PODER EXECUTIVO

MINISTÉRIO DA EDUCAÇÃO

UNIVERSIDADE FEDERAL DE RORAIMA

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

RELATÓRIO DO PROJETO: PROCESSADOR EK

ALUNOS:

Eduardo Henrique Freire Machado – 2020001617

Kelvin Araújo Ferreira - 2019037653

Março de 2022

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 EK

Março de 2022

Boa Vista/Roraima

Resumo

Este trabalho aborda o projeto e implementação de um processador EK, utilizando do *software* Quartus Prime, da linguagem *Assembly* MIPS e da linguagem VHDL (*Very High-Speed Integration Circuit HDL*). O processador em questão é um RISC (*Reduced Instruction Set Computer*) de 8 bits.

O relatório então servirá de instrumento de avaliação dos alunos para a disciplina de AOC (Arquitetura e Organização de Computadores), ministrada pelo professor Herbert Oliveira Rocha.

Conteúdo

Lista de Figuras

Lista de Tabelas

1 Especificação

Nesta seção é apresentado o conjunto de itens para o desenvolvimento do processador EK, bem como a descrição detalhada de cada etapa da construção do processador.

1.1 Plataforma de desenvolvimento

Para a implementacação do processador EK foi utilizada a IDE *Quartus Prime*:

Figura 1 - Especificações no Quartus Prime

Flow Status	Successful - Wed Mar 09 08:56:50 2022
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	CPU_EK
Top-level Entity Name	CPU_EK
Family	Arria II GX
Logic utilization	< 1 %
Total registers	40
Total pins	70 / 176 (40 %)
Total virtual pins	0
Total block memory bits	32 / 2,939,904 (< 1 %)
DSP block 18-bit elements	0 / 232 (0 %)
Total GXB Receiver Channel PCS	0 / 4 (0 %)
Total GXB Receiver Channel PMA	0 / 4 (0 %)
Total GXB Transmitter Channel PCS	0 / 4 (0 %)
Total GXB Transmitter Channel PMA	0 / 4 (0 %)
Total PLLs	0 / 4 (0 %)
Total DLLs	0 / 2 (0 %)
Device	EP2AGX45CU17I3
Timing Models	Final

Fonte: Elaborada pelos autores.

1.2 Conjunto de instruções

O processador EK possui 2 registradores: $s0 e $s1. Assim como 11 formatos de instruções de 8 bits cada, instruções do tipo R, I e J. Seguem algumas considerações sobre as estruturas contidas nas instruções:

- Opcode: indica ao processador qual a instrução a ser executada;
- 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.

Tipos de Instruções:

- Tipo R: Este tipo de instrução trata de operações aritméticas.
  - Formato para escrita de código na linguagem MIPS:

Opcode	$s0	$s1

  - Formato para escrita em código binário:

Instrução do tipo R		
Opcode	Reg1	Reg2
4bits	2bits	2bits
7-4	3-2	1-0

- Tipo I: Este tipo de instrução aborda carregamentos diretos na memória.
  - Formato para escrita de código na linguagem MIPS:

Opcode	$s0

  - Formato para escrita em código binário:

Instrução do tipo I		
Opcode	Reg1	Imediato
4bits	2bits	2bits
7-4	3-2	1-0

- Tipo J: Este tipo de instrução é responsável por desvios condicionais e incondicionais.
  - Formato para escrita de código na linguagem MIPS:

Opcode	$s0

  - Formato para escrita em código binário:

Instrução do tipo J	
Opcode	Endereço
4bits	4bits
7-4	3-0

Visão geral das instruções do Processador EK:

O número de bits do campo Opcode das instruções é igual a quatro, sendo assim obtemos um total de 16 Opcodes (0-15) que são distribuídos entre as instruções, assim como é apresentado na Tabela 1.

Tabela 1 - Opcodes suportados pelo processador EK

Opcodes	Sintaxe	Formato	Significado	Exemplos
0000	add	R	Soma	**add** $s0, $s1
0001	addi	I	Soma Imediata	**addi** $s0, 3
0010	sub	R	Subtração	**sub** $s0, $s1
0011	subi	I	Subtração Imediata	**subi** $s0, 6
0100	lw	I	Load Word	**lw** $s0 memoria (00)
0101	sw	I	Store Word	**sw** $s0 memoria (00)
0110	li	I	Load Imediato	**li** $s0 2
0111	beq	J	Desvio Condicional	**beq** endereço
1000	if	J	If	**if** $s0 $s1
1001	jump	J	Jump	**Jump** memoria

Fonte: Elaborada pelos autores.

1.3 Descrição do Hardware

Nesta seção são descritos os componentes do hardware que compõem o processador EK, incluindo uma descrição de suas funcionalidades, valores de entrada e saída.

1.3.1 ALU

O componente ALU (Unidade Lógica Aritmética) tem como principal objetivo efetuar as principais operações aritméticas (considerando apenas resultados inteiros), dentre elas: soma subtração e multiplicação.

Adicionalmente a ALU efetua operações de comparação de valor como igual ou diferente.

