PONTIFÍCIA UNIVERSIDADE CATÓLICA DE CAMPINAS

ENGENHARIA DE COMPUTAÇÃO

ARQUITETURA DE COMPUTADORES

Erick Matheus Lopes Pacheco – 18711630

Hiago Silva Fernandes – 18726455

Marcos Antônio Junior Vasconcellos – 18720920

Victor Reis – 18726471

HVEM 2.0 – PROCESSADOR PIPELINE EM VHDL

DOCUMENTAÇÃO, DETALHAMENTO E RESULTADOS DO PROJETO

CAMPINAS

Novembro 2019

SUMÁRIO

1. INTRODUÇÃO

Dá-se a proposta de projeto pelo professor: projetar e desenvolver um processador pipeline em VHDL, que seja capaz de realizar operações aritmética, leitura e escrita de instruções, jump e branch, com e sem valores imediatos atribuídos.

Desse modo, o grupo se organizou de maneira a todos trabalharem juntos, sem que houvesse falhas de comunicação ou conflito de atividades. Para isso utilizamos cinco ferramentas: **Discord** para comunicação VOIP do grupo, **Google Docs** e **Draw.io** para projetar a CPU, a máquina de estados e definir suas instruções; **Visual Studio Code** com o plugin Live Share para programação simultânea em VHDL, além do **Intel Quartus Prime** para a compilação e simulação do nosso código.

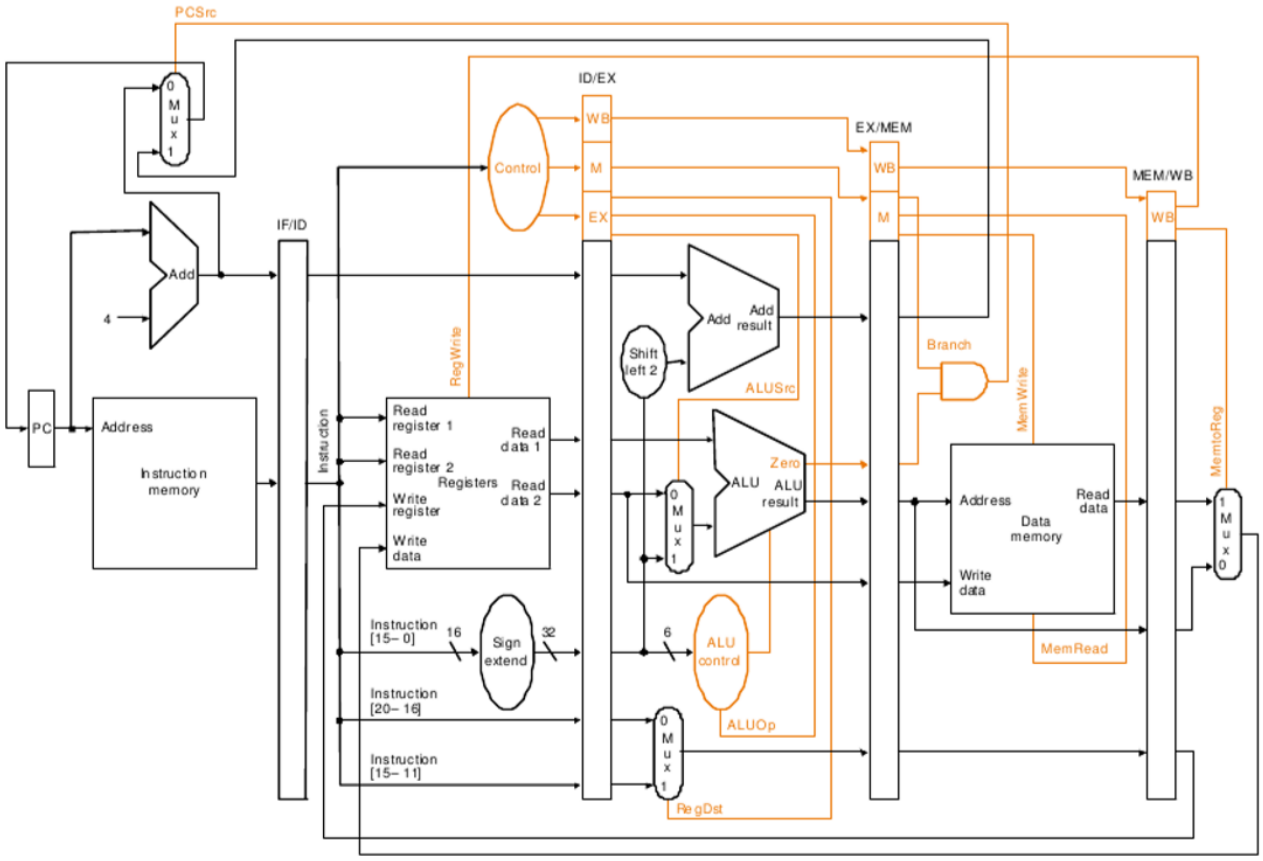
1.1. TOPOLOGIA DA CPU

Foi proposto um processador capaz de executar as seguintes instruções:

