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

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

RELATÓRIO DO PROJETO: PROCESSADOR JV

ALUNOS:
JASMIM SABINI - 2023010360
VINÍCIUS MARTINS - 2023001156

Março de 2024
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 JV

Dezembro de 2024
Boa Vista - Roraima

Resumo

Este trabalho aborda o projeto e implementação de um processador de 8 bits baseado na arquitetura mips monociclo. O projeto foi feito para a apresentação da disciplina de Arquitetura e Organização de Computadores visando obtenção de nota parcial do semestre. No processador haverá um total de 8 instruções e 4 registradores.

Conteúdo

1 Especificação

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

1.1 Plataforma de desenvolvimento

Para a implementação do processador JV foi utilizado a IDE: Quartus Prime

Flow Status	Successful - Mon Aug 27 17:33:50 2012
Quartus II Version	9.0 Build 184 04/29/2009 SP 1 SJ Web Edition
Revision Name	Quantum
Top-level Entity Name	Quantum_Pro
Family	Cyclone III
Met timing requirements	N/A
Total logic elements	3,123 / 15,408 (20 %)
Total combinational functions	2,148 / 15,408 (14 %)
Dedicated logic registers	2,137 / 15,408 (14 %)
Total registers	2137
Total pins	188 / 347 (54 %)
Total virtual pins	0
Total memory bits	0 / 516,096 (0 %)
Embedded Multiplier 9-bit elements	1 / 112 (< 1 %)
Total PLLs	0 / 4 (0 %)
Device	EP3C16F484C6
Timing Models	Final

Figura 1 - Especificações no Quartus

1.2 Conjunto de instruções

O processador JV possui 4 registradores: R0, R1, R2 e R3. Assim como 8 tipos de instruções, Instruções do R, I e J que seguem algumas considerações sobre as estruturas contidas nas instruções:

- **Opcode**: a operação básica a ser executada pelo processador, tradicionalmente chamado de código de operação;
- **RegDst**: o registrador contendo o primeiro operando fonte e adicionalmente para alguns tipos de instruções (ex. instruções do tipo R) é o registrador de destino;
- **RegSrc-Offset**: o registrador contendo o segundo operando fonte;

Tipo de Instruções:

- Formato do tipo R: Este formato aborda instruções de Load (exceto *load Immediately*), Store e instruções baseadas em operações aritméticas.

Formato para escrita de código na linguagem Quantum:

Tipo da Instrução	RegDest	Reg Src-Offset

Formato para escrita em código binário:

4 bits	2 bits	2 bits
7-4	3-2	1-0
Opcode	Reg2	Reg1

Visão geral das instruções do Processador JV:

O número de bits do campo **Opcode** das instruções é igual a quatro, sendo assim obtemos um total ($Bit 0 e 1^{NumeroTodaldeBitsdoOpcode} \therefore 2^4 = 16$) de 16 **Opcodes possíveis (0-15)** que são distribuídos entre as instruções, assim como é apresentado na Tabela 1.

Tabela 1 – Tabela que mostra a lista de Opcodes utilizadas pelo processador JV.

Opcode	Nome	Formato	Breve Descrição	Exemplo
0101	ADD	R	Soma	**add** $R0, $R1 ,ou seja, $R0 := $R0+$R1
0110	SUB	R	Subtração	**sub** $R0, $R1 ,ou seja, $R0 := $R0 - $R1
0010	ADDI	R	Soma Imediata	**addi** $R0, immediate, ou seja, $R0 := $R0 + immediate

0011	SUBI	R	Subtração Imediata	**subi** $R0, immediate, ou seja, $R0 := $R0 - immediate
0001	LW	I	Load Word	**lw** $R0 memória(00), ou seja: $R0 := valor ($R0) + memória(00)
1000	SW	I	Store Word	**sw** $R0 memória(00), ou seja: memória(00) := $R0
0111	BEQ	I	Desvio Condicional	**beq** endereço, ou seja: if($R0 == $R1)
0100	JUMP	J	Desvio Incondiciona	**jump** endereço(0000)

1.3 Descrição do Hardware

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

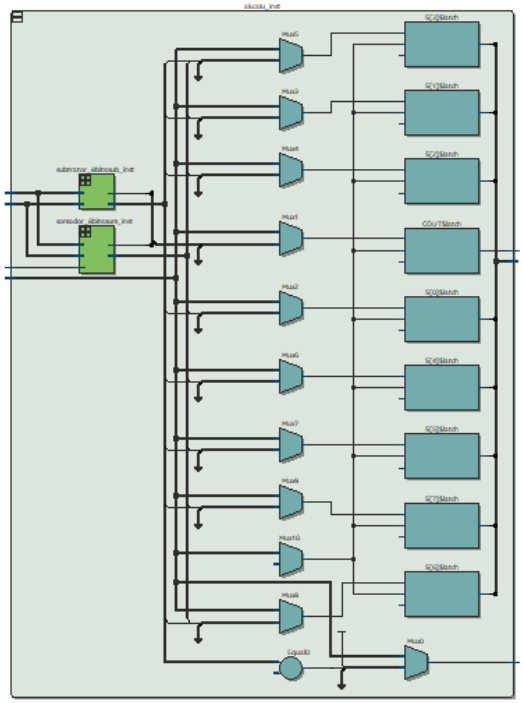
1.3.1 ALU ou ULA

O componente ALU (Unidade Lógica e Aritmética) é responsável por executar operações matemáticas e lógicas fundamentais dentro de um processador. Ele recebe dois operandos de 8 bits (A e B) e um código de operação de 4 bits (ALUOp) que determina a operação a ser realizada. Além disso, há uma entrada CIN para carry in e três saídas: S (resultado da operação), COUT (carry out ou borrow) e ZeroFlag (ativado quando o resultado é zero).

