

4.5.2 Banco de Registradores

Código do banco de registradores

```
module BR(
        // IN
        input[31:0] DR1_,         // Dado para escrita
        input[31:0] DJ,           // Data Jump
        input[ 4:0] RA1, RA2,     // Enderecos reg 1 e 2
        input[ 1:0] EW,           // Sinal de escrita
        input DF,                 // Data Flag

        // OUT
        output[31:0] DR1, DR2,    // Dado reg 1 e 2
        output CFL,               // Dado armazenado em flag

        // TIMER
        input clk                 // Clock
);

        // REGISTRADORES
        reg[31:0] register[31:0];  // 32x32 regs
        reg[31:0] RF;              // Register Flag
        parameter AJ = 5'b11111;   // Endereco Reg RJ


        always @(negedge clk) begin

                case(EW)
                // 2'b00 : Nao escrever nada
                        2'b01 : RF[0] <= DF;             // Escrever em Reg RF[0]
                        2'b10 : register[AJ] <= DJ;      // Escrever em Reg RJ
                        2'b11 : register[RA1] <= DR1_;   // Escrever em reg RA1
                        default: ;
                endcase

        end
```

```
34
35          // Repassa dados dos regs
36          assign DR1 = register[RA1];
37          assign DR2 = register[RA2];
38          assign CFL = RF[0];
39
40  endmodule
```

4.5.3 ULA

Código da ULA

```
1   module ULAS(
2           // IN
3           input[31:0] op1, op2,    // Operandos
4           input[4:0] smt,          // Shift amount
5           input[4:0] aluop,        // Controle
6
7           // OUT
8           output reg [31:0] r1,    // Resultado
9           output reg UF            // Flag
10          );
11
12          reg of;
13
14          always @* begin
15
16                  case(aluop)
17                          // ARITMETICO
18                          5'b00001 : begin
19                                  r1 = op1 + op2;
20                                  of = ((~op1[31] & ~op2[31] & r1[31]) | (op1[31] & op2[31]
                                        & ~r1[31]));  // OverFlow
21                                  UF = of;
22                          end   // Add
23
24
25                          5'b00010 : begin
26                                  r1 = op1 - op2;
27                                  of = ((~op1[31] & ~op2[31] & r1[31]) | (op1[31] & op2[31]
                                        & ~r1[31]));  // OverFlow
28                                  UF = of;
29                          end   // Sub
30
31                          // LOGICO
32                          5'b00011 : begin r1 = op1 & op2;    UF = 1'b0; of = 0; end   //
                                AND
33                          5'b00100 : begin r1 = op1 | op2;    UF = 1'b0; of = 0; end   //
                                OR
34                          5'b00101 : begin r1 = ~op1;         UF = 1'b0; of = 0; end   //
                                NOT
35                          5'b00110 : begin r1 = op1 ^ op2;    UF = 1'b0; of = 0; end   //
                                XOR
36                          5'b00111 : begin r1 = op1 << smt;   UF = 1'b0; of = 0; end   //
                                shiftL
37                          5'b01000 : begin r1 = op1 >> smt;   UF = 1'b0; of = 0; end   //
                                shiftR
38
```

```
39                          // COMPARACAO
40                          5'b01001 : begin UF = (op1 < op2);  r1 = 32'b0; of = 0; end  //
                                Less
41                          5'b01010 : begin UF = (op1 > op2);  r1 = 32'b0; of = 0; end  //
                                Grand
42                          5'b01011 : begin UF = (op1 == op2); r1 = 32'b0; of = 0; end  //
                                Equal
43                          5'b01100 : begin UF = (op1 != op2); r1 = 32'b0; of = 0; end  //
                                Not Equal
44                          5'b01101 : begin UF = (op1 <= op2); r1 = 32'b0; of = 0; end  //
                                Less Equal
45                          5'b01110 : begin UF = (op1 >= op2); r1 = 32'b0; of = 0; end  //
                                Grand Equal
46
47                          // LoadUp
48                          5'b01111 : begin
49                                  UF = 0;
50                                  of = 0;
51                                  r1 = op2 << 16;
52                          end
53
54                          default  : begin
55                                  r1 = op2;
56                                  UF = 0;
57                                  of = 0;
58                                  end
59                  endcase
60          end
61
62  endmodule
```

4.5.4 Memória de dados

Código da memória de dados

```
1   module MD(
2           // IN
3           input [31:0] AM,  // Endereco
4           input [31:0] DM_, // Dado de escrita
5           input EW,         // Sinal para escrita
6
7           // OUT
8           output[31:0] DM,  // Dado de leitura
9
10          // TIMER
11          input clk         // clock
12  );
13
14          reg[31:0] mem_d[31:0];  // Memoria de fato
15
16          always @(negedge clk) begin
17
18                  case (EW)
19                          1'b1 : mem_d[AM] <= DM_;  // Escrita habilitada
20                  endcase
21
22          end
23
```

```
24          assign DM = mem_d[AM];  // saida
25
26  endmodule
```

4.5.5 Entrada e saída de Dados (IO)

Como o módulo de entrada e saída é o que mais se distingue do modelo MIPS original, é mostrado o código e ele será melhor detalhado a seguir.

A instrução de *input* suportada pelo módulo é a instrução *get*: ela pega um dado em binário fornecido pelo usuário nas portas *in* e repassa o dado pela porta *du* para que a cpu recolha essa informação e salve na memória de dados.

Já a instrução de *output* suportada pelo módulo é a instrução *print*: ela pega um dado *dm* em binário da cpu (da memória de dados) e envia o resultado para 4 *displays* de 7 segmentos. O resultado é imprimido nos *displays*.

