PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE RORAIMA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

RELATÓRIO DO PROJETO: PROCESSADOR MK 10

ALUNOS:
KEVIN WILLYN CONCEICAO BARROS – 2019016958
MATHEUS NARANJO CORREA - 2019001187

Maio de 2021
Boa Vista/Roraima

PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE RORAIMA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

RELATÓRIO DO PROJETO: PROCESSADOR MK 10

Maio de 2021
Boa Vista/Roraima

RESUMO

O seguinte trabalho aborda o projeto que visa a implementação de um microprocessador sem estágios intertravados de pipeline (MIPS) utilizando uma arquitetura alternativa voltada para 8 bits, em consonância com a sua versão mais utilizada de 32 bits. Ao longo do trabalho serão descritos todos os estágios de funcionamento e desenvolvimento do microprocessador nomeado de MK 10, assim como as waveforms e resultados de testes implementados em cada componente que compõe o projeto. Ao todo, o MK 10 é composto de 13 instruções divididas em três formatos de instruções, que foram construídas utilizando a linguagem *VHSIC Hardware Description Language,* conhecida como VHDL.

Palavras-chave: MIPS, 8bits, processador, VHDL.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1 ESPECIFICAÇÃO

Nesta seção é apresentado o conjunto de itens para o desenvolvimento do processador MK 10, bem como a descrição detalhada de cada etapa da construção do processador. Em cada tópico, será abordado de maneira aprofundada os componentes que o compõe, visando simplificar e facilitar o entendimento da construção de um processador MIPS.

2 PLATAFORMA DE DESENVOLVIMENTO

Para a implementação do processador MK 10 foi utilizado a IDE ***Intel Quartus Prime Lite Edition***, na versão 20.1. Das especificações do programa temos:

Flow Status	Successful - Thu Apr 29 19:04:17 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	processador_MK10
Top-level Entity Name	processador_MK10
Family	Cyclone V
Device	5CGXFC7C7F23C8
Timing Models	Final
Logic utilization (in ALMs)	N/A
Total registers	72
Total pins	69
Total virtual pins	0
Total block memory bits	0
Total DSP Blocks	1
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0
Total DLLs	0

Figura 1 -Especificações no Intel Quartus Prime Lite Edition

Quanto ao simulador utilizado, optou-se pelo uso do simulador **ModelSim Altera**, utilizando a linguagem VHDL para a construção dos códigos.

2.1 CONJUNTO DE INSTRUÇÕES

O processador MK 10 possui 4 registradores: $S0, $S1, $S2, $S3. Assim como 3 formatos de instruções de 8 bits cada, Instruções do tipo R, I, J, seguem algumas

considerações sobre as estruturas contidas nas instruções:

- **Opcode**: a operação básica a ser executada pelo processador, tradicionalmente chamado de código de operação;
- **Reg1**: o registrador contendo o primeiro operando fonte e adicionalmente para alguns tipos de instruções (ex. instruções do tipo R) é o registrador de destino;
- **Reg2**: o registrador contendo o segundo operando fonte e que também é utilizado para realizar operações do tipo I, voltada para uso de operações imediatas.

2.1.1 TIPOS DE INSTRUÇÕES

Dentro da arquitetura MIPS, temos os seguintes tipos de instruções:

TIPO R: este formato é voltado para instruções que não requerem um endereço de destino, valor imediato ou mesmo deslocamento de bits. Com ele, podemos realizar operações aritméticas, como soma, subtração e multiplicação, além de operações lógicas. Originalmente nele, temos as divisões em seis campos (isso para o formato de 32 bits). Contudo, em uma adaptação para 8 bits, temos:

OPCODE	REG1	REG2
4 bits	2 bits	2 bits
7 - 4	3 - 2	1-0

Tabela 1 - Formatação de Instruções do Tipo R.

TIPO I: este formato permite operações imediatas, além de um operações de memória e manipulação de desvios condicionais. Nela, podemos realizar soma e subtração de forma imediata, não sendo necessário armazenar o valor dentro de um registrador e sim passar o número desejado diretamente, por exemplo. Temos então sua forma de escrita:

OPCODE	REG1	REG2
4 bits	2 bits	2 bits
7 - 4	3 - 2	1-0

Tabela 2 -Formatação de Instruções do Tipo I

TIPO J: Realiza apenas instruções de salto que ocorrem sempre que a instrução que o especificou é executada, também conhecidas como desvios incondicionais. Como exemplo, temos a instrução Jump. Sua forma de escrita fica assim:

OPCODE	ENDEREÇO
4 bits	4 bits
7 - 4	3-0

Tabela 3 -Formatação de Instruções do Tipo J

2.1.2 VISÃO GERAL DAS INSTRUÇÕES DO PROCESSADOR MK 10

O processador MK 10 é composto de um total de 13 instruções voltadas para uma estrutura limitada a 8 bits. Temos a tabela dividida em sua respectiva representação de instruções que vão de 0 até 12 em binário.

OPCODE	NOME	FORMATO	INSTRUÇÃO	EXEMPLO
0000	ADD	R	**SOMA**	ADD $S0, $S1
0001	ADDI	I	**SOMA IMEDIATA**	ADDI $S0, 2
0010	SUB	R	**SUBTRAÇÃO**	SUB $S0, $S1
0011	SUBI	I	**SUBTRAÇÃO IMEDIATA**	SUBI $S0, 5
0100	MUL	R	**MULTIPLICAÇÃO**	MUL $S0, $S1
0101	LW	I	**LOAD WORD**	LW $S0 MEM (00)
0110	SW	I	**STORE WORD**	SW $S0 MEM (00)
0111	MOVE	R	**MOVER**	MOVE $S0, $S1
1000	li	I	**LOAD IMEDIATO**	LI $S0 1
1001	BEQ	J	**DESVIO CONDICIONAL**	BEQ ENDEREÇO
1010	BNE	J	**DESVIO CONDICIONAL**	BNE ENDEREÇO
1011	IF BEQ E BNE	R	**CONDIÇÃO PARA DESVIO**	IF $S0 $S1
1100	J	J	**DESVIO INCONDICIONAL**	J ENDEREÇO (0000)

Tabela 4 Visão Geral de Instruções do Processador MK 10

3 DESCRIÇÃO DO HARDWARE

Passaremos a descrever cada um dos componentes que fazem parte do arcabouço que constitui no processador. Cada tópico temo por objetivo, descrever o funcionamento dos trilhos e sua importância para o funcionamento geral do projeto, seguindo uma certa ordem de complexidade.

3.1.1 ALU ou ULA

A ULA significa Unidade Lógica e Aritmética (ou em inglês, Arithmetic Logic Unit, por isso ALU) e é a parte do computador responsável por realizar operações lógicas e aritméticas sobre os dados. É a principal responsável por gerenciar partes fundamentais do computador, desde uma pequena conta de soma de bits até mesmo operações que envolvam um conjunto mais complexo operações matemáticas. Os demais componentes trabalham em função de fornecer dados para que a ULA processe e tome prosseguimento do que deve acontecer em seguida. Dentro dela, podemos realizar operações de soma de bits, subtração, multiplicação e operações booleanas AND, OR, XOR e NOT por exemplo. É o núcleo do computador. Ela recebe instruções de entrada com os dados provindos de outros componentes e gera os resultados a serem aplicados no resto do processador. A visão geral de nossa ULA possui este formato:

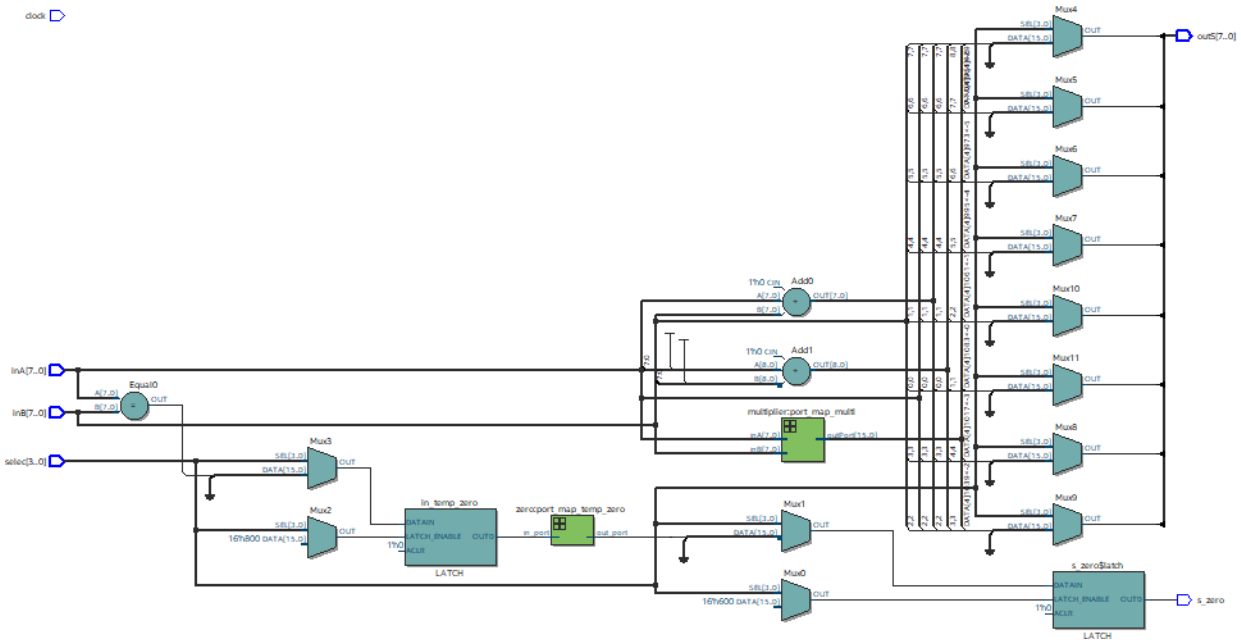


Figura 2 -RTL Viewer da Unidade Lógica Aritmética (ULA)

Em nossa ULA, recebemos valores provindos do banco de registradores que possuem um tamanho de 8 BITS e que servirão como variáveis X e Y para a realização das operações. Além disso, há uma entrada voltada para a entrada da flag ALUop que é responsável por comunicar qual Opcode está sendo inserido, indicando quais operações serão realizadas. Possuímos ainda o componente denominado zero (explicado mais abaixo) também está inserido dentro do componente da ULA. Os resultados de saída gerados pela ULA podem ser armazenados em dois lugares: Memória RAM ou no banco de registradores.

3.1.2 BANCO DE REGISTRADORES

Os registradores precisam ser alocados em algum local. O banco de registradores ser responsabiliza por isso. Seu funcionamento é simples: ele recebe duas entradas de registradores provindas da divisão de instrução, cada uma possuindo 2 bits de espaço.

Essas instruções podem ser repassadas para a ULA, logo, gera duas saídas de Bits que são repassadas para ULA. Como dito anteriormente, os resultados da ULA podem ser gravados novamente no banco de registrados, como em um caso de operação de soma, onde, por conta da arquitetura do MIPS, podemos escrever os dados diretamente da ULA para o banco de registradores.

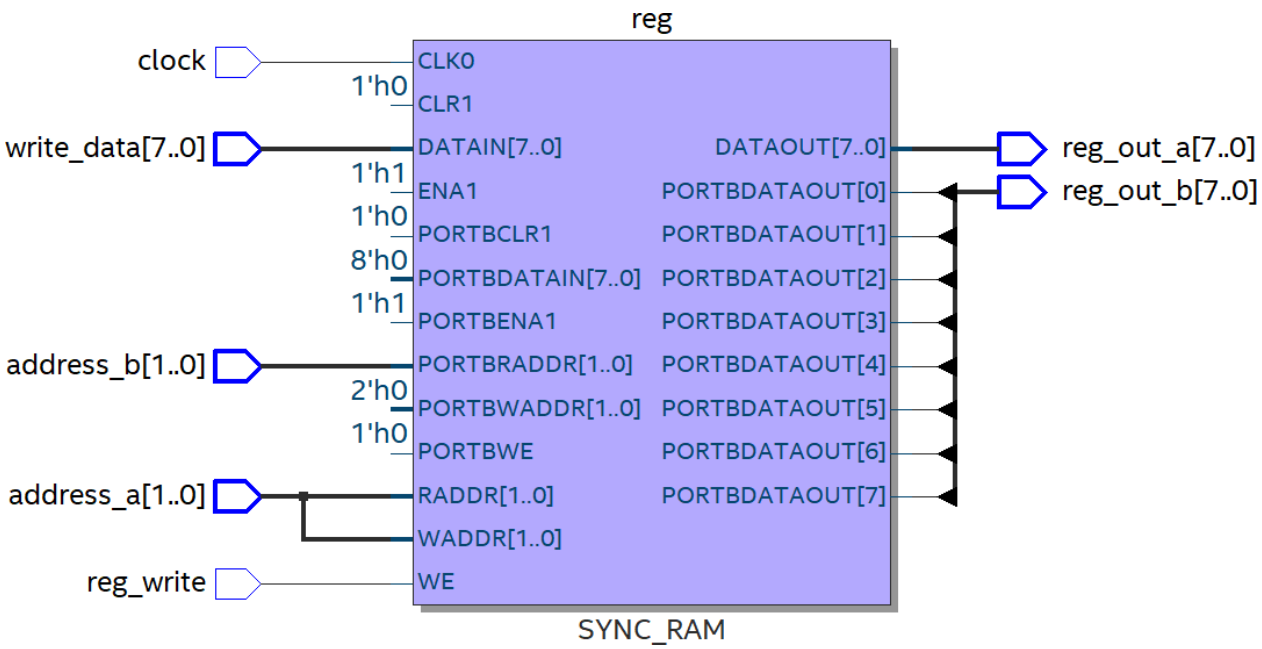


Figura 3 - RTL Viewer do Banco de Registradores

Explicando mais detalhadamente o funcionamento, temos, **address_a** e **address_b** que são as entradas recebidas, cada uma com 2 bits. O componente ainda possui uma Flag ligada a ele, que é ativada pelo **clock**. A **RegWrite** é responsável por indicar o uso do banco de registradores, sendo usada na maioria das operações existentes no MK 10. Quando possui valor 1 indica que o banco será utilizado para armazenar ou repassar as informações para ULA ou mesmo realizar ações de Load Word. Por fim, as saídas **reg_out_a** e **reg_out_b,** de 8 bits cada, recebem os valores armazenados nos dois registradores de 2 bits e se encaminham a ULA. No total, temos 4 registradores no processador, sendo eles $0 (00), $1(01), $S2 (10) e $3 (11).

3.1.3 CLOCK

Este componente gera uma taxa constante usada para quando os eventos ocorrem no hardware. No processador MK 10 o clock é simulado pela Simulator Waveform Editor, onde denominamos um valor para executar as operações do processador.

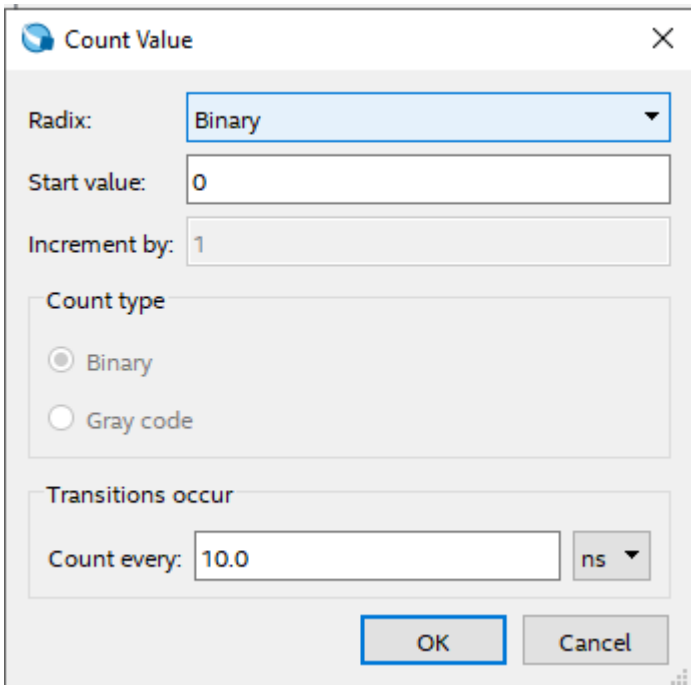


Figura 4 -- Clock Simulado no Intel Quartus

3.1.4 UNIDADE DE CONTROLE

Para a Unidade de Controle temos a utilização dos 4 bits usados no Opcode, onde a depender de cada valor passado, temos diferentes valores para as flags. São por meio delas que temos as instruções:

- **Jump –** Usado exclusivamente para operações de Jump
- **Branch –** Utilizado em estruturas de decisão como beq e bne
- **MemRead –** Flag usada para leitura de memória. Usada no Load Store
- **MentoReg –** Usado em operações envolvendo leitura de memória
- **AluOP –** Opcodes passados na instrução de memória e que definem que instrução está sendo executada
- **MemWrite –** Flag usada quando se trabalha com a memória Ram. Ativada, pode-se ser escrever o dado armazenado.
- **ALUsrc -** Decide se o valor a ser usado provém do extensor de sinal ou do banco de registradores.
- **RegWrite -** Utilizado para escrita de dados na memória. É nela que o banco de registradores escreve o dado provindo da ULA.