Category	Name	Instruction syntax	Meaning	Format	Notes
Arithmetic	Add	add $1,$2,$3	$1 = $2 + $3	R	Adds two registers
	Subtract	sub $1,$2,$3	$1 = $2 – $3	R	Subtracts two registers
	Add immediate	addi $1,$2,CONST	$1 = $2 + CONST	I	Used to add constants
	Sub immediate	subi $1,$2,CONST	$1 = $2 - CONST	I	Used to sub constants
Data Transfer	Load word	lw $1,CONST($2)	$1 = Memory[$2 + CONST]	I	Loads the word stored from: MEM[$s2+CONST] and the following 3 bytes
	Store word	sw $1,CONST($2)	Memory[$2 + CONST] = $1	I	Stores a word into: MEM[$2+CONST] and the following 3 bytes
Logical	And	and $1,$2,$3	$1 = $2 & $3	R	Bitwise and
	And immediate	andi $1,$2,CONST	$1 = $2 & CONST	I	
	Or	or $1,$2,$3	$1 = $2 | $3	R	Bitwise or
	Or immediate	ori $1,$2,CONST	$1 = $2 | CONST	I	
Conditional branch	Branch on equal	beq $1,$2,CONST	if ($1 == $2) go to PC+4+CONST	I	Goes to the instruction at the specified address if two registers are equal
Unconditional jump	Jump	j CONST	goto address CONST	J	Unconditionally jumps to the instruction at the specified address
	Jump register	jr $1	goto address $1	R	Jumps to the address contained in the specified register

Em detrimento ao projeto anterior, onde batizamos nossa CPU de “HVEM” (representando as iniciais dos membros integrantes do grupo), batizar-se-á com o mesmo nome, como sendo uma versão *melhorada* da CPU anterior

Devido ao paralelismo exigido, torna-se necessário a divisão do processador em estágios, com registradores capazes de armazenar as instruções e seus sinais para cada um desses estágios, de modo a não perder as informações. Seguindo essa linha de pensamento, nos foi fornecido o seguinte datapath base:



Devido ao fato de que o datapath base não executar as instruções de jump incondicional, mais a frente estaremos mostrando o datapath completo e adaptado para a implementação de todas as instruções propostas.

O processador operará com instruções e dados de 32 bits, ou seja, possuirá a capacidade de armazenar até 2^{32} (4.294.967.296) valores, o que torna possível a plena execução das instruções requisitadas, bem como a possibilidade de uma expansão de projeto para implementar mais instruções e armazenar valores maiores, caso desejado.

Na página seguinte, uma imagem esquematizada do datapath completo de uma CPU pipeline, baseada no datapath base fornecido pelo professor, que agora tem a plena capacidade de executar todas as instruções.

- Em **vermelho**, tudo referente a Unidade de Controle;
- Em **laranja**, os fios que estabelecem comunicação entre os componentes;

As instruções faltantes foram implementadas da seguinte maneira:

