Lab 2b: Projetando uma Unidade Lógica e Aritmética Simples

Prof. João Carlos Bittencourt

Monitor: Éverton Gomes dos Santos

Centro de Ciências Exatas e Tecnológicas

Universidade Federal do Recôncavo da Bahia, Cruz das Almas

Introdução

Este roteiro de laboratório consiste de várias etapas, cada uma construída a partir da anterior. Instruções detalhadas são fornecidas ao longo do texto. Códigos

cuidadosamente.

Você vai aprender a:

- Projetar um sistema hierárquico, com níveis de hierarquia muito mais profundos;
- Projetar circuitos lógicos e aritméticos;
- Usar constantes do tipo `parameter`;
- Simular circuitos Verilog, lidar com *testbench* e estímulos de entrada;

Modifique o Somador-Subtrator do Lab 2 para torná-lo Parametrizável

Tendo em vista customizar a quantidade de bits dos operandos do somador, apresento a seguir uma versão parametrizada do somador *ripple-carry*.

```
module adder #( parameter N=32 )
(
   input wire [N-1:0 ] A, B,
   input wire Cin,
   output wire [N-1:0] Sum,
   output wire FlagN, FlagC, FlagV
);

   wire [N:0] carry;
   assign carry[0] = Cin;

   assign FlagN = Sum[N-1];
   assign FlagC = carry[N];
   assign FlagV = carry[N] ^ carry[N-1];

   fulladder a[N-1:0] (A, B, carry[N-1:0], Sum, carry[N:1]);

endmodule
```

Neste caso, a palavra-chave `parameter` é usada para especificar uma constante com valor padrão, que pode ser sobrescrito ao declarar uma instância do circuito em um módulo de nível superior na hierarquia.

Ao modificar o parâmetro `N`, a largura do somador pode ser facilmente alterada. O valor `32` é especificado como valor `padrão` de `N`, mas pode ser sobrescrito por um novo valor quando este módulo é instanciado.

Note ainda que foram adicionadas mais saídas com o intuito de produzir as *flags* de `Negative`, `Carry out` e `Overflow`. A unidade Somador-Subtrator foi também modificada, como apresentado a seguir.

```
module addsub #( parameter N = 32 )
(
   input wire [N-1:0] A, B,
   input wire Subtract,
   output wire [N-1:0] Result,
   output wire FlagN, FlagC, FlagV
);

   wire [N-1:0] ToBornottoB = {N{Subtract}} ^ B;
   adder #(N) add(A, ToBornottoB, Subtract, Result, FlagN, FlagC, FlagV);
```

```
endmodule
```

Mais uma vez, o circuito Somador-Subtrator é parametrizado com a largura padrão de 32 bits. O parâmetro `N` é transferido para o módulo `adder` sempre que ele é instanciado. Deste modo, ao modificar o valor de `N` na linha acima do módulo `addsub`, digamos para 64, o novo valor do parâmetro é utilizado para o módulo somador, chamado `adder`.

O *testbench* do Lab 2 foi modificado para testar ambos os módulos.

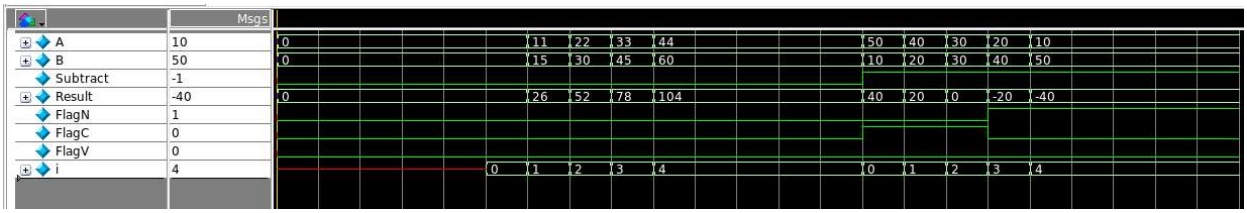
> 🤦 No testbench, onde você declara a *unit under test* (`uut`), é possível definir o valor do parâmetro como no exemplo abaixo:
>
> ```
> addsub #(width) uut ( ... );
> ```

O *testbench* fornecido no diretório `sim` (`addsub_param_tb`) utiliza uma palavra de 8-bits. Você deve tentar outras larguras (ex., 32 bits) também, simplesmente modificando o valor `localparam N` dentro do *testbench*.

