- Sobre o circuito em geral:

O objetivo do circuito é de atender a todas as especificações da atividade 2 através de software (no caso assembly) que roda dentro desse circuito.

Eu sei que existem diversas áreas do circuito que poderiam ser melhoradas em questão de complexidade (redução de chips e subcircuitos), porém, meu objetivo com o projeto é de conseguir implementar uma versão funcional do circuito que atenda as especificações da atividade 2 no menor tempo possível.

Como a implementação desse computador foi realizada seguindo os projetos do livro “The Elements of Computing Systems: Building a Modern Computer from First Principles”, tentei ao máximo me manter fiel a implementação vertical que parte apenas de uma porta lógica NAND. Em geral todos os circuitos e subcircuitos usados no projeto são construídos a partir de outros subcircuitos que eu implementei, e na base de todas essas implementações está a porta lógica NAND. Essa filosofia de implementação foi possível de ser realizada pela maior parte do circuito, mas devido a limitações do *logisim*, que no caso trava com implementações de RAM com mais de 64 registradores, eu tive que utilizar os chips de memória RAM e ROM padrões do *logisim*.

- Sobre o NTT_BasicGates.circ:

Essa biblioteca do *logisim* possui portas lógicas básicas utilizadas ao longo do projeto. A implementação delas foi feita apenas através de outras portas lógicas ou da porta lógica NAND.

- Sobre o NTT_ALU.circ:

Essa biblioteca possui a implementação dos chips que realizam lógica aritmética, sendo a peça central desse arquivo a ALU.

Por algum motivo esquisito que eu desconheço, o *logisim* aleatoriamente carrega o chip da ALU com erro, e durante a execução do programa erros em vermelho começam a aparecer vindos da ALU. Porém, ao deletar e acrescentar novamente a ALU ao circuito, os erros desaparecem. Vale lembrar que o chip da ALU não possui nenhum sistema sequencial dependente em estados anteriores, então eu acho plausível que o erro seja algum bug do *logisim*.

- Sobre o NTT_Memory.circ:

Nessa biblioteca foram implementados os chips de memória que irão ser utilizados no chip do computador final. Durante a implementação desses chips ficou evidente que o logisim não foi feito para simular chips de memória com alta quantidade de componentes, devido a isso, eu acabei decidindo implementar apenas os chips RAM8 e RAM64 para ao menos demonstrar como eu teria feito a implementação do resto dos chips, afinal, a lógica de implementação de chips com mais memórias se manteria a mesma, com demultiplexadores e multiplexadores controlando o acesso a diferentes endereços da memória. Para a implementação do maior chip de memória, o chip RAM16K, eu apenas encapsulei um chip de RAM padrão do *logisim*.

- Sobre o NTT_FullComputer.circ:

Esse é o arquivo principal do projeto dividido em duas partes: A primeira, contém os chips “FullMemory”, “CPU” e “HackPC”, que são a implementação do projeto do Capitulo cinco do livro onde o computador final é montado. A segunda, contém os chips “DebugHackPC” e “HackPCJockenpo”, que são os chips utilizados para atender os requisitos da atividade 2. No

caso, o “DebugHackPC” foi criado devido a necessidade de visualizar a operação de acesso e escrita da memória para que eu conseguisse “debugar” o programa, enquanto o “HackPCJockenpo” encapsula o circuito do “DebugHackPC” para fácil visualização e testes do programa.

- Desenvolvimento do software:

Como o computador desenvolvido atende a todos os requisitos dos projetos do livro, é possível utilizar o código de máquina gerado pelo “assembler" fornecido no site do livro. Por sorte os arquivos “.circ” salvos pelo *logisim* são na realidade arquivos XML, então editar e adicionar código para a ROM do computador foi fácil, bastou adicionar os comandos em hexadecimal dentro da área do XML que representava o conteúdo da ROM. O código em “assembly” usado no projeto se encontra no final desse documento.

- Como testar o programa:

Para testar o programa basta modificar os 8 bits de input dentro do arquivo “DebugHackPC” ou do arquivo “DebugHackPC”, que estão virados para baixo no circuito. Sua funcionalidade está descrita a baixo em ordem, da direita para a esquerda. Também descrito abaixo estão os bits de output, que também estão descritos da direita para a esquerda.