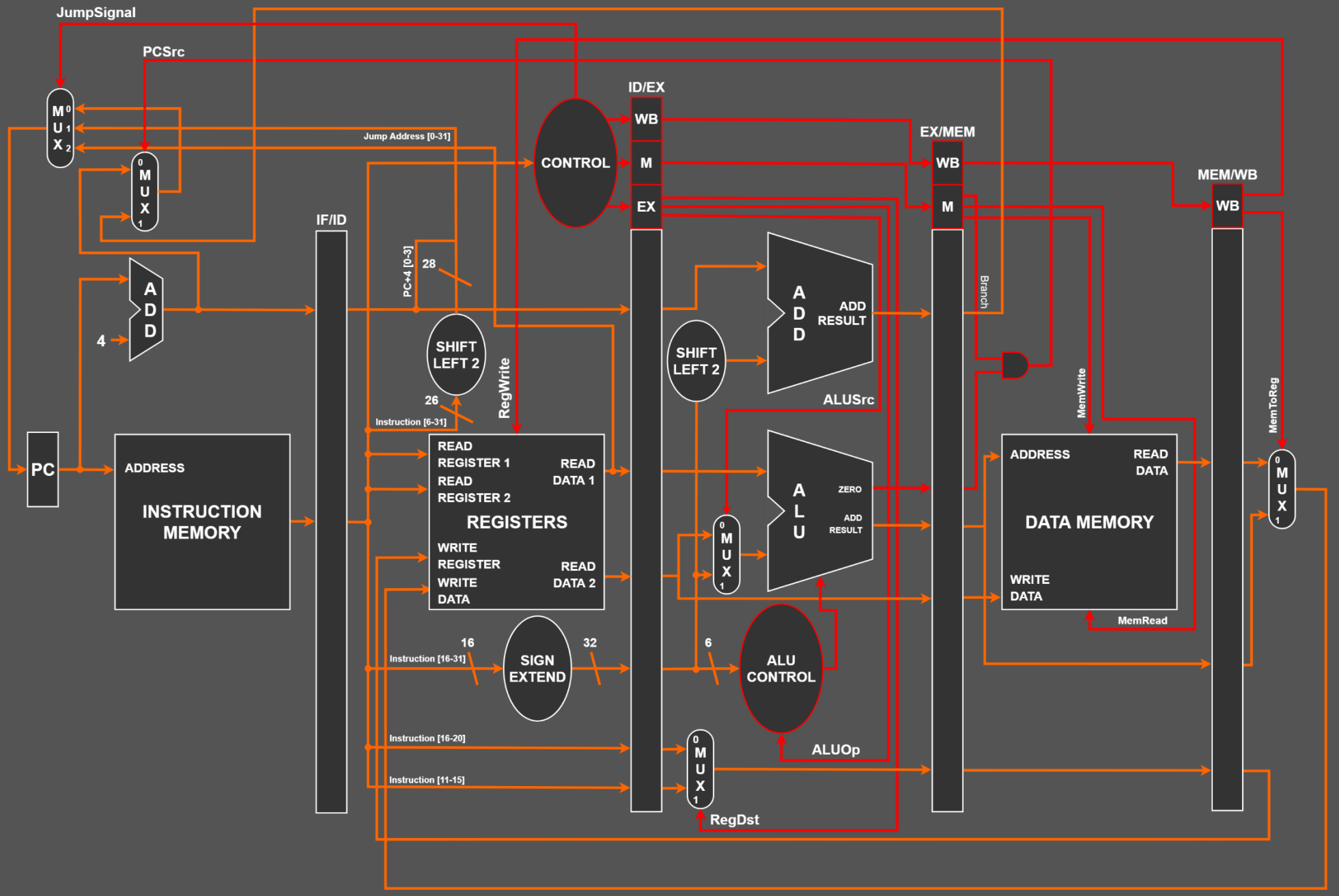
1. Jump
    - Adição um novo MUX 2 para 1;
    - Com um novo MUX, um novo sinal de controle para ele (JumpSignal);
    - O jump é calculado de maneira a concatenar os bits mais significativos (MSB) de PC+4 com os 26 bits da instrução, estendidos para 28;
    - O resultado da concatenação é ligado no MUX.

2. Jump Register
    - Aproveita-se o JumpSignal, tornando-o um sinal de 2 bits para a seleção entre Jump, o resultado do MUX entre Branch e PC+4 ou a nova instrução Jump Register;
    - Mudamos o MUX 2 para 1 adicionado anteriormente para um MUX 3 para 1;
    - O endereço é apenas puxado do conteúdo do registrador selecionado, ou seja, contido na saída READ DATA 1 da memória de registradores, e conectado na entrada do MUX.

PIPELINE CPU *HVEM* *(v2.0 | 32 bits)* - DATAPATH

By: Hiago Silva, Victor Reis, Erick Pacheco & Marcos Jr

A Pipeline CPU project to our Computer Architecture subject at the university.
The processor was built-in to support 32 bit instructions and performs it in parallel.



2. ESPECIFICAÇÃO

Dada a proposta do projeto e a estruturação do datapath base, traçamos nosso datapath de maneira que ela atenda as instruções com folga, isto é, definimos adequadamente o sistema de armazenamento (registradores e memória em geral), bem como a forma com que os blocos de memória seriam estruturados para o conjunto de registradores que possuímos, além da implementação das instruções faltantes (jump e jump register).

2.1. FORMATO DE INSTRUÇÃO

Temos os seguintes formatos de instrução (todos compondo **32 bits totais**):

TIPO	OPCODE	RS	RT	RD	UNUSED	FUNCT
R	0-5	6-10	11-15	16-20	21-25	26-31
	6	5	5	5	5	6

TIPO	OPCODE	RS	RT	IMED
I	0-5	6-10	11-15	16-31
	6	5	5	16

TIPO	OPCODE	ADDRESS
J	0-5	6-31
	6	26

Dado os formatos acima, temos que nossas instruções são identificadas pela unidade de controle a partir de um **código de operação de 6 bits** (OPCODE), da forma como indicada na tabela abaixo:

OPCODE	INSTRUÇÃO	SIGNIFICADO	DESCRIÇÃO
000001	Tipo R	$1 = $2 (op) $3	Operação lógica-aritmética
000010	ADDI	$1 = $2 + CONST	Add c/ imediato
000011	SUBI	$1 = $2 - CONST	Subtração c/ imediato
000100	ANDI	$1 = $2 & CONST	And c/ imediato
000101	ORI	$1 = $2 | CONST	Or c/ imediato
000110	LW	$1 = Mem[$2 + CONST]	Carrega palavra (word) de um endereço
000111	SW	Mem[$2 + CONST] = $1	Escreve palavra (word) em um endereço
001000	BEQ	if($1 == $2) go to PC+4+CONST	Branch se igual
001001	JUMP	go to address CONST	Jump para um endereço específico
001010	JR	go to $1	Jump para um endereço de um registrador específico

Para não descartamos o uso de uma unidade de controle própria da ULA, conforme presente no datapath base, foi optado pela criação de um **OPCODE único** para as instruções do tipo R, onde o controle da ULA a partir de cada OPCODE recebe um outro código **chamado de ALUOP de 3 bits**, que, quando identifica uma instrução do tipo R, fará a decodificação de qual operação realizar a partir do FUNCT presente no formato de instrução tipo R.

