Trabalho Prático 2

Leonardo Magalhães do Vale [1] - 2022035920
Lucas Albano Olive Cruz [1] - 2022036209
Mateus Gonçalves Moreira [1] - 2022035610

{leonardovale, lucasalbano, mateusmor}@dcc.ufmg.br

[1] Departamento de Ciência da Computação - Universidade Federal de Minas Gerais

1. Introdução

Este projeto busca integrar instruções de multiplicação, divisão, branch on equal e andi em um processador de 5 estágios usando Verilog. Para isso, foram feitas alterações no caminho de dados para suportar essas operações, implementadas a partir de modificações em um código pré-existente no Google Colab.

2. Desenvolvimento

2.1 Modificações na seção **Assembly**

Inicialmente, para implementar as instruções corretamente, foi necessário alterar a seção **Assembly** do código fornecido a fim de especificar quais os códigos, em binário, as novas instruções teriam. Em particular, foram especificados para cada instrução à qual classe de instruções elas pertencem, o *opcode*, o *funct3* e o *funct7,* a saber:

Instruction Name	**Instruction Format**	**Instruction Opcode**	**Instruction Funct3**	**Instruction Funct7**
Mul	R-Type	0110011	000	1000000
Div	R-Type	0110011	000	1100000
Beq	B-Type	1100011	000	N/A
Andi	I-Type	0010011	111	N/A

A última modificação necessária nessa parte do código foi realizada visando consertar um ***bug*** que não permitia o programa funcionar da forma esperada. Esse problema consistia em uma linha que atribuía o valor do *funct7* ao *funct3*, deixando as instruções do Tipo-R com 4 bits a mais, totalizando 36 bits. Dessa forma, na etapa de decodificar a instrução, o programa sempre exibia resultados inconsistentes. Sendo mais específico, o erro está logo abaixo da função ***r_type*** e consiste na seguinte linha:

```
funct7 = funct3 = instruction_funct7[instruction]
```

2.2 **Modificações na seção Control Unit**

A unidade **Control Unit,** no código, é responsável por interpretar o *opcode* da instrução de entrada e gerar sinais de controle adequados para a ALU, memória, registradores e instruções específicas.

Dito isso, foi necessário apenas incluir um ***case*** que representasse a instrução ***branch on equal*** para conseguir gerar os sinais necessários a fim de implementar essa operação. Em particular, quando o *opcode* for igual à 1100011, sendo a instrução *branch on equal*, foi necessário "*setar*" o sinal ***branch_eq*** para 1, assinalar o registrador ***aluop*** com o valor 1 e, além disso, calcular o valor do imediato. Todas essas ações foram necessárias para sinalizar para a ALU que realizaremos uma subtração para comparar os registradores, além de enviar os sinais necessários para a possível tomada do *branch*.

Vale ressaltar que essas alterações foram feitas, pois o código original não contemplava esse tipo de instrução — Tipo-B — e, para as outras operações, não foi necessário adaptar essa parte para além do que já foi dito, pois já havia implementado no código a lógica dos sinais tanto para instruções do Tipo-R, quanto para as do Tipo-I.

2.3 **Modificações na seção Pipeline Bar (Decode -> Exec)**

Nessa etapa foi necessário mudar quais bits do *funct7* e do *funct3* estavam sendo concatenados para as operações do Tipo-R, já que no código original era utilizado apenas o quinto bit do *funct7* e o *funct3* para fazer a decisão de qual operação a ALU iria realizar. Dito isso, tomamos a **decisão de usar os 2 últimos bits mais significativos do *funct7* e o *funct3*, totalizando 5 bits**, para sinalizar para ALU que agora podemos fazer as operações ***mul*** e ***div*.**

Em particular, o código em binário para a **multiplicação** agora é 10000, correspondendo a 16 em decimal e, para a divisão, o código resultou em 11000, que corresponde a 24 em decimal.

2.4 **Modificações na seção ALU Control and ALU**

2.4.1 **Módulo *alu_control***

Neste módulo, o tamanho do ***input funct*** foi aumentado para 5 bits para acomodar a nova concatenação de bits mencionada anteriormente. No primeiro bloco *always*, houve uma modificação para incluir casos em que **funct é igual a 16 ou 24, representando multiplicação ou divisão**. Para esses casos, *_funct* agora recebe 9 para multiplicação e 10 para divisão. A instrução ***andi*** foi implementada utilizando a estrutura da instrução *and*, enquanto a instrução ***beq*** aproveitou a lógica da operação de subtração. Portanto, *_funct* é definido como 0 para ***andi*** e, para a operação ***beq***, o valor dele será 6, como o da subtração.