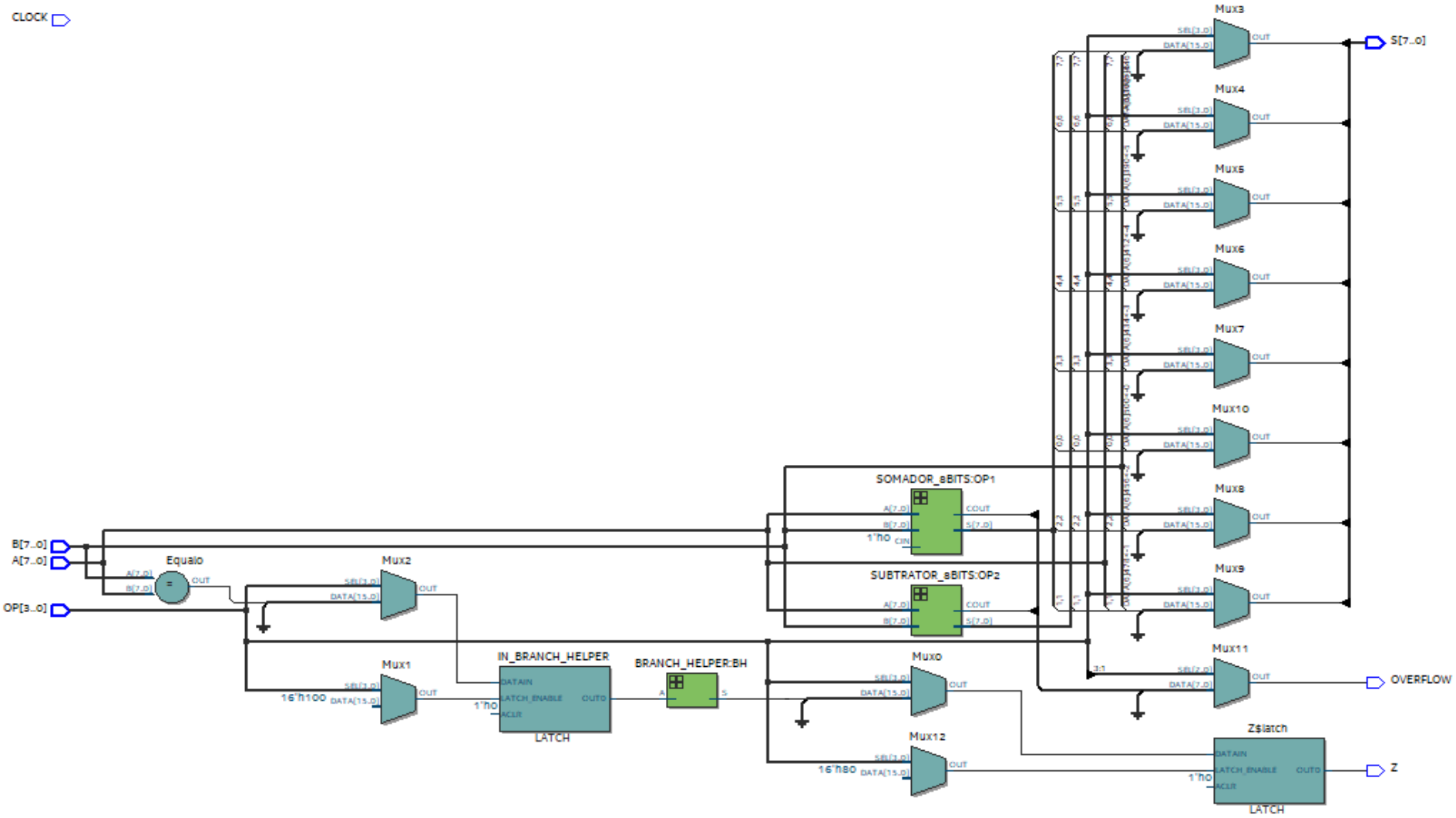
O componente ALU recebe como entrada três valores:

- **A:** dado de 8 bits para operação;
- **B:** dado de 8 bits para operação;
- **OP:** identificador da operação que será realizada de 4 bits.

O ALU também possui três saídas:

- **S:** identificador de resultado (1 bit) para comparações (1 se verdade e 0 caso contrário);
- **Overflow:** identificador de overflow caso a operação exceda os 8 bits;
- **Z:** saída com o resultado das operações aritméticas.

Figura 2 - Bloco simbólico do componente ALU gerado pelo Quartus Prime



Fonte: Elaborada pelos autores.

1.3.2 Bloco de Registradores

O componente banco_reg tem como principal objetivo armazenar e dizer aos registradores os valores que eles receberão além de instruir seu destino dentro do barramento.

O componente banco_reg recebe como entrada 4 valores:

- REG1_IN: valor imediato da instrução de 2 bits;
- REG2_IN: valor imediato da instrução de 2 bits;
- REG_WRITE: flag de controle para escrita no registrador;
- WRITE_DATA: dado de 8 bits a ser armazenado.