Código do módulo de *IO*

```
1   module IO(
2           // IN
3           input inop,
4           input outop,
5           input bt,
6
7           input[13:0] in,
8           input[31:0] dm,
9
10
11          // OUt
12          output[31:0] du,        // Data from user
13          output[27:0] display,
14          output reg await,
15
16          // TIMER
17          input clk,
18          input clk_state
19
20  );
21
22          // Button
23          wire bt_high;
24          reg[31:0] inbuff;
25          reg[31:0] outbuff;
26
27          // Button pressed
28          debounce db(.bt(bt), .out(bt_high), .clk(clk));
29
30          initial begin
31                  inbuff = 32'b00000000000000000000000000000000;
32                  outbuff = 32'b00000000000000000000000000000000;
33          end
34
35          always @(negedge clk) begin
36
```

```
37                  // Input
38                  if (inop) begin
39
40                          // Button pressed
41                          if (bt_high) begin
42                                  inbuff = in[13:0];
43                                  outbuff = outbuff;
44                                  await = 1'b0;
45                          end
46
47                          // Button not pressed
48                          else begin
49
50                                  // Se clock esta em nivel alto
51                                  if (clk_state) begin
52                                          await = 1'b1;
53                                          inbuff = inbuff;
54                                          outbuff = outbuff;
55                                  end
56
57                                  // Se clock esta em nivel baixo
58                                  else begin
59                                          await = await;
60                                          inbuff = inbuff;
61                                          outbuff = outbuff;
62                                  end
63                          end
64                  end
65
66                  // Output
67                  else if (outop) begin
68                          inbuff = inbuff;
69                          outbuff = dm;
70                          await = 1'b0;
71                  end
72
73                  // Nothing
74                  else begin
75                          inbuff = inbuff;
76                          outbuff = outbuff;
77                          await = 1'b0;
78                  end
79
80          end
81
82          // Display
83          bin2display b2d(.addr(outbuff[13:0]), .clk(clk), .data(display));
84
85          assign du = inbuff;
86
87  endmodule
```

Algumas nomenclaturas do código são explicadas abaixo:

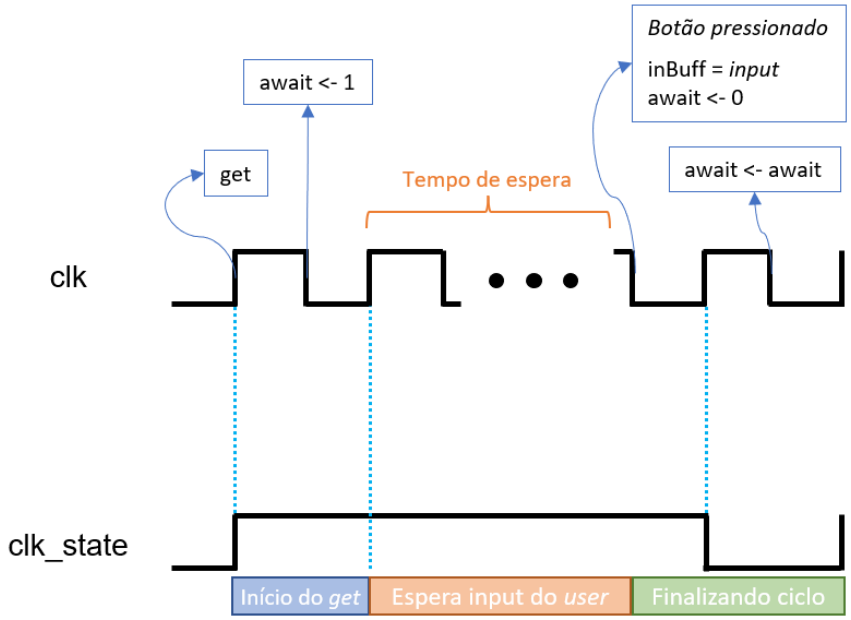
- *inop*/*outop*: Sinais de controle indicando que é uma instrução de *input*/*output*
- *bt*: Sinal indicando que um botão físico foi pressionado por um usuário para enviar

os dados de *in*

- *in*: dado em binário do usuário
- *dm*: dado vindo da memória de dados que será mostrado em um *display* através da instrução de *print*
- *clk*: *clock* do módulo de *IO*, ou seja, é o mesmo que *clockBasis*
- *clk_state*: é o estado atual do *clock* da cpu

A Figura 7 ajuda a entender o funcionamento do código com base no comportamento esperado para a instrução *get*.

Figura 7 – Planejamento do módulo de entrada e saída para instrução *get*



Fonte: Autor

Para a instrução *get*: Primeiramente, o sinal de *await* é atualizado em alto indicando que está aguardando o *input* do usuário. Assim que o usuário envia os dados para o *buffer*, o sinal de *await* é atualizado com 0, e o *clk_state* volta a seu funcionamento normal, passando seu sinal para nível lógico baixo. Durante esse ciclo, o sinal de *await* deve manter seu último valor (valor baixo), e por fim a instrução acaba.

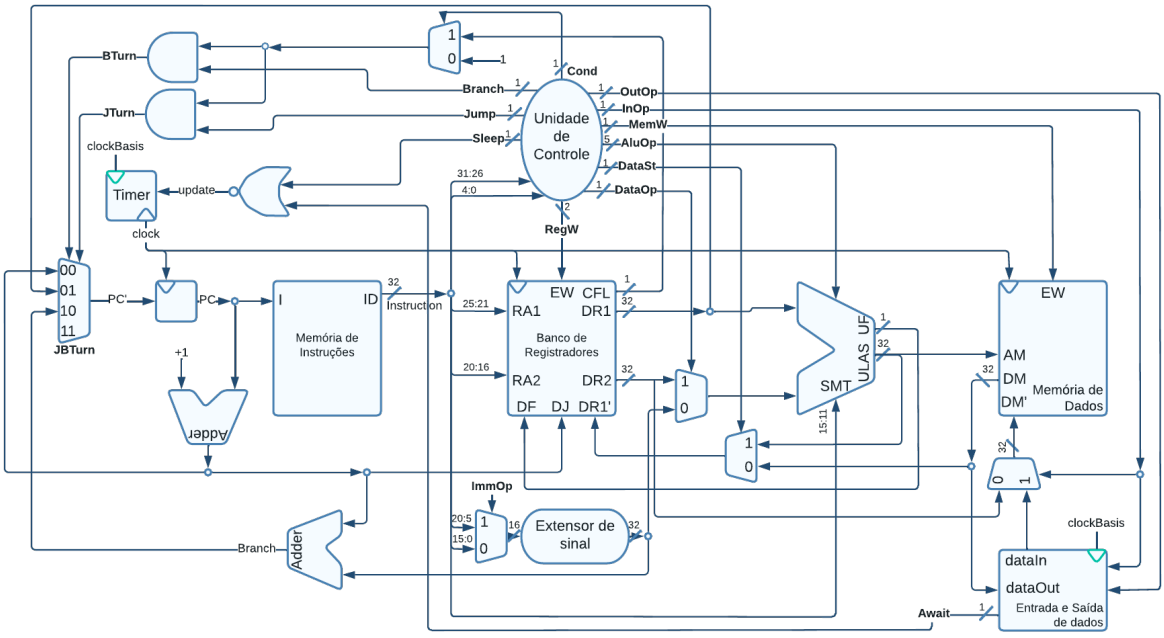
Para a instrução *print*: Nesse caso não há a necessidade de interrupções, dispensando o uso do *await*. Ela simplesmente escreve o dado *dm* no buffer de saída. Esse dado é automáticamente enviado através de um circuito combinacional para os *displays*.

