SUMÁRIO

1. INTRODUÇÃO

Uma unidade central de processamentos (CPU) é um circuito eletrônico que performa, seguindo instruções, performa operações aritméticas, lógicas, de controle ou de armazenamento. Esse circuito é programável por funcionar seguindo instruções, ou seja, pode executar uma sequência de ações de acordo com algum algoritmo moldado por elas. Deste modo, o poder de uma CPU encontra-se na sua capacidade de fazer diversas operações úteis de forma controlada por programas. De certa forma, pode ser considerado como o “cérebro” de um computador.

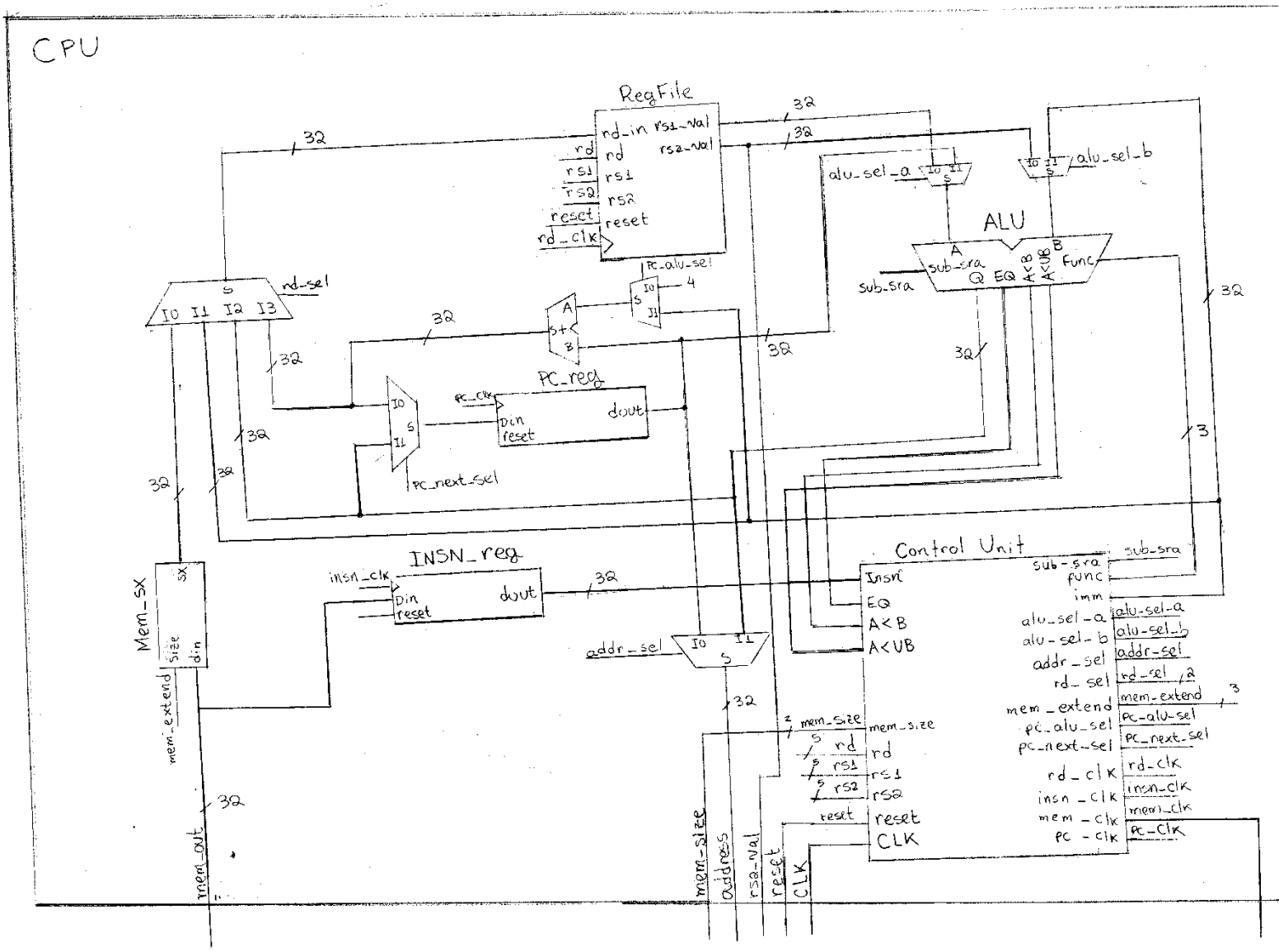
Uma das arquiteturas utilizadas em processadores é a Reduced Instruction Set Computer (RISC), a qual fornece um conjunto simples e pequeno de instruções que levam quase a mesma quantidade de tempo para serem executadas. Um exemplo de processador que a utiliza é o RISC-V, o qual implementa conjunto universal de instruções (Instruction Set Architecture – ISA), acomodando várias tecnologias de implementação, como FGPA, além de funcionar com linguagens de programação como o C. A variante RISC-V 32I é uma implementação do RISC-V que faz todas as suas operações com números inteiros de 32bits.

A ISA estabelece um modelo de como cada instrução, nesse caso entendida como uma palavra de 32bits, definirá como que o processador funcionará. Para RISC-V 32I (RV-32I, de forma mais abreviada), são definidos 10 tipos de instruções, cada um com um conjunto de delas que executam funções semelhantes. Um processador funcional deve ser capaz de receber uma palavra de 32bits codificada de acordo com a ISA e, a partir disso, executar a operação correta. Nesse documento, será descrita uma implementação do processador RV-32I descrito em verilog. Cada módulo presente no circuito será descrito em um arquivo, o qual será acompanhado de um testbench.

2. VISÃO GERAL

2.1 Diagrama de blocos

Primeiramente, para entender como que todo ciclo de execução é feito, é necessário esclarecer a estrutura principal do processador projetado. A qual pode ser vista no diagrama abaixo:



O diagrama acima foi feito adotando simplificações convenientes, pois seu objetivo, ao menos por agora, é de apenas revelar quais são os blocos principais do circuito. Os sinais foram, também, simplificados.

2.2 Blocos fundamentais

Cada bloco no diagrama é um circuito com uma função própria, podendo conter dentro dele vários outros blocos. Suas finalidades são:

Control Unit (CU): Gerar sinais de controle a partir da instrução recebida da memória. Cada instrução possui um opcode (um “identificador” do tipo da instrução), o qual serve de base para a CU decidir quais sinais enviar para os outros componentes. Assim, seu papel é decodificar a palavra recebida e gerar sinais como seletores, clocks, endereços ou códigos pertinentes que comandarão o funcionamento dos outros circuitos.

Regfile: Ser o elemento de memória principal do processador. É um conjunto de 32 registradores, os quais armazenam dados e podem ser lidos ou escritos de acordo com os sinais recebidos da CU. Todos eles são acessados através de seus endereços (palavras de 5 bits).

Multiplexadores (MUX): Selecionar qual entrada aparecerá na saída. É um circuito capaz de fazer decisões, permitindo a passagem de dados somente da entrada que for selecionada pelo seletor, o qual é gerado pela CU.

Elementos de memória: Armazenar valores que serão utilizados por outros blocos. São registradores especiais que guardam valores importantes para execução da instrução. Eles são Program Counter (PC) e Instruction Register (INSNReg), os quais serão melhor explicados mais adiante.

Unidade Lógico Aritmética (ALU): Realizar operações aritméticas e lógicas. Os valores em que as operações são feitas, o tipo de operação e as saídas da ALU são todos controlados por sinais emitidos da CU.

2.2 Funcionamento geral

Este processador segue o modelo de Von Neumann, onde as instruções de execução são armazenadas na mesma memória que as informações. A execução do processador se dá pelo ciclo Fetch – Decode – Execute.

Fetch: Na operação de Fetch, acessamos a memória usando como endereço o valor contido no registrador PC e extraímos uma instrução, a qual é então armazenada em um registrador de instruções.

Decode: Nesta operação, a instrução é passada para a unidade de controle que, baseando-se da codificação especificada na ISA do RV-32I, envia sinais e pulsos de Clock para os diversos componentes do circuito.

Execute: Agora, todos os sinais propagados pela unidade de controle são utilizados para executar a instrução que foi decodificada, sendo alguns sinais representando diferentes coisas, como a função que uma ALU deve desempenhar, e os Clocks mantendo a sincronia do circuito. Neste passo são utilizadas parte dos componentes, como a ALU caso precisemos realizar operações aritméticas ou lógicas, ou apenas os registradores, caso a instrução mande gravar um valor em um deles. Ao fim do ciclo execute, na maior parte das instruções, tirando algumas de Branch, sempre é somado 4 ao Program Counter, garantindo assim que o ciclo se inicie novamente, e o processador continue a executar instruções, até chegar ao fim.

2.3 ISA do RISC-V 32I

Existem ao todo 39 instruções, agrupadas em 10 tipos. Por padronização, alguns campos são reservados para guardar informações sobre coisas especificas. O opcode de cada instrução sempre está nos 7 lsb (least significant bits) da palavra. Instruções que tiverem estes campos, possuem sempre:

- Endereço para um registrado de destino (rd) codificado do bit 11 ao 7 da palavra
- Código func (especifica instrução dentro do seu tipo) do bit 14 ao 12.
- Endereço do registrador de saída 1 (rs1) codificado do bit 19 ao 15.
- Endereço do registrador de saída 2 (rs2) codificado do bit 24 ao 20.

Esse padrão facilita a decodificação, pois apenas sabendo o opcode é possível localizar essas informações sem maiores esforços. Os imediatos (imm – valores numéricos codificados diretamente na palavra), variam com os tipos de funções, necessitando trabalho extra para extrai-los da instrução. A seguir são expostos o formato geral de cada tipo, mostrando sua codificação em cada bit da palavra.

OBS: Nos campos dos imediatos (imm), estão especificados entre colchetes quais bits do valor do imediato estão representados por aqueles bits da palavra. No tipo I - JARL, por exemplo, IMM [11:0] indica que os bits [31:20] da palavra irão se localizar nos bits [11:0] do imediato, seguindo a ordem.

OBS: Nos exemplos dos códigos, as expressões após “//” indicam a operação feita

Tipo U - LUI:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
IMM [31:12]																				rd					0	1	1	0	1	1	1

Instrução load upper immediate (lui), a qual armazena no registrador rd o valor imm. Os bits [11:0] do imediato são preenchidos com 0. Formato: lui rd, imm

Exemplo de instrução: 000082B7 lui x05, 0x08 // x05 = 0x8000

Tipo U – AUIPC:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
IMM [31:12]																				rd					0	0	1	0	1	1	1

Instrução Add Upper Immediate to Program Counter (auipc), a qual soma o imm valor atual do PC e guarda o resultado no rd. Os bits [11:0] do imediato são preenchidos com 0. Formato: auipc rd, imm

Exemplo de instrução: 00001F97 auipc x1F, 0x01 // x1F = PC + 0x01000

Tipo J:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
IMM [20, [10:1], 11, [19:12]]																				rd					1	1	0	1	1	1	1

Instrução jump and link (jal), a qual guarda o valor da próxima instrução no rd e pula o PC para o valor “imm + PC”. Formato: jal rd, imm.

Exemplo de instrução: 008001EF jal x03, 0x08 // x03 = PC + 4, PC = PC + 8

Tipo I – JALR:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
IMM [11: 0]												rs1					0	0	0	rd					1	1	0	0	1	1	1

Instrução jump and link register (JALR), a qual guarda no rd o endereço da próxima instrução e pula o PC para o valor do imm somado com o valor do rs1. Formato: jalr rd, imm(rs1)

Exemplo de instrução: 00410267 jarl x04, 4(0x02) // x04 = PC + 4, PC = 4 + rs1

Tipo B:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
IMM [12, 10: 5]							rs2					rs1					func			IMM [4:1, 11]					1	1	0	0	0	1	1

Instruções de Branch, as quais verificam uma condição entre rs1 e rs2, pulando o PC para PC + imm caso seja verdadeira ou prosseguindo para próxima instrução caso seja falsa. Esse tipo de instrução possui um campo func, o qual especifica qual branch será feito. Formato: *branch* rs1, rs2, imm.

Valores de func possíveis:
000 – Branch equal (beq)
001 – Branch note qual (bne)
100 – Branch less than (blt)
101 – Branch greater or equal (bge)
110 – Branch less than unsigned (bltu)
111 – Branch greater or equal unsigned (bgeu)
Exemplo de instrução: 0040E463 bltu x01, x04, 0x08 // (rs1 < rs2)? PC = PC + 8 : PC + 4

Tipo I – LOAD:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
IMM [11: 0]												rs1					func			rd					0	0	0	0	0	1	1

Instruções de load, as quais pegam um valor da memória no endereço determinado por imm + rs1 e armazena ele no rd. Esse tipo de instrução possui um campo func, o qual especifica qual o tamanho da palavra extraída da memória, assim como se será feito signed ou unsigned extend nela caso seja menor que 32bits. Formato: *load* rd, imm(rs1).

Valores de func possíveis:
000 – Load byte (lb)
001 – Load half-word (lh)
010 – Load word (lw)
100 – Load byte unsigned (lbu)
101 – Load half-word unsigned (lhu)
Exemplo de instrução: 00C21403 lh x08, 12(x04) // x08 = sx(m16(12 + rs1))

Tipo S:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
IMM [11:5]							rs2					rs1					func			IMM [4:0]					0	1	0	0	0	1	1

Instruções store, as quais armazenam na memória, no endereço especificado por rs1 + imm, o valor contido no rs2. Esse tipo de instrução possui um campo func, o qual especifica qual o tamanho da palavra que será guardada na memória. Formato: *store* rs2, imm(rs1).

Valores de func possíveis:

000 – Store byte (sb)

001 – Store half-word (sh)

010 – Store word (sw)

Exemplo de instrução: 00521623 sh x05, 12(x04) // mem(12 + rs1) = rs2

Tipo I - ALU:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
IMM [11: 0]												rs1					func			rd					0	0	1	0	0	1	1

Instruções do tipo imediato, as quais fazem operações lógico aritméticas entre o valor do rs1 e o imm e armazena o resultado no rd. Esse tipo de instrução possui um campo func, o qual especifica qual operação será feita. Formato: *I* rd, rs1, imm.

Valores de func possíveis:

000 – Add immediate (addi)

010 – Set less than immediate (slti)

011 – Set less than immediate unsigned (sltiu)

100 – Xor immediate (xori)

110 – Or immediate (ori)

111 – And immediate (andi)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	S	0	0	0	0	0	shamt					rs1					func			rd					0	0	1	0	0	1	1

Para operações de shift em especifico, a magnitude do shift se localiza no campo shamt e, no bit 30 da instrução, se encontra um parâmetro que especifica se um shift right é lógico ou aritmético: será lógico se for 0 e aritmético caso seja 1

Valores de func possíveis:

001 – Shiftl left logical immediate (slli)

101 – Shift right logical immediate (srli)

101 – Shift right arithmetic immediate (srai)

Exemplo de instrução: 00109913 slli x12, x01, 0x01 // x12 = rs1 << 1

Tipo R:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	S	0	0	0	0	0	rs2					rs1					func			rd					0	1	1	0	0	1	1

Instrução do tipo register, as quais realizam operações lógico aritméticas entre os registradores rs1 e rs2 e guardam o resultado no rd. Esse tipo de instrução possui um campo func, o qual especifica qual operação será feita. Novamente, no bit 30 da palavra se encontra um parâmetro que especifica se um shift right será lógico ou aritmético, além de também especificar se será feita uma soma ou uma subtração. Formato: *R* rd, rs1, rs2

Valores de func possíveis:

000 – Add (add) –> Com S = 0

000 - Add (add) –> Com S = 1

001 – Shiftl left logical (sll)

010 – Set less than (slt)

011 – Set less than unsigned (sltu)

100 – Xor (xor)

101 – Shift right logical (srl)

101 – Shift right arithmetic (sra)

110 – Or (or)

111 – And (and)

Exemplo de instrução: 40D08B33 sub x16, x01, x0D // x16 = rs1 – rs2

Instruções CSR:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	0	0	0	0	0	0	0	0	0	0	S	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	1

Instruções ebreak e call, as quais, no contexto do processador em questão, apenas travam o PC no mesmo valor (ebreak) ou reseta o PC para o valor 0 (ecall). O parâmetro S no bit 20 diferencia as duas instruções. Formato: ebreak/ecall

Exemplo de instrução: 00100073 ebreak // PC = PC + 0

2.3.1 Executando uma instrução

Para esclarecer um pouco o funcionamento do processador, será simulada de forma simplificada a execução uma instrução bne.

00309463 bne x01, x03, 0x08 // (rs1 != rs2)? PC + 8 : PC + 4

Essa instrução é a palavra de 32bits apresentada a seguir:

0x00309463 = 00000000001000010010100011000011

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	0	1	0	1	0	0	0	1	1	0	0	0	1	1

Com:

- Opcode: 1100011
- Func: 001
- Rs1: 0x01
- Rs2: 0x03
- Imm: 0x08

Consideremos que os valores armazenados nos rs1 e rs2 sejam, respectivamente, 0xC4 e 0x14 (diferentes). Consideremos também o sinal periódico clock (clk), sendo que os circuitos sequenciais em questão são ativados na borda de subida/positiva dele.