Seguindo em frente, **para o segundo bloco *always*** deste módulo, que identifica o código dos 3 últimos bits do *funct — funct3 —* e atribui um valor para o *_functi*, foi adicionado um case para quando a operação for ***and immediate***, no caso o valor 111, o qual é 7 em decimal. Isso ocorre, pois como ela é uma instrução do tipo-I, não utilizamos o *funct7*. Em particular, quando essa operação for realizada, o *_functi* irá receber o valor 0.

Por fim, **no terceiro bloco *always***, que contém o case para o *aluop*, o registrador de saída ***aluctl***, poderá receber diferentes valores explicitados na tabela abaixo.

Instruction	**aluop (alu operation)**	**aluctl (alu control)**
Mul	2	*_funct* = 9
Div	2	*_funct* = 10
Beq	1	6
Andi	3	*_functi* = 0

Vale ressaltar que o registrador ***aluctl*** é a entrada ***ctl*** do módulo ***alu***, sendo ele o responsável por alterar os *cases* nesse módulo, escolhendo qual operação será realizada dentro da ALU.

Por fim, é importante deixar claro que tanto a instrução ***andi*** quanto a ***beq***, se aproveitam da lógica já implementada de outras operações do código, sendo o procedimento *and* para a primeira e *sub* para a segunda, mudando apenas os sinais de controle que saem da Unidade de Controle.

2.4.2 **Módulo *alu***

A ALU é o componente do processador responsável pela execução de operações aritméticas e lógicas. Ele executa tarefas como operações de adição, subtração, AND, OR e NOT, manipulando dados com base nas instruções fornecidas pela unidade de controle do processador, no caso deste código, o módulo ***alu_control***.

Em relação ao módulo ALU implementado no código, foram feitas pequenas modificações:

- Adicionado dois novos fios de saída, representando o ***mul*** e a ***div;***
- Adicionados os *cases* 9 e 10 para realizar a multiplicação e divisão entre dois registradores.

Para as outras duas operações, como ambas foram feitas em cima de instruções já implementadas, não foi necessário adaptar para além disso.

3. Testes

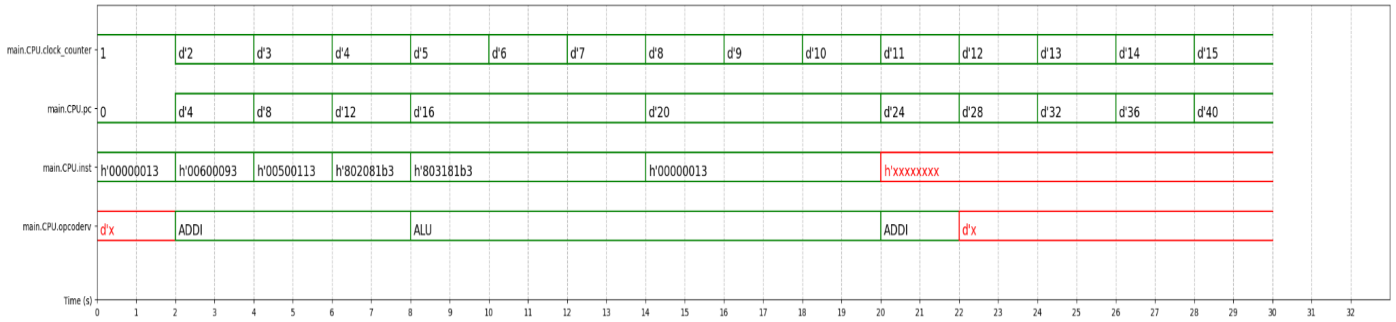
Inicialmente, é relevante destacar que todos os testes foram executados no ambiente do Google Colab, oferecendo a possibilidade ao avaliador de realizá-los manualmente, caso prefira essa abordagem. Os casos de teste estão estruturados na seção “Testes” do código, com cada nova instrução apresentando células de teste dedicadas. Vale ressaltar que a operação BEQ se destaca ao incluir dois testes: um simbolizando o cenário onde o branch é tomado e outro quando não é. Além disso, para facilitar a análise dos resultados, o código também disponibiliza waveforms que oferecem uma representação visual do comportamento do sistema durante os testes.