A tabela mostrando cada flag ativada para cada Opcode passado é demonstrada abaixo. A ativação correta de cada um será determinante para o correto funcionamento das operações. Dado o nosso número de instruções temos estes valores:

COMANDO	JUMP	BRANCH	MEM_READ	MEN_TO_REG	ALU_OP	MEM_WRITE	ALU_SRC	REG_WRITE
ADD	**0**	**0**	**0**	**0**	**0000**	**0**	**0**	**1**
ADDI	**0**	**0**	**0**	**0**	**0001**	**0**	**1**	**1**
SUB	**0**	**0**	**0**	**0**	**0010**	**0**	**0**	**1**
SUBI	**0**	**0**	**0**	**0**	**0011**	**0**	**1**	**1**
MUL	**0**	**0**	**0**	**0**	**0100**	**0**	**0**	**1**
LW	**0**	**0**	**1**	**1**	**0101**	**0**	**0**	**1**
SW	**0**	**0**	**0**	**0**	**0110**	**1**	**0**	**0**
MOVE	**0**	**0**	**0**	**0**	**0111**	**0**	**0**	**1**
LI	**0**	**0**	**0**	**0**	**1000**	**0**	**1**	**1**
BEQ	**0**	**1**	**0**	**0**	**1001**	**0**	**0**	**0**
BNE	**0**	**1**	**0**	**0**	**1010**	**0**	**0**	**0**
IF BEQ E BNE	**0**	**0**	**0**	**0**	**1011**	**0**	**0**	**0**
J	**1**	**0**	**0**	**0**	**1100**	**0**	**0**	**0**

Tabela 5 - Valores de Instruções passadas para a Unidade de Controle

3.1.5 MEMÓRIA ROM

A função deste componente funciona com o objetivo de armazenar instruções que venhamos a testar em nosso processador. Nossas instruções referents ao teste de factorial e de Fibonacci são descritas dentro dela, passando um array com os códigos necessários para a execução. No processador MK 10, temos um endereço de 8 bits denominada **address** e uma saída denominada **dout** também de 8 bits. Dentro dele, temos um array que pega de 0 a 255 posições de 8 bits, possibilitando assim a construção de sequências de instruções bem definidas.

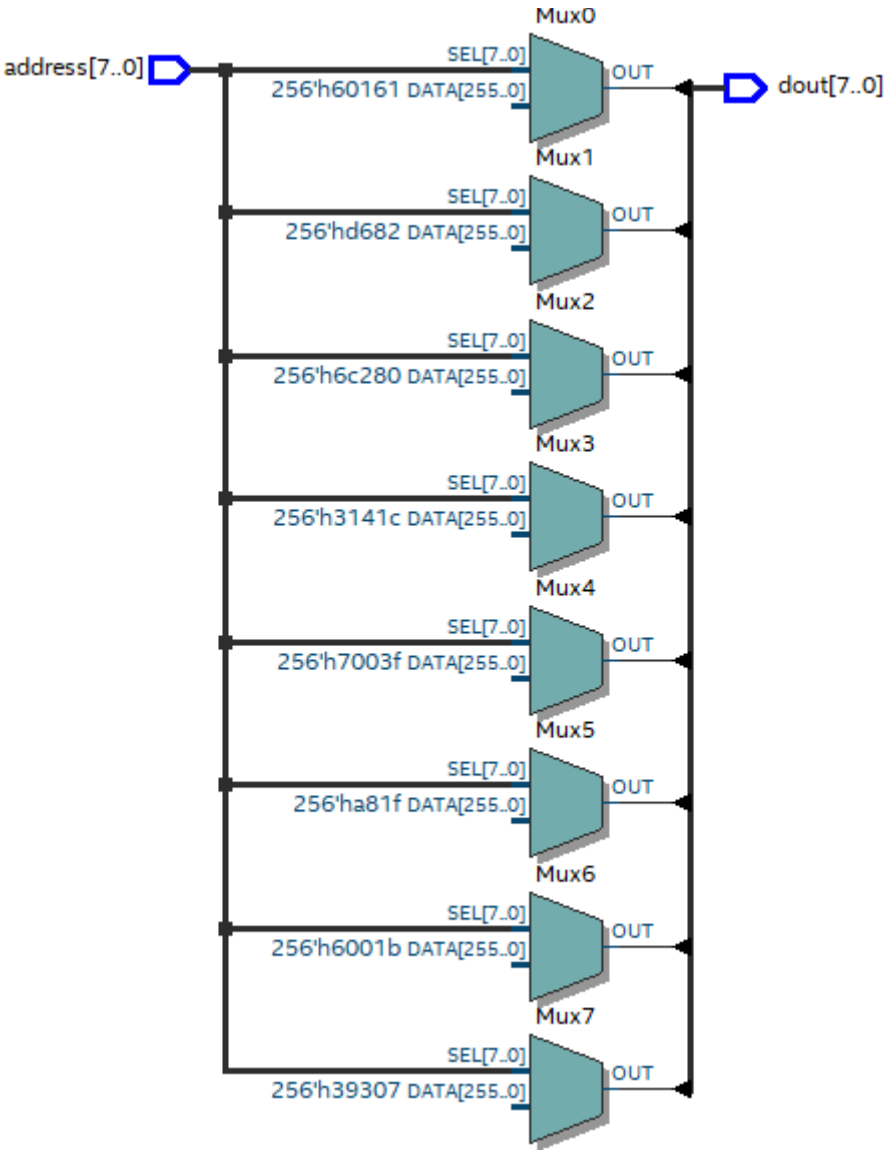


Figura 5 - RTL VIEWER da memória ROM

3.1.6 MEMÓRIA RAM

A memória RAM auxilia o processador na questão de armazenamento de dados nos registradores. Após os processamentos realizados na ULA, podemos aplicar os valores diretamente no banco de registradores ou passar para a memória RAM. Quando trabalhamos com as instruções LW (Load Word) e SW (Store Word), fazemos a manipulação de dados entre os registradores e a memória RAM. As flags **MEM_WRITE** quando ativadas indicam que a instrução está sendo escrita na memória RAM (é o que acontece com a instrução Store Word). Já quando usamos a flag **MEM_TO_REG** indicamos que desejamos usar tal dado no banco de registradores. Há ainda o uso da flag **MEM_READ** que também é usada. Nesse último caso, tanto a flag MEM_TO_REG quanto a MEM_READ são usadas na instrução de LW. Em nosso processador temos a seguinte construção do RTL VIEWER.