A ALU utiliza dois componentes auxiliares: um somador de 8 bits e um subtrator de 8 bits. Dependendo do valor de ALUOp, a ALU seleciona a operação correspondente. Algumas das principais operações são:

- 0101 (ADD): realiza a soma dos operandos, atualizando S com o resultado e COUT com o carry gerado.
- 0110 (SUB): realiza a subtração, ativando ZeroFlag se o resultado for zero, útil para operações de salto condicional (BEQ).
- 0010 (ADDI) e 0011 (SUBI): somam ou subtraem um valor imediato.
- 0100 (Salto Incondicional): força ZeroFlag para '1', garantindo que um salto ocorra.

Caso o código de operação não corresponda a nenhuma das instruções especificadas, a ALU retorna zero em todas as saídas.



1.3.2 Unidade de Controle

O componente Control 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:

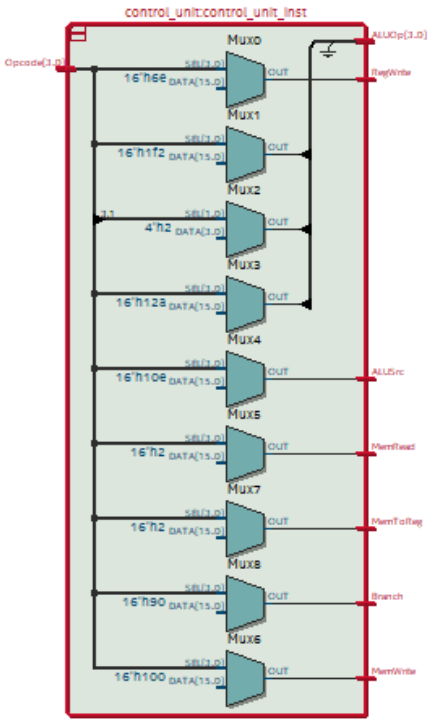
- **RegWrite**: Habilita ou desabilita a escrita em um registrador.
- **MemWrite**: Habilita ou desabilita a escrita na memória (RAM).
- **MemRead**: Habilita ou desabilita a leitura da memória.
- **MemToReg**: Define a origem do valor a ser escrito no registrador.
- **Branch**: Ativa ou desativa o salto condicional.
- **ALUSrc**: Controla qual será o operando B da ALU.
- **ALUOp:** Define a operação a ser realizada pela ALU.
-

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.

Comando	RegWrite	MemWrite	MemRead	MemToReg	Branch	ALUSrc	ALUOp
add	1	0	0	0	0	0	0101
sub	1	0	0	0	0	0	0110
addi	1	0	0	0	0	1	0010

subi	1	0	0	0	0	1	0011
lw	1	0	1	1	0	1	0101
sw	0	1	0	0	0	1	0101
beq	0	0	0	0	1	0	0110
jump	0	0	0	0	1	0	0100



1.3.3 Memória RAM

O componente RAM é responsável pelo armazenamento temporário de dados em uma memória de 8 bits, permitindo operações de leitura e escrita. Ele utiliza um clock para sincronizar as operações e garantir que os dados sejam acessados no momento correto.

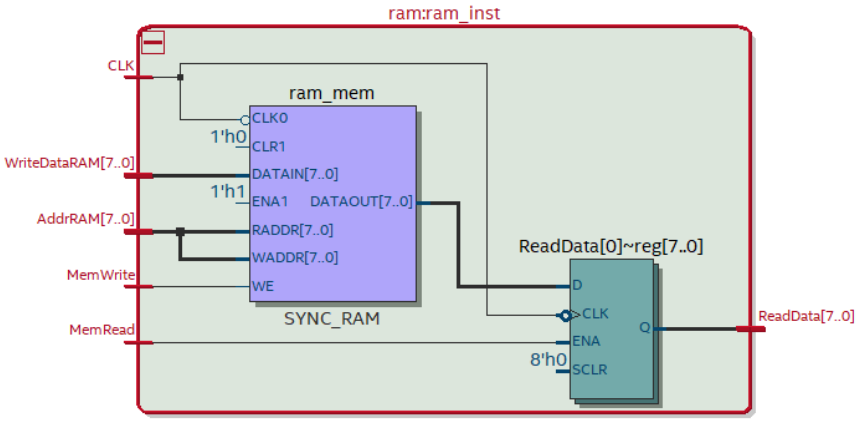
As entradas do componente incluem:

- CLK: sinal de clock que controla as operações de leitura e escrita;
- MemRead e MemWrite: sinais que indicam, respectivamente, se a memória deve ser lida ou escrita;
- AddrRAM: endereço de 8 bits que indica a posição na memória onde a operação será realizada;
- WriteDataRAM: dado de 8 bits que será armazenado na memória quando a escrita estiver habilitada.

A saída do componente é:

- ReadData: dado de 8 bits recuperado da memória quando a leitura é ativada.

A memória é implementada como um array de 256 posições, onde cada posição pode armazenar um valor de 8 bits. Durante a borda de descida do clock, se o sinal MemWrite estiver ativo, o dado fornecido na entrada WriteDataRAM será armazenado no endereço especificado. Da mesma forma, se o sinal MemRead estiver ativo, o dado armazenado na posição indicada será enviado para a saída ReadData.