3.1 **Teste para a operação *mul***

Instruções	Saída do programa

`nop`
`addi x1, x0, 6`
`addi x2, x0, 5`
`mul x3, x1, x2`
`mul x3, x3, x3`
`end: nop`
	`x1 = 6`
	`x2 = 5`
	`x3 = 900`

- Waveforms:



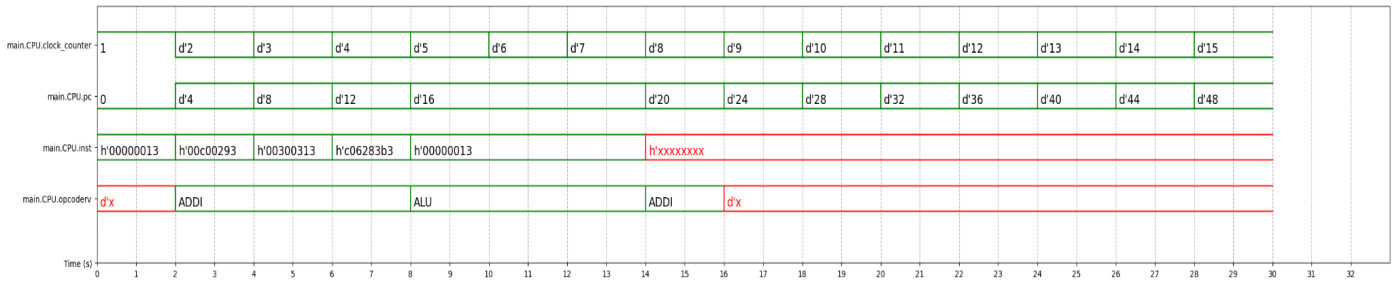
Como demonstrado pela saída do programa, a operação ***mul*** funciona corretamente, pois, primeiro carregados o valor 6 e 5 nos registradores x1 e x2 respectivamente, e logo após isso multiplicamos 6 por 5 e colocamos em x3 realizando, por fim, uma operação exponencial com esse valor usando a multiplicação.

3.2 **Teste para a instrução *div***

Instruções	Saída do programa

`nop`
`addi x5,x0,12`
`addi x6,x0,3`
`div x7,x5,x6`
`end: nop`
	`x5 = 12`
	`x6 = 3`
	`x7 = 4`

- Waveforms:



O programa adiciona 12 a x5 e 3 à x6, em seguida, executa uma operação de divisão, armazenando o resultado em x7. A saída correta demonstra que a operação de divisão funcionou conforme esperado, com x7 sendo 4, resultado da divisão de 12 por 3.

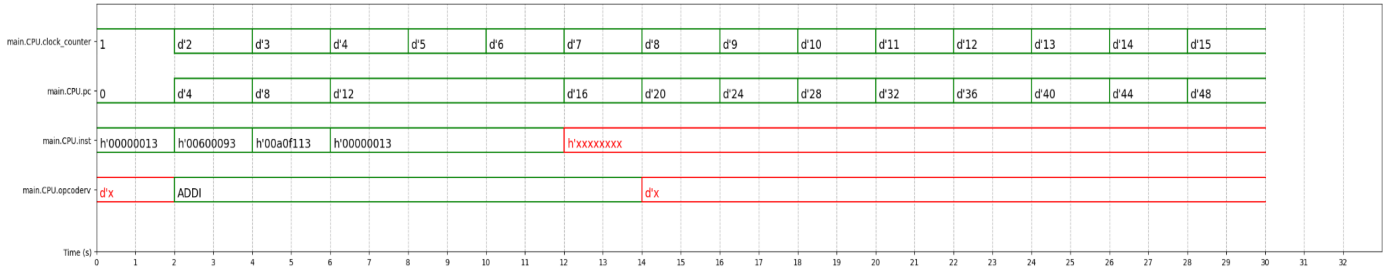
3.3 **Teste para a instrução *andi***

Instruções	**Saída do programa**

`nop`
`addi x1, x0, 6`
`andi x2, x1, 10`
`end: nop`
	`x1 = 6`
	`x2 = 2`

O programa adiciona 6 a x1 e realiza uma operação lógica AND entre x1 e 10, armazenando o resultado em x2. A saída correta, x2 sendo 2, confirma que a operação ANDI funcionou adequadamente ao extrair os bits compartilhados entre 6 e 10.