Em todos os outros casos, os valores dos *buffers* e do sinal de *await* são mantidos de acordo com seus últimos valores atualizados.

5 Resultados Obtidos e Discussões

A Figura 8 mostra o *design* final e completo, sem conexões ocultas e com o módulo de entrada e saída em conjunto com a cpu.

Figura 8 – Design IMIPs completo



Fonte: Autor

Como já foi dito, a arquitetura se baseia no modelo do MIPS monociclo, contudo existem suas peculiaridades. Uma diferenciação marcante é em relação ao formato de instruções. Como sabemos, o MIPS tem formato para suportar até 3 endereços de registradores em uma linha de instrução, enquanto que para a nova arquitetura são passados no máximo 2 endereços de registradores por instrução. Nesse caso, o resultado gerado por operações que ocorram entre dois registradores é armazenado sempre no endereço do primeiro registrador passado. Essa característica é um legado da arquitetura x86.

Sobre o formato de instruções, é importante saber exatamente a quantidade de bits para cada campo, além da quantidade de elementos que eles podem representar e a quantidade que já está sendo representada. Para manter o controle dessas informações, a Equação 3.1 é essencial. Como é possível ver nas Tabelas [6, 7, 8] todos os formatos de instruções começam com um campo de *OpCode* de 6 bits assim como o MIPS padrão. Esse campo suporta indexar até 64 diferentes instruções. Dado o total de 39 instruções previstas, os 6 bits para esse mapeamento é suficiente e ainda sobra uma margem para novas inserções futuras se necessário. Entretanto é importante relembrar que o formato de

instruções do tipo I e do tipo II, como mostram as Tabelas [6, 7] possuem o campo *funct* com 5 bits para indexação de operações dentro da ULA. Isso leva à 32 possíveis operações indexadas internas à ULA.

Os grupos de instruções Aritmético, Bitwise e Comparação somam ao todo 26 instruções e são os que ativam operações da ULA, eles usam apenas 3 valores de *OpCode* para a identificação de cada grupo enquanto a diferenciação interna se dá pelos valores de *funct*. Com isso, as 39 instruções que antes poderiam ser indexadas caíram para 16 (13 instruções e mais 3 conjuntos) elementos para serem mapeados pelo *OpCode*.

Com essas informações fica claro que os bits podem ser melhores utilizados com a redução da quantidade reservada para indexação.

5.1 Formato de onda para os módulos principais

Nessa seção é mostrado o formato de onda dos módulos desenvolvidos. Para melhor visualização, alguns valores são representados em sistemas numéricos diferentes. Além disso, quando propício, no início de cada subseção encontra-se uma descrição das várias siglas usadas no módulo em questão. Em geral, essas siglas são correspondente às usadas nos diagramas. Por fim, o *clock* foi frequentemente utilizado para a análise dos resultados, por conta disso, considere sempre que o primeiro ciclo de *clock* corresponde ao primeiro conjunto fechado (subida de *clock*, nível lógico alto, descida de *clock*, nível lógico baixo), logo a próxima subida de *clock* se refere ao segundo ciclo. Tome como exemplo, a Figura 9, a qual tem o seu primeiro ciclo de *clock* no intervalo semiaberto $[10.0ns, 30.0ns[$.

5.1.1 Banco de registradores

- $DR1$, $DR2$ (*output*): Dado(s) armazenado(s) no registrador 1, 2
- CFL (*output*): *Flag* de condição
- $RA1$, $RA2$ (*input*): Endereço do registrador 1, 2
- $DR1\_$ (*input*): Dado para escrita
- EW (*input*): Sinal de controle de escrita
- *clk* (*input*): *clock*

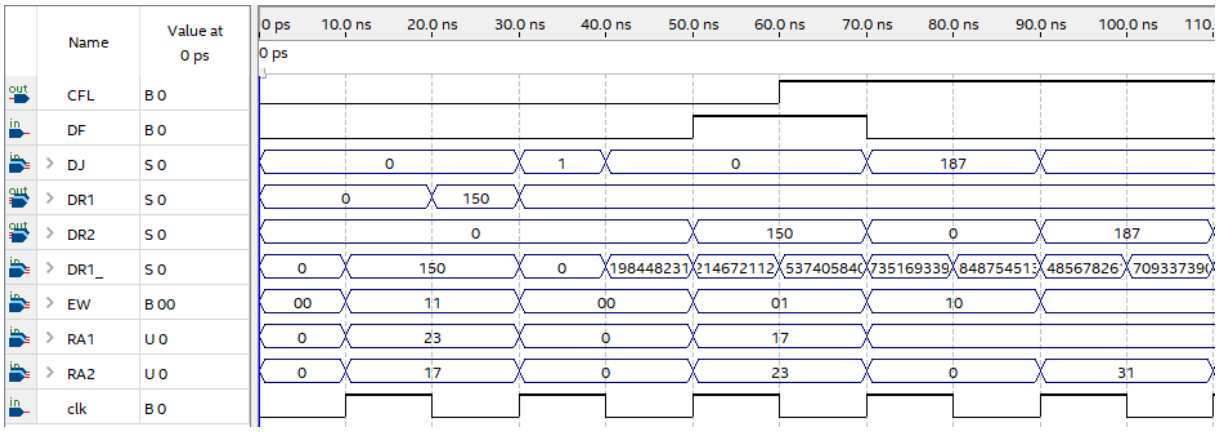
Caso teste 1: No primeiro ciclo de *clock*, um registrador com endereço $RA1 = 23_{10}$ armazena o valor $DR1\_ = 150_{10}$. O resultado pode ser visto pela porta $DR1$ assim que é armazenado, e durante o terceiro ciclo de *clock* quando $RA2$ recebe o endereço 23_{10}, e portanto a porta $DR2$ expressa seu valor.

Caso teste 2: Durante o terceiro ciclo de *clock*, o valor de DF tem nível lógico alto, fazendo com que CFL armazene esse nível lógico.

Casso teste 3: Durante o 4º ciclo de *clock*, o valor de $DJ = 187_{10}$ é armazenado no registrador de posição 31_{10}. O valor é acessado no próximo ciclo de *clock* pela porta $DR2$.

A Figura 9 mostra o formato de onda para o banco de registradores, onde é possível verificar os casos testes mencionados. É importante ressaltar que a leitura de dados é assíncrona, ou seja, é um circuito combinacional, onde os valores dos registradores são repassados automaticamente de acordo com os valores ativos da instrução. Esse efeito pode ser observado durante o caso teste 1 onde o registrador 1 apresenta os valores 0_{10} e 150_{10} durante o mesmo ciclo de *clock*.