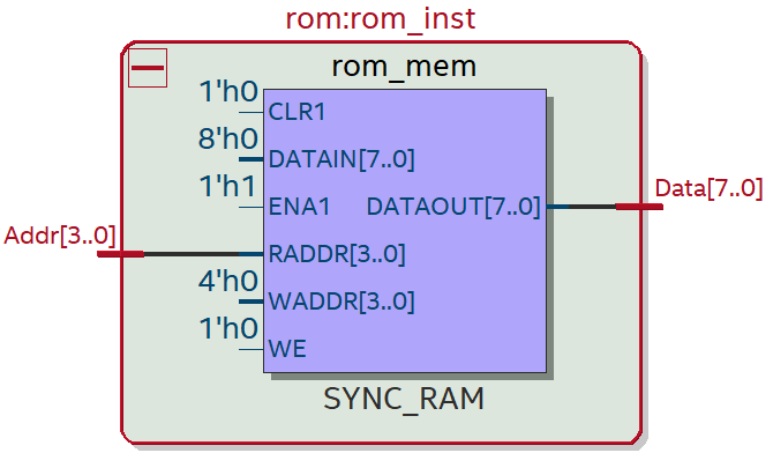
1.3.4 Memória ROM

O componente ROM representa uma memória somente leitura que armazena um conjunto fixo de instruções. Essa memória é essencial para armazenar programas ou dados pré-definidos que não podem ser modificados durante a execução.

As entradas e saídas do componente são:

- Addr: entrada de 4 bits que representa o endereço da instrução a ser lida, geralmente fornecido pelo contador de programa (PC);
- Data: saída de 8 bits que contém a instrução armazenada no endereço fornecido.

A ROM é implementada como um array de 16 posições, onde cada posição armazena uma instrução de 8 bits. Quando um endereço é recebido, a instrução correspondente é buscada e disponibilizada na saída Data. O conteúdo da ROM inclui um pequeno programa de teste com operações como adição, subtração e desvio, demonstrando seu uso no controle do fluxo de execução de um processador. Os espaços não utilizados são preenchidos com instruções NOP (00000000), que não realizam operações.



1.3.5 Somador

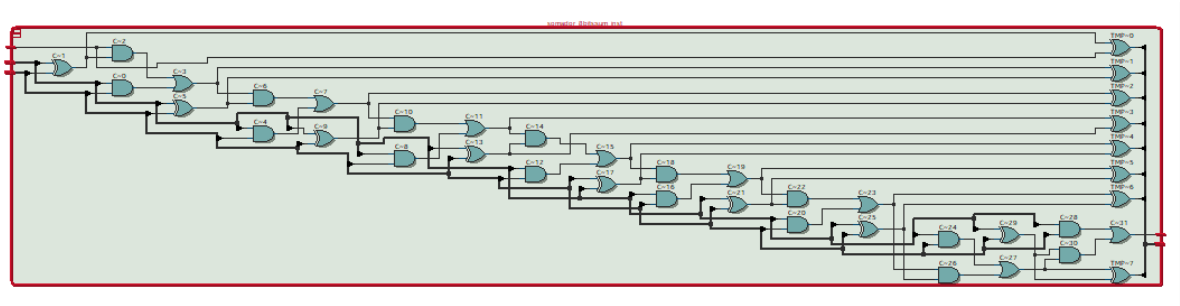
O componente somador_8bits tem a função de realizar a soma de dois valores de 8 bits.

As entradas do componente são:

- A e B: dois valores de 8 bits que serão somados;
- CIN: um bit que representa o carry de entrada.

As saídas do componente são:

- S: um valor de 8 bits que contém o resultado da soma;
- COUT: um bit que indica o carry de saída.



1.3.6 Subtrator

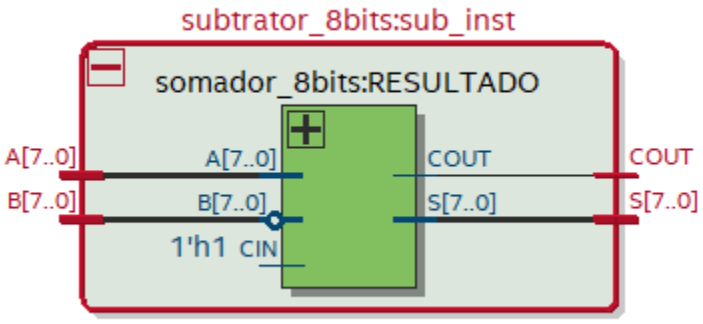
O componente subtrator_8bits tem a função de realizar a subtração entre dois valores de 8 bits.

As entradas do componente são:

- A e B: dois valores de 8 bits, onde B será subtraído de A.

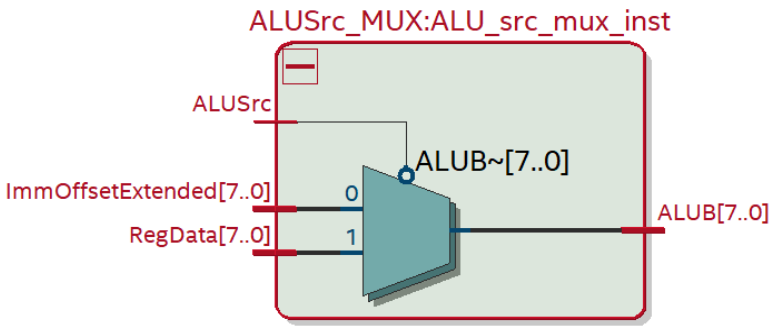
As saídas do componente são:

- S: um valor de 8 bits que contém o resultado da subtração;
- COUT: um bit que indica o carry de saída.