O banco_reg também possui 2 saídas de 8 bits cada:

- REG1_OUT: dado lido pela entrada;
- REG2_OUT: dado lido pela entrada;

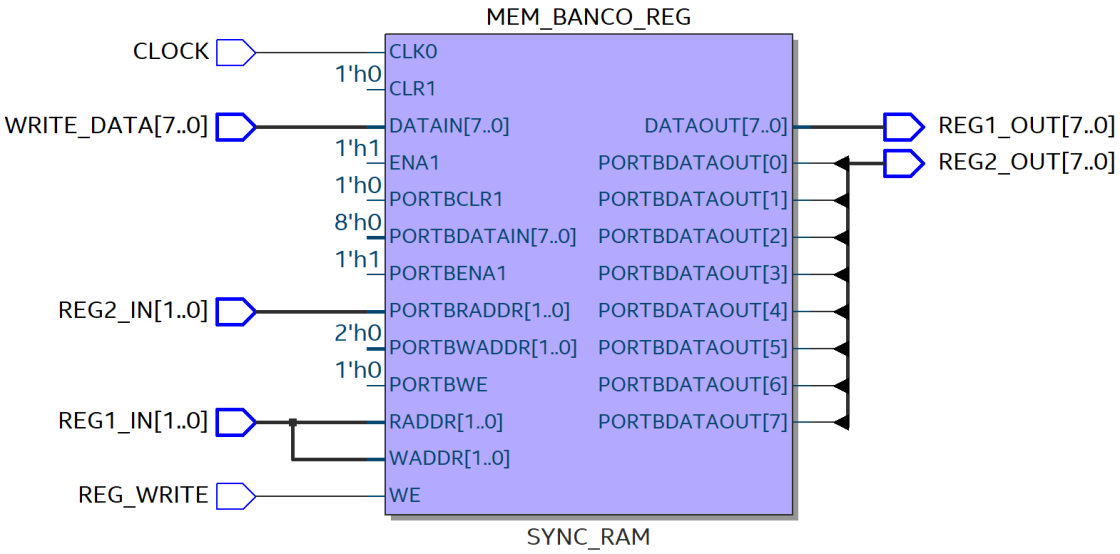


Figura 3 - Bloco simbólico do componente banco_reg gerado pelo Quartus

1.3.3 Controle

O componente unidade_de_controle é o responsável por dizer o que acontece e como se desenvolve dentro processador, portanto, tem como objetivo realizar o controle de todos os componentes do processador de acordo com o opcode, esse controle é feito através das flags de saída abaixo:

- **Jump**: Flag que é utilizada para operação de pulo de memória/desvio

condicional.

- **Branch**: É a flag responsável pela decisão de haver ou não um desvio.
- **memRead**: É a flag que decide se um valor será lido ou não na memória RAM.
- **ALUOp**: É a flag responsável por mandar o comando de qual operação a ALU irá executar.
- **memWrite**: É a flag responsável pela decisão de escrita na memória RAM.
- **AluSrc:** É o seletor que decide se o valor que a ALU irá receber vem do banco de registradores ou uma inserção imediata.
- **RegWrite**: É o que decide se haverá escrita ou não no banco de registradores.

Abaixo segue a tabela, onde é feita a associação entre os opcodes e as flags de controle:

Tabela 2 - Detalhes das flags de controle do processador

Opcode	Jump	Branch	MemRead	MemToreg	ALUOp
0000	0	0	0	0	0
0001	0	0	0	0	0
0010	0	0	0	0	0
0011	0	0	0	0	0
0100	0	0	0	0	0
0101	0	0	1	1	0
0110	0	0	0	0	0
0111	0	0	0	0	0
1000	0	1	0	0	0
1001	0	1	0	0	0
1010	1	0	0	0	0

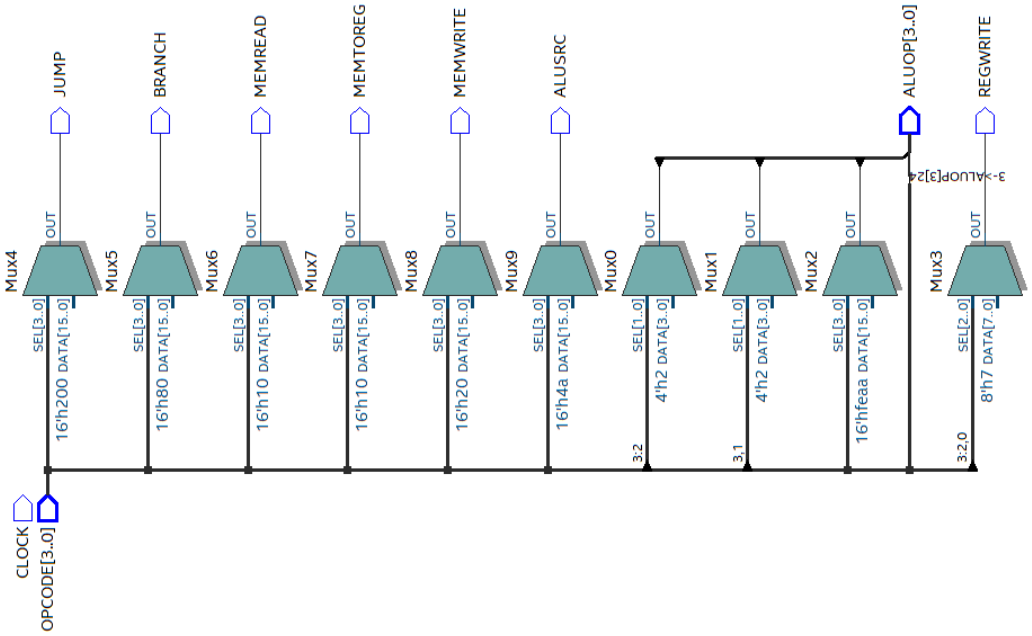
Fonte: Elaborada pelos autores.