O processador receberia a palavra da memória e a armazenaria no registrador de instruções em uma borda de subida do clock de instruções (gerado na CU) e, após outra borda, a enviaria para a CU. Dentro da Control Unit, o opcode seria decodificado, revelando que se trata de uma instrução de branch. Além disso, seria obtido diretamente o valor de func, indicado que se trata de um branch not equal. Assim, a Control Unit mandaria sinais para ALU, os quais: selecionaram os valores de entrada como sendo rs1 e rs2, selecionaram a operação como uma subtração (para funcionamento do comparador) e receberia a saída do resultado da comparação (sinal equal, EQ).

Como consideramos que os valores de rs1 e rs2 são diferentes, o sinal equal da ALU retornaria 0 (falso). Como não são iguais, deverá ocorrer o branch. Assim a Control Unit manda um sinal para que o PC recebe seu próprio valor somado com um imediato, operação que será feita por um pequeno somador dedicado. Na próxima borda do clock do PC, gerado pela CU, o novo valor do Program Counter seria registrado, terminando a execução a instrução atual.

3. IMPLEMENTAÇÃO

Tendo conhecimento da ISA e sabendo o funcionamento/relações entre os blocos básicos, é feita a implementação do processador utilizando o verilog. Nessa seção será detalhada a implementação de cada circuito, além da nomenclatura padrão adotada e as convenções para os sinais transmitidos.

3.1 Nomenclatura adotada

Aqui retomaremos algumas nomenclaturas, terminologias e abreviações e introduziremos outras, a fim de padronização:

OBS: Essas convenções adotadas são utilizadas em diversos lugares no código do projeto, como por exemplo no nome de sinais, em que "_sel" indica que o sinal é um seletor.

Reg: Um registrador. Circuito que armazena um valor, podendo ele ser lido ou escrito.

Regfile: Conjunto de 32 registradores do processador que armazenam dados, podendo ter valores lidos e escritos.

rd: Registrador de destino. Um registrador presente no regfile que será escrito.

rs1: Registrador de saída 1. Um registrador presente no regfile que será lido.

rs2: Registrador de saída 2. Um registrador presente no regfile que será lido.

Opcode: 7 lsb da instrução, código que diferencia os tipos de instruções.

Func: Código presente na instrução que diferencia instruções dentro de seus tipos.

Imm: Imediato. Valor numérico codificado em uma instrução

ALU: Unidade Lógico Aritmética, responsável por fazer diversas operações.

CU: Control Unit, responsável gerar sinais que controlam outros circuitos.

clk: Sinal de clock, um sinal periódico.

Mux: Multiplexer, bloco que seleciona qual entrada será passada para a saída

Sel: Abreviação de seletor, sinal que seleciona alguma entrada em um mux.

PC: Program Counter. Valor do endereço na memória de uma instrução.

addr: Endereço. Conjunto de bits que codificam a localização de algo em um conjunto. Ex: endereço na memória.

Val: Valor. Remete a um valor numérico.

Mem: Memória. Geralmente se refere à memória em que as instruções estão armazenadas.

Insn: Instrução.

Gate: Um multiplexador em que o seletor é um código de n bits, sendo n a quantidade de entradas.

Palavra/word: Conjunto de bits. Geralmente se refere a um conjunto de 32bits

Halfword: Se refere a uma palavra de 16bits.

3.2 Sinais

Para entendimento do funcionamento do processador, é essencial entender anteriormente quais são os sinais que relacionam todos os blocos e suas funções. A seguir, serão expostos os principais sinais presentes no circuito, além da explicação de sua finalidade

3.2.1 Clocks

Os clocks são sinais periódicos gerados para controlar circuitos sequenciais. Clocks presentes no circuito:

Clk: clock principal

Rd_clk: clock do rd. Toda vez que esse sinal emite uma borda de subida, o registrador destino selecionado irá receber o valor de entrada.

Mem_clk: clock da memória. Toda vez que esse sinal emite uma borda de subida, a memória irá gravar o conteúdo em sua porta de entrada no endereço selecionado.

Insn_clk: clock do insn reg. Toda vez que esse sinal emite uma borda de subida, o insn reg armazena uma nova instrução.

Pc_clk: clock do PC. Toda vez que esse sinal emite uma borda de subida, o PC atualiza seu valor (o qual será decido por seletores mencionados mais adiante).

3.2.2 Seletores

Esses sinais são responsáveis por selecionar as saídas dos mux.

Alu_sel_a: Porta responsável por escolher a primeira entrada da ALU (primária), podendo ser o registrador RS1 ou o Program Counter.

- Alu_sel_a = 0, seleciona o registrador RS1

- Alu_sel_a = 1, seleciona program counter PC

Alu_sel_b: Porta responsável por escolher a segunda entrada da ALU (primária), podendo ser o registrador RS2 ou um imediato.

- Alu_sel_b = 0, seleciona registrador RS2
- Alu_sel_b = 1, seleciona imm

Addr_sel: Sinal que seleciona se o endereço a acessar na RAM deve ser o Program Counter (possivelmente acessando uma instrução) ou o valor da ALU (possível executando um store).

- Addr_sel = 0, seleciona o PC para endereço atual
- Addr_sel = 1, selecionando resultado da ALU para o endereço atual

Rd_sel: Sinal que seleciona qual valor deve ir para a entrada da regfile.

- Rd_sel = 00, selecionando valor da memória para o RD
- Rd_sel = 01, selecionando imediato para o RD
- Rd_sel = 10, selecionando resultado da ALU para o RD
- Rd_sel = 11, Program Counter incrementado para o RD

Pc_next_sel: Sinal que seleciona qual deverá ser o próximo valor irá para o data do registrador PC (Program Counter).

- Pc_next_sel = 0, Program Counter recebe program counter + 4
- Pc_next_sel = 1, Program Counter recebe resultado da alu

Pc_alu_sel: Sinal que seleciona qual deve ser o valor que será enviado para a segunda entrada da ALU do Program Counter, podendo ser o 4 (para passar para a próxima instrução) ou um imediato.

- Pc_alu_sel = 0, incrementa Program Counter de 4
- Pc_alu_sel = 1, incrementa Program Counter de um valor arbitrário

Sub_sra: Sinal responsável por decidir se deve ser realizada uma subtração ao invés de uma soma ou um shift aritmético ao invés de um shift lógico. Não um seletor de MUX, mas seleciona uma operação.

- sub_sra = 0, operações convencionais
- sub_sra = 1, operações subtração e shift aritmético

mem_extend: Códigos para extensão de valores provenientes da memória, integrado na própria instrução. Não é um seletor de um MUX, mas seleciona qual extensão será feita no valor obtido.

- mem_extend = 000, guarda um byte extendido signed

- mem_extend = 001, guarda uma halfword extendida signed
- mem_extend = 010, guarda uma word, não é necessário extensão pois não existem zeros à esquerda
- mem_extend = 100, guarda um byte extendido unsigned
- mem_extend = 101, guarda uma halfword extendida unsigned

mem_size: Código que indica qual o tamanho da palavra que será armazenada na memória. Não é um seletor de um MUX, mas seleciona um tamanho.

- Mem_size = 00, armazena um byte.
- Mem_size = 01, armazena uma halfword
- Mem_size = 10, armazena uma word.

3.2.3 Valores e endereços

Sinais diversos que carregam valores numéricos ou endereços.

Addr: Sinal que carrega o endereço que será acessado na memória.

Insn: Sinal que carrega o valor da instrução.

Rd: Sinal que carrega o endereço do rd.

Rd_val: Sinal que carrega o valor que será armazenado no rd.

Rs1: Sinal que carrega o endereço do rs1.

Rs1_val: Sinal que carrega o valor que foi lido do rs1.

Rs2: Sinal que carrega o endereço do rs2.

Rs2_val: Sinal que carrega o valor que foi lido do rs2.

Imm: Sinal que carrega o valor do imediato.

Alu_val_a: Valor da primeira entrada numérica da ALU.

Alu_val_b: Valor da segunda entrada numérica da ALU.

Alu_val: Valor que sai da ALU.

Func: Valor do código func.

EQ: Sinal equal emitido pela ALU. Valor 1 caso A = B, 0 caso contrário.

LS: Sinal less than signed emitido pela ALU. Valor 1 caso A < B (signed), 0 caso contrário.

LU: Sinal less than unsigned emitido pela ALU. Valor 1 caso A < B (unsigned), 0 caso contrário.

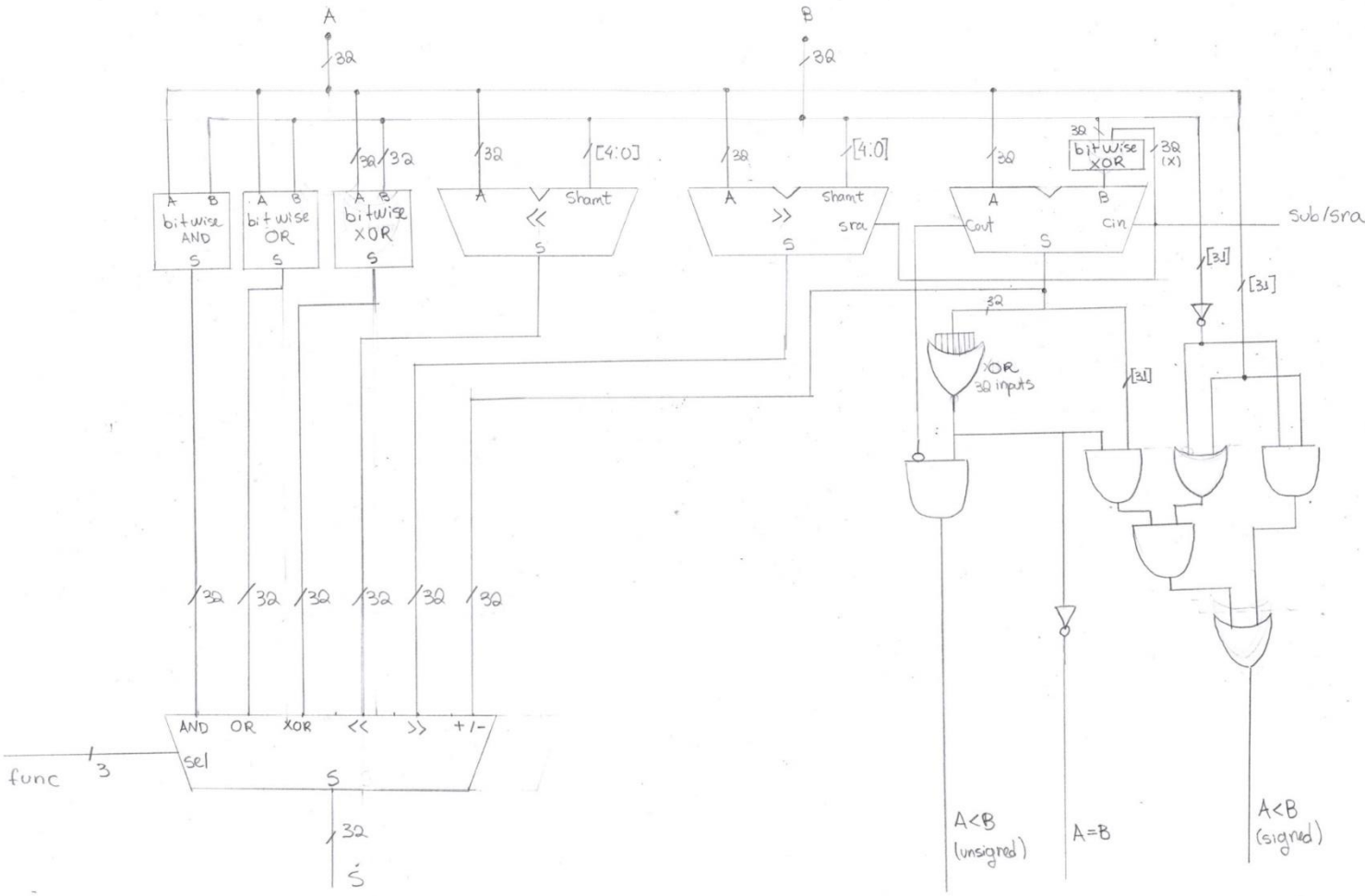
rs2_mem_i: Valor lido do rs2 que será armazenado na memória.

Data_i: Valor de 32bits que está entrando (input) na memória.

Data_o: Valor de 32bits que está saindo (output) da memória.

3.3 ALU

A ALU é o circuito responsável por fazer operações lógico aritméticas. A ALU do RV-32I é capaz de fazer as seguintes operações: add, subtract, shift left logical, shift right logical, shift right arithmetic, bitwise xor, bitwise and, bitwise or, equal, less than signed e less than unsigned. Um diagrama de blocos do circuito é apresentado a seguir:



Ela possui 4 entradas e 4 saídas, sendo elas:

ENTRADAS

- A – Uma entrada de 32 bits que pode ser tanto o valor do RS1 quanto o valor guardado no PC.
- B – Uma entrada de 32 bits que pode ser tanto o valor do RS2 quanto um imediato.

- Sub/sra – Bit 30 da instrução armazenada no INS_REG, responsável por indicar se será feita uma operação de subtração ou por indicar se será feita uma operação de shift aritmético para direita.
- Func – Sinal de 3 bits recebido da C.U. que selecionará qual será a saída, dentre todas os resultados possíveis das operações.

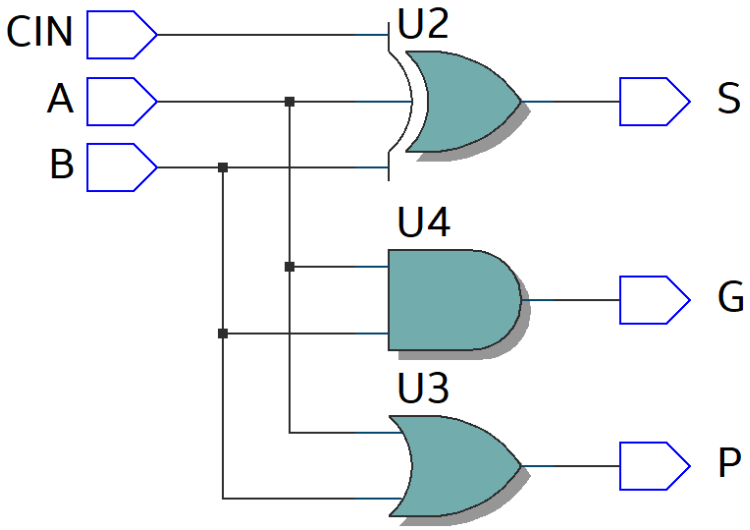
SAÍDAS

- Alu_val – Saída determinada pelo Func, pode ser o resultado de alguma das seguintes operações: Soma/subtração, AND, XOR, OR, shift left ou shift right.
- EQ – Saída que será 1 caso A = B.
- A < B – Saída que será 1 caso A < B (signed).
- A < UB – Saída que será 1 caso A < B (unsigned).

3.3.1 Adder de 32 bits carry look-ahead

A parte principal da ALU é um somador de 32bits que soma A com B ou com -B. Ele é formado por 4 somadores Carry Look-Ahead de 8 bits, ligados entre si na configuração Ripple Carry, que possuem sub/sra, A, B e CIN como entrada e S e COUT como saída. Esse circuito basicamente faz A + B, direciona o resultado na saída S e oferece o carry out da soma no COUT.