Figura 9 – Formato de onda para o banco de registradores



Fonte: Autor

5.1.2 ULA

- *op*1, *op*2 (*input*): Operando 1, 2
- *smt* (*input*): Quantidade de bits que serão deslocados na operação *shift*
- *aluop* (*input*): Sinal de controle para seleção da suboperação no módulo
- $r1$ (*output*): Resultado produzido
- UF (*output*): *Flag*

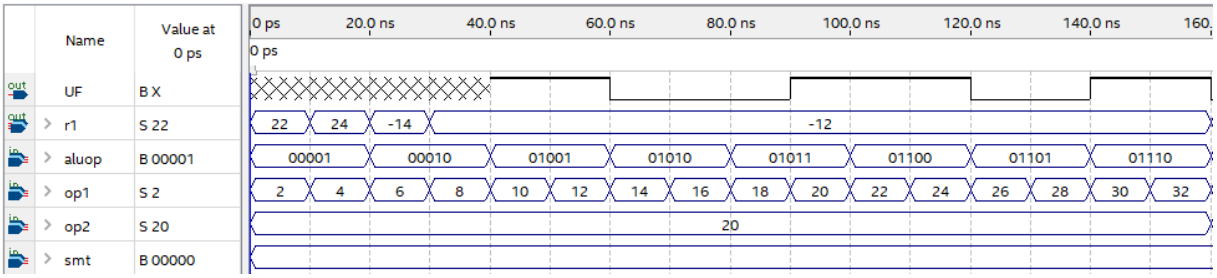
No caso da ULA, ela contém apenas circuitos combinacionais, logo seu formato de onda não depende do *clock*, e sim da combinação de todos os elementos atuais, quando um elemento tem seu valor alterado, o resultao de saída pode ser alterado.

Para facilitar, foi separado primeiro os testes voltados para operações aritméticas e de comparação, e em seguida os testes para operações bit a bit. Em ambos os casos vale lembrar que *aluop* representa a operação a ser realizada e que estão definidas na Tabela 9

A ULA aceita 2 operandos de entrada, tendo isso em vista, foi definido manter o operando 2 com um valor constante igual a 20_{10} enquanto o operando 1 teve seu valor variado periodicamente.

Os dois primeiros sinais da ULA mostrados na Figura 10 são respectivamente as operações de soma e subtração. Desse modo, até o tempo de simulação igual a $30.0ns$, o resultado armazenado em $r1$ é alterado de acordo. Após esse tempo, seu valor é mantido como o resultado da última operação aritmética realizada, mesmo que $op1$ tenha continuado alterando. Enquanto isso, a partir do $40.0ns$ tem-se as operações de comparação, dessa forma UF tem seu valor alterado de acordo. Como exemplo, tome o período de $40.0ns$ à $60.0ns$, nesse caso $aluop = 01001_2$, o que corresponde à comparação $(op1 < op2)$? Observando o valor dos dois operandos, conclui-se que essa comparação é verdadeira, e portanto UF recebe nível lógico alto.

Figura 10 – Formato de onda para operações aritméticas e de comparação



Fonte: Autor

A Figura 11 representa o segundo conjunto de testes. Nesse caso, os valores dos operandos estão representados em binário e os seus valores estão limitados a uma representação de 5 bits para facilitar a visualização dos dados.

É fácil verificar que as operações bit a bit têm os resultados esperados, pois seus cálculos são diretos e fáceis, basta definir os valores de operandos, calcular os resultados manualmente e verificar se os valores de simulação são equivalentes. Por conta disso, a Tabela 10 mostra um conjunto de operações booleanas realizadas manualmente.

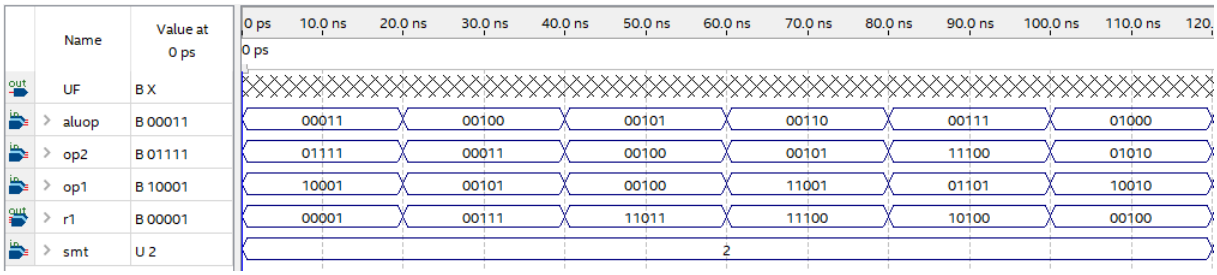
Tabela 10 – Exemplos de operações booleanas AND, OR, NOT, XOR

	AND	OR	NOT	XOR
$op2$	01111	00011	00100	00101
$op1$	10001	00101	$xxxxx$	11001
$r1$	00001	00111	11011	1110

Já na Figura 11, as operações de interesse definidas pelo *AluOp* são os 4 primeiros sinais distintos, representando, em sequência, AND, OR, NOT, XOR. Comparando os valores simulados e calculados, conclui-se que $r1$ apresenta o resultado esperado para todos os casos descritos.

Por fim, nos próximo dois sinais ocorrem *shift* esquerdo e *shift* direito respectivamente. Para essas operações de *shift*, os campos de interesse são o operando 1, e o de *smt*. Como *smt* possui valor igual a 2_{10} significa que $r1$ representa o resultado do operando 1 com todos os seus bits deslocados em 2_{10} para a esquerda (no caso de *shiftL*) ou para a direita (no caso de *shiftR*). Ainda vale notar que o valor de UF é indeterminado, pois nesse caso não houve nenhuma operação onde envolvesse esse elemento, e portanto, seu valor dependerá de resultados prévios.