Tabela 3 - Detalhes das flags de controle do processador

opcode	memWrite	AluSrc	RegWrite
0000	0	0	1
0001	0	1	1
0010	0	0	1
0011	0	1	1
0100	0	0	1
0101	0	0	1
0110	1	0	0
0111	0	1	1
1000	0	0	0
1001	0	0	0
1010	0	0	0

Fonte: Elaborada pelos autores.

Figura 4 - Bloco simbólico do componente unidade_de_controle gerado pelo Quartus



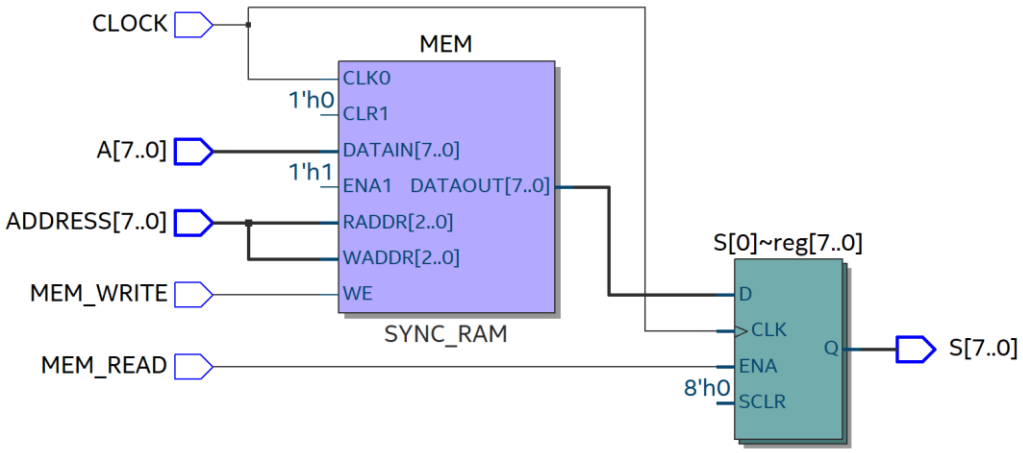
Fonte: Elaborada pelos autores.

1.3.4 Memória de Dados

O componente RAM, também conhecido como memória de dados é o responsável por armazenar bits de memória, portanto temos uma matriz de 8x8, ou seja, 8 posições com a capacidade de armazenar 8bits dinamicamente durante a execução do barramento, possui 5 entradas:

- Clock: sinal podendo ser de nivel logico alto ou baixo.
- A: valor de 8 bits representando a entrada
- MEM_WRITE: quando ativa, seta a memória RAM para modo de escrita (8bits)
- MEM_READ: quando ativa, seta a memória RAM para leitura, ou seja, a saída terá o valor endereçado pela entrada A (1bit)
- ADDRESS: endereço de trabalho (1bit), e uma única saída;
- S: Valor de 8 bits representando a saida.

Figura 5 - Bloco simbólico do componente RAM gerado pelo Quartus



Fonte: Elabodara pelos autores.

1.3.5 DIV_INSTRUÇÃO

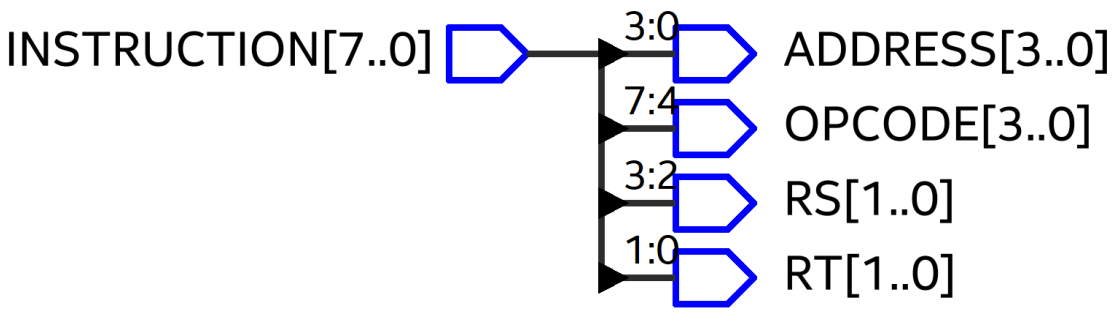
O componente memoria de instrução também conhecido como memória ROM é o responsável por dividir a instrução que sai dele em 4 trilhas, portanto possui uma porta de entrada de 8bits:

- INSTRUCTION: porta que recebe a instrução que vem da ROM;