Cada somador Carry Look-Ahead é formado por 8 Partial Full Adders, os quais recebem números de 1 bit, A, B e carry in (CIN), e tem como saída a sua soma S sem carry out, o sinal propagate P e o sinal generate G. O propagate serve para verificar se A e B propagam um carry, sinalizando 1 caso propague, 0 caso contrário. O generate, por sua vez, verifica se A e B geram um carry, sinalizando 1 caso gere e 0 caso contrário. O circuito desse somador é mostrado a seguir:



O módulo do Carry Look-Ahead de 8bits utiliza as saídas G e P do Partial Full Adder para fazer o circuito look-ahead, o qual calcula os carries fazendo

$$Ci = Gi + Pi*Ci-1$$

sendo:

Gi = Ai * Bi (generate, verifica se Ai e Bi geram um carry)

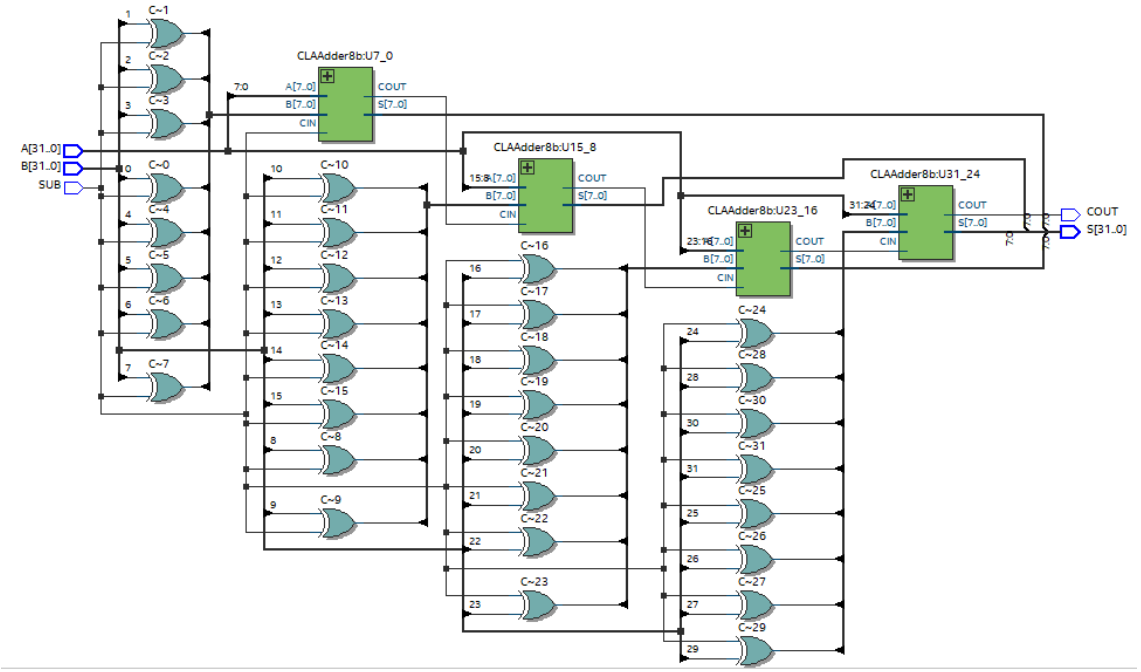
Pi = Ai + Bi (propagate, verifica se Ai e Bi propagam um carry)

Ci-1 = carry anterior

Cada carry i é somado com os bits Ai e Bi no somador parcial de um bit (PartialFullAdder1b). Exemplo do código que descreve o circuito que calcula o primeiro carry:

```
// C1 = G1 + P1*C0 (C0 = CIN)
and P01 (PC1,P0,CIN);
or C01 (C1, G1, PC1);
```

Desta forma, obtém-se um somador de números de 8bits. Juntando 4 deles em um esquema ripple-carry, no qual o carry out de um somador é o carry in do próximo, obtemos o somador de 32bits da ALU. No carry in o primeiro somador de 8 bits, é conectado o sinal sub_sra, o qual, se for 1, auxiliará no complemento de 2 do número B, já somando 1. Além disso, o sinal sub_sra, se for 1, complementa o valor de B através de um bitwise xor entre ele e cada bit de B. O esquema do circuito completo é apresentado a seguir:

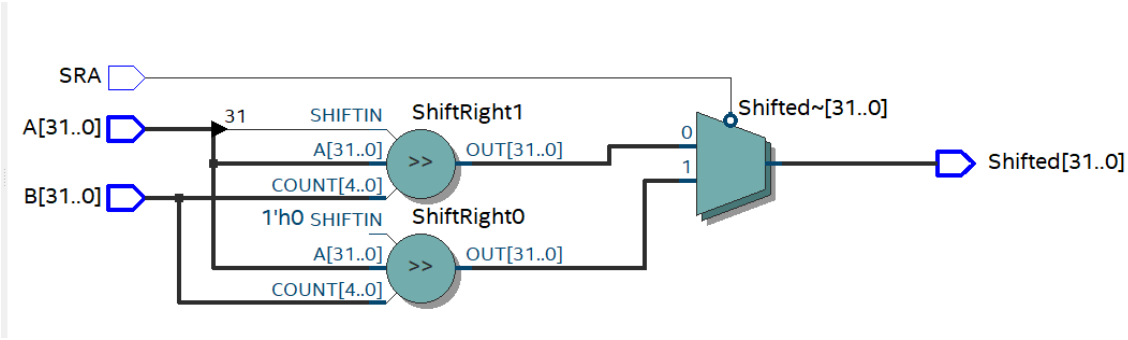


3.3.2 Shifters

Existem dois tipos de operação shift, logical ou arithmetic, e duas direções, left ou right. No logical shift, tanto left como right, os bits da palavra são shiftados na direção especificada e 0 são colocados nos espaços que são gerados. No arithmetic shift, o qual só tem na direção right, o bit mais significativo da palavra é copiado no espaço vazio gerado, preservando o sinal do valor numérico representado por aqueles bits da palavra.

Como no RV-32I há somente valores inteiros de 32 bits, o máximo que um número pode ser shiftado em qualquer direção é 32 bits, valor que é representado por 5 bits. Assim nos módulos de shift presentes na ALU, o valor A é shiftado em uma quantidade especificada pelos 5 lsb de B. Como somente há logical shift na direção left, há um módulo especifico que faz essa operação (observar no diagrama da ALU). Já para a direção right, novamente o sinal sub_sra aparece para especificar a operação. Será do tipo arithmetic e for 1, se for 0 será logical.

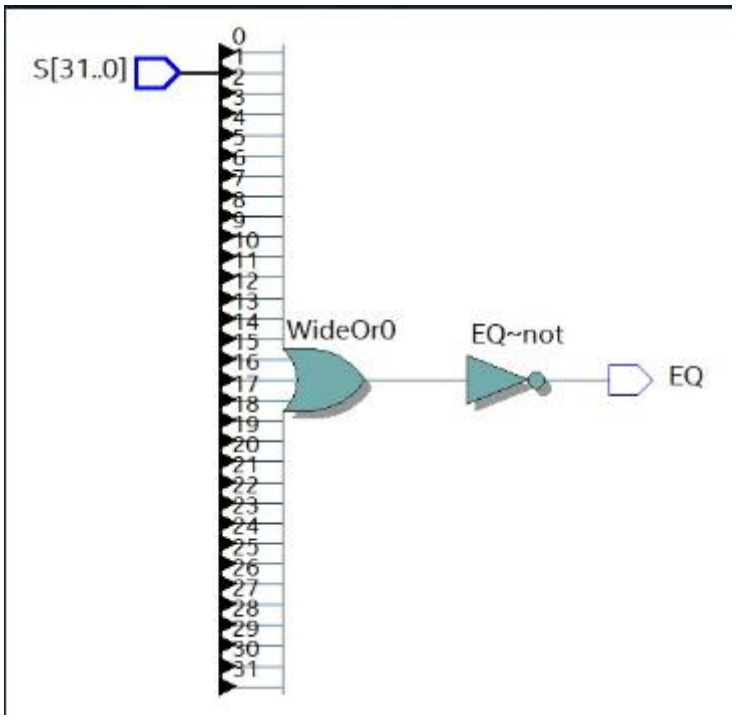
OBS: No projeto desenvolvido, não foram planejados circuitos mais profundos para os módulos de shift, ou seja, apenas foi adotado o que o verilog sintetizou/interpretou da descrição feita usando as operações padrões (<<, >> e >>>). Circuito do right shifter é apresentado a seguir junto com parte código de sua descrição, mostrando o resultado dessa abordagem:



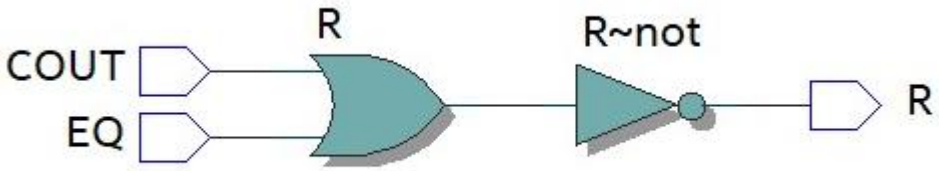

```
always @ (A, B, SRA)
    if (SRA == 0)
        Shifted = A >> B[4:0];
    else
        Shifted = A >>> B[4:0];
```

3.3.3 Comparadores

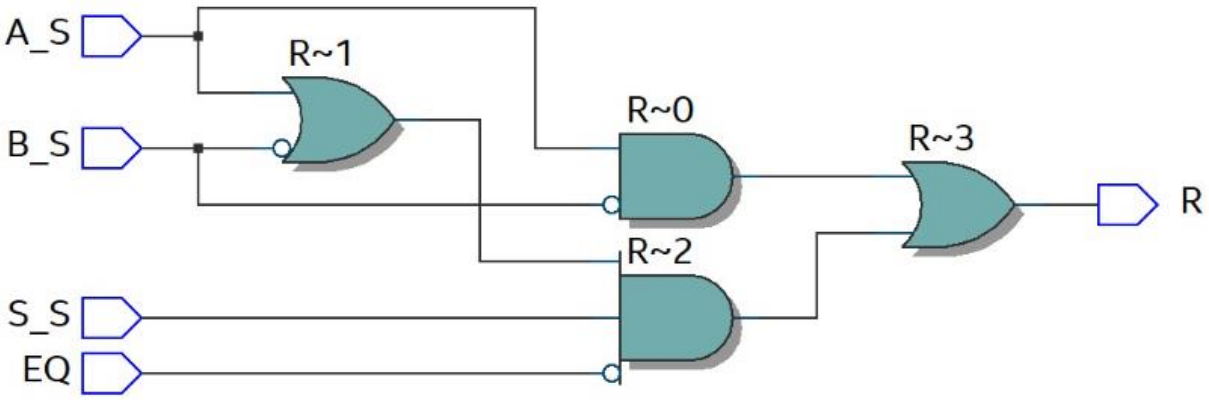
Para a realização das operações de comparações, poderia ter sido utilizado um circuito comparador próprio, entretanto, para poupar circuito, o comparador foi feito se aproveitando da operação de soma e subtração já presente na ALU. Para realizar a comparação de igual entre A e B, basta subtrair B de A e verificar se o resultado é igual a 0, caso ele seja, então A e B são iguais. Para realizar esta verificação, basta realizar OR entre todas entradas, caso seja 1, indica que o resultado foi diferente de 0, e que os números são diferentes.



Para realizarmos a operação menor que unsigned, novamente, basta subtrair B de A, mas desta vez, basta verificar se houve carry-out ou não, já que o carry só é gerado caso a soma de A com o complemento de 2 de B, que unsigned é o 2^32 – B, exceder 2^32, que só ocorre quando A > B, já que 2^32 -B + A > 2^32 ⇒ A > B. Desta forma, basta verificar o carry-out juntamente com o sinal de igualdade, já que a comparação é apenas menor que:



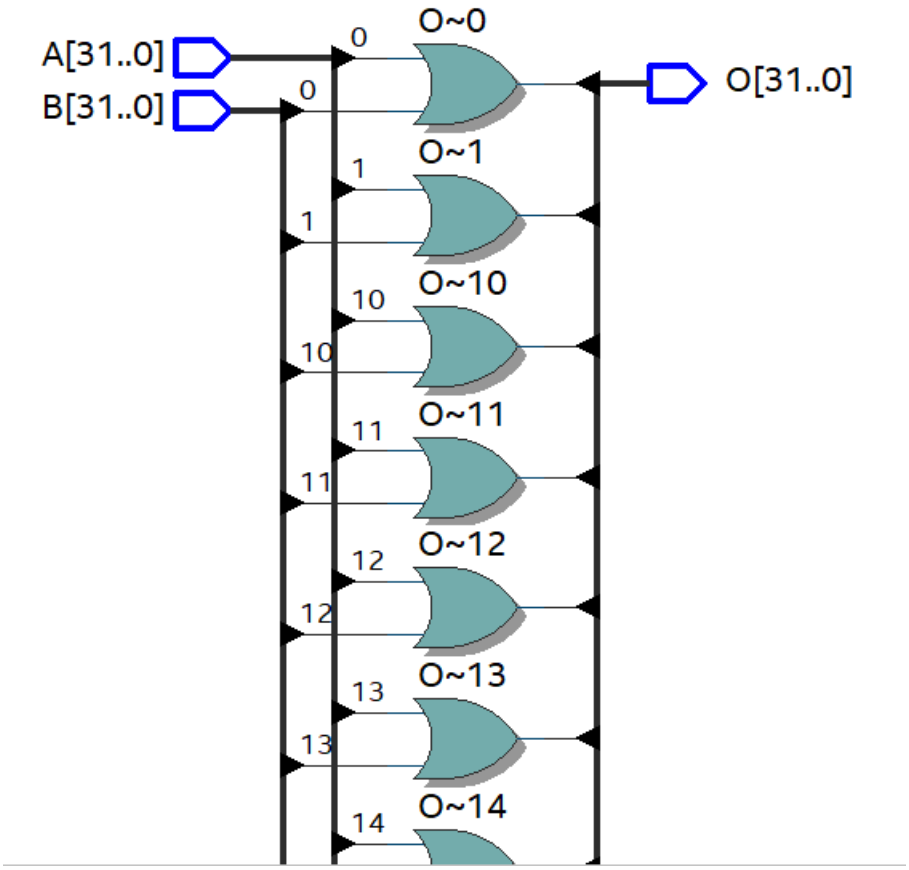
Para a comparação signed, iniciamos verificando o bit mais significativo, responsável pelo sinal, caso o sinal seja diferente, a comparação já está feita, caso os sinais sejam os mesmos, basta realizar uma subtração entre A e B, e verificar o sinal, por exemplo, caso ambos forem negativos, e a subtração A – B der positivo, implica que B é necessariamente maior que A. Então, para realizar a comparação, precisamos apenas do bit de sinal do primeiro número, do segundo número e do resultado e utilizando uma tabela verdade, é possível desenvolver o seguinte circuito:



3.3.4 Operações lógicas

Os módulos de operações são simples operações de AND, OR ou XOR entre cada bit correspondente das entradas A e B, ambas de 32bits. Cada bloco deles,

apresentado no diagrama da ALU, representa um conjunto de 32 portas lógicas que fazem suas operações, uma para cada bit. Por curiosidade, a seguir é exposto parte do interior do módulo do OR de 32bits.



3.3.5 Seleção da saída

Cada bloco mencionado anteriormente gera uma saída de forma independente, bastando receber valores A e B para que eles realizem suas operações. Na saída da ALU, porém, somente alguns resultados são selecionados. Por padrão, os resultados dos comparadores (EQ, LS e LU) são saídas diretas, sempre ativas. Já os outros dependem de qual operação foi requisitada pela instrução.

Dentro da ALU, um conjunto de MUXes selecionam qual resultado será passado na saída alu_val da ALU. O seletor é o um input do módulo da ALU chamado func, o qual os sinais são:

- Func = 000: Seleciona saída do Adder de 32bits.
- Func = 001: Seleciona saída do left shifter.
- Func = 010: Seleciona saída do comparador LS.
- Func = 011: Seleciona saída do comparador LU.

- Func = 100: Seleciona saída do bitwise XOR.
- Func = 101: Seleciona saída do right shifter.
- Func = 110: Seleciona saída do bitwise OR.
- Func = 111: Seleciona saída do bitwise AND.

Assim, de acordo com o func, alu_val recebe um valor diferente, o qual é requisitado pela instrução. A parte principal descrição desse circuito é apresentada a seguir.

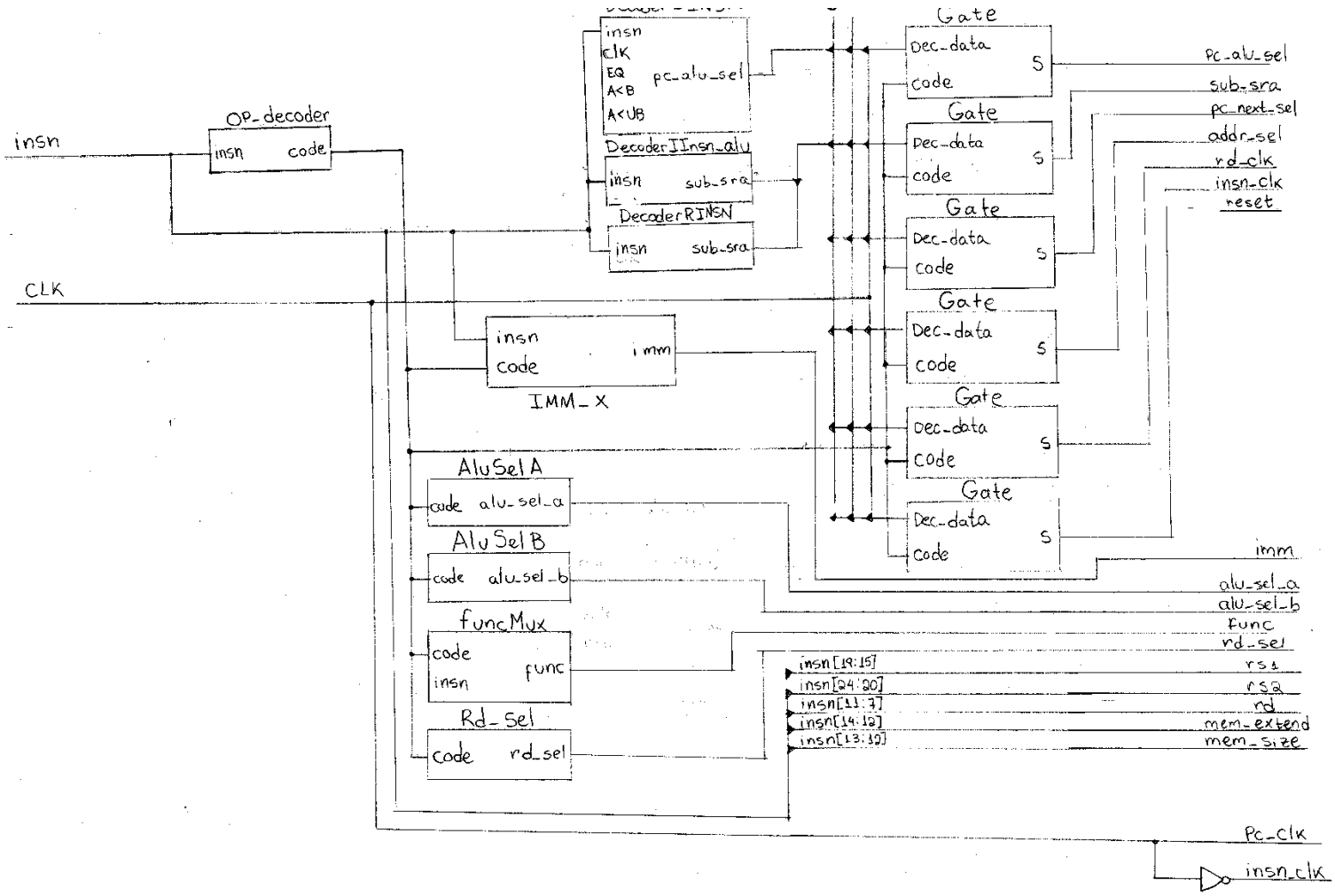
```
// Bloco always para síntese de multiplexador para selecionar as saídas
always @(*) begin
  // Case baseado nas instruções presentes na ISA, para poupar o máximo da C.U.
  case (func)
    3'b000: alu_val = ADD;
    3'b001: alu_val = SL;
    3'b010: alu_val = LS;
    3'b011: alu_val = LU;
    3'b100: alu_val = bXOR;
    3'b101: alu_val = SR;
    3'b110: alu_val = bOR;
    3'b111: alu_val = bAND;
  endcase
end
```

3.3.6 Relacionamento com outros componentes

A ALU recebe seus dois inputs de 32bits, A e B, de dois MUXes. Um deles seleciona entre o valor do rs1 e PC, enquanto o outro seleciona a entrada entre rs2 e o imm. Os outros dois inputs do circuito vem diretamente da CU, sendo sub_sra um sinal de seleciona operações a serem feitas e o func um sinal que seleciona a saída do módulo.