Figura 11 – Formato de onda para operações bit a bit



Fonte: Autor

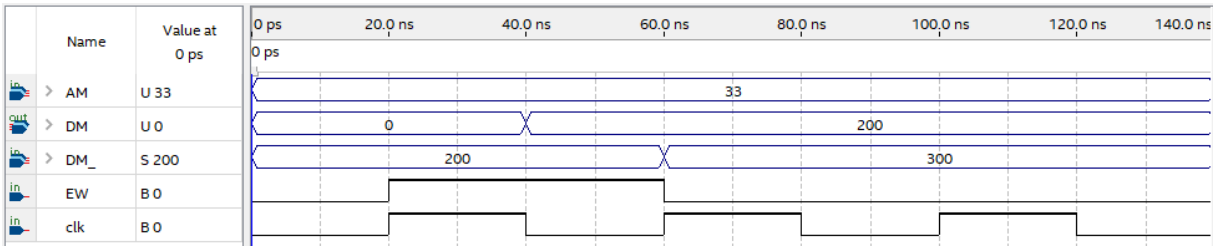
5.1.3 Memória de dados

- AM (*input*): Endereço de memória
- DM (*output*): Dado armazenado na memória
- $DM\_\_$ (*output*): Dado que pode vir a ser escrito
- EW (*input*): Sinal para habilitar a escrita
- *clk* (*input*): clock

No caso da memória de dados, as únicas operações possíveis é de acesso e salvamento. Como repassar os dados não depende de resultados anteriores, DM sempre estará representando o estado atual armazenado na memória de endereço AM. Por outro lado, a escrita é uma operação síncrona, e ocorre durante a descida do *clock*. É possível averiguar esses fatos de acordo com a Figura 12, onde de AM é constante e igual a 33_{10}, e como já explicado anteriormente, DM representa o valor armazenado na posição 33_{10}, o que inicialmente é 0. Em seguida, mesmo com o sinal de escrita em nível lógico alto, o valor só é

alterado quando ocorre a descida do *clock* no tempo $40ns$ e a escrita do valor $DM\_ = 200_{10}$ ocorre de fato.

Figura 12 – Formato de onda para a memória de dados



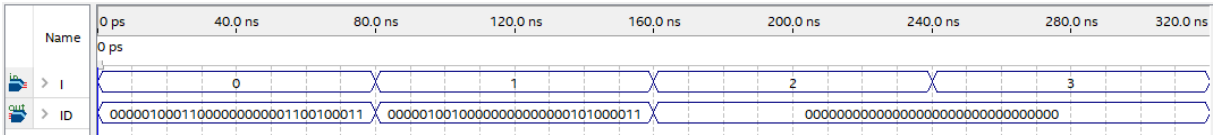
Fonte: Autor

5.1.4 Memória de instrução

- I (*input*): Endereço da instrução
- ID (*output*): Dado de instrução

Para a memória de instrução, o seu objetivo é apenas acessar instruções armazenadas e repassar esses dados. Para a realização do teste, foi carregado previamente duas instruções diferentes no endereço 1_{10} e 2_{10} enquanto os outros endereços têm todos os bits em 0 por padrão. A Figura 13 mostra o acesso das 4 primeiras posições, e como resultado, o valor de ID para as duas primeiras posições possuem valores diferentes de 0 enquanto as duas últimas têm o valor de 0 como esperado

Figura 13 – Formato de onda para a memória de instrução

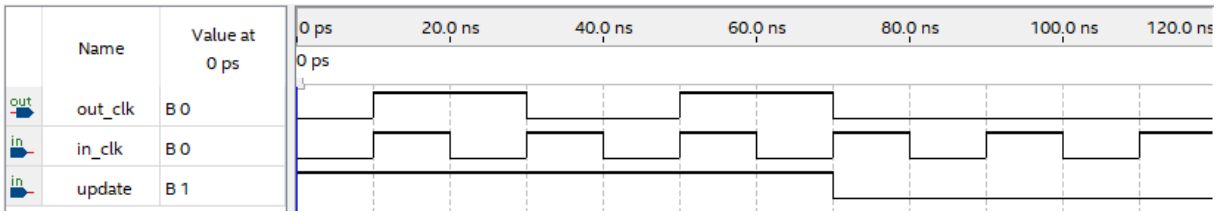


Fonte: Autor

5.1.5 Timer

- *in_clk* (*input*): *clock* de entrada. Para o projeto, $in\_clk = clockBasis$ da Figura 6
- *update* (*input*): Sinal indicando se o *out_clock* será atualizado normalmente (*update* = 1) ou será congelado (*update* = 0)
- *out_clk* (*output*): *clock* de saída. Para o projeto, $out\_clk = clock$ da Figura 6

Figura 14 – Formato de onda para o *Timer*



Fonte: Autor

É esperado que o *clock* de saída tenha metade da frequência do *clock* de entrada. A Figura 14 mostra como o resultado ocorre de acordo com o esperado, pois a saída é alterada sempre que a borda de subida do *clock* de entrada ocorre. Além disso, quando o sinal *update* está em baixo, o *clock* de saída mantém seu último valor.

5.2 Formato de onda para a ligação entre módulos

Nessa etapa, as simulações envolveram analisar passo a passo a ligação dos módulos elementares anteriormente testados. Para isso, a unidade de controle não foi levada em conta, implicando que os 6 primeiros bit de cada instrução (*OpCode*) não tivessem impacto nas simulações, portanto foram deixados em 0. E os sinais de controle, saídas da UC, são dados como entradas e manipulados manualmente.

5.2.1 Conexão entre MI e BR

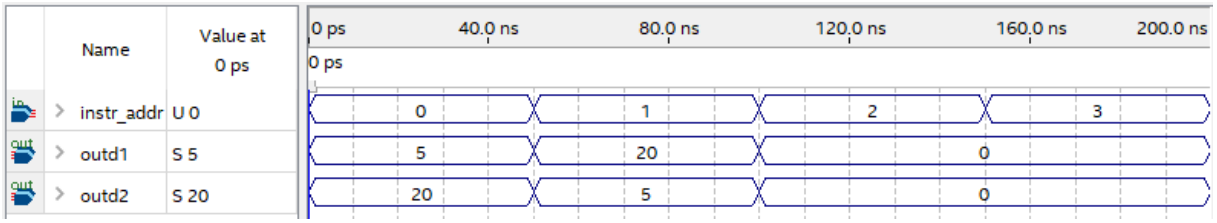
- *instr_addr* (*input*): Endereço de instrução
- *outd*1 (*output*): Valor armazenado no registrador 1
- *outd*2 (*output*): Valor armazenado no registrador 2

Considere reg_{x} como o registrador de endereço decimal x e $instrução_{\mathrm{y}}$ como a instrução de endereço decimal y. Para a realização da simulação, a $instrução_0$ possui valor carregado para acessar o reg_{30} e reg_{31}, nessa ordem, e a $instrução_1$ inverte a ordem dos registradores acessados. As demais instruções acessam outros registradores, que possuem valor 0 armazenado. Com isso, a Figura 15 mostra que a ligação entre o banco e a memória de instrução está de acordo.