4 PORTAS DE SAÍDA:

- ADDRESS: o endereço de memória que será usado na instrução do tipo jump e possui 4 bits, caso seja usado, utiliza os bits dos 2 registradores que não serão usados;
- OPCODE: o endereço de memória que será usado para o opcode do comando, possui 4 bits;
- RS: o endereço do primeiro registrador possui 2bits;
- RT: o endereço do segundo registrador possui 2bits;

Figura 6 - Bloco simbólico do componente memoria_de_instrução gerado pelo Quartus:



Fonte: Elaborada pelos autores

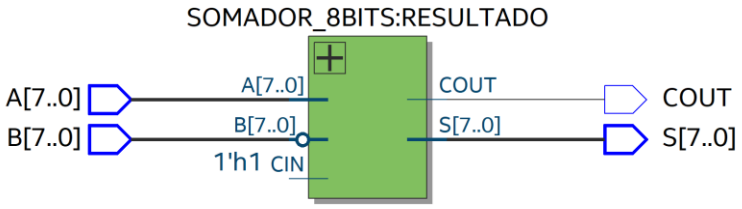
1.3.6 Somador

O componente somador 8 bits tem como principal objetivo efetuar operações de soma. É composto por 2 portas de entrada e uma porta de saída.

As portas de entrada são:

- **A:** recebe a informação de valor do primeiro registrador e possui 8 bits de tamanho;
- **B:** recebe a informação de valor do segundo registrador e possui 8 bits de tamanho, e porta de saída é:
- **S:** recebe o valor da operação de soma efetuada e possui também 8 bits de tamanho

Figura 7 - Bloco simbólico do componente somador8bits gerado pelo Quartus



Fonte: Elaborada pelos autores.

1.3.7 And

O componente AND é o responsável por realizar a operação lógica “E” e tem como saída o resultado da operação de 2 valores, sejam 0 ou 1, possui duas entradas:

- in_port_a: Vai introduzir o primeiro valor para operação;
- in_port_a: Vai introduzir o segundo um valor para operação, e tem uma única porta de saída:
- out_port: porta que vai portar o valor da operação E entre in_port_a e in_port_b;

1.3.8 Mux_2x1

O componente tem a função de um seletor de sinal, vai selecionar um sinal de entrada e vai deixar sair o sinal selecionado, seja ele 0 ou 1, possui 2 entradas de 8bits cada:

- A: porta que vai receber o sinal;
- B: porta que vai receber o sinal, e possui uma única saída de 8 bits:
- RESULT: porta que vai conter o sinal selecionado dentro do multiplexador;

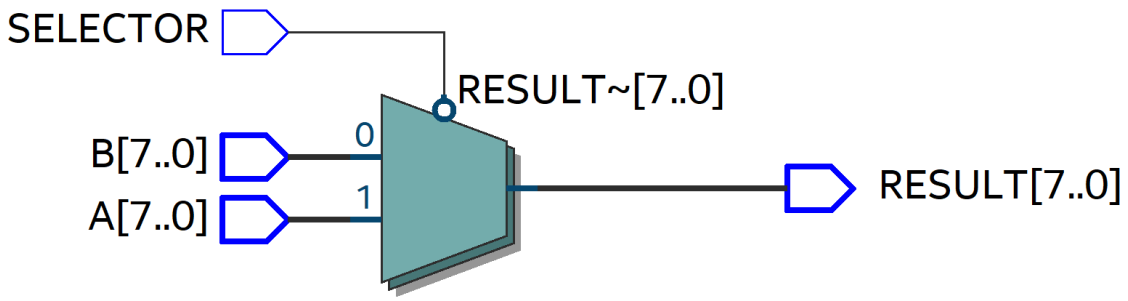


Figura 8 - Bloco simbólico do componente mult_2x1 gerado pelo Quartus

1.3.9 PC

O componente cprogram_counter funciona como o GPS dentro do processador, ele é que dirá qual é o endereço da próxima instrução a ser executada, por isso este componente só possui uma entrada de 8bits:

- ADDRESS_IN: recebe a informação que será destinada

E uma saída de 8bits também:

- ADDRESS_OUT: é o caminho que a informação já endereçada toma;

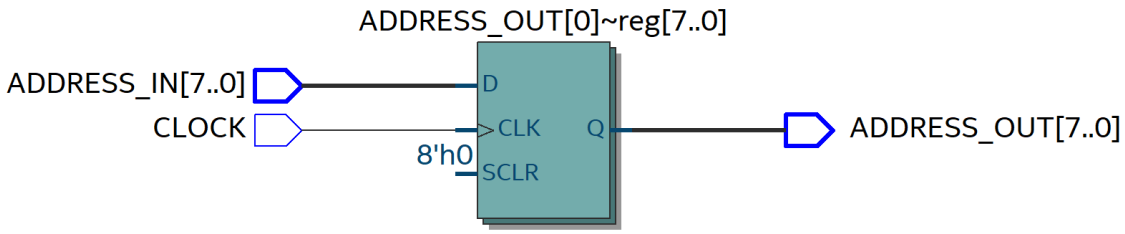


Figura 9 - Bloco simbólico do componente pc gerado pelo Quartus

1.3.10 Extensor

O componente extensor2_8 funciona como um extensor de sinal, transformando uma entrada de 2 bits em um sinal de 8 bits, possui, portanto, uma única entrada:

- A: recebe o sinal de 2 bits que será amplificado, e uma única saída:
- S: saída do sinal amplificado de 8 bits;

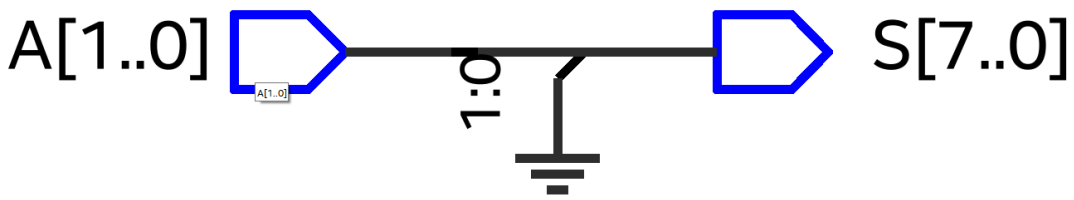


Figura 10 - Bloco simbólico do componente Extensor gerado pelo Quartus

1.4 Datapath

É a conexão entre as unidades funcionais formando um único caminho de dados e adicionando uma unidade de controle responsável pelo gerenciamento das ações que serão realizadas para diferentes classes de instruções.

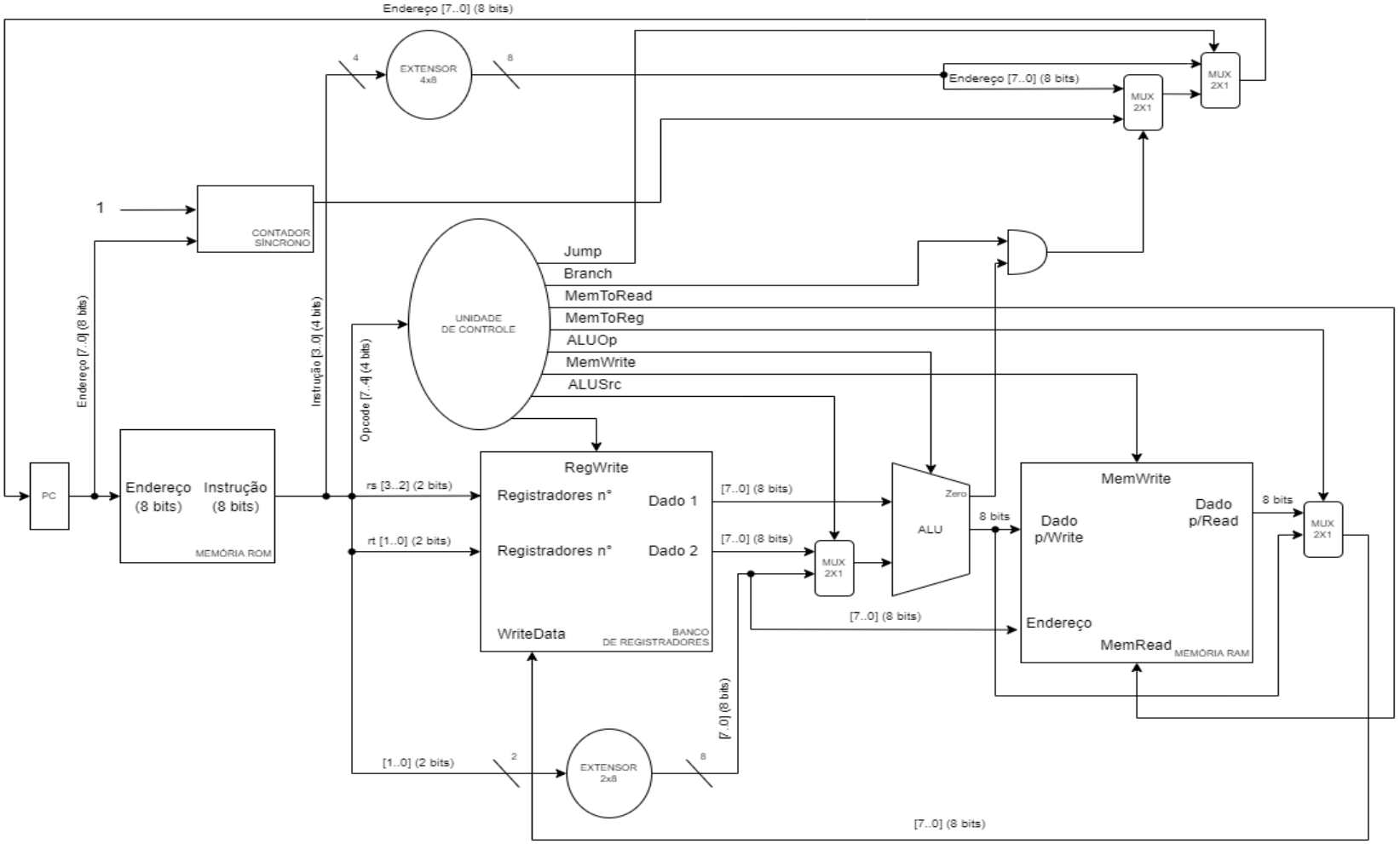


Figura 11 - Datapath gerado pelo Quartus do processador EK

2 Simulações e Testes

Objetivando analisar, verificando o funcionamento do processador, efetuamos alguns testes analisando cada componente do processador em específico, em seguida efetuamos testes de cada instrução que o processador implementa. Com a finalidade de demonstrar o funcionamento do processador EK. Utilizando como exemplo o código para calcular o número da sequência de Fatorial e Fibonacci.