Um `localparam` é outra forma de declarar constantes em Verilog. Todavia, ela é definida apenas como atributo **local** do módulo. Ou seja, essa constante não é visível ou modificável no módulo pai dentro da hierarquia. Observe no *testbench* como o `localparam` é declarado e utilizado.

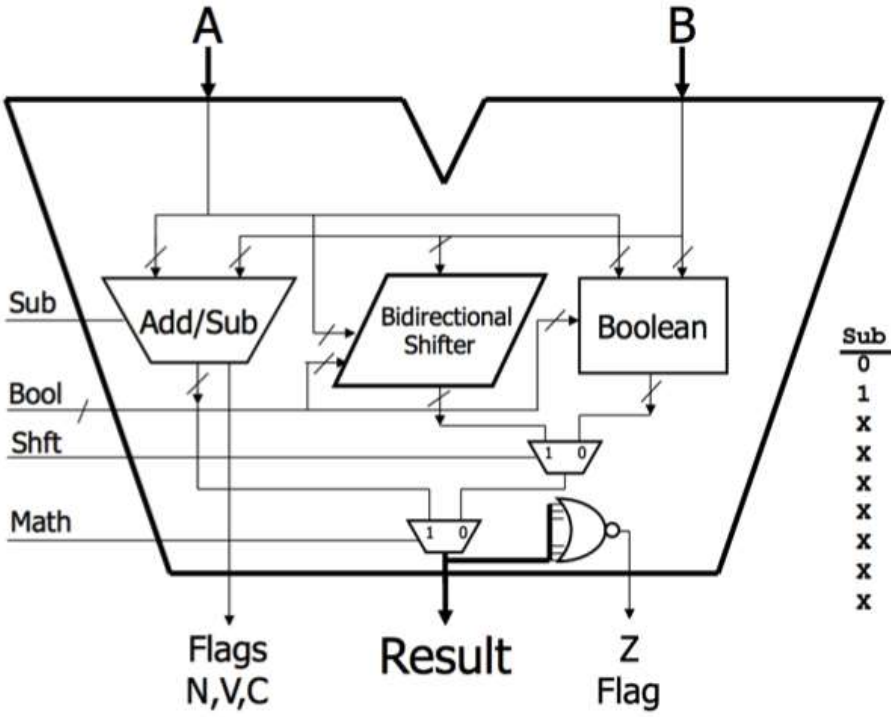
Você precisará definir a largura apenas **uma vez**, onde a `uut` é declarada. Isso sobrescreverá o valor padrão da largura a medida em que for transferida através da hierarquia.

A saída da sua simulação deve ser semelhante àquela apresentada a seguir (após definir a exibição para \textit{Decimal}).



Entendendo a estrutura e operação de uma ULA

A seguir apresentamos um diagrama de bloco para a ULA (Unidade Lógica e Aritmética), juntamente com seu controle de 5-bits.



5-bit ALUFN

Sub	Bool	Shft	Math	OP
0	XX	0	1	A+B
1	XX	0	1	A-B
X	00	1	0	B<<A
X	10	1	0	B>>A
X	11	1	0	B>>>A
X	00	0	0	A & B
X	01	0	0	A | B
X	10	0	0	A ^ B
X	11	0	0	$\overline{A \mid B}$

Sua tarefa é simplesmente revisar as informações sobre as estruturas acima e garantir que você entendeu como os 5-bits de controle codificam as operações, além de como os multiplexadores selecionam o resultado.

> ⚠️ Operações de comparação serão incorporadas na última parte do roteiro.

Módulos Lógicos e de Deslocamento

Utilize os modelos de código a seguir, para completar o circuito de uma unidade lógica e a unidade de deslocamento bi-direcional.

```
module logical #(parameter N=32) (
   input wire [N-1:0] A, B,
   input wire [1:0] op,
   output wire [N-1:0] R
);

   assign R = (op == 2'b00) ?      :
              (op ==      ) ?      :
              (op ==      ) ?      :
              (op ==      ) ?        :    ;

endmodule
```

Coloque o módulo lógico em um arquivo chamado `logical.sv`, e o deslocador em um arquivo chamado `shifter.sv` dentro do diretório `src`.

Você deve utilizar os operadores *bitwise* SystemVerilog que correspondam às quatro operações lógicas listadas na tabela para os 5 bits de controle acima.