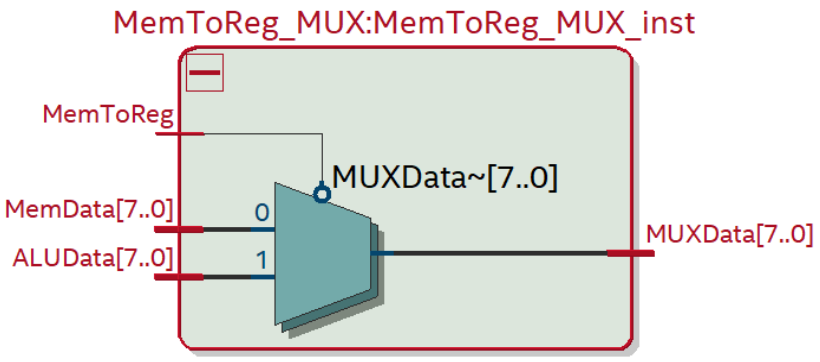
1.3.7 Multiplexador ALUSrc

O ALUSrc_MUX é um multiplexador responsável por determinar a entrada B da ALU, que pode vir de um registrador ou de um valor imediato estendido, dependendo da operação. Para operações como ADD e SUB, a entrada B vem de um registrador. Já para instruções como ADDI, SUBI, LOAD e STORE, a entrada B é um valor imediato estendido, que é manipulado pela ALU. Esse multiplexador garante que a ALU receba a fonte correta de dados para realizar a operação necessária, oferecendo flexibilidade nas operações aritméticas e lógicas.



1.3.8 Multiplexador MemToReg

O MemToReg_MUX é um multiplexador que decide a origem do valor a ser escrito em um registrador. Se a operação for de LOAD, o valor vem da memória. Para outras operações, como ADD, ADDI, SUB e SUBI, o valor vem da ALU. Esse multiplexador garante que o registrador receba o dado correto, seja ele proveniente da memória ou do resultado de uma operação aritmética ou lógica.

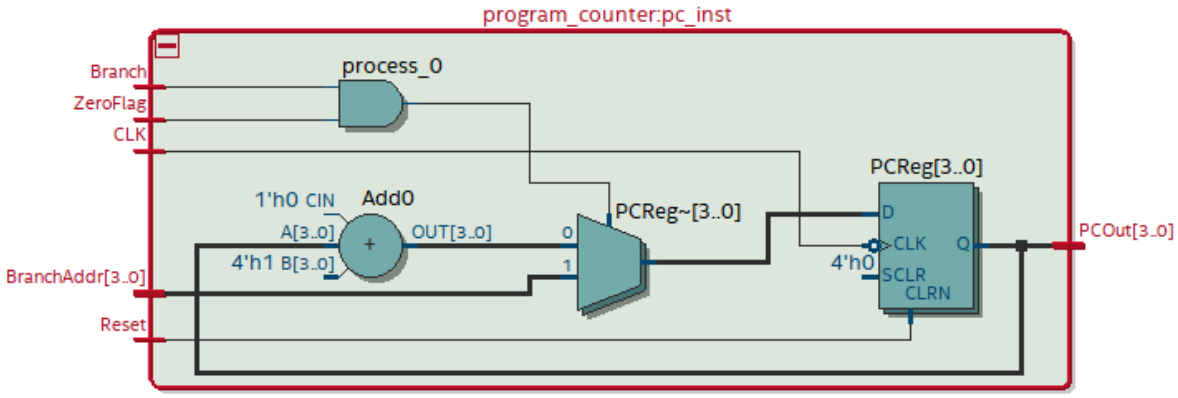


1.3.9 PC

O componente program_counter (PC) é responsável por controlar o endereço da instrução a ser executada em um processador, gerenciando o fluxo do programa. Ele recebe um clock (CLK) para determinar quando atualizar o valor do contador de programa, um reset para reiniciar seu valor, e um sinal de Branch para indicar se deve ocorrer um salto condicional baseado na ZeroFlag, que é ativada quando o resultado da ALU é zero.

O program_counter possui um registrador interno (PCReg) que armazena o endereço atual da instrução. Quando o reset é acionado, o PCReg é reiniciado para "0000". Caso contrário, em cada ciclo de clock, o PC é atualizado. Se o sinal de Branch estiver ativo e a ZeroFlag for igual a '1', o contador de programa será atualizado com o BranchAddr, que contém o endereço absoluto para o salto (geralmente usado em instruções do tipo BEQ). Caso contrário, o PC simplesmente é incrementado em 1 para apontar para a próxima instrução.

Por fim, o valor de PCReg é disponibilizado na saída PCOut, que reflete o endereço atual da instrução que está sendo executada.

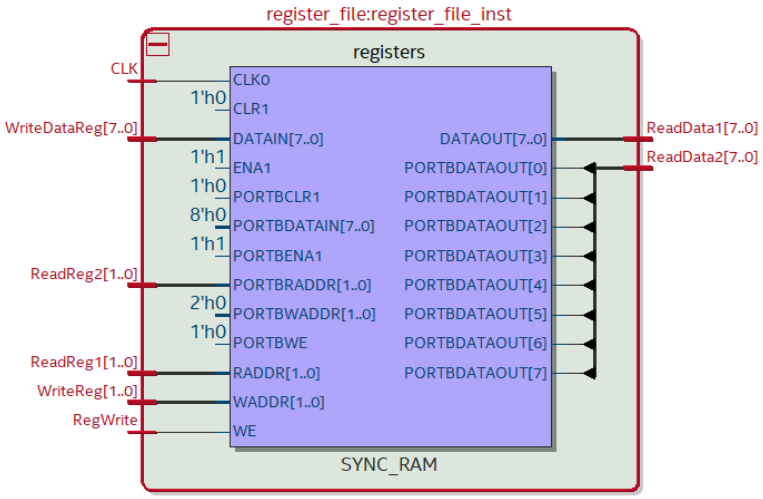


1.3.10 Banco de Registradores

O componente register_file representa um banco de registradores de 8 bits, contendo quatro registradores nomeados de R0 a R3. Ele permite a leitura e escrita de dados em registradores específicos, sendo controlado pelo clock (CLK) e pelo sinal de escrita (RegWrite).