2.1 Fibonacci

Na implementação do Fibonacci, observamos que a nossa arquitetura não suporta a operação move, portanto temos que salvar em memória uma constante zero que nos servirá somente para limpeza de registrador ao usarmos uma operação lw (load word), iniciamos dois registradores $s0 e $s1 com valores 0 e 1 respectivamente, prosseguindo para a próxima instrução iremos fazer uso do lw do endereço 00 no registrador $s2, ou seja, reescrevemos o registrador $s2 com o valor da constante 0 para então guardamos o valor de $s1, agora com todos esses estados executados podemos dar continuidade a operação de soma entre $s1 e $s0 que de fato irá dar origem ao Fibonacci, após o valor resultante da soma ser devolvida ao registrador $s1, temos que zerar o valor do registrador $s0 para então reatribuirmos o antigo valor de $s1 que está guardado no registrador $s2 e finalizamos com uma instrução jump para o endereço 0011.

Tabela 4 - Código Fibonacci para o processador EK

Código detalhado em MIPS para a realização do Fibonacci				
Endereço	Sintaxe	Instrução		
		Opcode	Rt	Rs
0	sw $s0 00	0110	00	00
1	li $s0 1	0111	00	01
2	li $s1 1	0111	01	01
3	lw $s2 00	0101	10	00
4	add $s2 $s1	0000	10	01
5	add $s1 $s0	0000	01	00

6	lw $s0 00	0101	00	00
7	add $s0 $s2	0000	00	10
8	j 0011	1010	0011	

Fonte: Elaborada pelos autores.

2.2 Testes Fibonacci

```
-- TESTE FIBONACCI
        0 => "00010000", -- ADDI S0 0
        1 => "01010000", -- SW S0
        2 => "00010001", -- ADDI S0 1
        3 => "00010101", -- ADDI S1 1
        4 => "01001100", -- LW S3 00
        5 => "00001101", -- ADD S3 S1
        6 => "00000100", -- ADD S2 S1
        7 => "01000000", -- LW S0 00
        8 => "00000011", -- ADD S0 S3
        9 => "10010100", -- J 0100
```

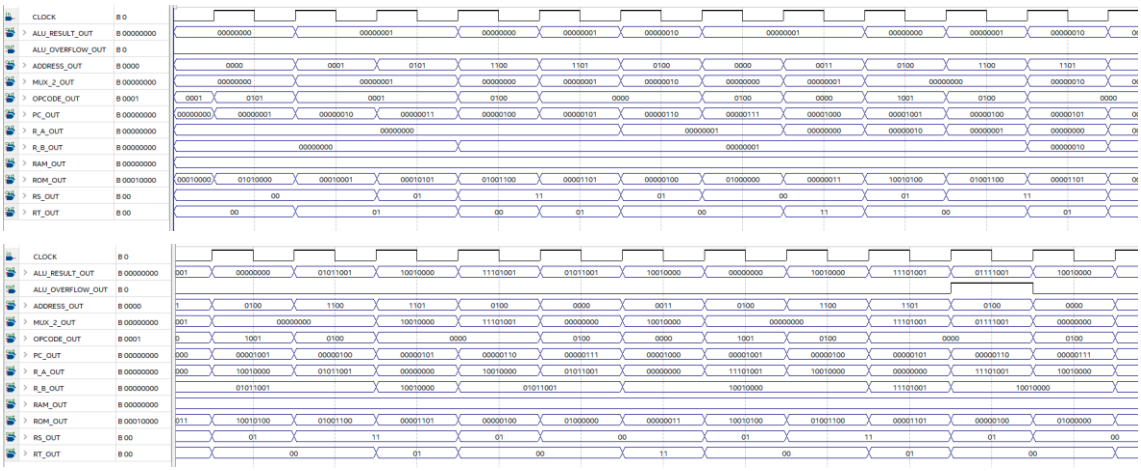


Figura 12 - Resultado na waveform. (Teste Fibonacci)

2.3 Testes de ADDI, SUB e SUBI

```
-- TESTE DE ADDI, SUB E SUBI
        0 => "00010011", -- ADDI S0 3
        1 => "00010101", -- ADDI S1 1
        2 => "00110001", -- SUBI S0 1
        3 => "00100001", -- SUB S0 S1
```