> 👷 *Nota de depuração*
>
> O último comando `else` no modelo de código acima não é realmente necessário. Você poderia modificar o código acima para eliminar a última construção `if-else`, ou pode deixar como está e utilizá-la como elemento de depuração.
>
> Especificamente, digamos que aconteceu um erro na codificação de uma instrução, e os dois bits de `op` recebidos foram `1x`. Esse valor não irá coincidir com nenhuma dos casos, e portanto irá cair na cláusula padrão final `else`.
>
> Atribuir um valor **padrão para todos** para esse tipo de situação pode ser útil no futuro. Em mais detalhes, digamos que seu valor padrão corresponde a todos os bits iguais a 1 (o que poderia ser escrito como `{N{1'b1}}`), e então, quando você estiver testando seu circuito completo, nota que a ULA está produzindo um resultado inesperado igual a vários `1's`. Isso poderia lhe direcionar ao problema oriundo de um valor de `op` inválido.

```
module shifter #(parameter N=32) (
   input wire signed [N-1:0] IN,
   input wire [$clog2(N)-1:0] shamt, // arredondando para log2
   input wire left, logical,
   output wire signed [N-1:0] OUT
);

   assign OUT = left ?  (IN << shamt) :
                        (logical ? IN >> shamt : IN >>> shamt);

endmodule
```

No código acima, observe que `IN` é do tipo `signed`, e que o tamanho do deslocamento pode variar de `0` até `N-1`. Portanto, a quantidade de bits em `shamt` foi arredondada usando a função:

$$ceiling(log_2(N))$$

> ✅ Você está usando operadores Verilog para os os três tipo de operação de deslocamento. Observe que o tipo de dado de `IN` é declarado como `signed`. Por que?

Por padrão, os tipos de dado em SystemVerilog são `unsigned`. Entretanto, para que operações de deslocamento aritmético à direita funcionem corretamente, a entrada deve ser declarada como do tipo `signed`, de modo que o bit mais significativo seja utilizado para indicar seu sinal.

Note ainda que o `shamt` só possui `$clog2(N)` bits (ex. 5 bits para operandos de 32 bits).

- `<<` é o operador de deslocamento lógico à esquerda (`sll`)
- `<<` é o operador de deslocamento lógico à direita (`srl`)
- `>>` é o operador de deslocamento aritmético à direita (`sra`)

Você deve agora analisar a estrutura de cada um dos dois módulos, descritos acima, através da visualização dos seus esquemáticos através do RTL Viewer.

Módulo da ULA (sem comparações)

Utilize o código abaixo para projetar uma ULA capaz de realizar operações de soma, subtração, deslocamento e lógicas.

```
module alu #(parameter N=32) (
   input    wire [N-1:0] A, B,
   output   wire [N-1:0] R,
   input    wire [4:0] ALUfn,
   output   wire FlagN, FlagC, FlagV, FlagZ
);

   wire subtract, bool1, bool0, shft, math;

   // Separando ALUfn em bits nomeados
   assign {subtract, bool1, bool0, shft, math} = ALUfn[4:0];

   // Resultados dos três componentes da ALU
   wire [N-1:0] addsubResult, shiftResult, logicalResult;

   addsub   #(N) AS(A, B,      ,         ,       ,      ,      );
   shifter  #(N) S(B, A[$clog2(N)-1:0],       ,       ,      );
   logical  #(N) L(A, B, {         },             );

   // Multiplexador de 4 entradas para selecionar a saída
   assign R =  (~shft & math) ?              :
               (shft & ~math) ?            :
               (~shft & ~math) ?             : 0;

   // Utilize o operador de redução aqui
   assign FlagZ =    ;

endmodule
```

> 🤦 *Dica Importante*
>
> Certifique-se de fornecer corretamente as entradas `left` e `logical` para o deslocador dentro do módulo da ULA. Observe primeiro quais valores entre `bool1` e `bool0` representam os deslocamentos `sll` , `srl` e `sra` . Agora determine como `left` e `logical` devem ser produzidos a partir de `bool1` e `bool0` .

> Isso pode ser um pouco complicado! Complete a tabela verdade a seguir, para determinar a relação que existe entre `{bool1, bool0}` e `{left,logical}`.