A fim de simplificar para melhor entendimento, segue a tabela abaixo referente ao controle da Unidade Lógica-Aritmética:

TIPO I			
INSTRUÇÃO	ALU OP	ALU FUNCTION	ALU CODE
LW	000	ADD	00
SW	000	ADD	00
BEQ	001	SUBTRACT	01
ADDI	011	ADD	00
SUBI	100	SUBTRACT	01
ANDI	101	AND	10
ORI	111	OR	11

Para instruções do tipo I, necessitamos apenas do ALU OP, que indicará para a ULA qual a devida operação através do ALU CODE.

	TIPO R			
	INSTRUÇÃO	FUNCT	ALU FUNCTION	ALU CODE
	ADD	100000	ADD	00
	SUB	100010	SUBTRACT	01
	AND	100100	AND	10
	OR	100101	OR	11
ALU OP	010			

Como se pode notar, o OPCODE envia um ALUOP único para o controle da ULA que, quando identificado, permite através dos sinais de FUNCT gerar o ALU CODE para cada respectiva operação lógica-aritmética do tipo R.

Por fim, e não menos importante, a tabela abaixo nos mostra como cada operação é identificada:

ALU CODE	OPERAÇÃO
00	ADD
01	SUB
10	AND
11	OR

3. DESENVOLVIMENTO

Discutidos os detalhes de projeto, é válido estendermos estes detalhes aos de implementação, apresentado a maneira como foram atribuídas as conexões, portas e o funcionamento interno dos componentes e registradores.

3.1. COMPONENTES DE CONTROLE

3.1.1. AluControl (Controle da Unidade Lógica-Aritmética)

É responsável pelo controle das operações da ULA, mapeando a utilização de 3 portas (ALU_OP, FUNCT, e ULA_CODE), onde o ALU_OP é um sinal de 3 bits que determina o tipo de instrução (tipo R, I ou J) a ser executada, o FUNCT é utilizado na seleção das operações aritméticas tipo R e o ULA_CODE é o sinal de saída que receberá a operação a ser feita.

3.1.2. ControlUnit (Unidade de Controle)

Responsável por mandar os sinais de controle de leitura e escrita de dados na memória, write back e execução. Mapeia-se a utilização de 5 portas (WB, MEM, EX, SIGNAL_JUMP e o INSTRUCTION). Esse componente recebe a instrução inteira (0 to 31) no estágio IF/ID e faz a verificação dos primeiros 6 bits (OP CODE) para que dessa forma os sinais de controle sejam setados em seus respectivos registradores (WB, MEM, EX) com os valores correspondentes a cada um deles no estágio ID/EX.

3.1.3. PCIncrement (Incremento do Contador de Programas)

Utilizado para incrementar o PC (Program Counter), o PCIncrement mapeia duas portas, uma de entrada (PC, logic vector de 32 bits) e uma de saída (X, que será o PC+4). O objetivo está em calcular o endereço da próxima instrução para o processo de fetch de instrução. O valor calculado é salvo em X.

3.1.4. ProgramCounter (Contador de Programas)

Componente responsável pela atualização do registrador contador de programas, possui três portas onde duas são de entrada (CLOCK e PC_INC) e uma de saída (PC). O funcionamento é simples: em um evento de clock, grava-se o conteúdo de PC_INC (aka PC+4) em PC.

3.1.5. ShiftLeft (Deslocamento Lógico à Esquerda)

O ShiftLeft possui duas portas: A, de entrada, que contém os dados a serem shiftados, e X, de saída, que irá receber o dado com o shift implementado. Utiliza-se 30 bits [2-31] (MSB) dos 32 bits de A para concatenar com dois bits “00” nos bits menos significativos (LSB), para que assim seja formado o valor com o equivalente a dois deslocamentos lógicos à esquerda.

3.1.6. ShiftLeft2_26to28 (Deslocamento Lógico à Esquerda 26 p/ 28 bits)

O ShiftLeft2_26to28 tem por objetivo dar o equivalente a dois shifts a esquerda no valor de entrada mapeado pela porta A. O shift é feito concatenando o todos os 26 bits de A com dois bits zeros à direita, de modo a estender o valor de 26 para 28 bits. O propósito deste componente está na implementação da instrução de jump, onde os 26 bits recebidos da instrução são estendidos para 28 para então serem concatenados com os 4 primeiros bits de PC+4.