5.2.2 Simulação entre MI, BR e ULA para instruções do tipo aritmética

- *immop* (*input*): Sinal do *mux ImmOp*

Figura 15 – Formato de onda para a integração entre a MI e o BR. Valores previamente carregados: $reg_{30} = 5$; $reg_{31} = 20$



Fonte: Autor

- *cdataop* (*input*): Sinal do *mux DataOp*
- *outr*1 (*output*): Resultado da ULA
- *id* (*output*): Valor da instrução

Dado a simulação da Figura 16 tem-se as seguintes análises:

$[0.0ns, 40.0ns]$ – Com o valor de *id* e com o formato de instrução da Tabela 6, é possível verificar que os endereços $r1 = 31$, $r2 = 30$. Além disso, *AluOp* varia sequencialmente entre soma e subtração, e *cdataop* seleciona $r2$. Como resultado, as operações $20 + 5 = 25$ e $20 - 5 = 15$.

$[40.0ns, 80.0ns]$ – $r1$ e o padrão de variação do *Aluop* se repetem. Nesse caso, *id* segue o formato de instrução da Tabela 7, e $immop = 1$ indica imediato pertencente a esse formato de instrução, concluindo $Imm = 10$. Além disso, *cdataop* seleciona operação com Imm. Como resultado, as operações realizadas $20 + 10 = 30$ e $20 - 10 = 10$.

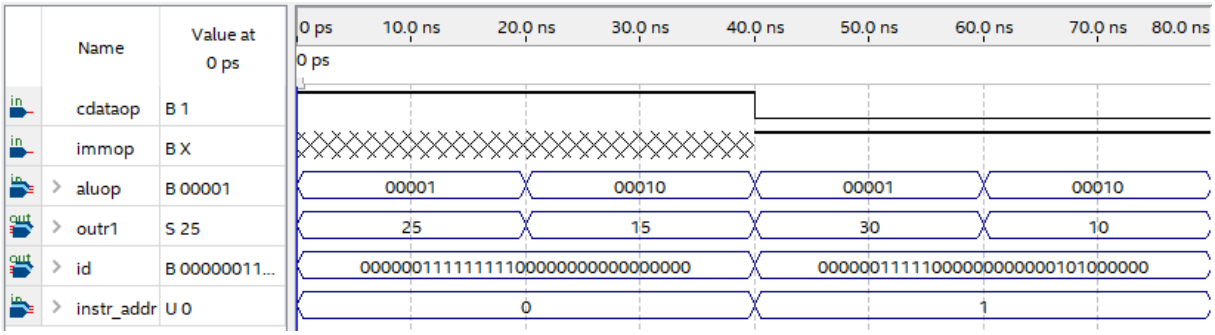
É importante notar que segundo o *design*, o resultado da ULA deve ser salvo no BR após uma operação ser realizada. Esse caminho envolve os *mux's RegW* e *DataSt*, e serão simulados em subseções futuras.

5.2.3 Simulação entre MI, BR, ULA e MD para uma instrução sw

- *memW* (*input*): Sinal para escrita na memória
- *outdm* (*output*): valor armazenado na memória

Considere a simulação da Figura 17. Utilizando a simulação da Figura 16 no período $[40.0ns, 80.0ns]$, repete-se os valores de *Aluop*, $r1$, Imm, e consequentemente a saída da ULA, *outr*1. A diferença se dá pois o formato de instrução é diferente, logo *immop* e *cdataop* são sempre mantidos em 0, já que o primeiro indica um imediato vindo do formato de instrução do tipo III e o segundo indica uma operação aritmética entre um registrador e um imediato. Por fim, $r2 = 30$ se mantém fixo.

Figura 16 – Formato de onda para simulação de operações de soma e subtração dado a integrção entre MI, BR e ULA. Valores previamente carregados: $reg_{30} = 5$; $reg_{31} = 20$

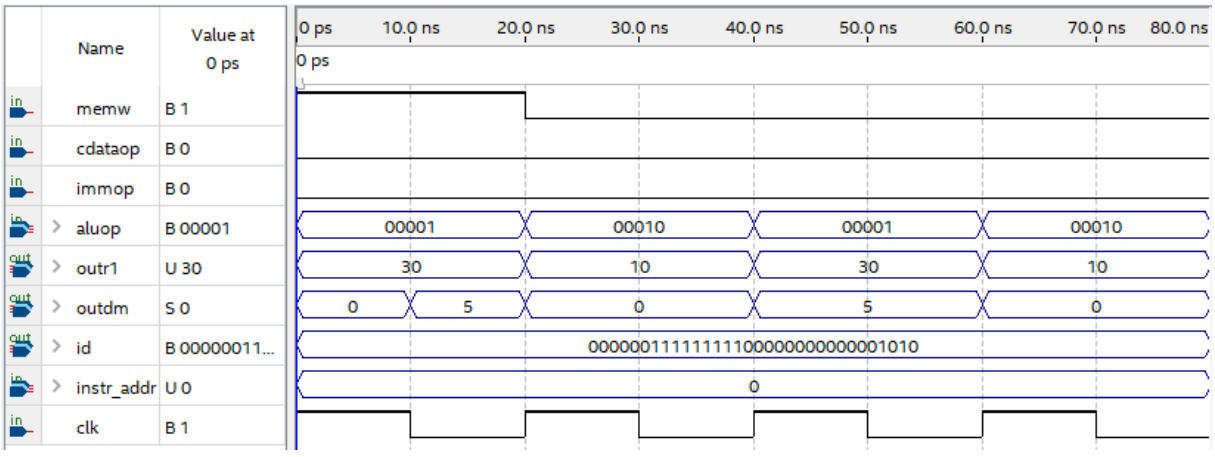


Fonte: Autor

$[0.0ns, 40.0ns]$ – O valor armazenado no registrador de endereço $r2$ é salvo na memória de dados na posição dada por $outr1$ durante a descida do $clock$

$[40.0ns, 80.0ns]$ – É possível checar que o valor armazenado na posição $r2$, quando o sinal $memW$ estava em nível lógico baixo não foram salvos na memória, enquanto o contrário é verdadeiro

Figura 17 – Formato de onda para instrução sw dado a integração entre MI, BR, ULA e MD.Valores previamente carregados: $reg_{30} = 5$; $reg_{31} = 20$



Fonte: Autor

5.2.4 Simulação entre MI, BR, ULA e MD para salvamento e acesso na memória

- $regw$ ($input$): Sinal para escrita no banco
- $cdatast$ ($input$): Sinal do mux $DataSt$