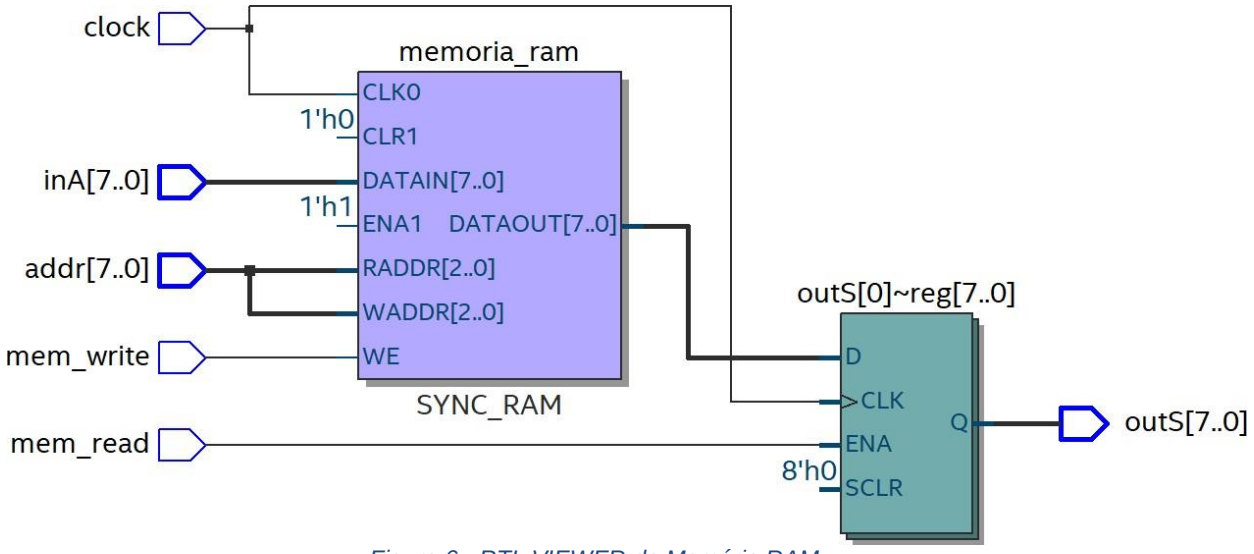


Figura 6 - RTL VIEWER da Memória RAM

Recebemos 5 variáveis de entrada, onde uma se refere ao Clock, utilizada para manter a consistência de relacionamento entre as execuções, além das duas flags mem_write e mem_read, que dependendo de estar igual a 1, realizará as instruções de escrita de dados ou a leitura. Se mem_read for igual a 1, a variável gera a saída de 8 bits que fica registrada em **outS.**

3.1.7 SOMADOR

Componente responsável por realizar a soma entre dois dados de entrada. Nele, recebemos duas entradas de valores correspondentes à 8 bits que serão somados bit a bit por meio do uso de variáveis auxiliares.

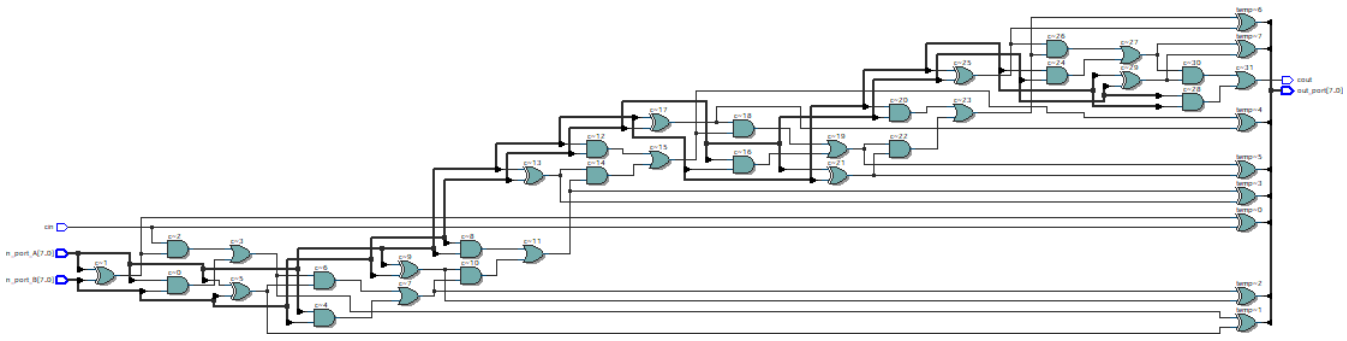


Figura 7- RTL VIEWER do somador

3.1.8 SUBTRADOR

Operações de subtração são necessárias para que outras como multiplicação venham a ser geradas. No MK 10, temos duas variáveis de entrada, denominadas **A** e **B** que são dados de entrada de 8 bits, que após serem subtraídas, geram a saída na variável **SUB** também de 8 bits.

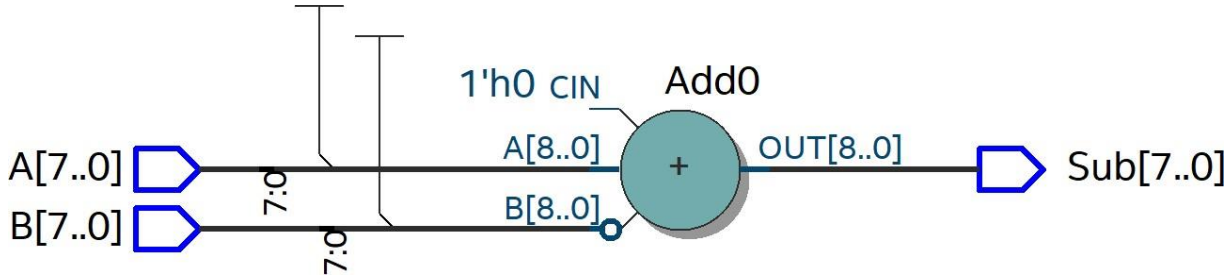


Figura 8 - RTL VIEWER do Subtrador

3.1.9 MULTIPLICADOR

Para realizar operações de Multiplicação em nosso processador, foi necessário a construção de um componente responsável exclusivamente para isso. Nosso multiplicador inicia com duas variáveis denominadas **in_port_A** e **in_port_B** que entram com os valores de 8 bits. Após uma série de operações de manipulação de bits, uma saída é gerada e gravada na variável **out_port** que possui o espaço de 16 bits, que é o tamanho necessário para guardar os resultados das operações de multiplicação. Apesar do tamanho do resultado final no RTL Viewer, o que realmente acontece é uma série de operações de comparação, soma e subtração de bits.

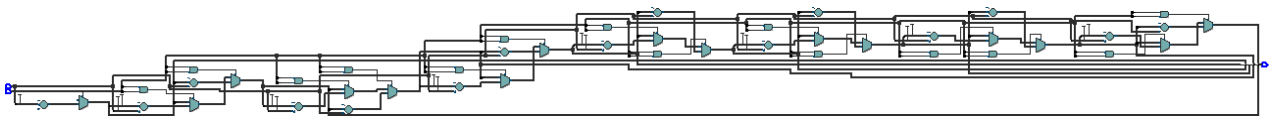


Figura 9 - RTL VIEWER do Multiplicador

3.1.10 AND

Por meio dele podemos realizar a operação lógica necessária para instruções voltadas a desvio condicional, por exemplo. Ele receberá duas entradas de bits (**in_port_A** e **in_port_B**) e verifica se as duas condições geral um valor verdadeiro que será posteriormente armazenado em **out_port**. Caso os dois valores seja, iguais a 1, retorna-se o valor 1. Nos demais casos, retorna-se 0.

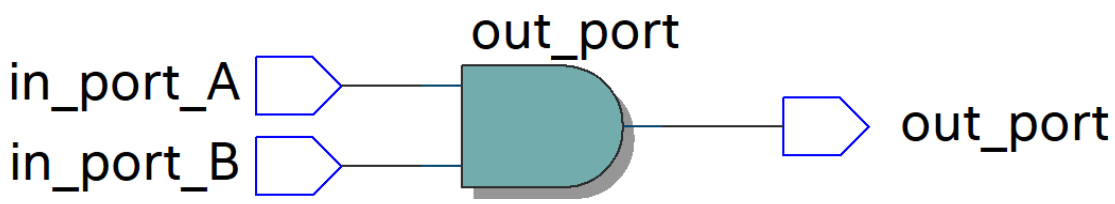


Figura 10 - RTL VIEWER da Porta AND

3.1.11 EXTENSOR DE SINAL 2 PARA 8 BITS

O uso do extensor de sinal se dá necessidade de realizar operações condicionais por exemplo. Seu uso é simples: uma variável denominada **in_port** entra com um valor de 2 bits e agrega-se a quantidade faltante. O após a concatenação de bits faltantes, o valor de saída fica gravado em **out_port**.