3.2. COMPONENTES DE MEMÓRIA

3.2.1. DataMemory (Memória de Dados)

O componente DataMemory possui seis entradas: ADDRESS, CLOCK, MEM_WRITE, WRITE_DATA, MEM_READ, READ_DATA. O ADDRESS é o endereço base onde começaremos a escrever o valor presente em WRITE_DATA no signal MEMORY, sendo que a cada evento de clock, se houver solicitação de escrita de memória (indicada por MEM_WRITE), salvamos em 4 posições de

memória a partir do endereço base + offset pegando 8 em 8 bits do conteúdo de WRITE_DATA, de modo que assim temos a divisão byte a byte da memória.

Em contrapartida, se houver requisição de dados (indicada por MEM_READ), salva-se 4 bytes (concatenando-os) no port de saída READ_DATA o que está gravado em ADDRESS+Offset de MEMORY.

3.2.2. InstructionMemory (Memória de Instrução)

O InstructionMemory possui 2 portas sendo elas o ADDRESS e o INSTRUCTION. O INSTRUCTION começa zerado pois quando se pegar o conteúdo que está no endereço na memória este sobrescrito e é convertido para inteiro e concatenado 4 bits a partir do ADDRESS e no final é salvo no INSTRUCTION.

3.3. COMPONENTES LÓGICOS

3.3.1. Alu (Unidade Lógica-Aritmética)

A Alu possui 5 portas sendo elas dois registradores A e B onde esses tem os valores a serem utilizados para as operações aritméticas, o ALU_CODE (instrução para operação aritmética), um ALU_OUT(onde este armazena o resultado da operação aritmética) e o ZERO é uma “booleana” que servirá para sinalizar se o resultado de uma operação é zero ou não, onde será utilizado nas instruções de branch. Tem-se no componente um sinal AUX que serve para receber o resultado da operação aritmética onde esse posteriormente é passado depois para o ALU_OUT. Caso o ALU_CODE não satisfaça nenhuma das condições o sinal AUX será setado como 0 e o sinal ZERO irá receber 1.

3.3.2. SignExtend (Extensor de Sinal)

Utilizado para estender um logic vector de 16 bits para 32. Mapeia duas portas, uma de entrada (A) e uma de saída (X), esse componente será implementado para o uso do imediato estendido. A extensão se dá pelo uso da função nativa “resize”, que permite redimensionar o tamanho de um logic vector

para uma quantidade N de bits. Utilizamos também a função “signed” para que se estenda o sinal do valor armazenado em A. O dado estendido é salvo na saída X.

3.3.3. Mux_2to1_32b e Mux_2to1_5b (Multiplexador 2 para 1)

Ambos os multiplexadores são 2 para 1 e assumem diversos usos, como na seleção de registradores e endereços para a próxima instrução (PC Source), são implementados de maneira que haja 4 portas mapeadas, sendo 3 de entradas (CONTROL, A e B) e uma de saída (X). A seleção dos valores se dá pelo valor contido em CONTROL (0 ou 1) que chaveia entre A e B, guardando o conteúdo na saída X. A única expressa diferença entre os dois é a quantidade de bits que são processados, onde um processa 5 bits e o outro 32.

3.4. COMPONENTES DE REGISTRADORES

3.4.1. FileRegister (Registrador de Arquivos)

Será o registrador de arquivos/dados do estágio IF/ID do pipeline, sendo utilizado para acesso e escrita rápida de informação, de modo a implementar as instruções sem acesso a memórias externas ao processador. Mapeia 9 portas, onde 6 registradores são de entrada (REGWRITE, CLOCK, READ_REGISTER_1, READ_REGISTER_2, WRITE_REGISTER, WRITE_DATA) e 3 de saída (READ_DATA_1, READ_DATA_2 e DEB_REGS), conta com 32 registradores internos (imitando a arquitetura MIPS), implementados no signal REGISTERS (que recebe uma matriz 32x32, ou seja, 32 registradores de 32 bits).

Ao evento de clock, verifica se há requisição de escrita de registrador (indicada por REG_WRITE) e se o endereço do registrador (indicado por WRITE_REGISTER) é diferente de 0, se sim, grava no endereço do registrador o valor contido em WRITE_DATA. Por fim, gravamos nas portas de saída os valores de Ri, Rj e um valor teste na porta de debug.

3.4.2. Reg_Pipe_IFID (Registrador de Pipeline - Estágio IF/ID)

Sendo o primeiro registrador de pipeline do datapath, ele possui a função de armazenar o fetch da instrução pega a partir do endereço PC, e o endereço da próxima instrução (PC+4), contando com duas portas de entrada para tal (IN_PC_MAIS_4 e IN_INSTR_MEM), e duas portas de saída (OUT_PC_MAIS_4, OUT_INSTR_MEM) para propagar esses dois dados para o próximo estágio.

3.4.3. Reg_Pipe_IDEX (Registrador de Pipeline - Estágio ID/EX)

O Reg_Pipe_IDEX mapeia 19 portas, das quais 10 portas são as entradas e as 9 restantes são as portas de saída. O componente visa fazer a transferência dos dados no evento de clock igual a 1.