O valor que sai da ALU pode ter destinos diferentes, dependendo da instrução. Por um caminho, ele pode ir para um MUX que seleciona qual valor será registrado no rd, podendo ser armazenado nele. Por outro caminho, o resultado da ALU pode ir para um MUX que decide qual será o próximo valor do PC, podendo mudar seu valor e fazer um jump. Uma outra última opção é o caminho que leva a um MUX que seleciona de onde vira o próximo endereço a ser visto na memória, podendo servir como um addr para uma função de store. Além disso, os sinais advindos dos comparadores vão para CU e são utilizados para decidir se é realizado ou não um Branch em uma instrução do tipo B.

3.4 CONTROL UNIT



A Control Unit possui 5 entradas e 18 saídas. As suas entradas são o sinal clk, uma instrução de 32bits e os sinais EQ, LS e LU provenientes da ALU, os quais carregam resultados das comparações. As saídas são: rd_clk, mem_clk, pc_clk, insn_clk, addr_sel, alu_sel_a, alu_sel_b, pc_next_sel, pc_alu_sel, sub_sra, imm, mem_size, mem_extend, func, rd_sel, rs1, rs2 e rd. A explicação do significado e função de cada sinal se encontra na seção 3.2 desse documento.

A finalidade principal desse bloco é gerar sinais corretos de acordo com a instrução recebida e, quando for necessário, dos resultados da ALU, sincronizando todo o processo por meio do sinal clk. Seu funcionamento interno se divide em: decodificar o opcode, obter o imediato, gerar sinais derivados do clk, decodificar a insn e gerar sinais que dependem do seu tipo e, por fim, gerar sinais que não dependem do tipo de insn.

3.4.1 Decodificação do opcode

Existem ao todo 10 opcodes diferentes, um para cada tipo de instrução. Para descobrir quais sinais devem ser emitidos, é necessário saber primeiramente o que a instrução quer que o processador faça. Desta forma, identificar o opcode é essencial para que a CU prossiga. Ele sempre se encontra nos 7 lsb da insn, sendo que o ideal seria decodificar esses 7 bits em um código (chamado de “code”) de 10bits, por exemplo, no seguinte formato:

- OPCODE: 1101111 --> Code = 10'b0000000001; // Instrução tipo J
- OPCODE: 1100111 --> Code = 10'b0000000010; // Instrução tipo I (Jarl)
- OPCODE: 0110111 --> Code = 10'b0000000100; // Instrução tipo U (LUI)
- OPCODE: 0010111 --> Code = 10'b0000001000; // Instrução tipo U (AUIPC)
- OPCODE: 1100011 --> Code = 10'b0000010000; // Instrução tipo B
- OPCODE: 0110011 --> Code = 10'b0000100000; // Instrução tipo R
- OPCODE: 0100011 --> Code = 10'b0001000000; // Instrução tipo S
- OPCODE: 0010011 --> Code = 10'b0010000000; // Instrução tipo I (ALU)
- OPCODE: 0000011 --> Code = 10'b0100000000; // Instrução tipo I (LOAD)
- OPCODE: 1110011 --> code = 10'b1000000000; // Instrução tipo I (CSR)

Em que cada bit é 0 com exceção de um, o qual a sua posição indica de qual tipo de instrução aquele opcode é. Nesse caso, um decoder padrão para os 7 bits de entrada seria um circuito gigantesco e impraticável. Por isso, para aplicação nesse processador, foi desenvolvido um decoder especial para o opcode, chamado criativamente de OPDecoder. Nele, a entrada é a insn e a saída é um código de 10 bits, o code. O "algoritmo" de decodificação é descrito abaixo:

Cada bit de saída de code é gerada por uma porta AND, a qual a finalidade é gerar 1 apenas quando a entrada tiver o opcode que corresponde a sua posição (verificar formato da codificação acima). Por exemplo, o AND do décimo bit de saída só gera 1 quando a entrada for o opcode 1110011, da insn do tipo CSR. O processo foi feito identificando/separando os OPCODE em grupos com certos padrões, até que ao final da separação cada grupo tivesse apenas um elemento.

Quando um opcode é identificado/separado em um grupo, todos os AND de saída dos outros grupos recebem 0, portanto geram 0. Ao final, sobrará somente um grupo com um opcode, o qual gerará o sinal 1 (com os demais gerando 0).

1. Primeiramente, os OPCODE são separados em dois grupos: os que o AND dos 3 LSB resulta em 1 e os que resulta em 0.

- Grupo 1 (resulta em 1): 1101111, 1100111, 0110111 e 0010111.
- Grupo 2 (resulta em 0): 1100011, 0110011, 0100011, 0010011, 0000011 e 1110011.

2. Após isso, eles são separados em outros dois grupos: os que o XOR entre os bits 6,5 e 3 de INSN resultam em 1 e os que resultam em 0.

- Grupo 1 (resulta em 1): 1101111, 0110111, 0110011, 0100011.
- Grupo 2 (resulta em 0): 1100111, 0010111, 1100011, 0010011, 0000011, 1110011.

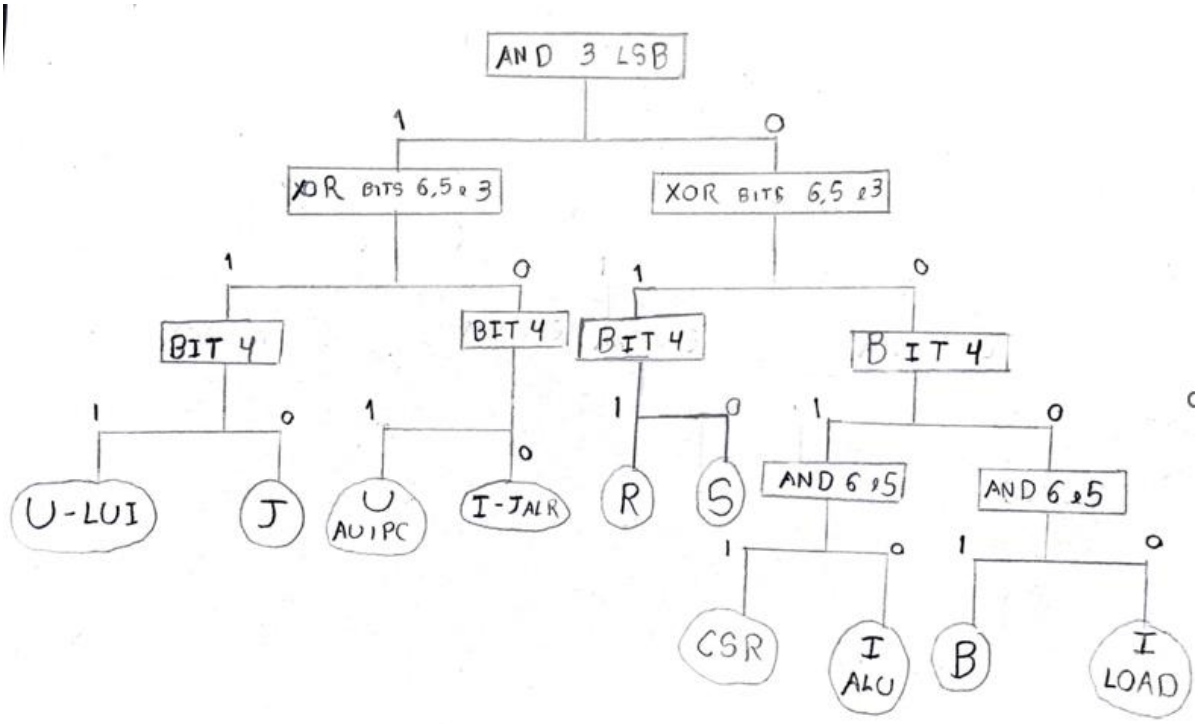
3. Seguindo, são separados em mais dois grupos: os que o AND entre os bits 6 e 5 de INSN resulta em 1 e os que resulta em 0;

- Grupo 1 (resulta em 1): 1101111, 1100111, 1100011, 1110011,
- Grupo 2 (resulta em 0): 0110111, 0110011, 0100011, 0010111, 0010011, 0000011.

4. Por fim, separados nos grupos: os que o bit 4 de INSN é 1 e os que é 0.

- Grupo 1 (é 1): 1110011, 0110111, 0110011, 0010111, 0010011.
- Grupo 2 (é 0): 1101111, 1100111, 1100011, 0100011, 0000011.

Esses “grupos” foram escolhidos arbitrariamente. A ideia por trás de sua criação era estabelecer critérios que separassem os opcodes, utilizando operações lógicas, até que cada um deles pudessem ser unicamente identificados. Abaixo tem um diagrama de uma “árvore” em que cada nó é uma operação lógica e que possui como folhas as instruções/opcodes.



Exemplo:

OPCODE: 1110011 - Instrução tipo I (CSR)

1. Entra no grupo 2. Grupo atual: 1100011, 0110011, 0100011, 0010011, 0000011 e 1110011.

2. Se enquadra no grupo 2. Grupo atual: 1100011, 0010011, 0000011, 1110011.

3. Se enquadra no grupo 1. Grupo atual: 1100011, 1110011.

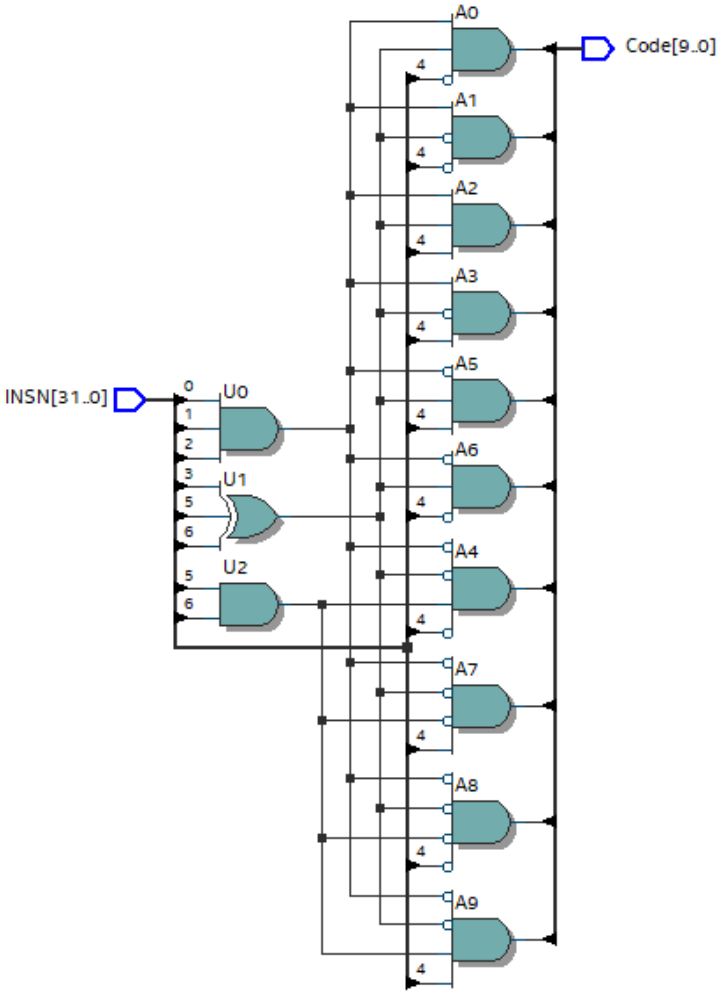
4. Se enquadra no grupo 1. Grupo atual: 1110011.

Assim, como toda vez que um opcode "entra em grupo" os AND de saída dos OPCODE de outros grupos geram 0, temos que o único que gerará 1 será o AND de saída do OPCODE da instrução tipo I (CSR).

Utilizando apenas 4 operações, foi possível separar o conjunto de códigos em 10 unitários, gerando um código que identifica facilmente o tipo de instrução. Por ter poucas operações, o circuito resultante é simples e melhor que um decoder genérico para o caso em questão. Essa solução, porém, é extremamente especifica para a ISA do RV-32I, sendo que para extensões seria necessário adicionar mais operações à arvore.

OBS: Vale destacar que, para extensões que adicionariam mais opcodes, não seriam necessárias adicionar muitas operações, pois a quantidade de folhas de uma árvore desse tipo dobra a cada nível de profundidade.

O circuito do OPDecoder é apresentado a seguir:



3.4.2 Obtenção do imediato

A codificação do imediato, ao contrário do rd por exemplo, não possui uma posição fixa na palavra da instrução. Assim, para cada tipo de instrução que contém um imm, os bits que correspondem ao imediato mudam de lugar.

Tipo J:

Instrução

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	rd					1	1	0	1	1	1	1

Imediato:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
A	A	A	A	A	A	A	A	A	A	A	A	M	N	O	P	Q	R	S	T	L	B	C	D	E	F	G	H	I	J	K	0

Tipo I (JALR, ALU e LOAD):

Instrução

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00

A	B	C	D	E	F	G	H	I	J	K	L	rs1	func	rd	0	0	1	0	0	1	1

Imediato:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	B	C	D	E	F	G	H	I	J	K	L

Tipo U (LUI e AUIPC):

Instrução:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	rd					0	1	1	0	1	1	1

Imediato:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	0	0	0	0	0	0	0	0	0	0	0	0

Tipo B:

Instrução:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
A	B	C	D	E	F	G	rs2					rs1					func			H	I	J	K	L	1	1	0	0	0	1	1

Imediato:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	L	B	C	D	E	F	G	H	I	J	K	0

Tipo S e CSR:

Instrução:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
A	B	C	D	E	F	G														H	I	J	K	L							

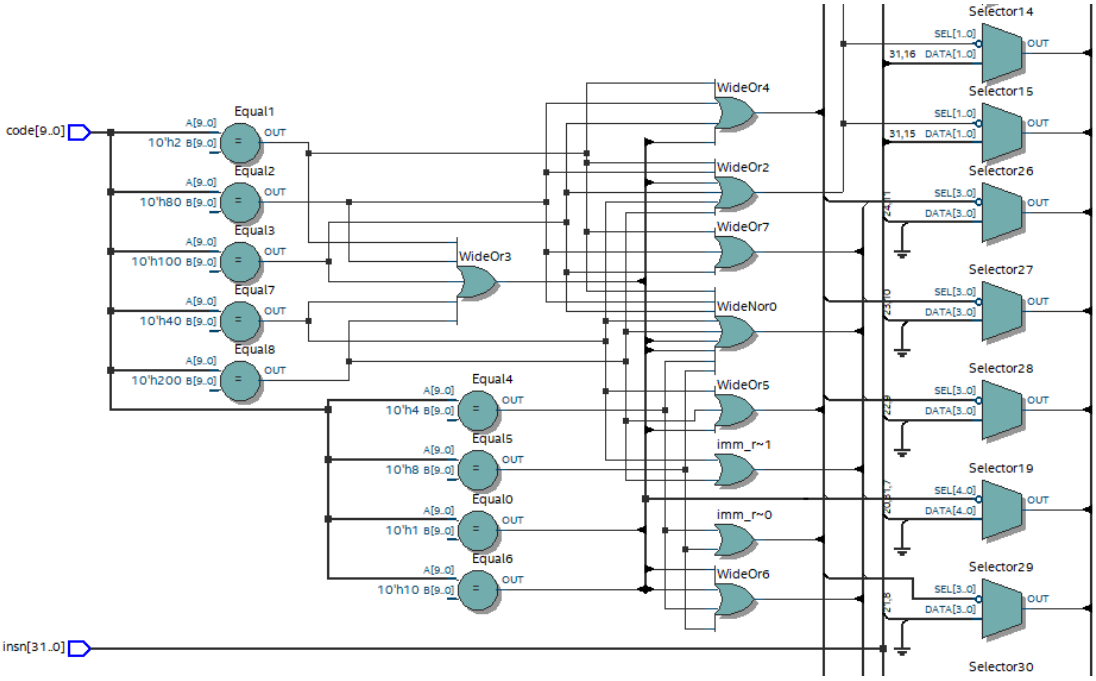
Imediato:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	B	C	D	E	F	G	H	I	J	K	L

A implementação da decodificação do imediato, assim como nos shifters, não se aprofundou em circuitos mais específicos. Ele recebe a insn, o código code e gera o sinal com valor do imediato na saída. O circuito foi descrito por meio de um case e, na síntese, gerou alguns comparadores de igualdade, além de uma série de MUXes.