-----Bits de input-----	-----Bits de Output-----
Bit 1: J1Pedra	Bit 1: PontuacãoJ1
Bit 2: J1Tesoura	Bit 2: PontuacãoJ1
Bit 3: J1Papel	Bit 3: PontuacãoJ2
Bit 4: J2Pedra	Bit 4: PontuacãoJ2
Bit 5: J2Tesoura	Bit 5: VencedorJ1
Bit 6: J2Papel	Bit 6: VencedorJ2
Bit 7: Jogar	Bit 7: Erro
Bit 8: Reset	Bit 8: Processando

Obs1: O bit8 de output “Processando” acende se o computador está processando as instruções enviadas pelo input, e desliga quando o sistema estiver pronto para receber o próximo input. É com base nesse bit que é possível identificar se a jogada foi empate.

Obs2: O bit 7 de input “Jogar” precisa ser desligado e ligado novamente para fazer com que a jogada seja computada. Isso impede que jogadas indesejadas não sejam realizadas.

Obs3: Existe um Bit de “ResetCPU” que reseta o registrador de contagem do programa. Se ativado, ele reseta o programa para o inicio sem alterar os registradores da memoria RAM. Em geral não deve ser usado.

- Links:

Nand2Tetris: https://www.nand2tetris.org/

- Código em “assembly”:

```
(LISTEN_INPUT)
    @KBD
    D=M
    @R0 //input salvo no R0
    M=D
    @64 //0100-0000->Play
    D=D&A
    @LISTEN_INPUT
    D;JEQ
    @R3 //Zera o registrador de output
    M=0
    @R4 //zera o registrador usado para calcular os erros
    M=0
(CHECK_FOR_RESET)
    @R0
    D=M
    @128
    D=D&A
    @CHECK_FOR_VICTORY
    D;JEQ //se D não tem o bit reset ligado, não roda o codigo a seguir
    @R1
    M=0
    @R2
    M=0
    @R3
    M=0
    @R4
    M=0
    @OUTPUT_RESULT
    0;JMP

(CHECK_FOR_VICTORY)
    @R1
    D=M
    @768
    D=A-D
    @PLAYER1_WIN //se o jogador 1 ganhou, nao roda o codigo a seguir
    D;JEQ
    @R2
    D=M
    @3072
    D=A-D
    @PLAYER2_WIN //se o jogador 2 ganhou, nao roda o codigo a seguir
    D;JEQ
(CHECK_FOR_ERROR)

    @R0
    D=M
    @1
    D=D&A
    @SKIP_SUM_1
    D;JEQ
    @R4
    M=M+1

    (SKIP_SUM_1)
    @R0
    D=M
    @2
```

```
    D=D&A
    @SKIP_SUM_2
    D;JEQ
    @R4
    M=M+1

    (SKIP_SUM_2)
    @R0
    D=M
    @4
    D=D&A
    @SKIP_SUM_3
    D;JEQ
    @R4
    M=M+1
    (SKIP_SUM_3)

    @R4
    D=M
    M=0
    D=D-1
    @CHECK_PLAYER2_INPUT
    D;JEQ //se a quantidade de inputs for igual a 2, pula para play sem deixar o erro exec

    @16384
    D=A
    @R3
    M=D

    @OUTPUT_RESULT
    0;JMP

    (CHECK_PLAYER2_INPUT)
    @R0
    D=M
    @8
    D=D&A
    @SKIP_SUM_4
    D;JEQ
    @R4
    M=M+1

    (SKIP_SUM_4)
    @R0
    D=M
    @16
    D=D&A
    @SKIP_SUM_5
    D;JEQ
    @R4
    M=M+1

    (SKIP_SUM_5)
    @R0
    D=M
    @32
    D=D&A
    @SKIP_SUM_6
    D;JEQ
    @R4
    M=M+1
```

```
    (SKIP_SUM_6)

    @R4
    D=M
    @1
    D=D-A
    @PLAY
    D;JEQ //se a quantidade de inputs for igual a 2, pula para play sem deixar o erro exec

    @16384
    D=A
    @R3
    M=D

    @OUTPUT_RESULT
    0;JMP
(PLAY)
    //-----Player 1 pedra-----//
    @R0
    D=M
    @1
    D=D&A
    @PLAYER1_PEDRA
    D;JNE