Na operação de escrita, quando o clock detecta uma borda de subida (rising_edge(CLK)) e RegWrite está ativo ('1'), o valor de WriteDataReg é armazenado no registrador especificado por WriteReg.

A leitura ocorre de forma assíncrona, ou seja, os valores dos registradores são recuperados imediatamente quando ReadReg1 ou ReadReg2 são alterados. Os dados lidos são disponibilizados nas saídas ReadData1 e ReadData2. Caso os registradores de leitura estejam em um estado indefinido (UU), o componente retorna 00000000.

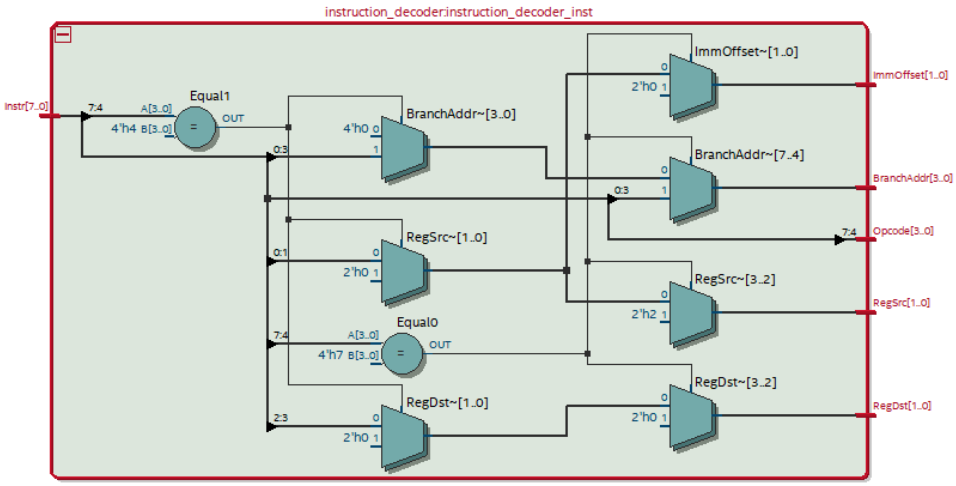


1.3.11 Decodificador de Instruções

O componente instruction_decoder é responsável por decodificar instruções de 8 bits, extraindo os campos necessários para execução no processador. Ele separa a instrução em diferentes partes, incluindo o código de operação (Opcode), os registradores de destino (RegDst) e origem (RegSrc), valores imediatos ou deslocamentos (ImmOffset) e endereços de salto (BranchAddr).

Se a instrução for um BEQ (branch if equal), os últimos 4 bits representam um endereço absoluto de salto, enquanto os registradores de destino e origem são fixos. Já no caso de um Jump, o campo de endereço também é extraído, mas os registradores não são utilizados, pois a ALU apenas ativa a flag de zero para indicar o salto.

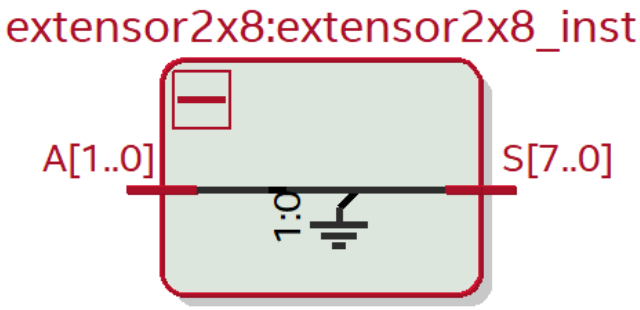
Para outras operações, o decodificador segue um formato padrão, onde os bits são divididos entre Opcode (4 bits), RegDst (2 bits) e RegSrc/ImmOffset (2 bits). Dessa forma, o processador pode interpretar corretamente os diferentes tipos de instruções.



1.3.12 Extensor de Sinal

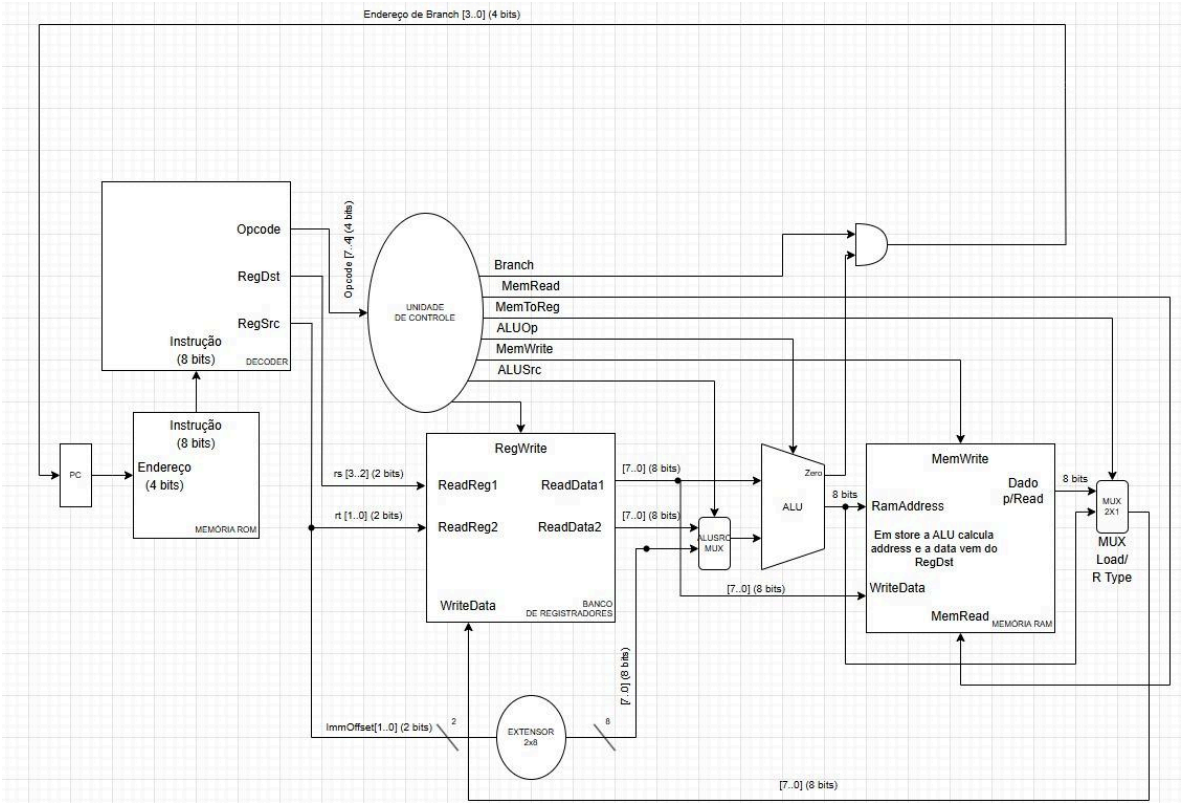
O componente extensor2x8 tem como objetivo expandir um vetor de 2 bits (A) para um vetor de 8 bits (S), mantendo os dois bits mais significativos de A nas duas posições correspondentes de S e preenchendo o restante com zeros.