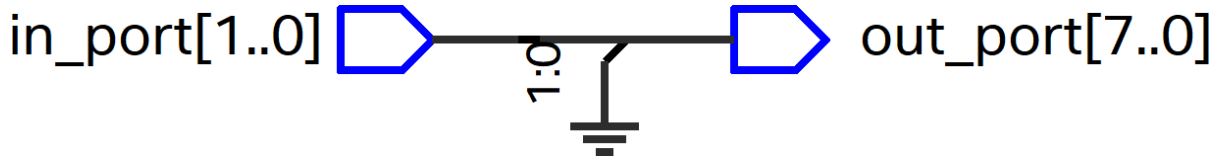


Figura 11 - RTL VIEWER do Extensor de Sinal de 2 para 8 bits

3.1.12 EXTENSOR DE SINAL 4 PARA 8 BITS

Seu funcionamento é semelhante ao extensor de 2 para 8 bits, sendo que a única diferença é que agora temos como valor de entrada um dado de 4 bits.

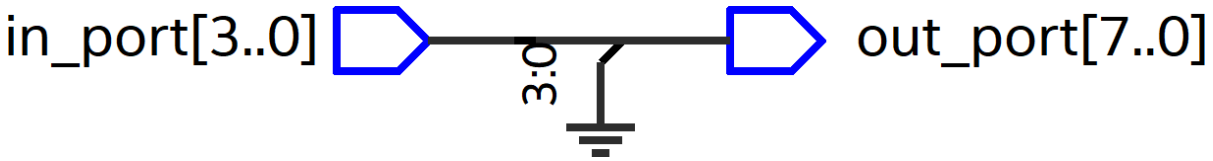


Figura 12 - RTL VIEWER do Extensor de 4 para 8 Bits

3.1.13 MUX_2X1

Tem como objetivo realizar a seleção de trilhas a serem manipuladas dentro do processador. Recebe-se em **in_A** e **in_B**, cujas entradas possuem 8 bits, respectivamente e aplica-se o **inPort** que servirá de seletor de trilhas. O resultado será atribuído a saída em **outPort**.

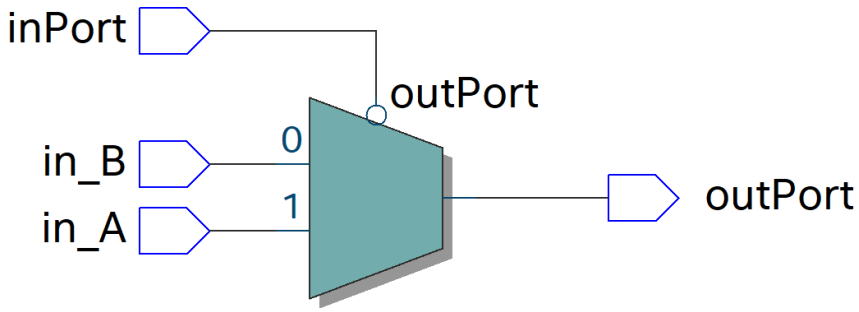


Figura 13 - RTL VIEWER do Multiplexador 2X1

3.1.14 PC

Também conhecido como **program counter** ou **contador de programa**, serve como registrador de endereço de instrução. É por meio dele que podemos apanhar o próximo par de instrução de memória. Ele recebe um endereço de 8 bits provindos de **inPort** além da entrada **clock**, que irá realizar a ativação do componente inPort gerando a saída **out_port** que dará continuidade a execução de código com estes bits.

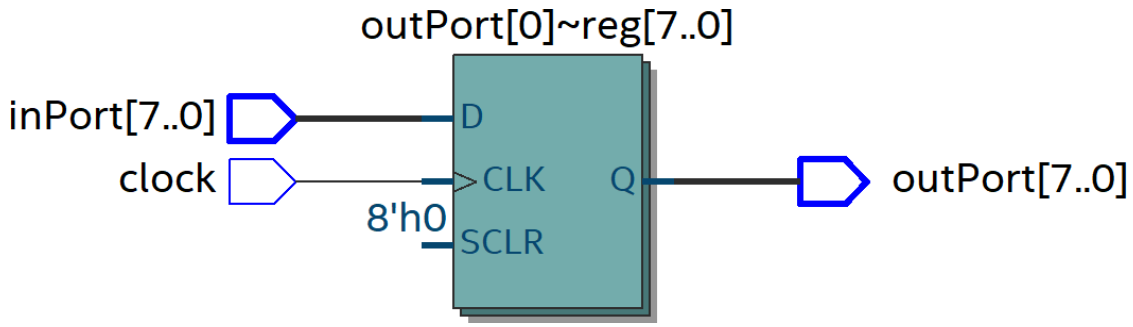
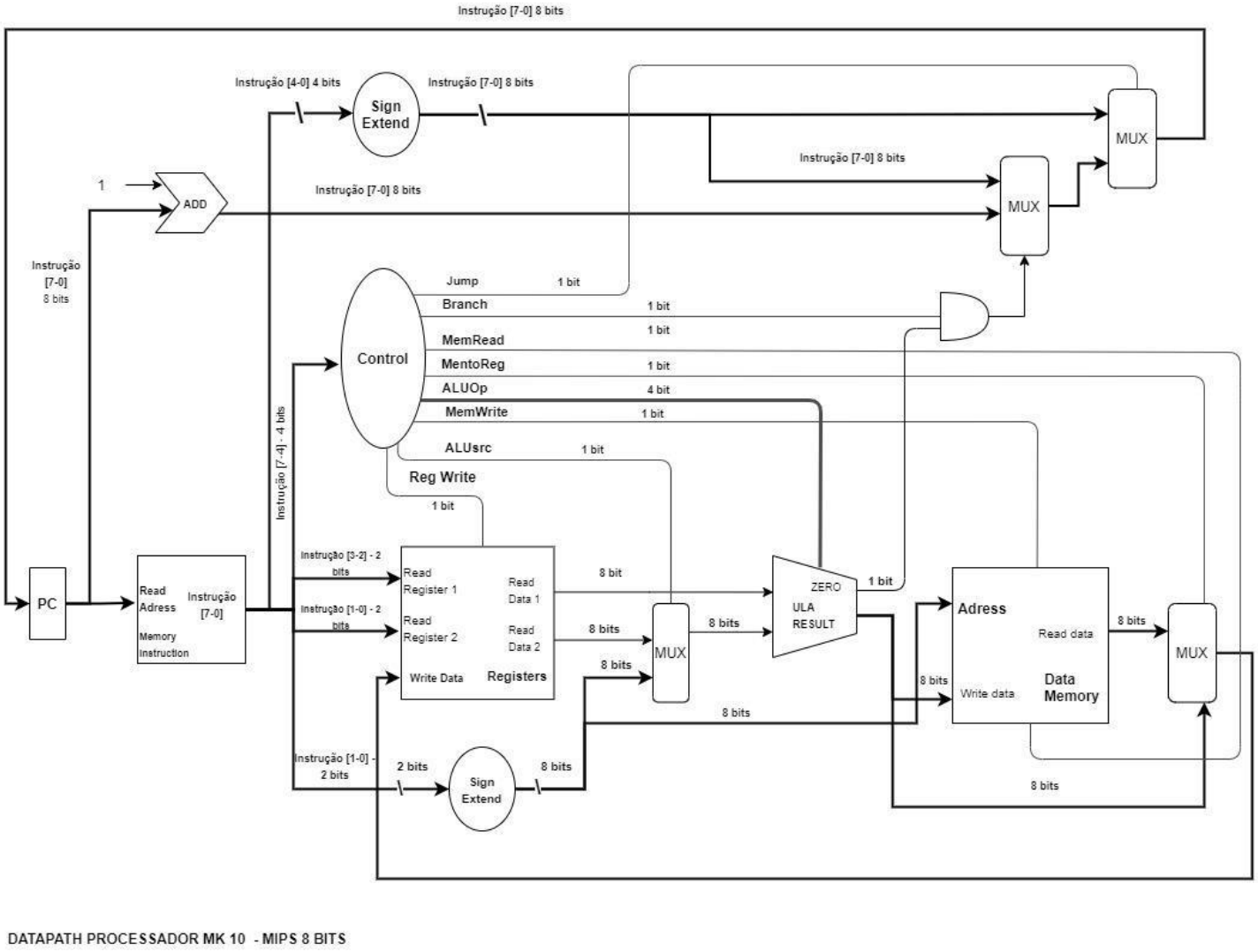


Figura 14 - RTL VIEWER do Program Counter (PC)

3.2 DATAPATH

É a conexão entre as unidades funcionais formando um único caminho de dados e acrescentando uma unidade de controle responsável pelo gerenciamento das ações que serão realizadas para diferentes classes de instruções. Abaixo, pode-se visualizar todo o caminho realizados. Temos duas imagens, uma do Datapath gerado para o projeto e outra do resultado que o RTL VIEWER gerou após a conexão entre os componentes do MK 10.