    //-----Player 1 pedra-----//

    //-----Player 1 tesoura-----//
    @R0
    D=M
    @2
    D=D&A
    @PLAYER1_TESOURA
    D;JNE
    //-----Player 1 tesoura-----//

    //-----Player 1 papel-----//
    @R0
    D=M
    @4
    D=D&A
    @PLAYER1_PAPEL
    D;JNE
    //-----Player 1 papel-----//

(PLAYER1_PEDRA)
    //-----Player 2 pedra-----//
    @R0
    D=M
    @8
    D=D&A
    @DRAW
    D;JNE
    //-----Player 2 pedra-----//

    //-----Player 2 tesoura-----//
    @R0
    D=M
    @16
    D=D&A
    @PLAYER1_SCORE
    D;JNE
```

```
    //-----Player 2 tesoura-----//

    //-----Player 2 papel-----//
    @R0
    D=M
    @32
    D=D&A
    @PLAYER2_SCORE
    D;JNE
    //-----Player 2 papel-----//
(PLAYER1_TESOURA)
    //-----Player 2 pedra-----//
    @R0
    D=M
    @8
    D=D&A
    @PLAYER2_SCORE
    D;JNE
    //-----Player 2 pedra-----//

    //-----Player 2 tesoura-----//
    @R0
    D=M
    @16
    D=D&A
    @DRAW
    D;JNE
    //-----Player 2 tesoura-----//

    //-----Player 2 papel-----//
    @R0
    D=M
    @32
    D=D&A
    @PLAYER1_SCORE
    D;JNE
    //-----Player 2 papel-----//
(PLAYER1_PAPEL)
    //-----Player 2 pedra-----//
    @R0
    D=M
    @8
    D=D&A
    @PLAYER1_SCORE
    D;JNE
    //-----Player 2 pedra-----//

    //-----Player 2 tesoura-----//
    @R0
    D=M
    @16
    D=D&A
    @PLAYER2_SCORE
    D;JNE
    //-----Player 2 tesoura-----//

    //-----Player 2 papel-----//
    @R0
    D=M
    @32
    D=D&A
```

```
    @DRAW
    D;JNE
    //-----Player 2 papel-----//

(PLAYER1_SCORE)
    //-----Pontua-----//
    @R1
    D=M
    @256
    D=D+A
    @R1
    M=D
    //-----Pontua-----//

    //-----Verifica vitoria geral-----//
    @R1
    D=M
    @768
    D=A-D
    @PLAYER1_NO_WIN //se o jogador 1 nao ganhou, nao roda o codigo a seguir
    D;JNE

    //-----PLAYER1 WIN-----//
    (PLAYER1_WIN)
    @R3
    D=M
    @4096
    D=D+A
    @R3
    M=D
    //-----PLAYER1 WIN-----//

    (PLAYER1_NO_WIN)
    //-----Verifica vitoria geral-----//
    @OUTPUT_RESULT
    0;JMP
(PLAYER2_SCORE)
    //-----Pontua-----//
    @R2
    D=M
    @1024
    D=D+A
    @R2
    M=D
    //-----Pontua-----//

    //-----Verifica vitoria geral-----//
    @R2
    D=M
    @3072
    D=A-D
    @PLAYER2_NO_WIN //se o jogador 2 nao ganhou, nao roda o codigo a seguir
    D;JNE

    //-----PLAYER2 WIN-----//
    (PLAYER2_WIN)
    @R3
    D=M
    @8192
    D=D+A
    @R3
```

```
    M=D
    //-----PLAYER2 WIN-----//

    (PLAYER2_NO_WIN)
    //-----Verifica vitoria geral-----//
    @OUTPUT_RESULT
    0;JMP

(DRAW)
(OUTPUT_RESULT)
    //-----Seta o placar-----//
    @R1
    D=M
    @R3
    M=M+D
    @R2
    D=M
    @R3
    M=M+D
    //-----Seta o placar-----//

    //-----Passa o R3 para output-----//
    @R3
    D=M
    @KBD
    M=D
    //-----Passa o R3 para output-----//

    @LISTEN_INPUT //volta a ouvir o input
    0;JMP


//-----FIM-----//
(END)
@END
0;JMP
//-----FIM-----//
```