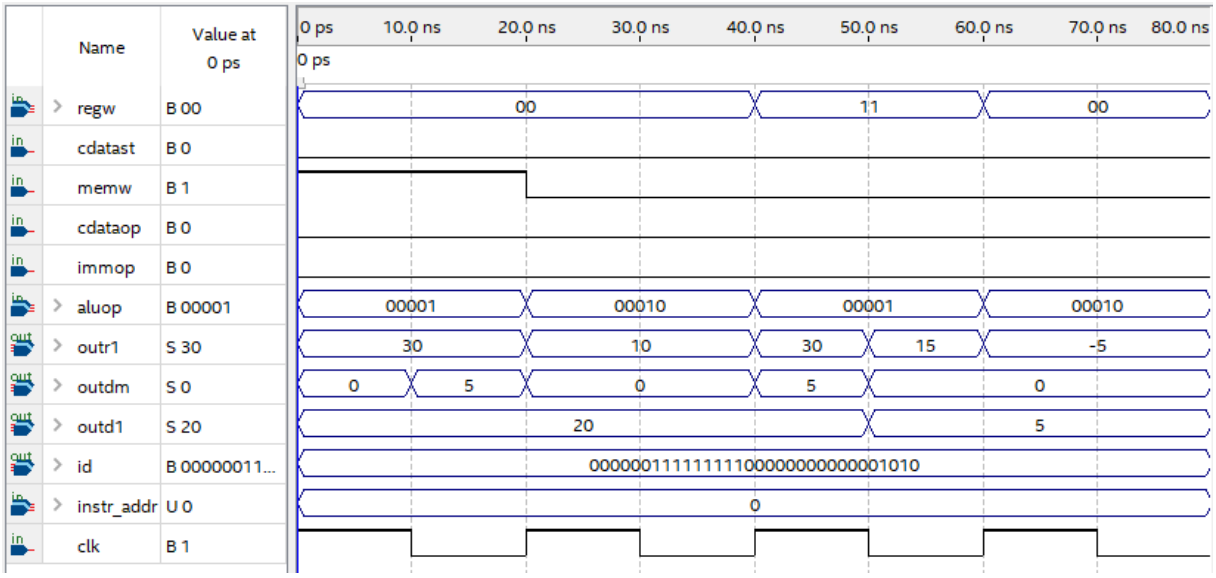
Para a simulação de acesso da memória de dados, o intervalo $[0.0ns, 40.0ns]$ foi simulado igualmente a Figura 17, logo os resultados nessa etapa também são iguais. Além disso, *cdatast* está sempre em nível baixo, pois indica que o dado a ser escrito no registrador virá da memória de dados.

$[0.0ns, 40.0ns]$ – Como já foi mencionado, segue os mesmos resultados da simulação para a instrução de *sw*. O valor 5 é armazenado na memória de posição 30

$[40.0ns, 50.0ns]$ – Durante a descida do *clock*, em $50.0ns$, o valor armazenado anteriormente é recuperado e escrito no endereço do registrador 1. $regW = 11$ é o sinal que permite a escrita.

$[50.0ns, 80.0ns]$ – A ULA realiza soma e subtração entre $r1 = 5$ (Novo valor) e $Imm = 10$. Resultando corretamente em: $5 + 10 = 15$ e $5 - 10 = -5$

Figura 18 – Formato de onda para a integração entre MI, BR, ULA e MD para uma instrução simulada *lw*



Fonte: Autor

5.3 Teste de usuário

Para realizar um teste a nível de usuário, foi desenvolvido um programa para calcular o i-ésimo valor da sequência de fibonacci onde o valor 'i' é um *input* do usuário. Esse programa foi previamente carregado na memória de instrução. O código em verilog, juntamente com o programa desenvolvido, foram sintetizados e carregados no kit FPGA EP4CE115F29C7.

A Figura 19 mostra o código em diferentes níveis de abstração. As colunas de **[A:C]** mostram o programa em um pseudo-código. As colunas de **[E:G]** mostram como o

Figura 19 – Código da sequência de fibonacci

	A	B	C	D	E	F	G	H
1	if __name__ == '__main__':			0	START			00000000000000000000000000000000
2		n = input()		1		mvi $r0, 0		00010100000000000000000000000000
3				2		get $r0		01000000000000000000000000000000
4				3		lw $r0		00011100000000000000000000000000
5		res = fib(n)		4		bal FIB		00111000000000000000000000000101
6		print(res)		5		mv $r1, $r13		00010000001011010000000000000000
7				6		mvi $r0, 0		00010100000000000000000000000000
8				7		sw $r0, $r1		00011000000000010000000000000000
9				8		print $r0		01000100000000000000000000000000
10				9		STOP		11111100000000000000000000000000
11	def fib(n):				FIB			
12				10		mv $r10, $r0		00010001010000000000000000000000
13		int contador = 0		11		mvi $r11, 0		00010101011000000000000000000000
14		int a0 = 1		12		mvi $r12, 1		00010101100000000000000000100000
15		int a1 = 1		13		mvi $r13, 1		00010101101000000000000000100000
16		if(n <= 2)		14		leqi $r10, 2		00001101010000000000000001001011
17			return a1	15		jc $r31		00110011111000000000000000000000
18		else		16		NOP		00000000000000000000000000000000
19			for(contador < n - 2):	17		subi $r10, 2		00000101010000000000000001000100
20						FOR		
21				18			geq $r11, $r10	00001101011010100000000000000110
22				19			bc AFTERFOR	00111100000000000000000000000101
23			temp = a1	20			mv $r14, $r13	00010001110011010000000000000000
24			a1 = a0 + a1	21			add $r13, $r12	00000101101011000000000000000001
25			a0 = temp	22			mv $r12, $r14	00010001100011100000000000000000
26				23			addi $r11, 1	00000101011000000000000000100011
27				24			branch FOR	00110100000000001111111111111001
28						AFTERFOR		
29	return a1			25		jc $r31		00110011111000000000000000000000