Entradas		Saídas	
`bool1`	`bool2`	`left`	`logical`
0	0		
0	1		
1	0		
1	1		

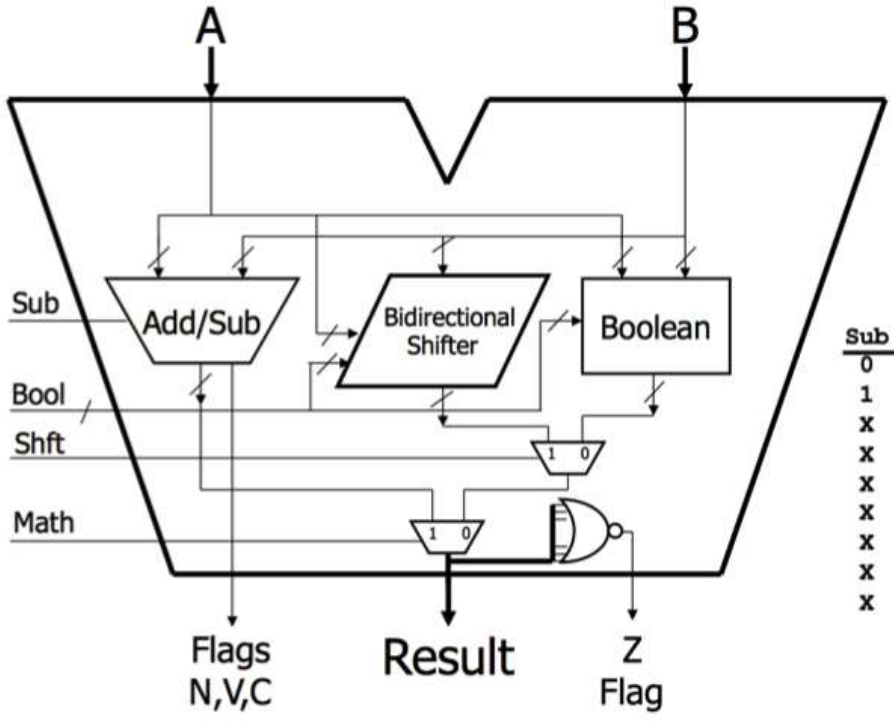
Por fim, escreva uma equação lógica para `left`, e uma para `logical`, em termos de `bool1` e `bool0`. Utilize as expressões para `left` e `logical` que você acabou de desenvolver para completar o módulo `alu`.

Use o *testbench* fornecido (`alu_simple`) para testar a ULA completa. Lembre-se de selecionar **Decimal** como modo de representação (**Radix**), para exibir as entradas `A` e `B`, e a saída `R`. Selecione a representação binária para `ALUfn`.

Modifique a ULA e Inclua Comparações

A seguir são apresentados dois diagramas de blocos para a nossa ULA: aquele que você implementou, e uma versão modificada que você irá desenvolver agora.

