

DOCUMENTAÇÃO CALCULADORA:

```
***** Parser Output *****
Parsing successful!
Calculadora | 1 + | 2 - | 3 * | 4 / -> | 5 Stop -> 1
Digite um numero: 25
Digite um numero: -5
0 resultado eh: 20

Calculadora | 1 + | 2 - | 3 * | 4 / -> | 5 Stop -> 2
Digite um numero: 25
Digite um numero: -5
0 resultado eh: 30

Calculadora | 1 + | 2 - | 3 * | 4 / -> | 5 Stop -> 3
Digite um numero: 25
Digite um numero: -5
0 resultado eh: -125

Calculadora | 1 + | 2 - | 3 * | 4 / -> | 5 Stop -> 4
Digite um numero: 25
Digite um numero: -5
0 resto eh: 0
0 resultado eh: -5

Calculadora | 1 + | 2 - | 3 * | 4 / -> | 5 Stop -> 5
```

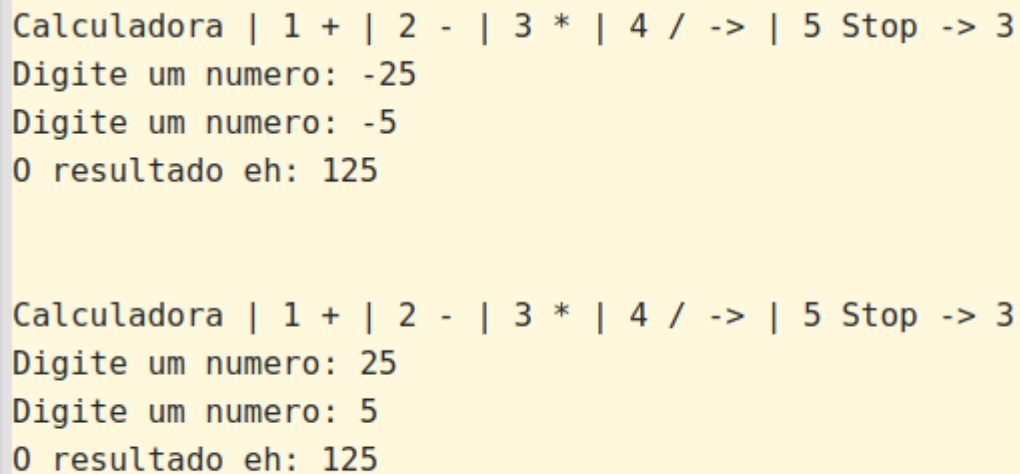
a) Calculadora funcionando com todas suas operações

Aqui temos um exemplo de output para cada uma das operações que a calculadora realiza. Foi decidido para esse código a execução repetida de operações até que o usuário deseje parar, ou seja, após realizar uma operação, voltamos para o laço inicial onde é pedida uma nova operação. Essa decisão foi tomada pois como se trata de uma calculadora faz sentido querer utilizar mais de uma vez sem ter que reiniciar o processo todas as vezes, apesar de que isso não interfere no fluxo do código.

O código é capaz de lidar com os sinais e o exemplo da imagem a) evidencia isso bem.

$25 + (-5) = 20$, enquanto $25 - (-5) = 25 + 5 = 30$. Esses casos tem o sinal tratado apenas na hora de obter o resultado, sendo assim: se resultado < 0 , então invertemos ele para seu complemento, imprimimos o “-” e o valor obtido.

Para o caso da divisão e multiplicação a mesma ideia é aplicada, mas para esses casos é feita uma contagem dos números negativos e, se temos apenas 1 dos operandos negativos, então o resultado também é negativo, para demonstrar isso melhor segue a imagem:



The image contains two screenshots of a calculator interface. The first screenshot shows the calculator menu with options 1 (+), 2 (-), 3 (*), 4 (/), and 5 (Stop). The user has selected the multiplication function (3 *). The prompt 'Digite um numero:' is followed by '-25'. The second prompt 'Digite um numero:' is followed by '-5'. The final output is 'O resultado eh: 125'. The second screenshot shows the same calculator menu. The user has selected the multiplication function (3 *). The prompt 'Digite um numero:' is followed by '25'. The second prompt 'Digite um numero:' is followed by '5'. The final output is 'O resultado eh: 125'.

b) Checagem de sinal para caso de multiplicação / divisão

Podemos ver que a validação de sinal funciona e ela é independente da ordem da entrada dos operandos, ou seja, $(-25) * 3$ ou $25 * (-3)$ possui resultados equivalentes. Isso acontece porque verificamos os operandos fornecidos, então qualquer número negativo é transformado na sua representação positiva.