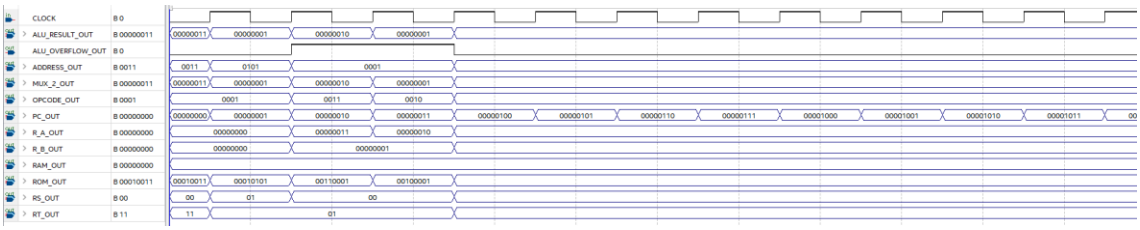


Figura 13 - Resultado na waveform. (Teste de ADDI, SUB E SUBI)

2.4 Testes de ADD e ADDI

```
-- TESTE DE ADD E ADDI
    0 => "00010011", -- ADDI S0 3
    1 => "00010111", -- ADDI S1 3
    2 => "00000001", -- ADD S0 S1
```

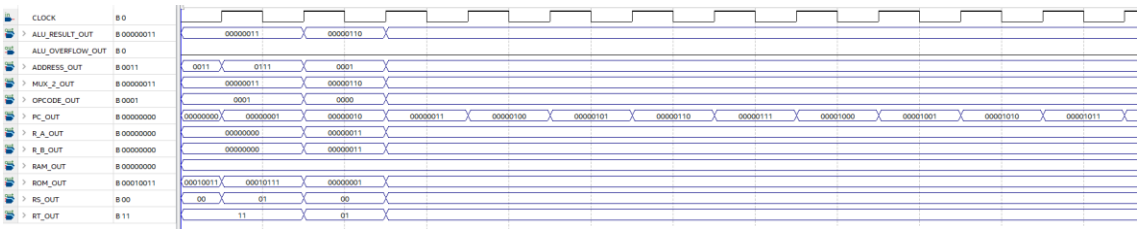


Figura 14 - Resultado na waveform. (Teste de ADD e ADDI)

2.5 Testes de BEQ

```
-- TESTE BEQ
    0 => "00010010", -- LI S0 2
    1 => "00010110", -- LI S1 2
    2 => "10000001", -- IF S0 == S1
    3 => "01110101", -- BEQ S0 == S1 JUMP 0101
    4 => "00010001", -- ADDI S0 1
    5 => "00010010", -- ADDI S0 2
```

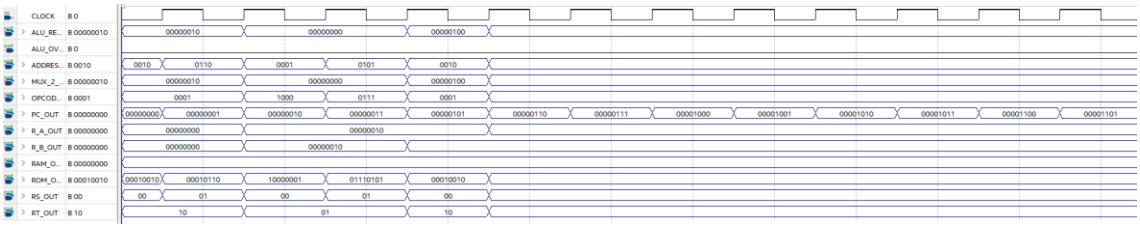


Figura 15 - Resultado na waveform. (Teste de BEQ)

2.6 Testes de LI

```
-- TESTE LI
    0 => "01100010", -- LI S0 2
    1 => "00010001", -- ADDI S0 1
```

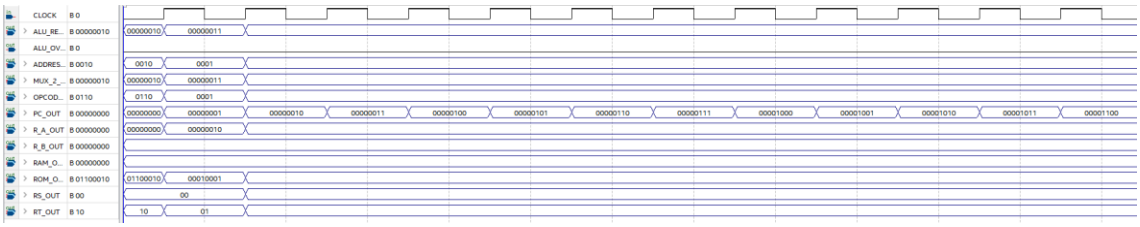


Figura 16 - Resultado na waveform. (Tesde de LI)

3 Considerações finais

Este trabalho apresentou o projeto e implementação do processador de 8 bits denominado de EK, que foi uma rica oportunidade para pôr em prática o que nos foi ensinado na disciplina de AOC, e esclarecer diversos pontos que antes estavam difíceis de se entender. Uma das maiores dificuldades encontradas foi justamente a de ter que lidar com um baixo número de bits.

No entanto, houve implementações que ficaram inacabadas devido às decisões tomadas durante o desenvolvimento. Exemplos dessas é a operação de *move*, que é simulada através de um sw e um lw, e um limite de salto de 15 endereços para instruções do tipo J.

4 Repositório

https://github.com/ed-henrique/AOC_Eduardo_Kelvin_UFRR_2022