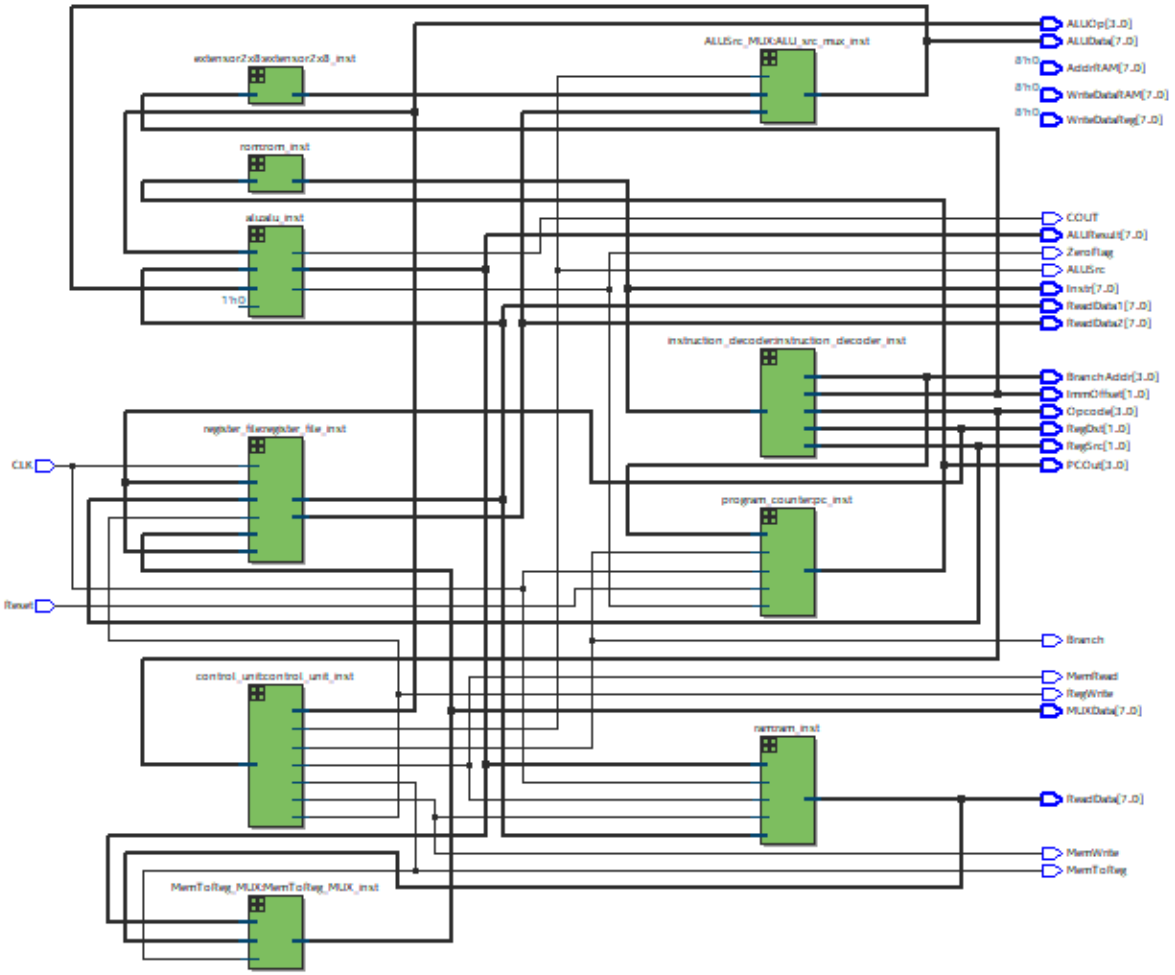
A entrada A é um vetor de 2 bits, e a saída S é um vetor de 8 bits. No processo de expansão, os seis bits mais significativos de S (de S(7) a S(2)) são definidos como '0', enquanto os dois bits menos significativos de S (S(1) e S(0)) recebem os valores correspondentes de A.



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.