Outra funcionalidade interessante de ser analisada é a divisão

```

***** Parser Output *****
Parsing successful!
Calculadora | 1 + | 2 - | 3 * | 4 / -> | 5 Stop -> 4
Digite um numero: -30
Digite um numero: 6
0 resto eh: 0
0 resultado eh: -5

Calculadora | 1 + | 2 - | 3 * | 4 / -> | 5 Stop -> 4
Digite um numero: 34
Digite um numero: 7
0 resto eh: 6
0 resultado eh: 4

Calculadora | 1 + | 2 - | 3 * | 4 / -> | 5 Stop -> 4
Digite um numero: 100
Digite um numero: 0
Divisao por 0

```

c) Operações de divisão

Notamos que ele fornece a informação de qual é o resto obtido bem como o “resultado inteiro” da divisão. No primeiro caso temos uma divisão exata, cujo um dos operandos é negativo, então temos o resto 0 e o resultado negativo. No segundo, por sua vez, a divisão possui um resto, então podemos obter as duas informações pela calculadora. Por último, existe a validação de divisão por 0, nesse caso, o usuário é alertado e nenhuma operação é calculada.

No mais, a calculadora funciona para as operações mencionadas (números pequenos foram usados nos exemplos para facilitar a validação dos resultados). A execução do programa é simples, basta que o usuário digite **CORRETAMENTE** uma das operações possíveis e fornecidas pela mensagem do menu. Após isso, será solicitado a entrada de dois números - para essa calculadora, tratamos os números decimais - por fim, o usuário terá o resultado exibido na tela.

Um atalho para os usuários que desejam executar o programa no simulador (<https://ascslab.org/research/briscv/simulator/simulator.html>). Adicione um **breakpoint** na linha 56 (label loop) e linha 65 (solicitação da operação), após isso também é possível adicionar um breakpoint no ecall dos valores. Fazendo isso, é possível pular a execução e ver os resultados mais rápido.

DOCUMENTAÇÃO CONVERSÃO:

```
1 dec -> hex | 2 hex -> dec | 3 dec -> bin | 4 bin -> dec | 5 bin -> hex | 6 hex -> bin. -> 1
Insira o numero:30
0 resultado eh: 0x0000001e

1 dec -> hex | 2 hex -> dec | 3 dec -> bin | 4 bin -> dec | 5 bin -> hex | 6 hex -> bin. -> 2
Insira o numero:0x0000001e
0 resultado eh: 30

1 dec -> hex | 2 hex -> dec | 3 dec -> bin | 4 bin -> dec | 5 bin -> hex | 6 hex -> bin. -> 3
Insira o numero:30
0 resultado eh: 11110

1 dec -> hex | 2 hex -> dec | 3 dec -> bin | 4 bin -> dec | 5 bin -> hex | 6 hex -> bin. -> 4
Insira o numero:11110
0 resultado eh: 30

1 dec -> hex | 2 hex -> dec | 3 dec -> bin | 4 bin -> dec | 5 bin -> hex | 6 hex -> bin. -> 5
Insira o numero:11110
0 resultado eh: 0x0000001e

1 dec -> hex | 2 hex -> dec | 3 dec -> bin | 4 bin -> dec | 5 bin -> hex | 6 hex -> bin. -> 6
Insira o numero:0x0000001e
0 resultado eh: 11110
```

d) Funcionamento das operações de conversão

Inicialmente, como não existiu uma especificação explícita para o uso sequencial de operações de conversão, o programa suporta uma conversão por vez.

O uso é simples, basta que o usuário escolha qual é a operação e após isso entre com um valor válido na base digitada, sendo:

- **Decimal:** Um número decimal
- **Binário:** Uma sequência de 0's e 1's - Até 32 bits
- **Hexadecimal:** Um número no formato 0x... (Note que o 0x **DEVE** ser digitado).

Após isso, o algoritmo fará todas as operações necessárias para a conversão. Muitas funções foram reutilizadas entre as mudanças de bases, algumas funções poderiam ser reescritas para serem reutilizadas, porém em questão de legibilidade e custo benefício muitas ideias acabaram não sendo implementadas!

Na imagem d) podemos ver uma execução de cada conversão, o mesmo número (30) foi usada para facilitar a visualização.

- **Decimal:** 30
- **Binário:** 11110
- **Hexadecimal:** 0x1E

Um atalho para os usuários que desejam executar o programa no simulador (<https://ascslab.org/research/briscv/simulator/simulator.html>). Adicione um **breakpoint** na linha 88 (solicitação da conversão desejada) e também é possível adicionar um breakpoint no ecall dos valores. Fazendo isso, é possível pular a execução e ver os resultados mais rápido. Depois disso, o código irá finalizar, mas basta clicar na opção de “restart” do simulador.