3.4.4. Reg_Pipe_EXMEM (Registrador de Pipeline - Estágio EX/MEM)

O Reg_Pipe_EXMEM mapeia 15 portas, onde 8 portas são de entrada e 7 são de saída. Nesse componente a sua função é a de transferir os dados dos registradores de entrada para os de saída quando o evento de clock for 1.

3.4.5. Reg_Pipe_MEMWB (Registrador de Pipeline - Estágio MEM/WB)

Como este registrador é o último registrador de pipeline do datapath, ele possui a função de salvar os valores dos resultados para que dessa forma não sejam sobrescritas as instruções posteriores a ele. O Reg_Pipe_MEMWB mapeia 9 portas das quais 5 são de entrada e 4 são de saída. Com um evento de clock de subida ocorrendo, este componente apenas propaga os sinais de controle para os multiplexadores de PC e de resultados.

4. RESULTADOS

Os resultados obtidos durante o desenvolvimento do projeto cumpriram às expectativas iniciais, quais envolviam tempo despendido no projeto, funcionamento e devida implementação das instruções requisitadas bem como o efetivo término dentro do prazo de entrega.

Abaixo, uma série de testes de instruções sendo realizadas pela CPU:

4.1. PRIMEIRO TESTE

O seguinte código foi desenvolvido em C:

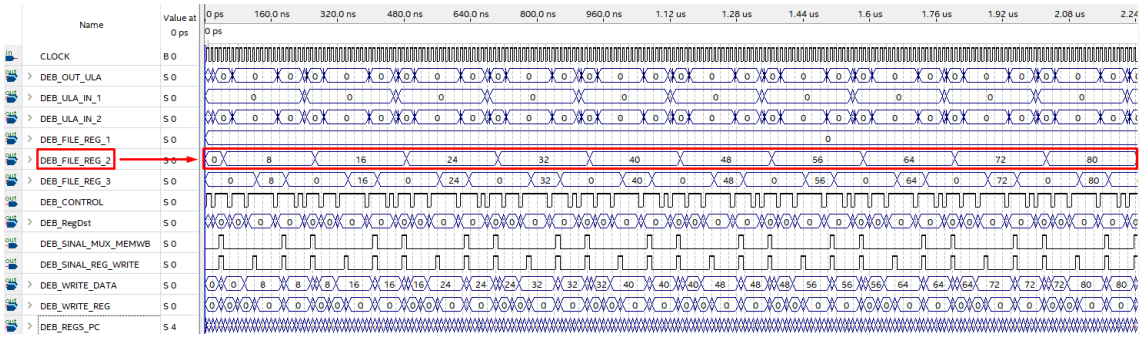
```c
int main() {
    int reg2 = 0, reg3 = 0, reg4 = 0, reg6 = 0;

    while(!reg3) {
        reg2 = reg2 + 8;
        reg4 = reg2;
        reg3 = reg4;
        reg3 = reg6 + 0;
    }
}
```

Montado para nossa CPU da seguinte maneira:

```
ADDI $2, $2, 8   # $2 = $2 + 8, onde $2 inicialmente é 0
SW $2, 0($4)     # $4 = $2, salvo em $4 o que está em $2
LW $3, 0($4)     # $3 = $4, carrego em $3 o que está em $4
ADDI $3, $6, 0   # $3 = $6 + 0, ou seja, $3 = 0
JR $3            # Jump para o endereço de $3, ou seja, 0 (inicio do programa)
```

Resultado esperado: Tabuada infinita de 8 presente no registrador $2

Resultado obtido:



(zoom p/ ampliar)

Ou seja, para essa nossa primeira instrução, o resultado foi condizente, visto que até o último instante de período mostrado no waveform, os valores presentes no registrador $2 são condizentes com o de uma tabuada do número 8. Caso estendêssemos o período mostrado, obteríamos a continuação de tal tabuada infinitamente.

Veridito final: ✅

4.2. SEGUNDO TESTE

O seguinte código foi desenvolvido em C:

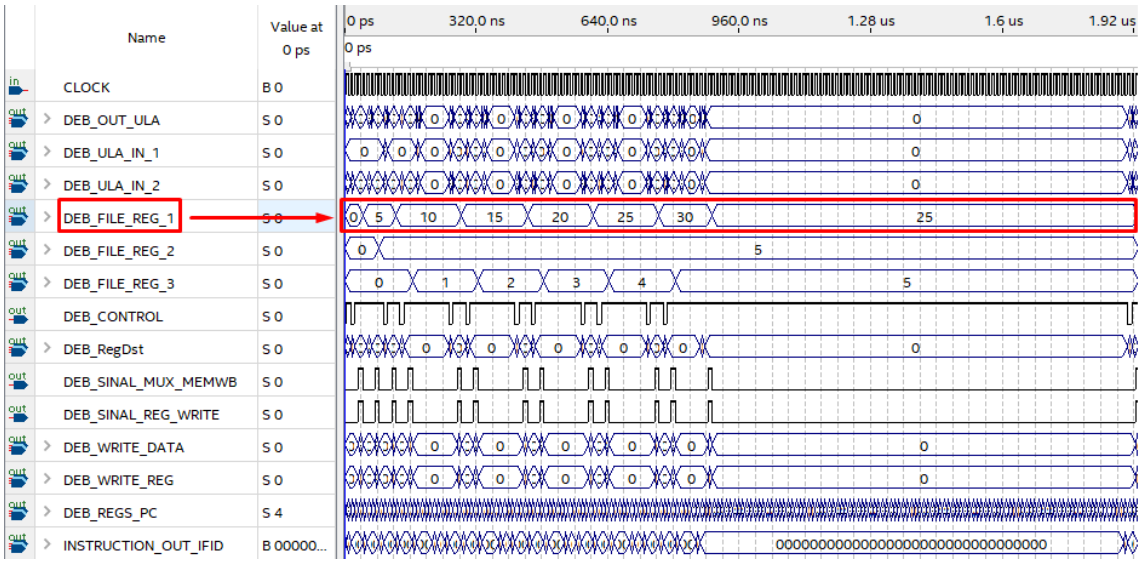
```c
int main() {
    int reg1 = 5, reg2 = 0, reg3 = 0, reg4 = 0;

    reg2 = reg1;

    while(reg3 < reg2) {
        reg1 = reg1 + 5;
        reg3 = reg3 + 1;
    }
    reg1 = reg1 - reg2;
}
```

Montado para nossa CPU da seguinte maneira:

```
ADDI $1, $1, 5          # $1 = $1 + 5, onde $1 inicialmente é 0
ADD $2, $2, $1          # $2 = $2 + $1, onde $2 inicialmente é 0, logo $2 = 5
LOOP:
    ADDI $1, $1, 5      # $1 = $1 + 5
    ADDI $3, $3, 1      # $3 = $3 + 1
    BEQ $3, $2, J_SUB   # Branch para a label J_SUB se $3 == $2
    J LOOP              # Jump para a label LOOP
J_SUB:
    SUB $1, $3, $1      # $1 = $3 - $1
```

Resultado esperado: Valor 25 presente no registrador $1

Resultado obtido:



(zoom p/ ampliar)

Como é possível observar, o programa permaneceu dentro do laço de repetição até que o registrador 3 (*$3*) usado como contador chegasse em 5 e fosse constatado no *Branch if Equal* que seu valor era igual ao do registrador 2 (*$3*), realizando um branch para o endereço da instrução *SUB*, e assim, finalizando o programa subtraindo o conteúdo do registrador 1 (*$1*) que a cada repetição somava 5 com o conteúdo do registrador 2 (*$2*), que era 5. Enquanto o valor do registrador 3 não chegasse em 5, a instrução de jump era responsável por pular novamente para a label *LOOP*, e como o registrador 3 começava em 0, esperava-se que o resultado presente em $1 até a última repetição fosse de 30, sendo subtraído 5 após o término do loop.

Veridito final: ✅

4.3. TERCEIRO TESTE

O seguinte código foi desenvolvido em C:

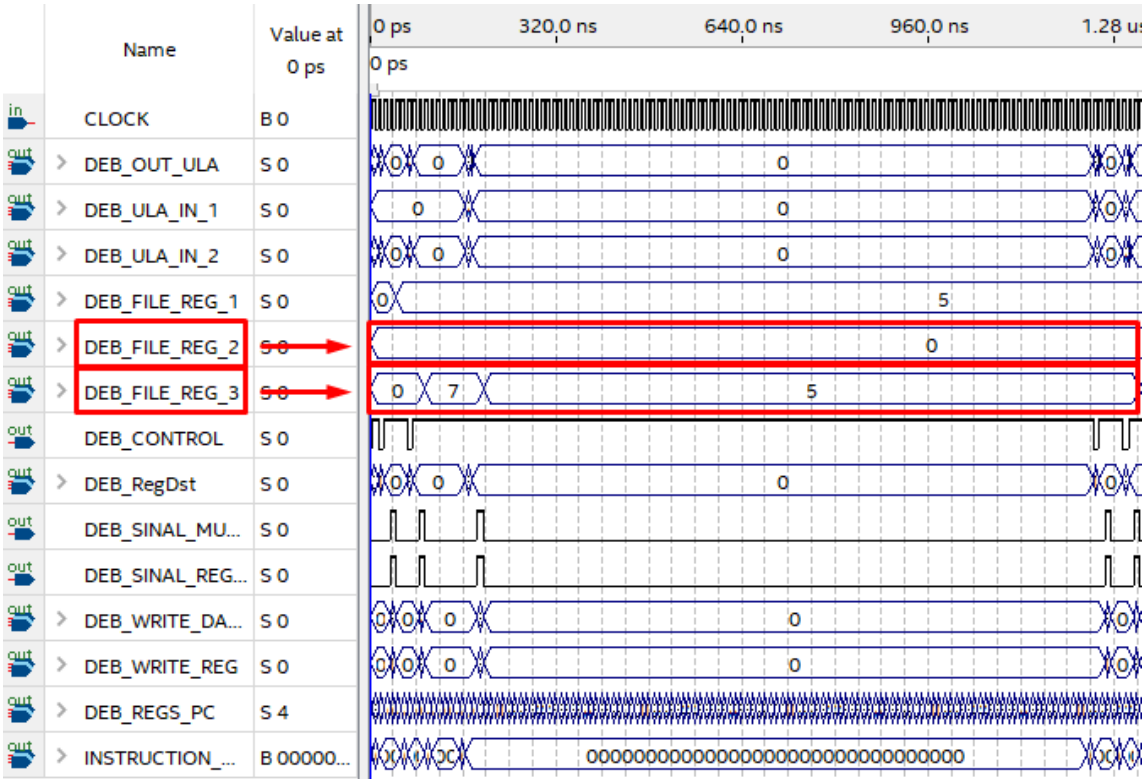
```c
int main() {
    int reg1 = 0, reg2 = 0, reg3 = 0;

    reg1 = reg1 | 5;
    reg3 = reg3 + 7;

    goto NEXT_INSTR;

    reg2 = reg1 - 1;

    NEXT_INSTR:
    reg1 = reg1 & reg3;
}
```