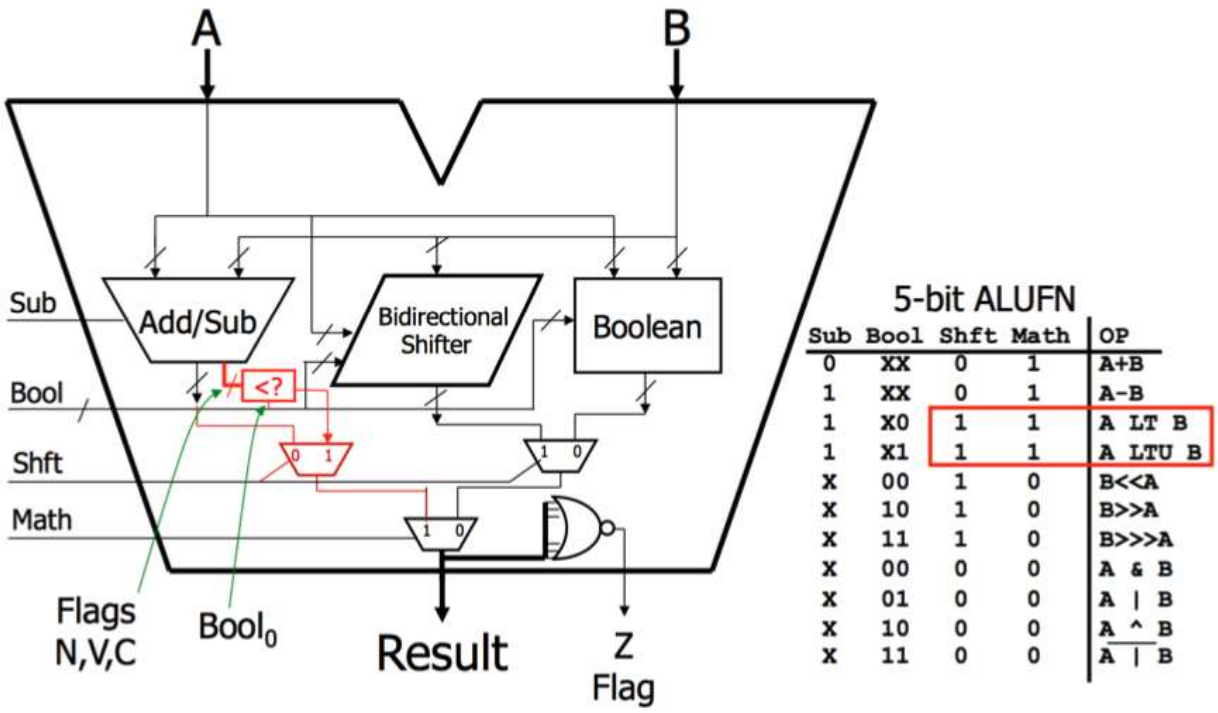
ULA Sem Comparação



5-bit ALUFN

Sub	Bool	Shft	Math	OP
0	XX	0	1	A+B
1	XX	0	1	A-B
X	00	1	0	B<<A
X	10	1	0	B>>A
X	11	1	0	B>>>A
X	00	0	0	A & B
X	01	0	0	A | B
X	10	0	0	A ^ B
X	11	0	0	$\overline{A \mid B}$

ULA Com Comparação:



Sub	Bool	Shft	Math	OP
0	XX	0	1	A+B
1	XX	0	1	A-B
1	X0	1	1	A LT B
1	X1	1	1	A LTU B
X	00	1	0	B<<A
X	10	1	0	B>>A
X	11	1	0	B>>>A
X	00	0	0	A & B
X	01	0	0	A | B
X	10	0	0	A ^ B
X	11	0	0	$\overline{A \mid B}$

A figura acima inclui algumas funcionalidades adicionais. Especificamente, as operações comparação entre operandos `A` e `B`. Ambas as comparações, `signed` e `unsigned` são implementadas:

- *Less-then signed* (LT); e
- *Less-than-unsigned* (LTU).

Perceba as diferenças entre as duas ULAs (novas funcionalidades foram destacadas em **vermelho**).

A comparação entre dois operandos `A` e `B`, é realizada a partir da subtração (`A-B`) seguida da geração das *flags* (`N`, `V`} e `C`).

Observe que o bit `Sub` de `ALUFN` está ativo. O bit menos significativo de `Bool` determina quando a comparação é `signed` ou `unsigned`. Quando comparando números `unsigned`, o resultado para `A-B` é negativo se e somente se *o carry out mais significativo do somador-subtrator (ou seja, a `flag C`) é igual a `0`*.

Não pode haver *overflow* quando dois números positivos são subtraídos, mas quando os números estão sendo comparados como *signed* (ou seja, na notação de complemento de 2), o resultado da operação `A-B` é negativo se:

1. tanto o resultado possui seu bit negativo (*flag* `N` ativo) **e** não houve *overflow* na operação; ou
2. o resultado é positivo e não houve *overflow* (*flag* `V` ativa).

Para implementar a nova ULA, primeiro crie um módulo chamado `comparator` em um novo arquivo SystemVerilog chamado `comparator.sv`. Aqui está o esqueleto do código:

```
module comparator (
   input wire FlagN, FlagV, FlagC, bool0,
   output wire comparison
);

   assign comparison =                ;

endmodule
```

Em seguida, modifique o módulo `alu` para incluir uma instância do comparador que você acabou de projetar, e, em seguida, modifique a linha `assign R` para torná-lo um multiplexador de 4 entradas, no lugar do multiplexador de 3 entradas original.

> 🎯 O último `else` da ULA pode ser usado agora para lidar com o resultado do comparador!

Tenha em mente que o resultado do comparador é um único bit `0` ou `1`, mas o resultado da ULA é multi-bit. Para evitar alertas do compilador neste quesito, *dentro do módulo alu*, você deve completar o resultado de 1 bit do comparador com `(N-1)` zeros à esquerda. A expressão SystemVerilog que contempla tal operação é:

```
{{(N-1){1'b0}}, compResult}
```

A expressão `{(N-1)\{1'b0\}\}` replica um bit zero `N-1` vezes e então concatena o resultado de 1 bit do comparador a ele.