Por não apresentar complexidade absurda, essa solução foi adotada. A seguir é mostrado parte principal do código e parte do circuito sintetizado.

```
/* Para cada code, monta o imediato concatenando bits específicos da instrução de acordo com a ISA. */
always @(insn) begin
    case (code)
        10'b0000000001: imm_r = {{12{insn[31]}}, insn[19:12], insn[20], insn[30:21], 1'b0};
        10'b0000000010, 10'b0010000000, 10'b0100000000: imm_r = {{20{insn[31]}}, insn[31:20]};
        10'b0000000100, 10'b0000001000: imm_r = {insn[31:12], 12'b0};
        10'b0000010000: imm_r = {{20{insn[31]}}, insn[7], insn[30:25], insn[11:8], 1'b0};
        10'b0001000000, 10'b1000000000: imm_r = {{20{insn[31]}}, insn[31:25], insn[11:7]};
        default: imm_r = 32'hxxxxxxxx;
    endcase
end
```



3.4.3 Geração do clock e seus derivados

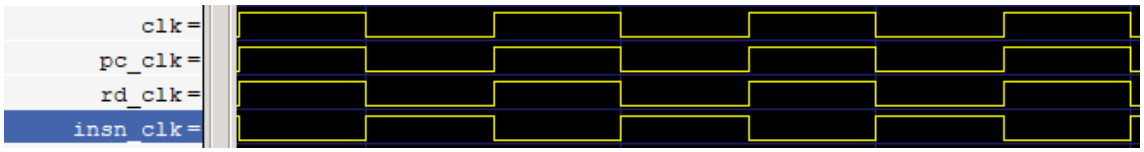
O sinal de clk é gerado por módulo com parte principal descrita da seguinte forma:

```
always
    #delay clk = ~clk;
```

Em que o delay é o período entre duas edges do clock. Esse sinal é um dos inputs da CU, do qual são derivados outros clks: mem_clk, rd_clk, pc_clk e insn_clk. A explicação da função desses clocks se encontra na seção 3.2 desse documento.

Para o funcionamento do processador, o insn_clk precisa ter uma borda de subida antes que os demais, pois primeiro se obtem a instrução e executa as operações e somente depois se registram os resultados. Assim, os clks do pc, rd e da

mem precisam ter a borda de subida um tempo depois da subida do insn_clk, tempo necessário para executar o que foi pedido. A solução adotada foi fazer o insn_clk ser igual ao clk invertido, enquanto os outros sinais, quando necessário, possuem sinais iguais ao clk. O diagrama a seguir revela melhor o comportamento dos sinais:



Desta forma, uma instrução sempre é lida da memória um período “#delay” antes dos clks rd e pc registrarem os resultados. Vale ressaltar que o rd_clk e o mem_clk não estão sempre oscilando conforme o clk. O rd_clk permanece sempre em 0 em instruções que não armazenam valores em um rd e o mem_clk permanece em 0 em qualquer instrução que não seja de store. Já os insn_clk e o pc_clk são sinais sempre periódicos.

```
assign insn_clk = ~clk;
assign pc_clk = clk;
```

OBS: A maneira como a CU decide os valores dos sinais mem_clk e rd_clk irá ser melhor explicada na seção 3.4.5 “Sinais que dependem da instrução”.

3.4.4 Sinais que não dependem do tipo da instrução

Alguns sinais podem ser gerados independentemente de qual instrução se trata. Eles são retirados diretamente da insn ou do clk, sendo que os da insn se referem àqueles que possuem um campo fixo na palavra: rd, rs1, rs2, mem_extend e mem_size. Esses sinais saem direto da CU e vão para os outros blocos do processador, sem passar por outro circuito interno. Verificar a seção 2.3 “ISA do RISC-V 32I” desse documento para entendê-los melhor. Os detalhes sobre os clk se encontram na seção 3.4.3.

OBS: mem_extend e mem_size são retirados do campo func. Seus significados e funções são explicados na seção 3.2.

```
// Sinais que não dependem do opcode, podem ser extraídos diretamente da instrução
assign rs1 = insn[19:15];
assign rs2 = insn[24:20];
assign rd = insn[11:7];
assign mem_extend = insn[14:12];
assign mem_size = insn[13:12];
assign insn_clk = ~clk;
assign pc_clk = clk;
```

Vale destacar que, mesmo se uma instrução não tiver um campo para um dos sinais, a CU ainda o geraria. Um exemplo são as instruções do tipo B, as quais não possuem um campo para rd, porém a CU continuaria gerando esse sinal. Entretanto, não há problemas com isso, pois seletores e clks garantem que apenas os valores corretos estão sendo passados as e operações certas estão sendo feitas.

No exemplo citado, o sinal rd_clk, permanecendo sempre em 0, garantiria que nada iria ser gravado no endereço dado pelo rd. Numa instrução de store, o mem_clk (sempre em 0) e addr_sel fariam o mem_size não ter significância. Alu_sel_a e alu_sel_b evitariam que rs1 e rs2 enviassem valores errados para ALU. Por fim, o rd_sel (junto do rd_clk) não permitiria que o mem_extend tivesse algum significado ou interferisse em algo.

3.4.5 Sinais que dependem da instrução

Boa parte dos sinais assumem valores que dependem diretamente do tipo de instrução dada. Eles fazem o maior trabalho de controle do processador, selecionando saídas de MUXes, selecionando operações ou controlando armazenamentos. Eles são: addr_sel, alu_sel_a, alu_sel_b, pc_next_sel, pc_alu_sel, rd_clk, mem_clk, sub_sra, func e rd_sel. A seguir, especificaremos o comportamento deles para cada tipo de instrução, com exceção do alu_sel_a, alu_sel_b, rd_sel e func, os quais serão melhor detalhados na seção 3.4.5.2 “Gates”.

Tipo U – LUI

- sub_sra = 1'bx; // não importa pois não é realizada operação aritmética
- addr_sel = 0; // O endereço da memória continua sendo o program counter
- pc_next_sel = 0; // O PC continua recebendo o seu valor incrementado em 4
- pc_alu_sel = 0; // A ALU do PC contínua recebendo 4

- mem_clk = 0; // É setado para 0, para evitar problemas com gravação de memória.
- rd_clk = CLK; // O rd_clk é igual ao CLK do processador, que ao subir grava o valor.

Tipo U – AUIPC

- sub_sra = 0; // deve ser 0, pois é realizada uma soma.
- addr_sel = 0; // O endereço da memória continua sendo o program counter.
- pc_next_sel = 0; // O PC continua recebendo o seu valor incrementado em 4.
- pc_alu_sel = 0; // A ALU do PC contínua recebendo 4.
- mem_clk = 0; // É setado para 0, para evitar problemas com gravação de memória.
- rd_clk = CLK; // O rd_clk é igual ao CLK do processador, que ao subir grava o valor.

Tipo J

- sub_sra = 0; // sempre deverá ser realizada uma soma.
- addr_sel = 0; // O endereço da memória continua sendo o program counter.
- pc_next_sel = ~CLK; // O PC continua recebendo o seu valor incrementado em 4.
- pc_alu_sel = CLK; // A ALU do PC recebe valor imediato.
- mem_clk = 0; // É setado para 0, para evitar problemas com gravação de memória.
- rd_clk = CLK; // O rd_clk é igual ao CLK do processador, que ao subir grava o valor.

Tipo I – JALR

- sub_sra = 0; // como haverá uma soma, devemos setar sub_sra com 0.
- addr_sel = 0; // o endereço de memória deve continuar recebendo o pc.
- pc_next_sel = 1; // O PC continua deve receber o valor provindo da alu.
- pc_alu_sel = 0; // A ALU do PC contínua recebendo 4
- mem_clk = 0; // É setado para 0, para evitar problemas com gravação de memória.

- rd_clk = CLK; // O rd_clk é igual ao CLK do processador, que ao subir grava o valor.

Tipo B

- addr_sel = 0; // o endereço de memória deve continuar recebendo o pc.
- sub_sra = 1'b1; // como todas as operações de branches necessitam de comparação, e os comparadores foram feitos se aproveitando da subtração, sub_sra deverá sempre ser 1.
- pc_next_sel = 0; // O PC continua recebendo o seu valor incrementado em 4.
- mem_clk = 0; // É setado para 0, para evitar problemas com gravação de memória.
- rd_clk = 0; // O rd_clk é igual a 0, já que não deve ocorrer nenhuma gravação em registrador.

Tipo I – LOAD

- sub_sra = 1'b0; // Deve ser zero, já que queremos soma sempre.
- addr_sel = ~CLK; // O endereço de memória deve iniciar com o resultado da alu, e em seguida deve retornar para o pc, para prosseguir o programa, o que alinha com o clock do processador.
- pc_next_sel = 0; // O PC continua recebendo o seu valor incrementado em 4.
- pc_alu_sel = 0; // A ALU do PC contínua recebendo 4.
- rd_clk = CLK; // O rd_clk é igual ao CLK do processador, que ao subir grava o valor.
- mem_clk = 0; // É setado para 0, para evitar problemas com gravação de memória.

Tipo S

- addr_sel = ~CLK; // o endereço de memória deve receber o resultado da alu inicialmente, entretanto em seguida deve voltar para o pc, o que coincide com o sinal de clock invertido do processador.
- sub_sra = 0; // como haverá soma, sempre será 0.
- pc_next_sel = 0; // O PC continua recebendo o seu valor incrementado em 4.
- pc_alu_sel = 0; // A ALU do PC contínua recebendo 4.

- rd_clk = 0; // como o rd não é utilizado, deve ser deixado em 0 para evitar problemas de gravação.
- mem_clk = CLK; // Deve ter rising edge quando o addr_sel tem falling edge, coincidindo com o sinal de clock.

Tipo I – ALU

- addr_sel = 0; // o endereço de memória deve continuar recebendo o pc.
- sub_sra = (~func[2] & func[1]) | (func[2] & ~func[1] & func[0] & insn[30]); // a subtração deve existir nos casos que precisamos de funções de comparação, como slti e sltiu, e no shift direito quando o segundo bit mais significativo for 1.
- pc_next_sel = 0; // O PC continua recebendo o seu valor incrementado em 4.
- pc_alu_sel = 0; // A ALU do PC contínua recebendo 4.

- rd_clk = CLK; // O rd_clk é igual ao CLK do processador, que ao subir grava o valor.
- mem_clk = 0; // É setado para 0, para evitar problemas com gravação de memória.

Tipo R

- sub_sra = ((~insn[14]) & insn [13]) | insn[30]; // Está codificada no segundo bit mais significativo da instrução, e também deverá ser 1 caso a instrução necessite de uma comparação.
- addr_sel = 0; // O endereço da memória continua sendo o program counter.
- pc_next_sel = 0; // O PC continua recebendo o seu valor incrementado em 4.
- pc_alu_sel = 0; // A ALU do PC contínua recebendo 4.
- rd_clk = CLK; // O rd_clk é igual ao CLK do processador, que ao subir grava o valor.
- mem_clk = 0; // É setado para 0, para evitar problemas com gravação de memória.

3.4.5.1 Instruction Decoders

Para a maioria das instruções, os todos os sinais são valores fixos ou algum derivado do clk. Como por exemplo, para instrução do tipo J, addr_sel sempre é 0 e o rd_clk sempre é igual ao clk. Para 3 tipos, entretanto, há sinais que variam de acordo com a instrução, precisando ser decodificados de forma especial. Eles são: Tipo B, R e I – ALU.

Tipo B:

Os branchs precisam decidir se incrementam ou não o valor do PC com base nos resultados da ALU. Assim, o sinal pc_alu_sel não pode ser fixo, pois depende do EQ, LS ou LU. Dentro de um módulo especial para decodificação desse sinal, o sinal pc_alu_sel é escolhido com base no tipo de Branch, determinado pelo func:

```
always @(insn, EQ, LS, LU) begin
  /* realizamos as comparações, e baseado no opcode, ou a alu do pc recebe 4 normalmente e
  não realiza um branch, ou recebe o valor imediato e avança/retorna para algum endereço */
  case (func)
    3'b000: pc_alu_sel = (EQ);
    3'b001: pc_alu_sel = (~EQ);
    3'b100: pc_alu_sel = (LS);
    3'b101: pc_alu_sel = (~LS);
    3'b110: pc_alu_sel = (LU);
    3'b111: pc_alu_sel = (~LU);
    default: pc_alu_sel = 1'bx;
  endcase
end
```

OBS: Verificar seção 2.3 “ISA do RISC-V 32I” para conhecer o significado de cada valor de func para a instrução do tipo B.

Tipo R:

Algumas instruções do tipo R, como visto na seção 2.3, possuem um parâmetro no bit 30 da insn, o qual seleciona qual operação será feita pela ALU. Como ele não é fixo e não são todas as instruções do tipo R que o utilizam, foi feito um circuito que emite o sinal corretamente para as funções necessárias:

```
assign sub_sra = ((~insn[14]) & insn[13]) | insn[30];
/* Está codificada no segundo bit mais signifcativo da instrução,
e também deverá ser 1 caso a instrução necessite de uma comparação */
```

Tipo I – ALU

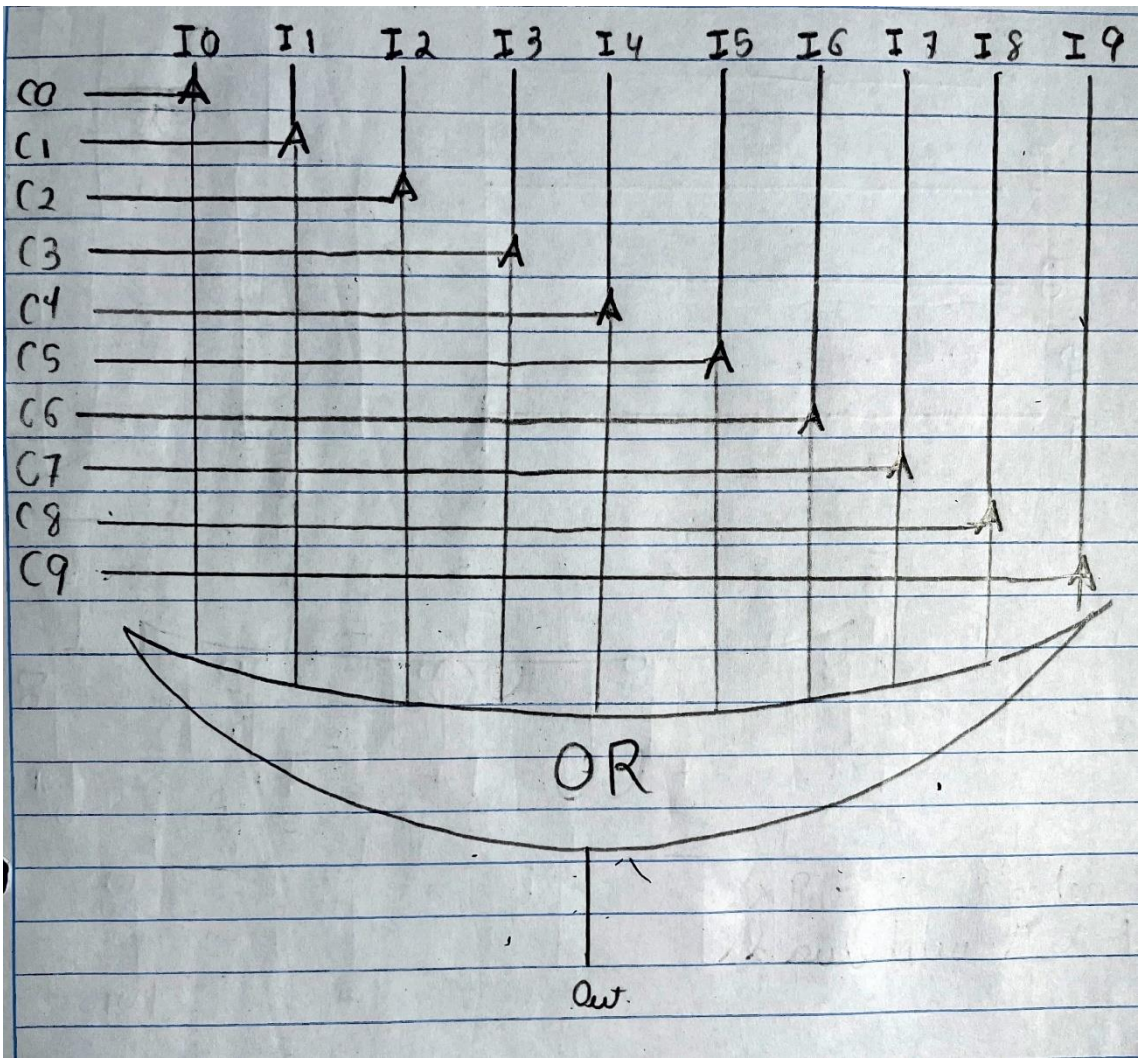
Igual ao caso das instruções do tipo R, possuem o parâmetro no bit 30 da insn. Também foi feito um circuito que emite o sinal corretamente para as funções necessárias:

```
assign sub_sra = (~func[2] & func[1]) | (func[2] & ~func[1] & func[0] & insn[30]);
/* a subtração deve existir nos casos em que precisamos de funções de comparação,
como slti e sltiu, e no shift right quando o segundo bit mais significativo for 1 */
```

3.4.5.2 Gates

No contexto desse processador, chamamos de “gate” um MUX que funciona de forma levemente diferente. Ao invés de ter um seletor de n bits, o qual seleciona até 2^n entradas, temos como seletor o code, um código de 10 bits em que é diferente de 0 somente em um deles. A entrada é uma palavra de 10 bits e a saída de 1 bit. A ideia é selecionar para a saída somente um bit da entrada, aquele que está na mesma posição que o bit = 1 do code. Por exemplo, se code = 0100000000, então somente o bit [8] da entrada será selecionado para a saída.