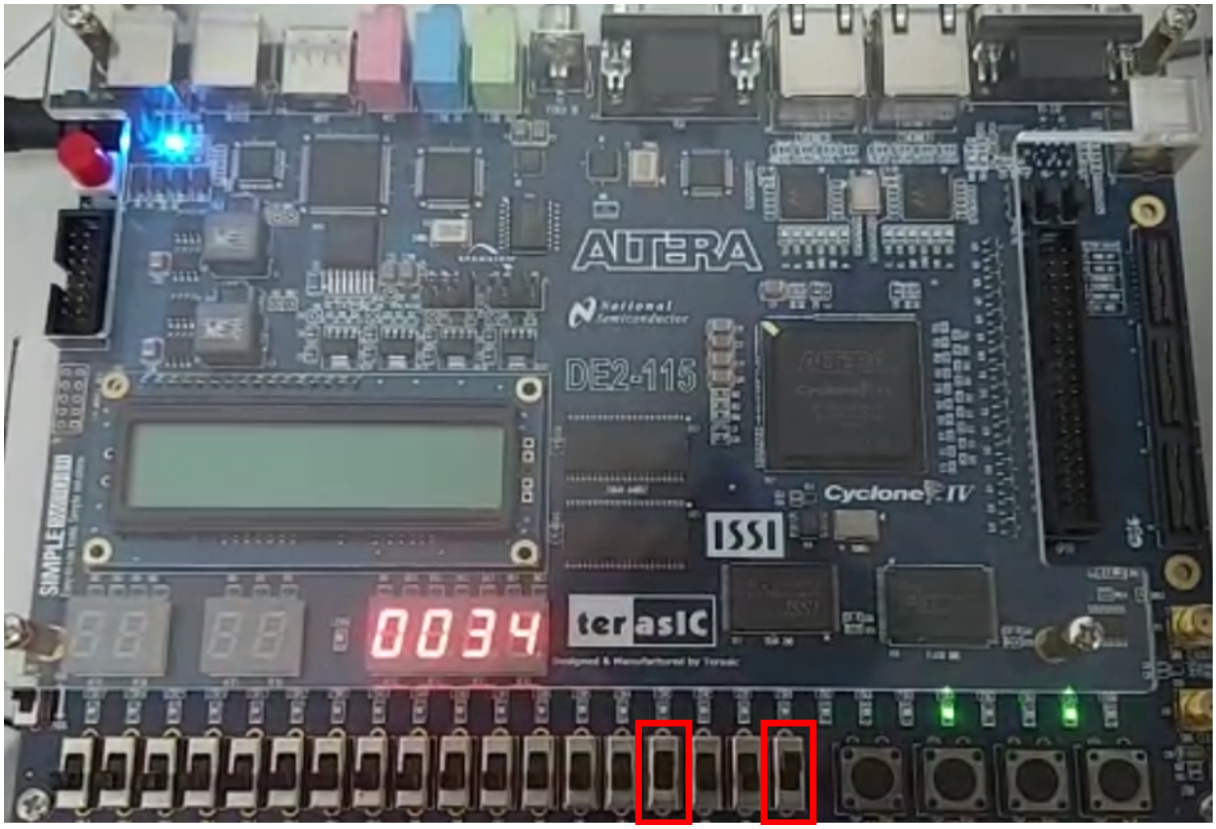
Fonte: Autor

programa seria desenvolvido na linguagem do I-Mips. A coluna **D** representa a posição da memória de instruções. Por fim a coluna **H** mostra o código em linguagem de máquina que foi carregado no FPGA.

A cor das linhas indicam os blocos de código do pseudo-código que equivalem a uma mesma tarefa do código na linguagem do I-MIPS.

A Figura 20 mostra o resultado final. A marcação em vermelho mostra os dois *switches* que tem nível lógico alto. Assim o input do usuário foi $00000000001001_2 = 9_{10}$. A saída mostrada no *display* é 34_{10}, indicando que o 9º elemento da sequência de fibonacci é 34. Logo $x_9 = 34$, o que condiz com a sequência de fibonacci.

Figura 20 – Resultado do programa desenvolvido



Fonte: Autor

6 Considerações Finais

As ações realizadas pelo conjunto de instrução se assemelham muito ao do modelo MIPS, dessa forma é fácil verificar que mesmo com as alterações feitas o conjunto é suficiente para que ao final de toda a implementação seja possível ter uma programação satisfatória.

Os objetivos previsto para todos os pontos de checagem foram atingidos, desde as definições básicas até o planejamento da arquietura e organização, com o conjunto de instrução mapeado, divido em formatos de instruções e com os tipos de endereçamentos bem definidos. Além de uma quantidade satisfatória de simulações em formato de onda, tanto dos módulos principais isoladamente, quanto da integração de todos os módulos, até o teste final a nível de usuário.

A melhor utilização de bits para indexação de algumas instruções é um ponto que pode ser melhorado mas não é um problema para o fluxo de execução.

Talvez seja interessante algumas alterações no futuro para que a programação seja mais fácil e simples, mas essas alterações não são importantes para a cpu no que diz respeito ao seu funcionamento correto.

Os resultados apresentados no modo do formato de onda correspondem as saídas esperadas e estão de acordo com a estrutura planejada, permitindo o funcionamento correto de todos os elementos que compõe a cpu e o seu periférico (Entrada e saída de dados)

Dessa forma o projeto foi bem sucedido, e a cpu desenvolvida é correta e suficiente para sua utilização futura no desenvolvimento de um compilador.

Referências

CHAVES, M. D. de M.; QUEIROZ, A. F. de; PINTO, F. A.; SANTOS, G. B. dos. A evolução da ihc na história da computação. **Revista Diálogos Acadêmicos IESCAMP**, v. 2, n. 1, p. 86–101, 2019.

CREPALDI, C.; COSTA, L. V.; ESCOBAL, A. A. A história da computação: Das máquinas de calcular aos computadores quânticos. **Instituto de Física da Universidade de São Paulo, IF-USP**, v. 9, 2017.

FERLIN, E. P. O avanço tecnológico dos processadores e sua utilização pelo software. **Revista da Vinci**, v. 1, n. 1, p. 43–60, 2004.

FILHO, C. F. **História da computação: O Caminho do Pensamento e da Tecnologia**. [S.l.]: EDIPUCRS, 2007.

FREITAS, A. R.; JUNIOR, C. R.; SOUZA, M. Z.; GONÇALVES, R. A. Arquitetura mips: desenvolvimento de um simulador. **ANAIS do 9º. FÓRUM DE INFORMÁTICA E TECNOLOGIA DE MARINGÁ**, p. 44, 2010.

KOWALTOWSKI, T. Von neumann: suas contribuições à computação. **Estudos Avançados**, SciELO Brasil, v. 10, n. 26, p. 237–260, 1996.

OLIVEIRA, P. de S. Comparação do desempenho multi-core de arquiteturas risc e cisc: Um estudo de caso entre computador desktop e o raspberry pi. **Engenharia de Computação**, n. 1, 2017.

RANDELL, B.; BABBAGE, C. The origin of digital computers. In: **Institute of Mathematics and its Applications, November/December**. [S.l.]: SpringerVerlag, 1973. p. 335–346.

SANTOS, H. G. dos; MARIOTTO, R. Antigas máquinas de calcular: Uma pré-história dos computadores. In: **11º CONGRESSO DE INICIAÇÃO CIENTÍFICA E TECNOLÓGICA DO IFSP**. [S.l.: s.n.], 2020. p. 1.