Montado para nossa CPU da seguinte maneira:

```
ORI $1, $1, 5       # Faz $1 or 5, que resulta em 5 e salva em $1
ADDI $3, $3, 7      # Soma $3 com 7, e salva o resultado em $3
J NEXT_INSTR        # Jump para a label NEXT_INSTR
SUBI $2, $1, 1      # Subtrai $1 com 1 e salva em $2
NEXT_INSTR:
    AND $3, $3, $1  # Faz $3 and $1 e salva o resultado em $3
```

Resultado esperado: Valor 5 no registrador $3 e 0 no registrador $2

Resultado obtido:



(zoom p/ ampliar)

Ambos os resultados obtidos condizem com o esperado, visto que a intenção de nosso programa era realizar um jump que pulasse a instrução de *SUBI* presente logo após esse jump, executando, no lugar, a última instrução apenas. Isso faz com que esperemos que o resultado final presente no registrador 2 (*$2*) seja 0, tal como é por padrão. Esperamos também que o registrador 3 (*$3*) possua no final o valor 5, que é resultado de um AND de 5 (contido em *$1*) com 7 (contido em *$3*), sendo que tal resultado também é observado no waveform.

Veridito final: ✅

5. CONCLUSÕES

A proposta do projeto consistida em implementar um processador pipeline de modo a colocar em prática as aulas teóricas ministradas em sala de aula é de efetiva necessidade, conclui o grupo.

Dentre as razões, há se pontuar a melhor compreensão do funcionamento em geral das arquiteturas de computadores, em especial do MIPS. No segundo projeto foi possível compreender melhor toda a estrutura funcional e de instrução de um processador, neste terceiro, todavia, estende-se ainda mais a profundidade destes pontos além de nos introduzir ao mundo das memórias e paralelismo de processamento com a proposta de pipeline.

Inicialmente, o grupo acreditava que a dificuldade de desenvolvimento nos seria impactada, porém numa análise mais detalhada, podemos concluir que apenas o nível de complexidade de extensão do projeto é posto à mesa, onde com paciência e dedicação é possível tornar funcional o projeto.

O trabalho equipe desempenhado entre as partes foi plenamente síncrono e atenderam, juntamente com as ferramentas utilizadas, as necessidades exigidas por tal complexidade, sendo a CPU devidamente implementada conforme solicitada na proposta do projeto.

Obstáculos de desenvolvimento e mudanças de planos válidas de serem mencionadas são:

- Falta de familiarização com a sintaxe e semântica da linguagem VHDL;
- Abstração máxima dos componentes da CPU era prevista, porém descartada para que nos encaixássemos dentro do prazo de entrega;
- Pequenos erros de implementação que nos fizeram despender mais tempo que o previsto fazendo debug.

Apesar dos obstáculos, o projeto foi um sucesso e de muito nos serviu para que expandíssemos nosso grau de conhecimento e experiência prática na disciplina de Arquitetura de Computadores.

6. BIBLIOGRAFIA

PANNAIN, Ricardo. “**CAPÍTULO 2 – INSTRUÇÕES: LINGUAGEM DO COMPUTADOR**”; Google Drive. Disponível em <https://drive.google.com/file/d/13e3GzMN3Vykn8YYVTCWpS96o3nmLRgh9/view?usp=sharing>. Acesso em novembro de 2019.

PANNAIN, Ricardo. “**CAPÍTULO 4 – O PROCESSADOR**”; Google Drive. Disponível em <https://drive.google.com/file/d/1EWK2zLVzMs2ON1PAyg_bJHlp-w9Ce9mC/view?usp=sharing>. Acesso em novembro de 2019.

“**HOW TO CONVERT 8 BITS TO 16 BITS IN VHDL?**”, Stack Overflow. Disponível em <https://stackoverflow.com/q/17451492>. Acesso em novembro de 2019.