O code, como visto na seção 3.4.1, representa uma instrução, assim se a entrada for a concatenação de sinais de instruções, na ordem prevista por ele, será possível selecionar qual deles irá para saída. Desta forma, se a concatenação de todos os sinais pc_next_sel for a palavra 0001010100, e a instrução atual tiver code igual a 0000010000, o gate selecionará o bit [4], o qual será igual a 1. Já se code for igual a 1000000000, a saída será 0 pois irá selecionar o bit [9]. O formato de um gate é demonstrado a seguir, onde os “I” são inputs, os “C” são do code e os “A” são portas AND.



Tendo “dec_data” como input e S o output, a descrição em verilog é:

```
assign S = |(code & Dec_Data);
```

Na CU, a concatenação dos sinais foi feita já considerando os sinais fixos, os derivados de clk e os gerados pelos instruction decoders:

```
// concatenação de todos os sinais de saída para ser utilizado no Gate
assign addr_sel_c = {1'b0, ~clk, 1'b0, ~clk, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0};
assign pc_next_sel_c = {1'b1, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1, ~clk};
assign sub_sra_c = {1'b0, 1'b0, sub_sra_IA, 1'b0, sub_sra_R, 1'b1, 1'b0, 1'bx, 1'b0, 1'b0};
assign pc_alu_sel_c = {1'b0, 1'b0, 1'b0, 1'b0, 1'b0, pc_alu_sel_B, 1'b0,  1'b0, 1'b0, clk};
assign rd_clk_c = {1'b0, clk, clk, 1'b0, clk, 1'b0, clk, clk, clk, clk};
assign mem_clk_c = {1'b0, 1'b0, 1'b0, clk, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0};
```

OBS: cada posição na concatenação corresponde à posição do bit = 1 do code para cada instrução. A instrução J por exemplo tem sinais enviados no bit menos significativo pois seu code é 0000000001. Os sinais enviados foram já expostos e explicados na seção 3.4.5.

Após isso, o sinal final que será enviado da CU para outros blocos é selecionado pelo gates:

```
// instanciação dos gates para cada sinal, os quais retornam diretamente as saídas.
Gate G0 (.code(code), .Dec_Data(addr_sel_c), .S(addr_sel));
Gate G1 (.code(code), .Dec_Data(pc_next_sel_c), .S(pc_next_sel));
Gate G2 (.code(code), .Dec_Data(sub_sra_c), .S(sub_sra));
Gate G3 (.code(code), .Dec_Data(pc_alu_sel_c), .S(pc_alu_sel));
Gate G4 (.code(code), .Dec_Data(rd_clk_c), .S(rd_clk));
Gate G5 (.code(code), .Dec_Data(mem_clk_c), .S(mem_clk));
```

3.4.6 Casos especiais: rd_sel, alu_sel_a, alu_sel_b e func

Esses sinais foram gerados em módulos com circuitos derivados de tabelas verdade. Com base no code, possuem saídas esperadas, as quais são especificadas as seguir.

Rd_sel:

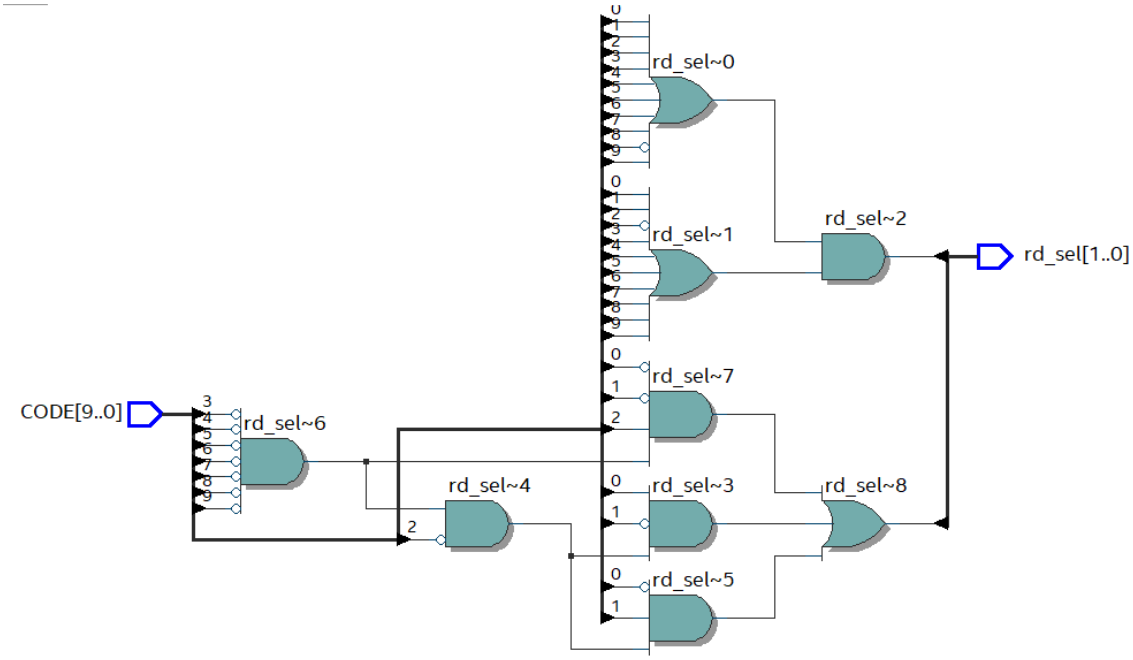
Módulo responsável por gerar o sinal rd_sel a partir do código code:

- Code = 10'b0000000001, gera sinal: 11 // Instrução tipo J
- Code = 10'b0000000010, gera sinal: 11 // Instrução tipo I (Jarl)
- Code = 10'b0000000100, gera sinal: 01 // Instrução tipo U (LUI)
- Code = 10'b0000001000, gera sinal: 10 // Instrução tipo U (AUIPC)
- Code = 10'b0000010000, gera sinal: xx // Instrução tipo B
- Code = 10'b0000100000, gera sinal: 10 // Instrução tipo R
- Code = 10'b0001000000, gera sinal: xx // Instrução tipo S
- Code = 10'b0010000000, gera sinal: 10 // Instrução tipo I (ALU)
- Code = 10'b0100000000, gera sinal: 00 // Instrução tipo I (LOAD)
- Code = 10'b1000000000, gera sinal: xx // Instrução tipo I (CSR)

A tabela verdade para esse circuito é a seguinte:

```
0000000001  1 / 1                                / (~A&~B&~C&~D&~E&~F&~G&~H&~I&J)
0000000010  1 / 1                                / (~A&~B&~C&~D&~E&~F&~G&~H&I&~J)
0000000100  0 / 1 (A|B|C|D|E|F|G|~H|I|J) / (~A&~B&~C&~D&~E&~F&~G&H&~I&~J)
0000001000  1 / 0
0000010000  x / x
0000100000  1 / 0
0001000000  x / x
0010000000  1 / 0
0100000000  0 / 0 (A|~B|C|D|E|F|G|H|I|J)
1000000000  x / x
```

Com circuito sintetizado sendo:



Alu_sel_a:

Módulo responsável por gerar o sinal alu_sel_a a partir do código code

- code = 10'b0000000001, gera sinal: 1 // Instrução tipo J
- code = 10'b0000000010, gera sinal: 0 // Instrução tipo I (Jarl)
- code = 10'b0000000100, gera sinal: x // Instrução tipo U (LUI)
- code = 10'b0000001000, gera sinal: 1 // Instrução tipo U (AUIPC)
- code = 10'b0000010000, gera sinal: 0 // Instrução tipo B
- code = 10'b0000100000, gera sinal: 0 // Instrução tipo R
- code = 10'b0001000000, gera sinal: 0 // Instrução tipo S
- code = 10'b0010000000, gera sinal: 0 // Instrução tipo I (ALU)
- code = 10'b0100000000, gera sinal: 0 // Instrução tipo I (LOAD)

* code = 10'b1000000000, gera sinal: 1 ou 0 // Instrução tipo I (CSR)

OBS: Para o último code, referente as instruções CSR, o valor de alu_sel_a deve ser 0 caso a instrução seja um ecall e 1 caso seja um ebreak

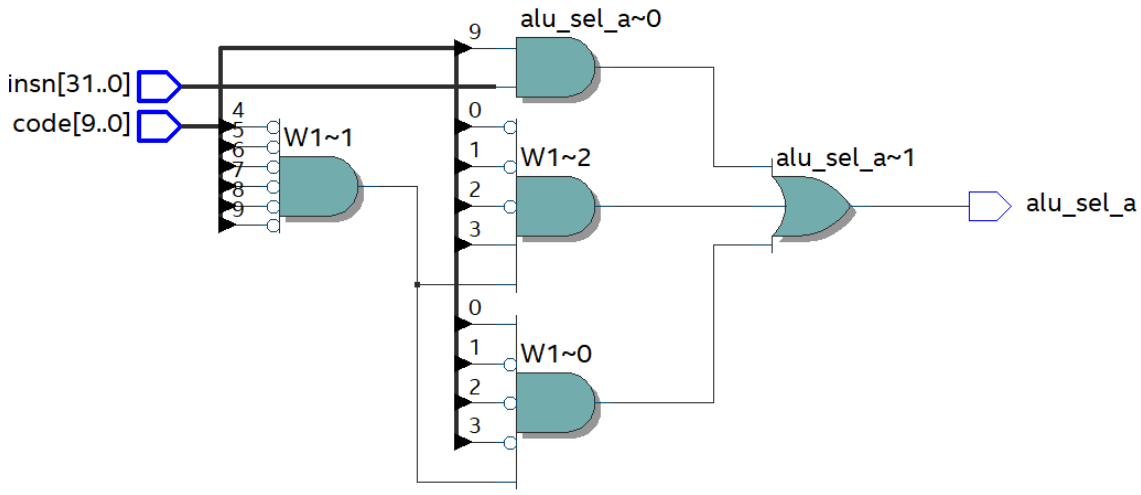
A tabela verdade para esse circuito é a seguinte:

```
Tabela verdade do circuito (função em mintermos com A sendo msb e J lsb):

    0000000001  1 (~A&~B&~C&~D&~E&~F&~G&~H&~I&J)
    0000000010  0
    0000000100  x
    0000001000  1 (~A&~B&~C&~D&~E&~F&G&~H&~I&~J)
    0000010000  0
    0000100000  0
    0001000000  0
    0010000000  0
    0100000000  0
```

Para o caso das instruções CSR, o valor do alu_sel_a é determinado pelo bit 20 da insn. Sendo 0 para ecall e 1 para ebreak:

```
/* Se a função acima de devolveu 1, apenas passar o sinal
Caso contrário, devolver 0 para todos os codes != 1000000000 e 1
 somente quando o bit 20 de insn for 1 (diferenciador entre ecall e ebreak) */
assign alu_sel_a = W1 | (code[9] & insn[20]);
```

O circuito sintetizado foi:



Alu_sel_b:

Módulo responsável por gerar o sinal alu_sel_b a partir do código code

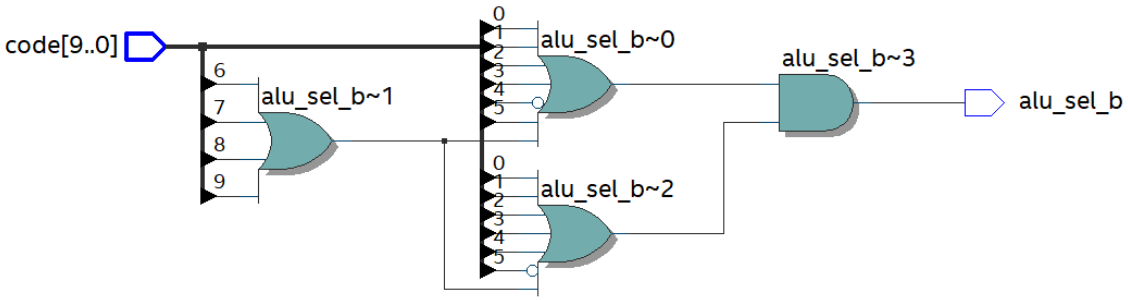
- Code = 10'b0000000001, gera sinal: 1 // Instrução tipo J
- Code = 10'b0000000010, gera sinal: 1 // Instrução tipo I (Jarl)
- Code = 10'b0000000100, gera sinal: x // Instrução tipo U (LUI)
- Code = 10'b0000001000, gera sinal: 1 // Instrução tipo U (AUIPC)
- Code = 10'b0000010000, gera sinal: 0 // Instrução tipo B
- Code = 10'b0000100000, gera sinal: 0 // Instrução tipo R
- Code = 10'b0001000000, gera sinal: 1 // Instrução tipo S

- Code = 10'b0010000000, gera sinal: 1 // Instrução tipo I (ALU)
- Code = 10'b0100000000, gera sinal: 1 // Instrução tipo I (LOAD)
- Code = 10'b1000000000, gera sinal: 1 // Instrução tipo I (CSR)

A tabela verdade para esse circuito é a seguinte:

```
Tabela verdade do circuito (função em maxtermos com A sendo msb e J lsb):

    0000000001  1
    0000000010  1
    0000000100  x
    0000001000  1
    0000010000  0 (A|B|C|D|E|~F|G|H|I|J)
    0000100000  0 (A|B|C|D|~E|F|G|H|I|J)
    0001000000  1
    0010000000  1
    0100000000  1
    1000000000  1
```

Com circuito sintetizado sendo:



Func:

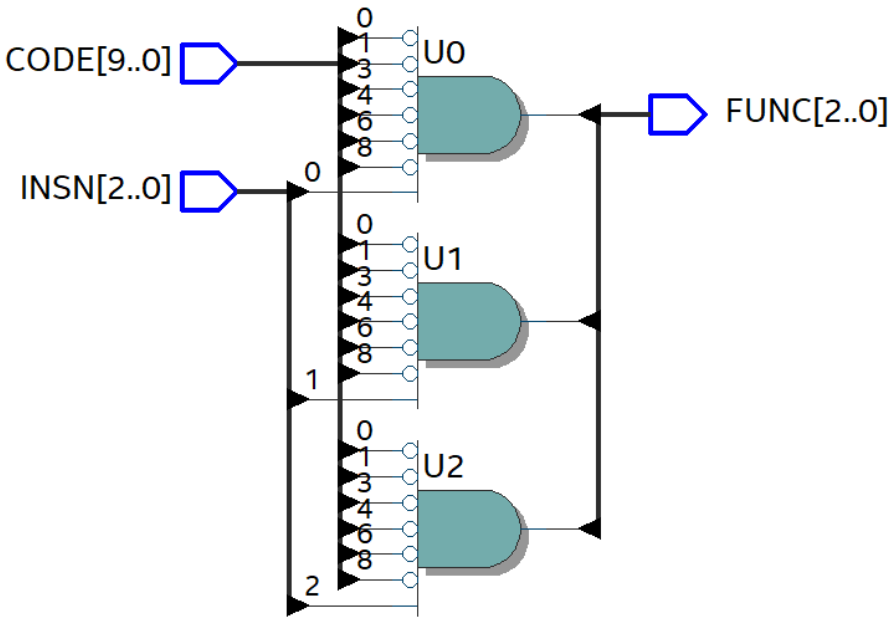
Esse é um caso especial de “gate”. Algumas instruções necessitam que o valor de func seja igual a 000 para poderem fazer operações de soma/subtração na ALU, porém nem sempre o campo func da insn dessas instruções possui esse valor. Por isso, é necessário selecionar qual valor de func a CU emitirá para os outros componentes.