DATAPATH PROCESSADOR MK 10 - MIPS 8 BITS

Figura 15 - DataPath do Processador MK 10

O RTL VIEWER do processador que fora gerado:

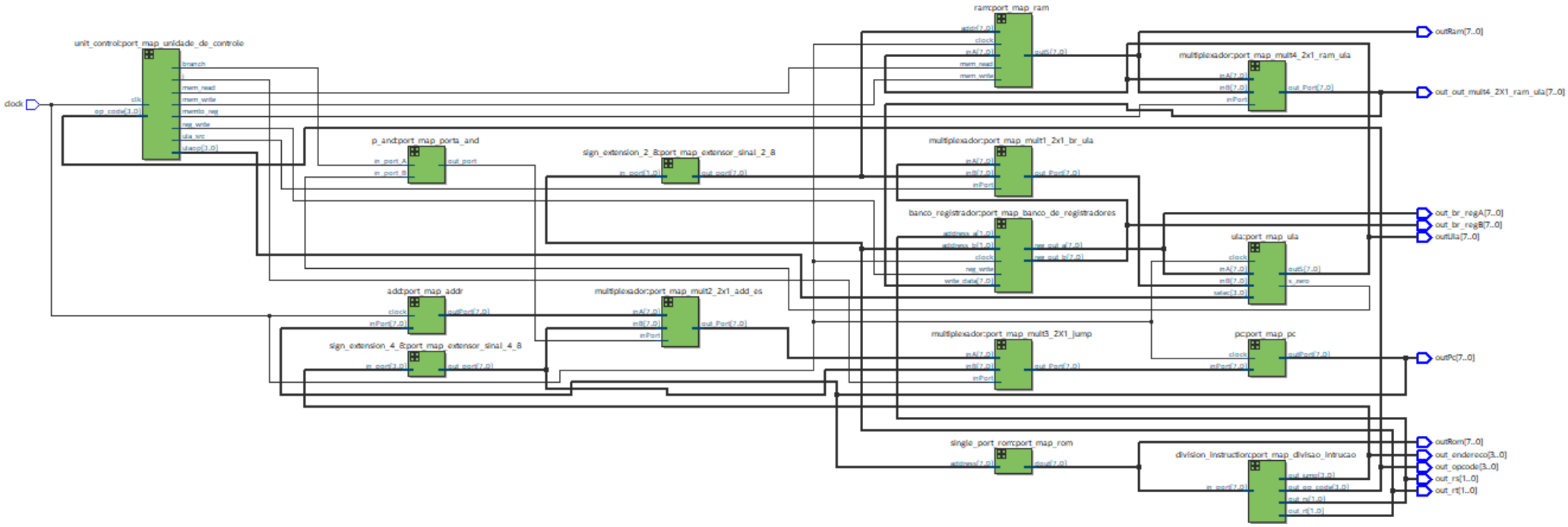


Figura 16 - RTL VIEWER do processador MK 10

4 SIMULAÇÕES E TESTES

Com o objetivo de verificar o funcionamento dos componentes que compõem o processador MK 10, geramos dois testes que utilizam todas as instruções implementadas, dessa forma podemos ver como o processador funciona na prática e ainda podemos verificar se os cálculos estão executando corretamente.

4.1 TESTE DE FIBONACCI

Comecemos então pelo teste de Fibonacci, onde iremos verificar se instruções relacionadas a soma, multiplicação e SW, LW, LI e ADDI estão funcionando corretamente. O código que passamos para a memória ROM do processador tem esse formato:

```
1   0 =>  "10001111", --li S3 3
2   1 =>  "01001111", --mul S3 S3
3   2 =>  "00011101", --addi S3 1
4   3 =>  "00011110", --addi S3 2
5   4 =>  "00011110", --addi S3 2
6   5 =>  "10001000", --li S2 0
7   6 =>  "10000000", --li S0 0
8   7 =>  "01100000", --sw S0 RAM(00)
9   8 =>  "10000001", --li S0 1
10  9 =>  "01100001", --sw S0 RAM(01)
11  10 => "01010000", --lw S0 RAM(00)
12  11 => "00000100", --add S1 S0
13  12 => "01010001", --lw S0 RAM(01)
14  13 => "00000100", --add S1 S0
15  14 => "01100000", --sw S0 RAM(00)
16  15 => "01100101", --sw S1 RAM(01)
17  16 => "00011001", --addi S2 1
18  17 => "10111011", --if S2 S3
19  18 => "10101010", -- bne 1010
```

O que em uma transcrição para Mips, seria isso:

ENDEREÇO	INSTRUÇÃO	BINÁRIO		
		OPCODE	REG 1	REG 2
			ENDEREÇO	
1	li S3 3	1000	11	11
2	mul S3 S3	0100	11	11
3	addi S3 1	0001	11	01
4	addi S3 2	0001	11	10
5	addi S3 2	0001	11	10

6	li S2 0	1000	10	00
7	li S0 0	1000	00	00
8	sw S0 RAM(00)	0110	00	00
9	li S0 1	1000	00	01
10	sw S0 RAM(01)	0110	00	01
11	lw S0 RAM(00)	0101	00	00
12	add S1 S0	0000	01	00
13	lw S0 RAM(01)	0101	00	01
14	add S1 S0	0000	01	00
15	sw S0 RAM(00)	0110	00	00
16	sw S1 RAM(01)	0110	01	01
17	addi S2 1	0001	10	01
18	if S2 S3	1011	10	11
19	bne 1010	1010	1010	

Tabela 6 - Código de Fibonacci usado no Processador MK 10

O funcionamento do código acima ocorre da seguinte maneira: Do endereço 0 ao endereço 9 ocorre a preparação para a execução do Fibonacci, onde primeiro e armazenado no registrador s3 o número 3, logo após ocorre a multiplicação do valor do registrador s3 com ele mesmo, logo após é somado mais 1ao valor do registrador s3, depois mais 2 e de pois mais 2, em seguida é salvo o valor 0 nos registrado S2 e S0, em seguida é carregado a valor do registrador S0 na posição 00 da memória RAM, depois é armazenado 1 no registrado S0, e em seguida o registrador S0 e carregado na posição 01 da memória R.

Após isso no endereço 10é carregado o valor de S0 na posição 00 da memória ram, em seguida ocorre a soma entre o registrador S1 e S0, no endereço 12 ocorre o carregamento do registrador S0 na posição 01 da memória ram, no endereço 13 ocorre novamente asoma de S1 e S0, no endereço 14 é carregado o valor do registrador S0 na posição00 da memória ram, logo após no endereço 15 é carregado o valor do registrado S1na posição da memória ram, depois no endereço 16 ocorre uma soma imediata com o valor do registrador S2 com o número 1 para que seja usado no loop onde no endereço 17 ocorre a comparação entre os registradores S2 e S3 e no endereço18 se encontra a instrução bne que compara se o valor do registrador S2 ou S3, secaso o valor for de s2 for menor que do s3 então ocorre um salto para o endereço 10, e caso os valores forem iguais ocorre o encerramento do programa.

O que temos então é a saída de valores que vão até a sequeência de número 14 de Fibonacci.