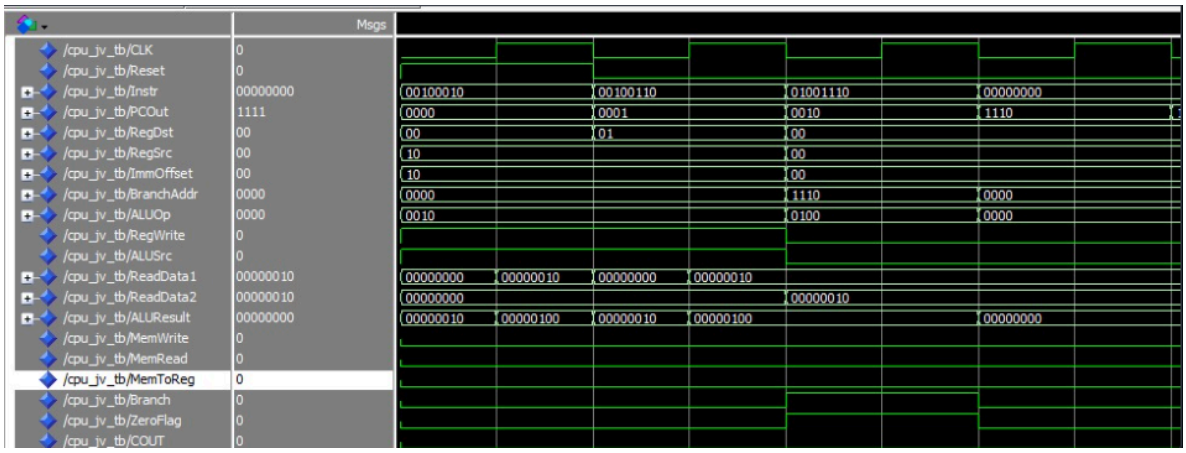



ALUOp[1..0]
ALUData[7..0]
8'h0
AddrRAM[7..0]
8'h0
WriteDataRAM[7..0]
8'h0
WriteDataReg[7..0]
ALUSrc_MUX:ALU_src_mux_inst
extension2x8extension2x8_inst
romrom_inst
alualu_inst
1'h0
COUT
ALUResult[7..0]
ZeroFlag
ALUSrc
Instr[7..0]
ReadData1[7..0]
ReadData2[7..0]
instruction_decoderinstruction_decoder_inst
BranchAddr[3..0]
ImmOffset[1..0]
Opcode[3..0]
RegDst[1..0]
RegSrc[1..0]
PCOut[3..0]
register_fileregister_file_inst
CLK
Reset
program_counterpc_inst
Branch
control_unitcontrol_unit_inst
MemRead
RegWrite
MUXData[7..0]
ramram_inst
ReadData[7..0]
MemWrite
MemToReg
MemToReg_MUX:MemToReg_MUX_inst

2 Simulações e Testes

Objetivando analisar e verificar o funcionamento do processador, efetuamos um teste utilizando as instruções add, sub, addi, subi, beq e jump:

```
-- Teste 2 (Teste com ADD, SUB, ADDI, SUBI, BEQ)
"00100010", --addi de +2 para r0 (r0 = 0 + 2 = 2)
"00100110", --addi de +2 para r1 (r1 = 0 + 2 = 2)
"01010001", --add (r0 = r0 + r1 = 2 + 2 = 4)
"01010001", --add (r0 = r0 + r1 = 4 + 2 = 6)
"01100001", --sub (r0 = r0 - r1 = 6 - 2 = 4)
"00110010", --subi (r0 = r0 - 2 = 4 - 2 = 2
-- Nesse ponto, tanto r0 quanto r1 devem ser 2.
"01111000", --beq (o beq sempre compara r0 e r1, o endereço de salto é 1000 nesse caso)
"00000000", --Instrução N 0111, o beq deve pular para 1000
"01000000", --Jump para 0000, que é o inicio, formando um loop.
```



Descrição dos Sinais

- Instr: Instrução recebida no momento.
- PCOut: Posição da instrução lida na ROM.
- RegDst: Registrador de destino (00 - r0, 01 - r1, 10 - r2, 11 - r3).
- RegSrc: Registrador de origem (00 - r0, 01 - r1, 10 - r2, 11 - r3).
- ImmOffset: Offset para instruções ADDi, SUBi, LOAD e STORE. O campo ImmOffset passa pelo extensor 2x8 para a ALU.
- ALUOP: Código da operação na ALU.
- ReadData 1: Registrador lido no espaço 1 (Selecionado por RegDst).
- ReadData 2: Registrador lido no espaço 2 (Selecionado por RegSrc).
- ALUResult: Resultado da ALU no instante. Importante: A ALU tem funcionamento assíncrono devido a problemas encontrados ao utilizá-la com a borda de subida. Os registradores são atualizados na borda de subida, pegando o último valor encontrado na ALU na borda de descida. Portanto, qualquer valor em ALUResult na subida deve ser ignorado, pois não influencia em nada.

Descrição do Teste:

Momento 1: "00100010" (addi de +2 para r0)

Intervalo: 0 a 10 ns

Na borda baixa, o PC recebe o endereço da instrução a ser lida e decodificada.

- Opcode: 0010
- Registrador Destino: 00 (r0)
- Imediato: 10 (2)
- ReadData 1 (r0): 00000000
- ALUResult (borda baixa): 00000010

Na borda alta, o valor lido em ReadData 1 (r0) é atualizado para 2, pegando o ALUResult da borda baixa como resultado a ser escrito.Comentário:Na borda baixa, o PC recebe o endereço da instrução a ser lido e decodificado.

Comentário :
Na borda baixa, o PC recebe o endereço da instrução a ser lido e decodificado.