- Waveforms:



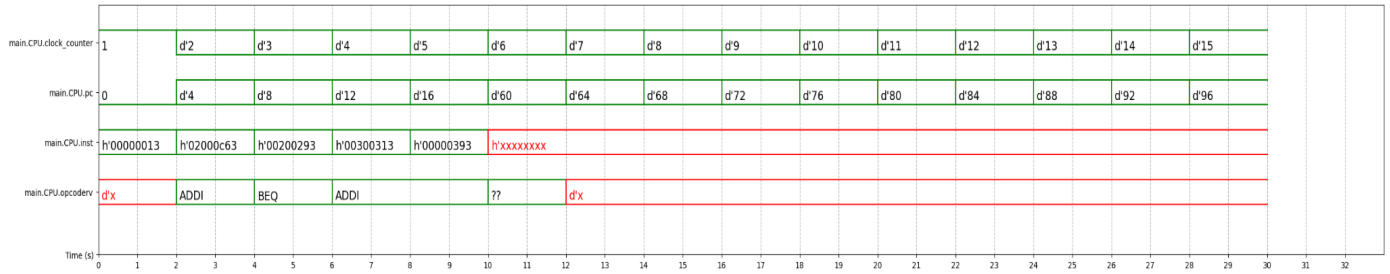
3.4 **Testes para a instrução *beq***

Instruções	**Saída do programa**

`nop`
`beq x0, x0, end`
`addi x5, x0, 2`
`addi x6, x0, 3`
`addi x7, x0, 0`
`addi x7, x7, 1`
`addi x7, x7, 1`
`addi x7, x7, 1`
`end: nop`
	`x1 = 0`
	`x2 = 0`
	`x3 = 0`
	`.`
	`.`
	`.`
	`.`
	`.`
	`.`
	`.`
	`.`
	`.`
	`x31 = 0`

O programa acima utiliza a instrução ***beq*** para criar um desvio condicional. Como a condição ***beq x0, x0*** é sempre verdadeira (x0 é sempre igual a si), o programa desvia para a instrução "end". As instruções **antes de *end*** não são executadas, resultando em saídas zeradas para todos os registradores de x1 a x31, pois as instruções de adição não foram processadas. Isso demonstra que a operação *beq* está funcionando quando a condição for verdadeira.

- Waveforms:

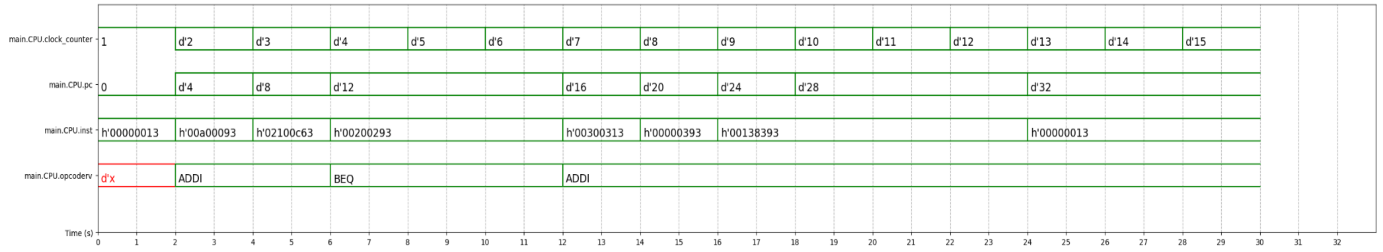


Instruções	Saída do programa

nop
addi x1, x0, 10
beq x0, x1, end
addi x5, x0, 2
addi x6, x0, 3
addi x7, x0, 0
addi x7, x7, 1
addi x7, x7, 1
end: nop
	x1 = 10
	
	x5 = 2
	
	x6 = 3
	
	x7 = 2

A instrução ***beq x0, x1, end*** compara x0 e x1, não sendo iguais, então o programa continua com as instruções de adição. A saída do programa mostra os valores corretos, indicando que as instruções após o desvio condicional foram executadas, pois x1 é diferente de zero, sendo a condição da instrução *beq*, falsa. Isso demonstra o funcionamento adequado do desvio condicional.

- Waveforms:



4. Conclusão

Este projeto teve como principal objetivo solidificar os conhecimentos em relação à como o processador funciona, além de fornecer uma familiaridade com a linguagem HDL Verilog. No desenvolvimento, ajustes na seção Assembly, Control Unit, Pipeline Bar e ALU foram realizados para integrar instruções adicionais, incluindo a expansão da seção de controle para a instrução *branch on equal*. Além disso, adaptações na concatenação de bits na Pipeline Bar foram feitas para sinalizar corretamente operações de multiplicação e divisão. No módulo ALU Control, o *input funct* foi expandido, incorporando casos específicos para as operações de multiplicação e divisão.

Por fim, a instrução beq utilizou a subtração na comparação de registradores e a andi se aproveitou da estrutura do procedimento and, precisando de poucas modificações para incorporar as quatro novas operações, sendo o maior desafio a interpretação do código já existente.