Para isso, o módulo recebe o code e força a saída ser 000 caso code seja um entre: 0000000001, 0000000010, 0000001000, 0000010000, 0001000000, 0100000000. Caso contrário, apenas transmite direto o sinal recebido da instrução insn. A descrição de um circuito que faz isso é a seguinte:

```
input [2:0] insn; // Instrução
input [9:0] code; // Código gerado pelo OPDecoder
output [2:0] func; // Sinal func gerado

and U0 (func[0], insn[0], ~code[0], ~code[1], ~code[3], ~code[6], ~code[8], ~code[4]);
and U1 (func[1], insn[1], ~code[0], ~code[1], ~code[3], ~code[6], ~code[8], ~code[4]);
and U2 (func[2], insn[2], ~code[0], ~code[1], ~code[3], ~code[6], ~code[8], ~code[4]);
```

OBS: Repare que code[0], code[1], code[3], code[4], code[6] e code[8] são posições em que o 1 aparece nos codes em que a saída deve ser forçada para 000.

O circuito sintetizado foi:



3.4.7 Relacionamento com outros componentes

Na ALU, a CU é responsável por controlar as operações, os valores de entradas (por meio de seletores de MUXes) e a saída. Os principais MUXes do processador possuem como seletores os sinais dela. O Regfile possui entradas, saídas e armazenamento controlados por ela. Além disso, a control unit também envia sinais que decidem qual valor o PC irá receber, bem como quando isso acontecerá. Por fim, direciona as entradas da memória e seleciona qual endereço dela será acessado.

A CU está ligada a cada um dos componentes do processador. Por ser justamente o “centro de controle”, do qual partem todos os sinais importante, ela a peça chave que coordena todo o funcionamento do circuito. Seus inputs vêm do insn_reg, do gerador de clk e da ALU. Seus outputs seguem para os demais blocos. A CU é o coração do processador, componente que torna possível executar diferentes operações com base em diferentes instruções, dando o poder do circuito de ser programável.

3.5 DATAFLOW

Entende-se por dataflow os componentes que recebem sinais da CU e executam operações. São os circuitos por ordem as informações fluem e são armazenadas. No processador em questão, esses blocos são: MUXes, elementos de memória, regfile e módulo para extensão da memória.

OBS: A ALU faz parte do dataflow, porém nesse documento ela foi tratada separadamente devido a sua importância e complexidade.

3.5.1 Multiplexadores

São circuitos que seleciona uma das entradas para ir na saída, com base em um seletor. No processador, os MUXes principais são de dois tipos: de 2 entradas e de 4 entradas de 32bits. Implementados da seguinte forma:

Duas entradas

```
reg [31:0] data_o_r; // reg para ser utilizado no bloco always

always @(*) begin
  case (select)
    1'd0: data_o_r = I0; // caso o seletor for 0, I0 deve passar para a saída
    1'd1: data_o_r = I1; // caso o seletor for 1, I1 deve passar para a saída
  endcase
end

assign data_o = data_o_r;
```

Quatro entradas:

```
reg [31:0] data_o_r; // reg para ser utilizado no bloco always

always @(*) begin
  case (select)
    2'd0: data_o_r = I0; // caso o seletor for 00, I0 deve passar para a saída
    2'd1: data_o_r = I1; // caso o seletor for 01, I1 deve passar para a saída
    2'd2: data_o_r = I2; // caso o seletor for 10, I2 deve passar para a saída
    2'd3: data_o_r = I3; // caso o seletor for 11, I3 deve passar para a saída
  endcase
end
assign data_o = data_o_r;
```

Sendo os “I” os inputs e data_o o output do circuito.

Os MUXes principais são:

- **PCNEXTSEL**: multiplexador de 2 entradas para selecionar o próximo valor do PC.

- **PCALUSEL**: multiplexador de 2 entradas para seleção da segunda entrada da ALU do Program Counter.
- **ALUASEL**: multiplexador de 2 entradas para selecionar qual valor irá entrar na ALU geral, podendo ser o PC ou o valor do rs1.
- **ALUBSEL**: multiplexador de 2 entradas para selecionar qual valor irá entrar na ALU geral, podendo ser um imediato ou o valor do rs2.
- **ADDRSEL**: multiplexador de 2 entradas para selecionar endereço, utilizado em função de store para gravar algum valor.
- **RDSE**L: multiplexador de 4 entradas para selecionar qual valor deverá ser gravado no registrador destino presente na Regfile.

3.5.2 Elementos de memória

São dois registradores de 32bits especiais que armazenam informações que são utilizadas diretamente pela CU/memória. São controlados por sinais especiais advindos da CU. Esses circuitos são, assim como os registradores do regfile, conjuntos de 32 latches que armazenam dados de 1 bit cada.

3.5.2.1 Instruction register

Esse registrador é responsável por armazenar o valor da insn retirado da memória. Ele recebe um sinal de clk advindo da CU, o insn_clk, no qual a cada borda de subida ele armazena o conteúdo proveniente da memória. O seu output, a insn, segue diretamente para CU. Como esse sinal é igual ao clk negado, sua borda de subida ocorre meio período de clock antes, fazendo com que a instrução seja decodificada e que as operações corretas sejam realizadas antes que os sinais que armazenam os resultados (rd_clk, pc_clk e mem_clk) tenham uma borda de subida.

3.5.2.2 Program Counter

Esse registrador armazena o endereço que será acessado na memória em busca da próxima instrução. Ele recebe um sinal de clk advindo da CU, o pc_clk, no qual a cada borda de subida o registrador armazena o valor advindo do PCNEXTSEL. Seu output é o valor que irá para o ADDRSEL.

Sobre esse registrador vale comentar os possíveis valores que ele pode assumir dependendo dos seletores. O PCNEXTSEL seleciona um valor entre o valor

advindo do PCALUSEL e valor advindo da ALU. Já o PCALUSEL seleciona um valor entre PC + 4 (próxima instrução) e PC + imm (PC incrementado). Desta forma, dependendo dos seletores enviados pela CU, o PC pode assumir tanto seguir para próxima instrução, pular para um valor resultado de alguma operação na ALU ou ser incrementado em alguma quantidade especificada em um imm.

OBS: O incremento do “PC + imm” é feito sem passar pela ALU, executado por um adder de 32 bits presente no dataflow, dedicado a essa finalidade.

3.5.3 Regfile

Conjunto de 32 registradores que armazenam palavras de 32bits. Todos são acessados por endereços de 32bits, podem ser lidos ou escritos. O circuito recebe um sinal de clk da CU, o rd_clk, no qual a borda de subida armazena o valor presente no input “rd_val” no registrador de endereço “rd”. A leitura é feita acessando registradores de dois endereços, rs1 e rs2, enviando seus valores nas saídas. Esse componente possui um sinal reset síncrono, o qual, em uma borda de subida do rd_clk, se for 1 ele zera todos os registradores e faz nada se for 0. Apesar de não ser muito complexo, é o maior circuito presente no processador.

3.5.4 Extensão da memória

Instruções do tipo I – LOAD podem retirar da memória valores que não são de 32 bits, sendo necessário que sejam estendidos. O circuito responsável por fazer isso recebe um sinal da CU, o mem_extend, que especifica qual tipo de extensão deve ser feita. Uma extensão signed completa os bits restantes copiando o msb bit, ou seja, o bit de sinal. Já a extensão unsigned apenas completa com 0 os bits restantes.

- Mem_extend = 000 // Extensão signed de um byte
- Mem_extend = 001 // Extensão signed de uma halfword
- Mem_extend = 010 // Palavra de 32bits, não realiza extensão
- Mem_extend = 100 // Extensão unsigned de um byte
- Mem_extend = 101 // Extensão unsigned de uma halfword

Descrição do circuito:

```
always @(*) begin
    case (mem_extend)
        3'b000: mem_extended_r = {{24{mem_value[7]}},mem_value[7:0]};
        3'b001: mem_extended_r = {{16{mem_value[15]}}, mem_value[15:0]};
        3'b010: mem_extended_r = mem_value;
        3'b100: mem_extended_r = {{24{1'b0}}, mem_value[7:0]};
        3'b101: mem_extended_r = {{16{1'b0}}, mem_value[15:0]};
        default: mem_extended_r = 32'hxxxxxxxx;
    endcase
end
```

O circuito sintetizado a partir dessa descrição foi um conjunto de MUXes com entradas de 8 bits.

3.5.5 Relacionamento com outros componentes

O dataflow como um todo (considerando também a ALU) se relaciona com a CU e a memória. Para a memória, ele envia um endereço, um sinal mem_clk, um valor de 32bits data_i e o sinal mem_size. Caso queira armazenar algum valor na memória, ele manda o valor necessário no endereço especificado e, na borda de subida do mem_clk, o armazena no tamanho especificado pelo mem_size. Da memória, ele recebe o valor presente no endereço especificado, o qual pode ser uma instrução ou um número a ser usado numa instrução LOAD. Já a CU recebe do dataflow o valor da instrução que ele retirou da memória, além dos sinais provenientes da ALU. De volta, a Control Unit envia todos os sinais seletores, de endereço, de valor ou de clks necessários para o dataflow executar suas operações.

3.6 Relacionamento entre CPU e RAM

O processador recebe as instruções que estão armazenadas em um elemento de memória externo, no caso uma RAM. A memória considerada armazena em cada endereço palavras de 8 bits (bytes). Assim, uma instrução de 32bits sempre está codificada em 4 endereços, por isso que o PC sempre avança de 4 em 4. A memória em questão possui um sistema de endereçamento little-endian, no qual os bytes menos significativos da instrução são armazenados em endereços menores da memória. Desta forma, a instrução 00109913 se encontraria na memória da seguinte forma:

Endereço: 00 – Byte: 13

Endereço: 01 – Byte: 99

Endereço: 02 – Byte: 10

Endereço: 03 – Byte: 00

A saída da memória é uma concatenação de 4 bytes, do formato {addr + 3, addr+ 2, addr + 1, addr}. Note que nesse caso, a instrução acima sairia no formato correto (00109913), pois sua leitura na memória se deu da direita para esquerda.

O processador acessa um elemento da memória por meio de um endereço e recebe o valor presente nele concatenado com os próximos 3 endereços (na forma mostrada acima). Caso queira armazenar um valor, ele é mandado no input da memória, junto a um sinal que indica o tamanho da palavra que será armazenada: um byte (um endereço), uma halfword (dois endereços) ou uma word (4 endereços). O valor é gravado na memória na borda de subida do mem_clk.

4. FUNCIONAMENTO

O processador foi totalmente descrito em verilog. No git disponibilizado no link presente na capa desse documento encontram-se, além dos códigos em verilog, documentos que explicam como compilar eles, como executar os testbenchs e como fazer sua inspeção. O funcionamento de cada módulo pode ser verificado compilando os arquivos e simulando os testbenchs. Alguns testbench mostram valores no terminal, cabendo à pessoa executando-o analisar os dados e verificar se estão corretos. A maioria dos testbenchs, porém, funcionam no regime “self check”, mostrando no terminal somente os erros, se for encontrado algum.

OBS: No caso dos TB “self check”, o próprio módulo verifica se os valores recebidos do circuito são iguais aos esperados. Esse método, entretanto, depende que o testbench faça verificações corretas. Desta forma, uma forma de descobrir se um circuito faz o esperado é verificar se o seu testbench executa os testes de forma certa.

Para o teste do processor completo, alguns passos devem ser seguidos. Primeiramente, escolhe qual o conjunto de instruções serão testadas. Isso é feito dentro do módulo RAM_mod.v, o qual possui comentadas as orientações de como prosseguir para selecionar os testes. Com a escolha feita, compila-se o arquivo (ou todos os arquivos, através da makefile) e executa-se a simulação do CPU_test.v (vsim -c work.CPU_TB).

Esse testbench possui um formato diferente dos outros. Ele gera um arquivo “CPU.vcd” o qual pode ser aberto por um software para a inspeção de todos os sinais presentes no processador. O software utilizado no projeto para analisar os vcd foi o GTKWave. Com esse vcd gerado e aberto, o funcionamento do processador pode ser observado através do timing diagram

4.1 Funcionamento das instruções

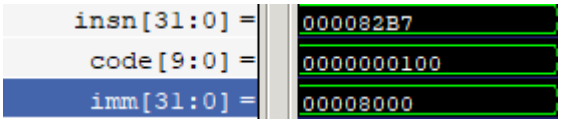
Nesta seção explicaremos brevemente e de forma simplificada o funcionamento de cada instrução, mostrando o timing diagram do processador as executando. O programa testado foi o “testeInstrucoes”, o qual testa cada uma das instruções presentes na ISA do RV-32I.

4.1.1 Tipo U – LUI

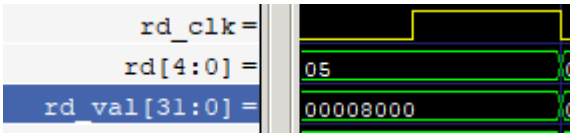
Instrução testada: 000082B7 lui x05, 0x08 // x05 = 0x8000

Nela, o registrador 5 deve receber o valor imediato codificado conforme exposto na seção 3.4.2 “obtenção do imediato”.

A insn 000082B7 entra na CU, a qual obtém o imediato, o endereço rd e manda o valor do imm para o rd_val.



Na borda de subida do sinal rd_clk, o valor do rd_val (0x8000) será armazenado no endereço rd (5).

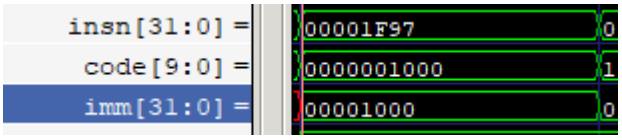


4.1.2 Tipo U - AUIPC

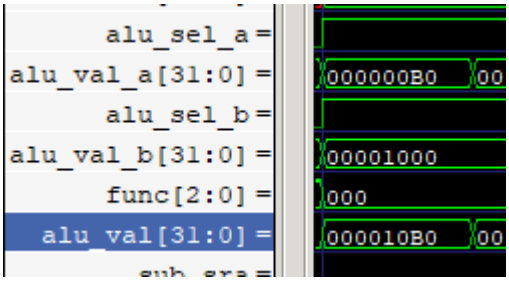
Instrução testada: 00001F97 auipc x1F, 0x01 // x1F = 176 + 0x01000 = 0x10b0

Nessa instrução, o rd x1F recebe o valor do PC + 0x01. No programa, essa instrução está no endereço 176 da memória, sendo esse o valor do PC.

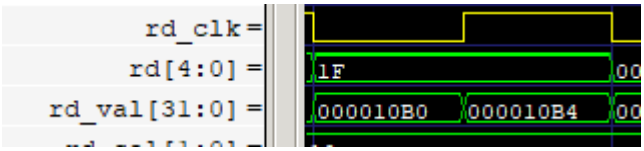
A CU obtém o imediato da insn



Após isso, ela seleciona as entradas da ALU como sendo A = PC e B = imm. A operação é a de soma, logo func = 000.



Por fim, manda o resultado da ALU (0x10B0) para o rd_val, armazenando o no endereço rd (x1F) na borda de subida do rd_clk.

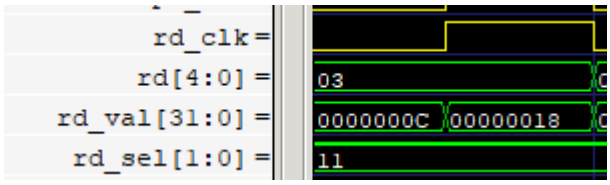


4.1.3 Tipo J

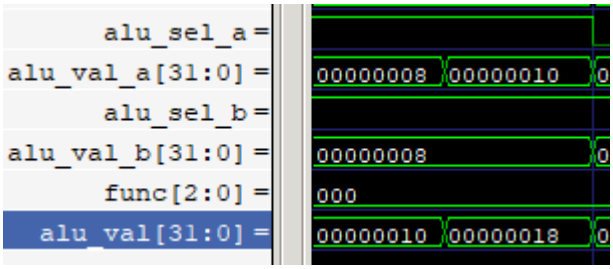
Instrução testada: 008001EF jal x03, 0x08 // x03 = 8 + 4 = 0x0c, pc = 8 + 8 = 0x10

Essa instrução armazena o valor da próxima instrução no rd (x03) e soma um imm ao PC. No programa, essa instrução está no enderenço 8, sendo esse o valor do PC.

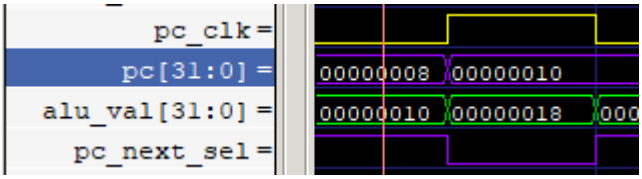
A CU manda o rd_sel = 11, para o o mux RDSEL selecionar como entrada o valor da próxima instrução (8 + 4 = 0x0C), a qual será armazenada no rd (3) na próxima borda de subida do rd_clk.



Ao mesmo tempo, a CU seleciona as entradas da ALU como sendo A = PC e B = IMM. A operação é de soma, logo func = 000.