Momento 2: "00100110" (adicional de +2 para r1)

- Intervalo : 11 a 20 minutos
- Na borda baixa, o PC recebe o endereço da instrução a ser lida e decodificada.
    - Código de operação : 0010
    - Registrador Destino : 01 (r1)
    - Imediato : 10 (2)
    - Dados de leitura 1 (r1) : 00000000
    - ALUResult (borda baixa) : 00000010
- Na borda alta, o valor lido em ReadData 1(r1) é atualizado para 2, pegando o ALUResultda borda baixa como resultado a ser escrito.

Momento 3: "01010001" (adicionar: r0 = r0 + r1 = 2 + 2 = 4)

- Intervalo : 21 a 30 minutos
- Na borda baixa, o PC recebe o endereço da instrução a ser lida e decodificada.
    - Código de operação : 0101
    - Registrador Destino : 00 (r0)
    - Registrador Origem : 01 (r1)
    - Dados de leitura 1 (r0) : 00000010

    - Dados de leitura 2 (r1) : 00000010
    - ALUResult (borda baixa) : 00000100
- Na borda alta, o valor lido em ReadData 1(r0) é atualizado para 4, pegando o ALUResultda borda baixa como resultado a ser escrito.

Momento 4: "01010001" (adicionar: r0 = r0 + r1 = 4 + 2 = 6)

- Intervalo : 31 a 40 minutos
- Na borda baixa, o PC recebe o endereço da instrução a ser lida e decodificada.
    - Código de operação : 0101
    - Registrador Destino : 00 (r0)
    - Registrador Origem : 01 (r1)
    - Dados de leitura 1 (r0) : 00000100
    - Dados de leitura 2 (r1) : 00000010
    - ALUResult (borda baixa) : 00000110
- Na borda alta, o valor lido em ReadData 1(r0) é atualizado para 6, pegando o ALUResultda borda baixa como resultado a ser escrito.

Momento 5: "01100001" (sub: r0 = r0 - r1 = 6 - 2 = 4)

- Intervalo : 41 a 50 ns
- Na borda baixa, o PC recebe o endereço da instrução a ser lida e decodificada.
    - Código de operação : 0110
    - Registrador Destino : 00 (r0)
    - Registrador Origem : 01 (r1)
    - Dados de leitura 1 (r0) : 00000110
    - Dados de leitura 2 (r1) : 00000010
    - ALUResult (borda baixa) : 00000100
- Na borda alta, o valor lido em ReadData 1(r0) é atualizado para 4, pegando o ALUResultda borda baixa como resultado a ser escrito.

Momento 6: "00110010" (subi: r0 = r0 - 2 = 4 - 2 = 2)

- Intervalo : 51 a 60 ns
- Na borda baixa, o PC recebe o endereço da instrução a ser lida e decodificada.
    - Código de operação : 0011
    - Registrador Destino : 00 (r0)
    - Dados de leitura 1 (r0) : 00000100
    - Dados de leitura 2 (r1) : 00000010

    - ALUResult (borda baixa) : 00000010
- Na borda alta, o valor lido em ReadData 1(r0) é atualizado para 2, pegando o ALUResultda borda baixa como resultado a ser escrito.

Momento 7: "01111000" (beq: compara r0 e r1, salto para 1000)

- Intervalo : 61 a 70 segundos
- Na borda baixa, o PC recebe o endereço da instrução a ser lida e decodificada.
    - Código de operação : 0111
    - Endereço da agência : 1000
    - Filial : 1
    - ZeroFlag : 1 (A subtração na ALU deu zero)
    - ALUResult (borda baixa) : 00000000
- Ocorre beq, pois a ZeroFlagestá ativado, fazendo o salto para o endereço 1000.

Momento 8: "00000000" (Instrução N 0111, o beq deve pular para 1000)

- Intervalo : 71 a 80 ns
- A instrução foi devidamente pulada graças ao BEQ.

Comentário :
A instrução foi legalmente pulada graças ao BEQ.

Momento 9: "01000000" (Salte para 0000, início do loop)

- Intervalo : 81 a 90 segundos
    - Código de operação : 0100
    - Endereço de Jump : 0000
    - Filial : 1
    - ZeroFlag : 1 (Não depende de subtrações)
    - ALUResult (borda baixa) : U (A ALU não realiza cálculo em casos de Jump, apenas ativa a ZeroFlage o salto ocorre)

Comentário :
A ALU não realiza nenhum cálculo em casos de Jump, apenas ativa o ZeroFlag e o salto ocorre.

3 Considerações finais

Este trabalho apresentou o projeto e implementação do processador de 8 bits denominado JV, detalhando sua arquitetura, conjunto de instruções e funcionamento dos principais componentes. Através da especificação em VHDL, foi possível validar seu comportamento e compreender melhor o fluxo de execução das operações. O desenvolvimento desse processador contribui para o aprendizado sobre Arquitetura e Organização de Computadores.

Além disso, a implementação permitiu explorar técnicas de controle e processamento de dados, como a interação entre registradores, memória e a unidade de controle. A simulação dos componentes e a análise dos sinais de controle demonstraram a importância de cada módulo na execução eficiente das instruções. Infelizmente, devido a problemas de inicialização de variáveis (WriteDataRAM) (AddrRAM) e (ReadData (ram)), que nossa equipe não conseguiu corrigir mesmo com muito esforço, não conseguimos fazer a análise no caso de sw e lw. Portanto, as operações LOAD e STORE estão com problema e não podem ser exibidas no nosso processador. Vale dizer que individualmente, a RAM funciona, e o teste perfoma como esperado.

Dessa forma, o projeto do processador JV se mostrou uma experiência enriquecedora, proporcionando uma visão prática dos princípios que fundamentam os sistemas computacionais modernos.

Repositório:
https://github.com/vinimartinsufrr/AOC_ViniciusMartins_JasmimSabini_UFRR2024_ProjetoFinal/tree/main