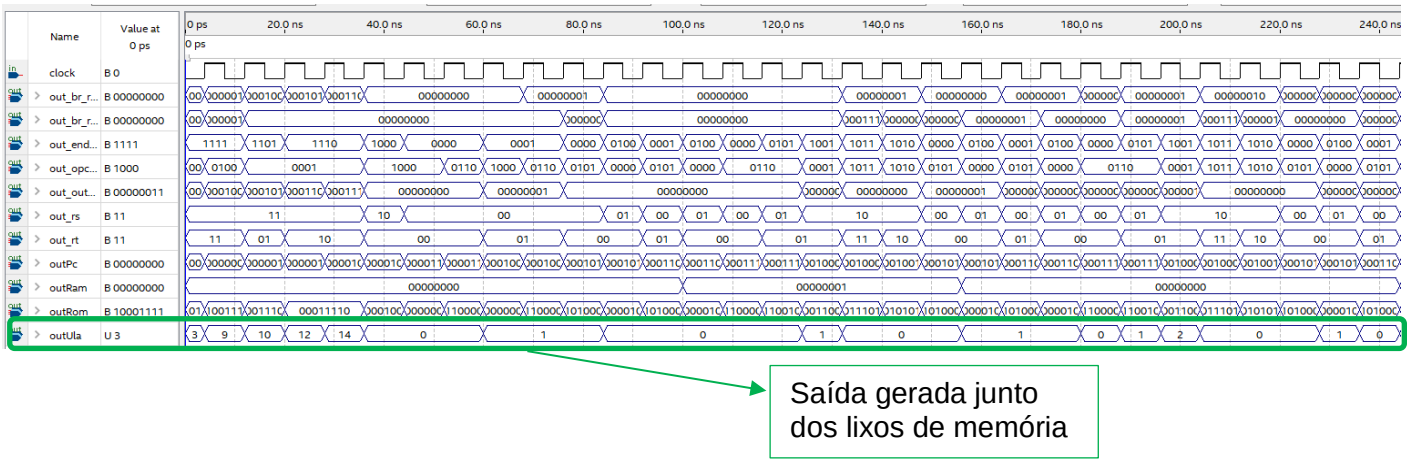


Figura 17 - Teste de Fibonacci parte 1

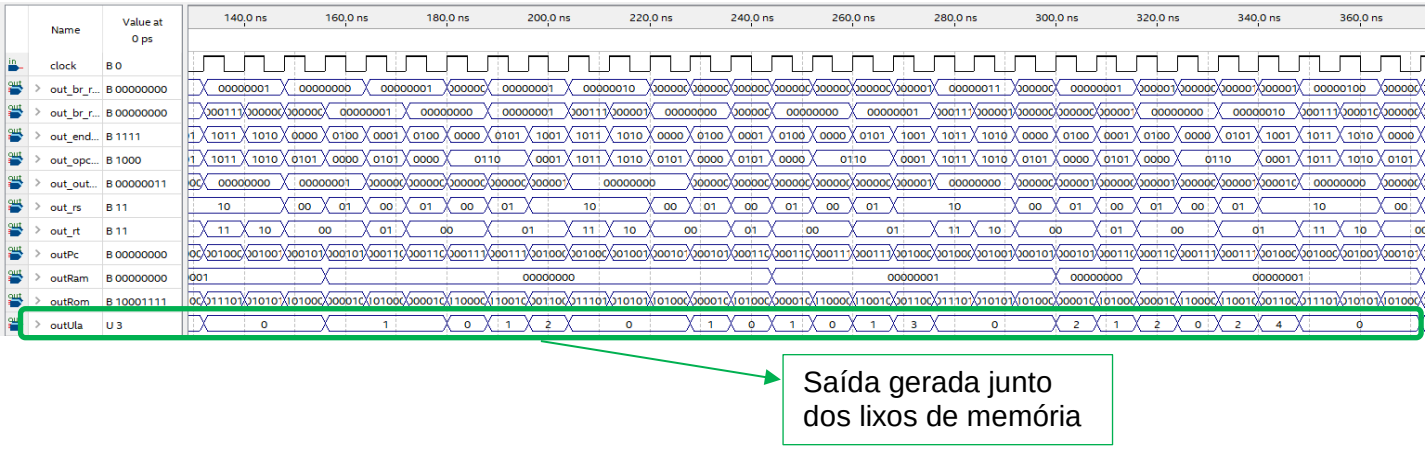


Figura 17 - Teste de Fibonacci parte 2

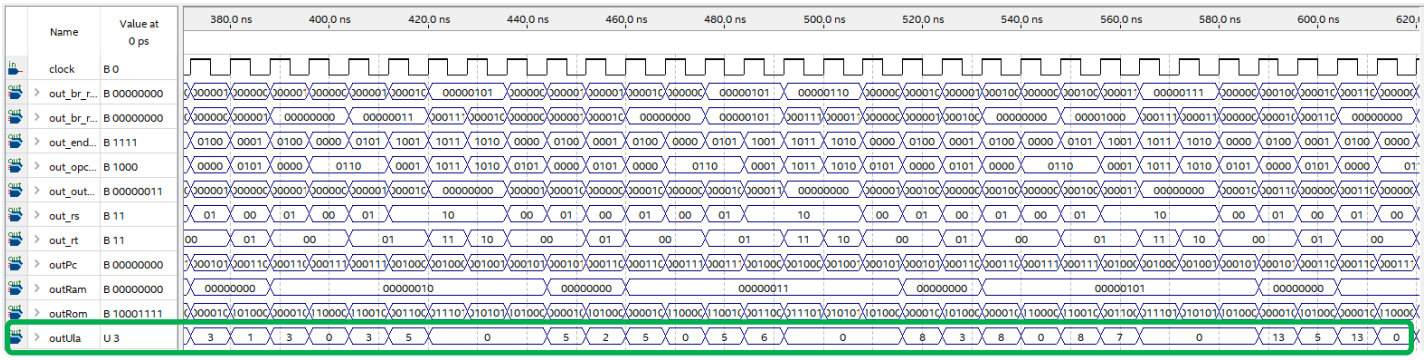


Figura 19 - Teste de Fibonacci parte 3

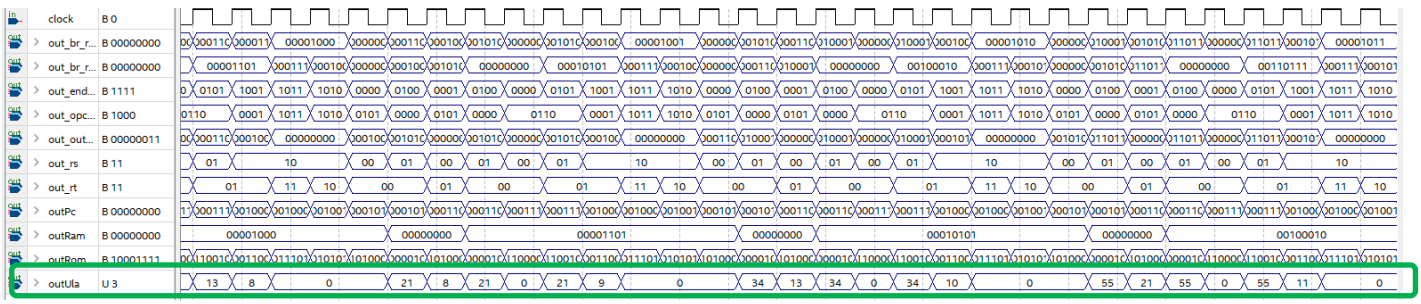


Figura 20 - Teste de Fibonacci parte 4

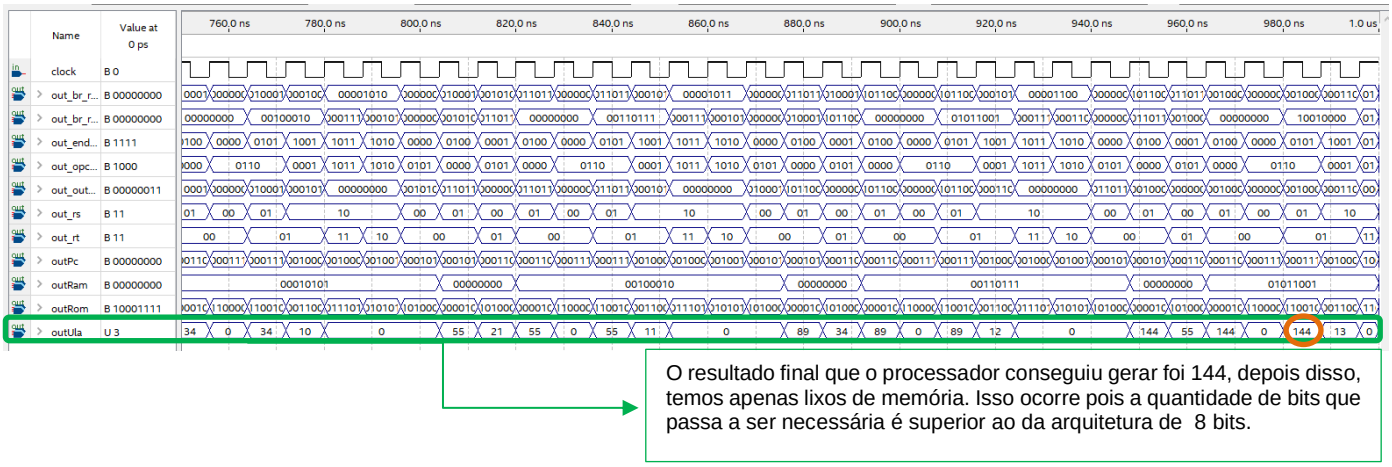


Figura 2118 - Teste de Fibonacci parte 5

4.2 TESTE DE FATORIAL

O teste de fatorial trabalhará com o uso de adições, multiplicações, Jump e condicionais. O código passado para a memória ROM tem este formato:

```
1   -- Fatorial que vai até 120 (depois estoura a memoria)
2       0 => "10001111"  -- li S3 3
3       1 => "00011111"  -- addi S3 3 == 6
4       2 => "10001001"  -- li S2 1
5       3 => "10000001"  -- li S0 1
6       4 => "01000010"  -- mul S0 S2
7       5 => "10111011"  -- if S2 == S3
8       6 => "00011001"  -- addi S2 1
9       7 => "10100100"  -- bne S2 != S3 jump 0100
10      8 => "10000000"  -- li S0 0
11      9 => "10000100"  -- li S1 0
12      10=> "10001000"  -- li S2 0
13    OTHERS=> "00000000"
```

Em Mips temos isso:

ENDEREÇO	INSTRUÇÃO	BINÁRIO		
		OPCODE	REG 1	REG 2
				ENDEREÇO
1	li S3 3	1000	11	11
2	Addi S3 3	0001	11	11
3	Li S2 1	1000	10	01
4	Li S0 1	1000	00	01

5	Mul S0 S2	0100	00	10
6	If S2 == S3	1011	10	11
7	Addi S2 1	0001	10	01
8	Bne S2 != S3 jump	1010	0100	
9	Li S0 0	1000	00	00
10	Li S1 0	1000	01	00
11	Li S2 0	1000	10	00

Tabela 7 - Código Teste de Fatorial

O código acima funciona da seguinte: recebemos no registrador S3 o valor 3que é armazenado. Logo depois, realizamos a soma do valor salvo no registrador S3 com mais 3. Em seguida, duas novas variáveis são criadas (em S0 e S2 que recebem 1) e que auxiliarão na passagem de valores futuros. Multiplicamos osvalores então e realizamos uma verificação entre S2 e S3, sendo verdadeira, iremosrealizar outras operações como no caso o de soma imediata do operador