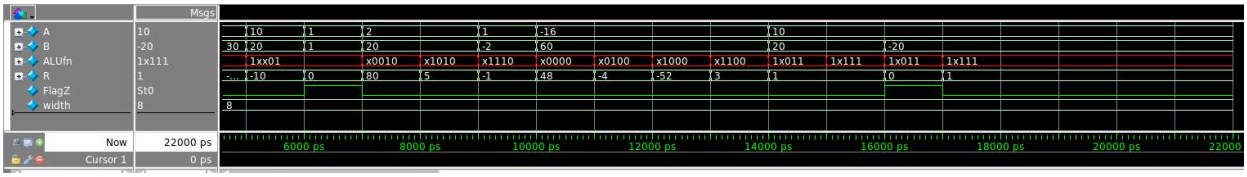
Finalmente, elimine as *flags* `N`, `V` e `C` da saída da ULA. Essas *flags* são usadas apenas para instruções de comparação, e, uma vez que o comparador foi incluído dentro da ULA, elas não são mais necessárias. A *flag* `Z`, entretanto, ainda precisa ser uma saída.

Simulando a ALU

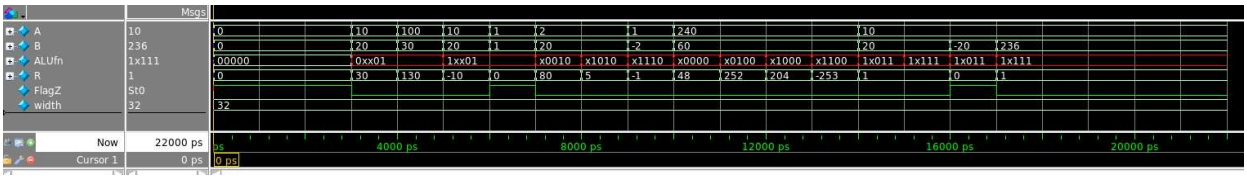
Utilize o *testbench* fornecido junto com os arquivos de laboratório para testar sua ULA (em ambas as versões de 8-bits e 32-bits).

Lembre-se de selecionar a representação "decimal" para exibir as entradas `A` e `B`, e a saída `R`. Selecione a representação "binária'' para representar `ALUfn`.

Primeiro, você deve testar sua ULA com a largura definida como 8 bits (use o arquivo `alu_8bit_tb.sv`). Se tudo der certo, você deve visualizar uma forma de onda como a que segue.



Em seguida, teste seu projeto utilizando a versão 32-bits do testador (`alu_32bit_tb.sv`). Se tudo correr bem, novamente você deve visualizar uma forma de onda como a que segue.



Alguns resultados podem mudar, como por exemplo, a operação `100+30` , que provocava um *overflow,* quando em 8 bits, graças à limitação na representação com sinal, não causará o mesmo efeito com 32 bits.

> 🤦 Observe, cuidadosamente, todas as diferenças entre as saídas de 8 bits e de 32 bits e verifique se você é capaz de explicá-las.

Visualização Esquemático da ALU

Encerre a simulação, e crie uma visualização do esquemático da sua ULA. Tente relacionar o esquemático produzido com a estrutura desses componentes.

> 💬 Você pode também comparar com o esquemático da ULA apresentado no roteiro.

> ✅ Você ainda vai aprender muito sobre como relacionar o esquemático do circuito com a descrição Verilog, e como identificar as correspondências entre os componentes do circuito e as linhas do código Verilog. Isso será importante para os futuros projetos!

Acompanhamento (entrega: quarta-feira 11 de maio, 2022)

O que entregar:

- Sua resposta para a pergunta na seção Módulos Lógicos e de Deslocamento
- Uma captura de tela das formas de onda da simulação, mostrando claramente **os resultados da simulação final tanto para a versão 8-bits, quanto para a versão 32-bits**.
- Seu código Verilog para os módulos: `ula` , `comparator` , `logical` e `shifter` .
- Uma figura do esquemático do seu circuito para o projeto final (32-bits).

Como entregar:

Submeta suas respostas através da tarefa no Gradescope

Agradecimentos

Este laboratório é o resultado do trabalho de docentes e monitores de GCET231 ao longo dos anos, incluindo:

- **18.1:** Caio França dos Santos
- **18.2:** Matheus Rosa Pithon
- **20.2:** Matheus Rosa Pithon
- **21.1:** Matheus Rosa Pithon, Éverton Gomes dos Santos
- **21.2:** Éverton Gomes dos Santos