S2 com 1. Na linha 8 realizamos uma comparação entre os registradores S2 e S3, que casoseja verdadeira dará um jump para a instrução da linha 10. Nas linhas de 9 e 10 realizam a atribuição de 0 aos registradores S0 e S2, respectivamente. Tais comandos são executadas até o estouro da memória. O resultado pode ser vistologo abaixo. Como podemos conferir, temos então a geração de alguns lixos de memóriaque foram gerados na impressão dos valores. Isso ocorre porque necessitamos ir manipulando ao longo das execuções, inúmeras instruções que vão se modificandoao longo do programa.

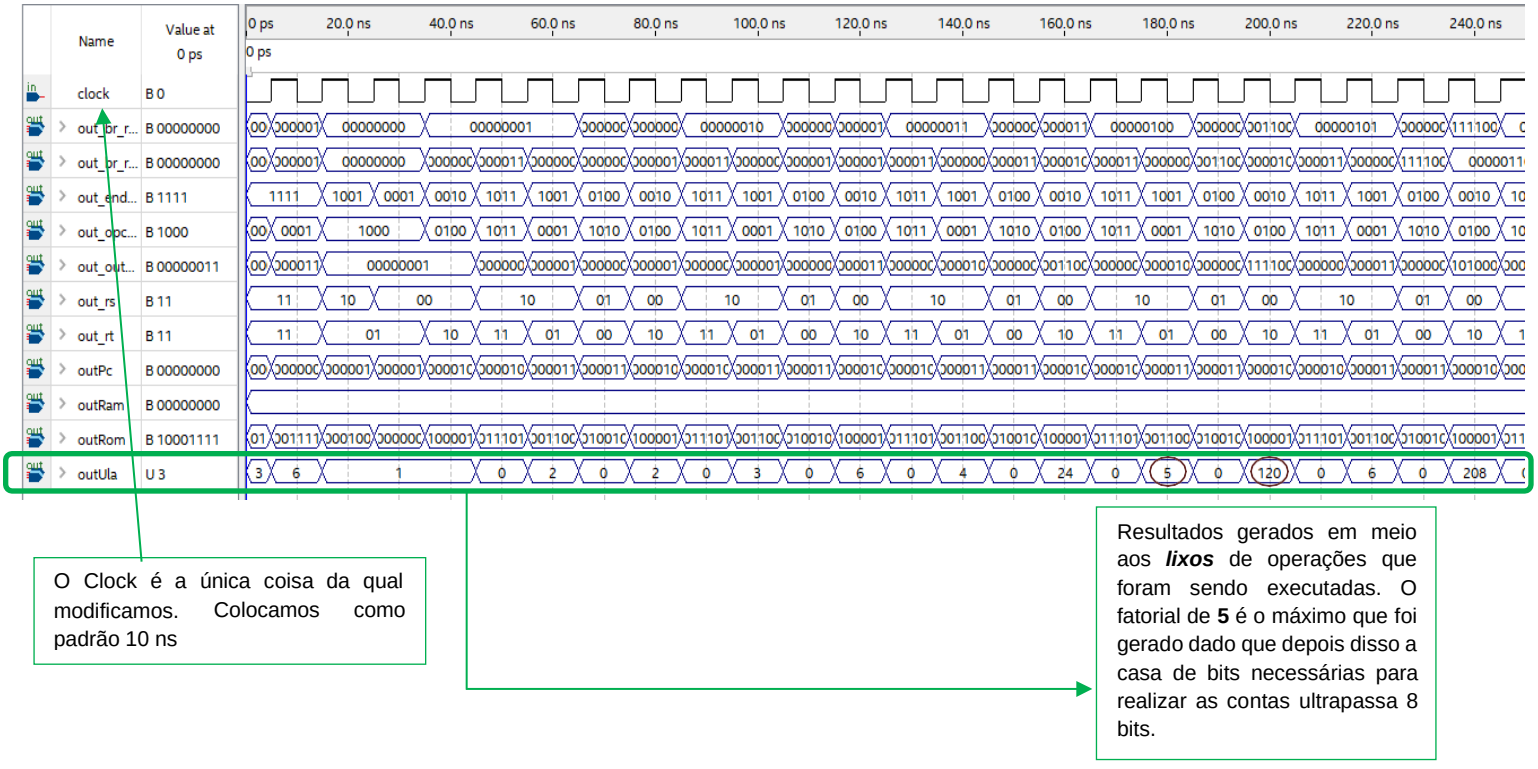


Figura 22 - Teste de Fatorial

5 CONSIDERAÇÕES FINAIS

Este trabalho apresentou a implementação do processador de 8 bits denominado de MK 10, um desafio complexo que foi feito com base em muito estudo e pesquisa. Cada componente foi realizado com base em profundas análises, pesquisas e horas de estudo. O processador apesar da sua poucacapacidade de processamento e de instruções, foi capaz de obter um desempenhorelativamente animador, trazendo ao campo de estudo e pesquisa. Com maiores implementações, em um projeto de 16 bits, por exemplo, teríamos muito mais possibilidades, como a implementação das operações de divisão, left shift dentre outros. Contudo, como um dia já fora no campo da computação, o uso de apenas 8bits é algo bastante surpreendente, principalmente se levarmos em conta toda a questão de evolução computacional gerada nos últimos 70 anos.

O processador MK 10 é um projeto oriundo de pesquisas de grandes conceitos que foram lapidados na ciência da computação pelos primeiros computadores, cuja capacidade de armazenamento não ultrapassava alguns Kilobytes e que hoje conseguem realizar até mesmo processamentos quânticos. Sua implementação buscou ser didática para aqueles que desejam compreender mais acerca do funcionamento de hardware e que ainda pode evoluir muito mais no futuro.