A CU manda o sinal pc_next_sel selecionar o resultado da ALU como próximo valor do PC. Na próxima borda de subida do do pc_clk, o valor presente no alu_val irá ser armazenado no PC.

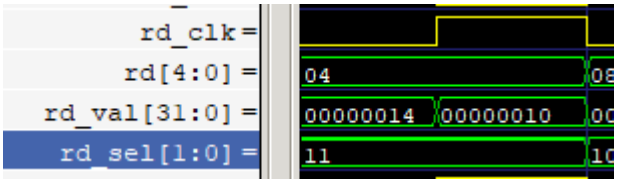


4.1.4 Tipo I – JALR

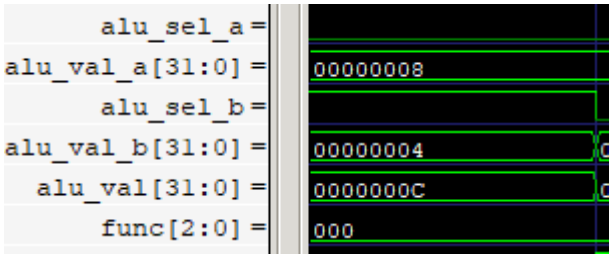
Instrução testada: 00410267 jalr x04, 4(0x02) // x04 = 16 + 4 = 0x14, pc = 4 + 0x08 = 0x0c

Nessa instrução o rd (x04) recebe o valor da próxima instrução, depois pula o PC para o valor do imm (4) + rs1 (0x02). No programa, o endereço dessa instrução, portanto o PC, vale 16 e o valor armazenado dentro do rs1 0x02 é 8.

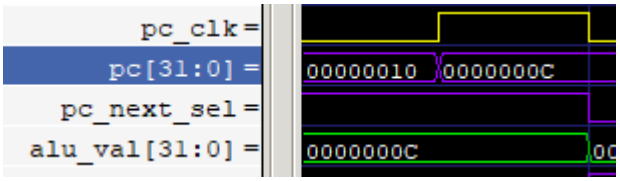
A CU manda o rd_sel = 11, para o o mux RDSEL selecionar como entrada o valor da próxima instrução (14 + 4 = 0x14), a qual será armazenada no rd (4) na próxima borda de subida do rd_clk.



Ao mesmo tempo, a CU seleciona as entradas da ALU como sendo A = rs1 e B = IMM. A operação é de soma, logo func = 000.



A CU manda o sinal pc_next_sel selecionar o resultado da ALU como próximo valor do PC. Na próxima borda de subida do do pc_clk, o valor presente no alu_val irá ser armazenado no PC.

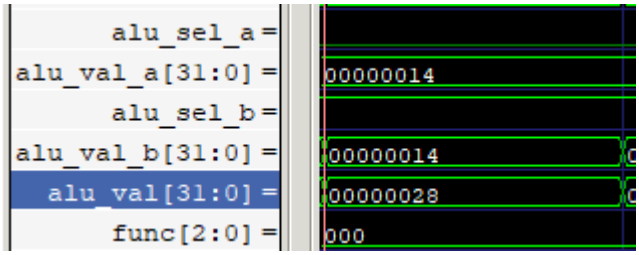


4.1.5 Tipo I – LOAD

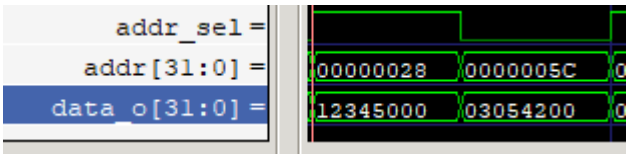
Instrução testada: 01422483 lw x09, 20(x04) // x09 = m32(20 + 0x14) = 0x12345000

Nessa instrução, o rd (x09) recebe o valor que está armazenado no endereço de memória igual a imm (20) + rs1. No programa, o valor armazenado no registrador x04 é 0x014 e, no endereço 20 + 0x14 está armazenado o valor 0x12345000.

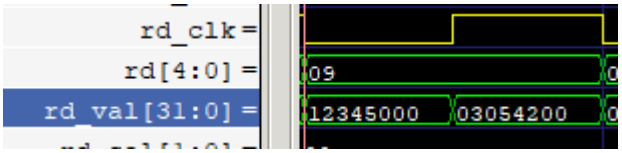
A CU seleciona as entradas da ALU como sendo A = rs1 e B = IMM. A operação é de soma, logo func = 000.



Após isso, o ADDRSEL seleciona o endereço como sendo o alu_val e acessa a memória, a qual saída está no sinal data_o



O sinal data_o então passa para o rd_val, armazenando-o no endereço rd (x09) na borda de subida do rd_clk.

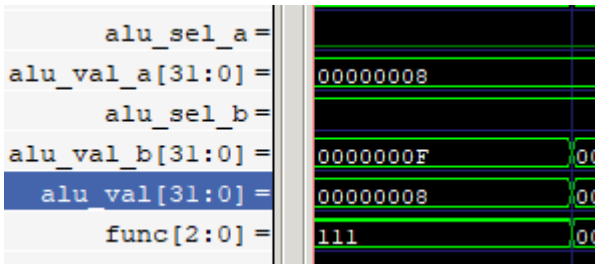


4.1.6 Tipo I – ALU

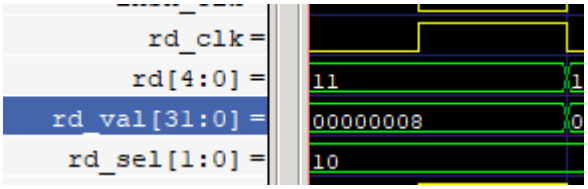
Instrução testada: 00F0F893 andi x11, x01, 0x0F // x11 = 0x8

Nessa instrução, o rd (0x11) recebe o resultado de um bitwise AND entre o rs1 (x01) e o imm (0x0F). Nesse programa, o registrador x01 tem armazenado o valor 8 (1000 em binário).

A CU seleciona as entradas da ALU como sendo A = rs1 e B = IMM. A operação é de bitwise AND, logo func = 111.



O resultado alu_val então passa para o rd_val, armazenando-o no endereço rd (x11) na borda de subida do rd_clk.

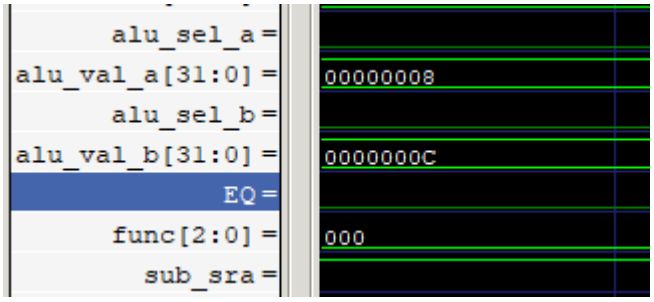


4.1.7 Tipo B

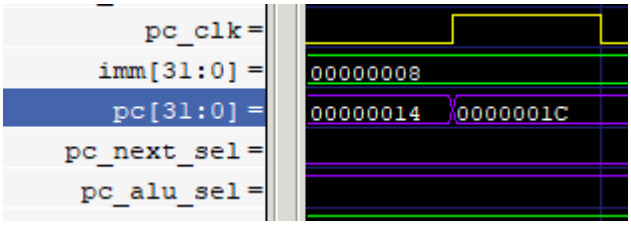
Instrução testada: 00309463 bne x01, x03, 0x08 // pc = 20 + 0x08 = 0x1C

Nessa instrução, se rs1 (x01) for diferente do rs2 (x03), o PC recebe o valor de PC + imm (0x08), se não for o PC segue para próxima instrução. Nesse programa, o endereço dessa instrução, portanto o PC, vale 20. O valor armazenado dentro do rs1 é 8 e dentro do rs2 é 0x0C.

A CU seleciona as entradas da ALU como sendo A = rs2 e B = rs2. A operação é de comparação, logo func = 000 e sub_sra = 1.



Recebendo o sinal EQ da ALU (EQ = 0), a CU envia o sinal pc_alu_sel decidindo se vai ou não ocorrer um Branch. Como nesse caso ocorrerá, então na borda de subida do pc_clk o PC recebe PC + imm (20 + 0x08 = 0x1C), operação feia no adder 32bits dedicado.

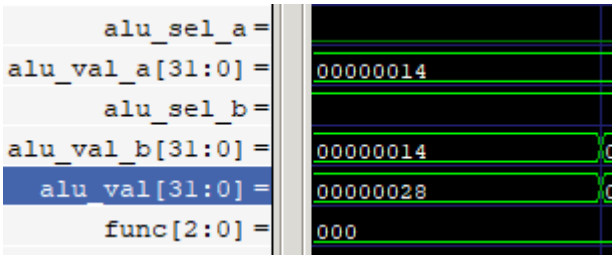


4.1.8 Tipo S

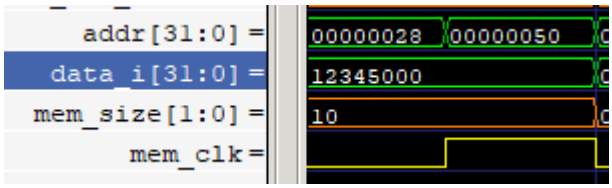
Instrução testada: 00622A23 sw x06, 20(x04) // mem(20+ 0x14) = 0x12345000

Nessa instrução, o valor do rs2 (x06) é armazenado na memória a partir do endereço dado por imm (20) + rs1 (x04). Nesse programa, o valor armazenado no rs1 é 0x14 e no rs2 é 0x12345000.

A CU seleciona as entradas da ALU como sendo A = rs1 e B = imm. A operação é de adição, logo func = 000.



A CU envia o sinal addr_sel para o mux ADDRSEL selecionar como endereço o resultado indo da ALU. O input da memória no data_i é o valor do rs2, o qual será armazenado na memória na borda de subida do mem_clk, no tamanho especificado por mem_size.

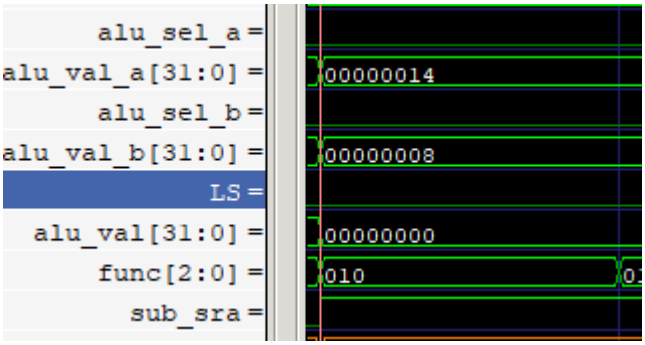


4.1.9 Tipo R

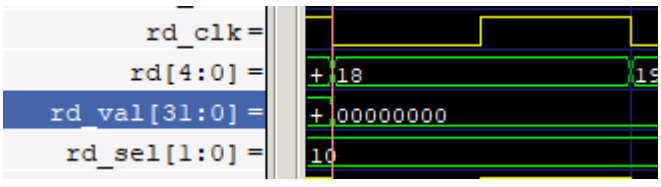
Instrução testada: 00122C33 slt x18, x04, x01 // x18 = 0

Nessa instrução, o rd (x18) recebe 1 caso rs1 (x04) for menor que rs2 (x01), recebe 0 caso contrário. Nesse programa, o registrador x04 armazena o valor 0x14 e o x01 armazena 0x08.

A CU seleciona as entradas da ALU como sendo A = rs1 e B = rs2. A operação é de comparação, logo func = 000 e sub_sra = 1.



O resultado alu_val (0, pois rs1 > rs2) então passa para o rd_val, armazenando-o no endereço rd (x18) na borda de subida do rd_clk.

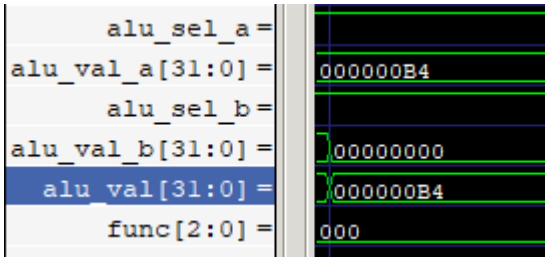


4.1.10 CSR instructions

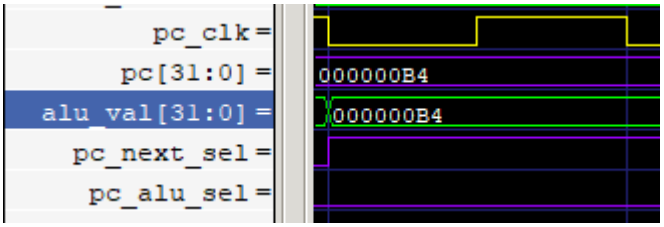
Instrução testada: 00100073 ebreak // PC = PC + 0

Nessa instrução o PC recebe PC + imm. Nesse programa, o PC nessa ocasião vale 0xB4.

A CU seleciona as entradas da ALU como sendo A = PC e B = imm. A operação é de soma, logo func = 000.



A CU manda o sinal pc_next_sel para o mux PCNEXTSEL, selecionando o resultado da ALU como próximo valor do PC. Assim alu_val será armazenado no PC na próxima borda de subida do pc_clk.



OBS: Essa instrução trava o programa, voltando sempre para a mesma instrução.

4.2 Outros testes

Os testes realizados no projeto consistem em pré-carregar a memória RAM, por meio de um arquivo .hex, com todas as instruções, e em seguida executar a simulação com estas instruções, utilizando os métodos de análise de timing diagram e análise de memória para verificar se o resultado obtido foi o esperado.

4.2.1 Testes de instruções:

O teste mais fundamental realizado, foi a execução e verificação de cada tipo de instrução, para isso, foi escrito um programa em C que gera o código binário de uma instrução baseado nas entradas do usuário, que foram então compilados em um arquivo .hex. Ao fim da execução, foi gerado um arquivo .vcd, que foi inspecionado utilizando GTKWave, para verificar se as instruções estavam realmente sendo executadas da forma esperada. As instruções testadas foram:

```
0  00800093 addi x1, x0, 0x08 // x01 = 0 + 0x08 = 0x08
4  00800113 addi x2, x0, 0x08 // x02 = 0 + 0x08 = 0x08
8  008001EF jal x03, 0x08 // x03 = 8 + 4 = 0x0c, pc = 8 + 8 = 0x10
12 00208463 beq x01, x02, 0x08 // pc = 12 + 8 = 0x14
16 00410267 jarl x04, 4(0x02) // x04 = 16 + 4 = 0x14, pc = 4 + 0x08 = 0x0c
20 00309463 bne x01, x03, 0x08 // pc = 20 + 0x08 = 0x1C
24 00000000
28 0030C463  blt x01, x03, 0x08 // pc = 28 + 8 = 0x24
32 00000000
36 0020D463 bge x01, x02, 0x08 // pc = 36 + 8 = 0x2C
40 00000000
44 0040E463 bltu x01, x04, 0x08 // pc = 44 + 8 = 0x34
48 00000000
52 00127463 bgeu x04, x01, 0x08 // pc = 52 + 8 = 0x3C
56 00000000
60 00420223 sb x04, 4(x04) // mem (4 + 0x14) = 0x14
64 000082B7 lui x05, 0x08 // x05 = 0x8000
68 00521623 sh x05, 12(x04) // mem(12 + 0x14) = 0x8000
72 12345337 lui x06, 0x12345 // x06 = 0x12345000
76 00622A23 sw x06, 20(x04) // mem(20+ 0x14) = 0x12345000
80 00420383 lb x07, 4(x04) // x07 = sx(m8(4 + 0x14)) = 0x14
84 00C21403 lh x08, 12(x04) // x08 = sx(m16(12+0x14)) = 0xFFFF8000
88 01422483 lw x09, 20(x04) // x09 = m32(20 + 0x14) = 0x12345000
92 00420503 lb x0A, 4(x04) // x0A = zx(m8(4 + 0x14)) = 0x14
96 00C25583 lhu x0B, 12(x04) // x0B = zx(m16(12+0x14)) = 0x8000
100 00C08613 addi x0C, x01, 12 // x0C = 12 + 0x08 = 0x14
104 0100A693 slti x0D, x01, 0x10 // x0D = 1
108 0100B713 sltu x0E, x01, 0x10 // x0E = 1
112 00F0C793 xori x0F, x01, 0x0F // x0F = 0x07
116 00F0E813 ori x10, x01, 0x0F // x10 = 0x0F
120 00F0F893 andi x11, x01, 0x0F // x11 = 0x8
124 00109913 slli x12, x01, 0x01 // x12 = 0x10
128 0010D993 srli x13, x01, 0x01 // x13 = 0x04
```

132 40145A13 srai x14, x08, 0x01 // x14 = 0xFFFFC000

136 00D08AB3 add x15, x01, x0D // x15 = 0x08 + 1 = 0x09

140 40D08B33 sub x16, x01, x0D // x16 = 0x08 - 1 = 0x07

144 00409BB3 sll x17, x01, x04 // x17 = 0x800000

148 00122C33 slt x18, x04, x01 // x18 = 0

152 00123CB3 sltu x19, x04, x01 // x19 = 0

156 0180CD33 xor x1A, x01, x18 // x1A = 0x08

160 00145DB3 srl x1B, x08, x01 // x1B = 0x00FFFF80

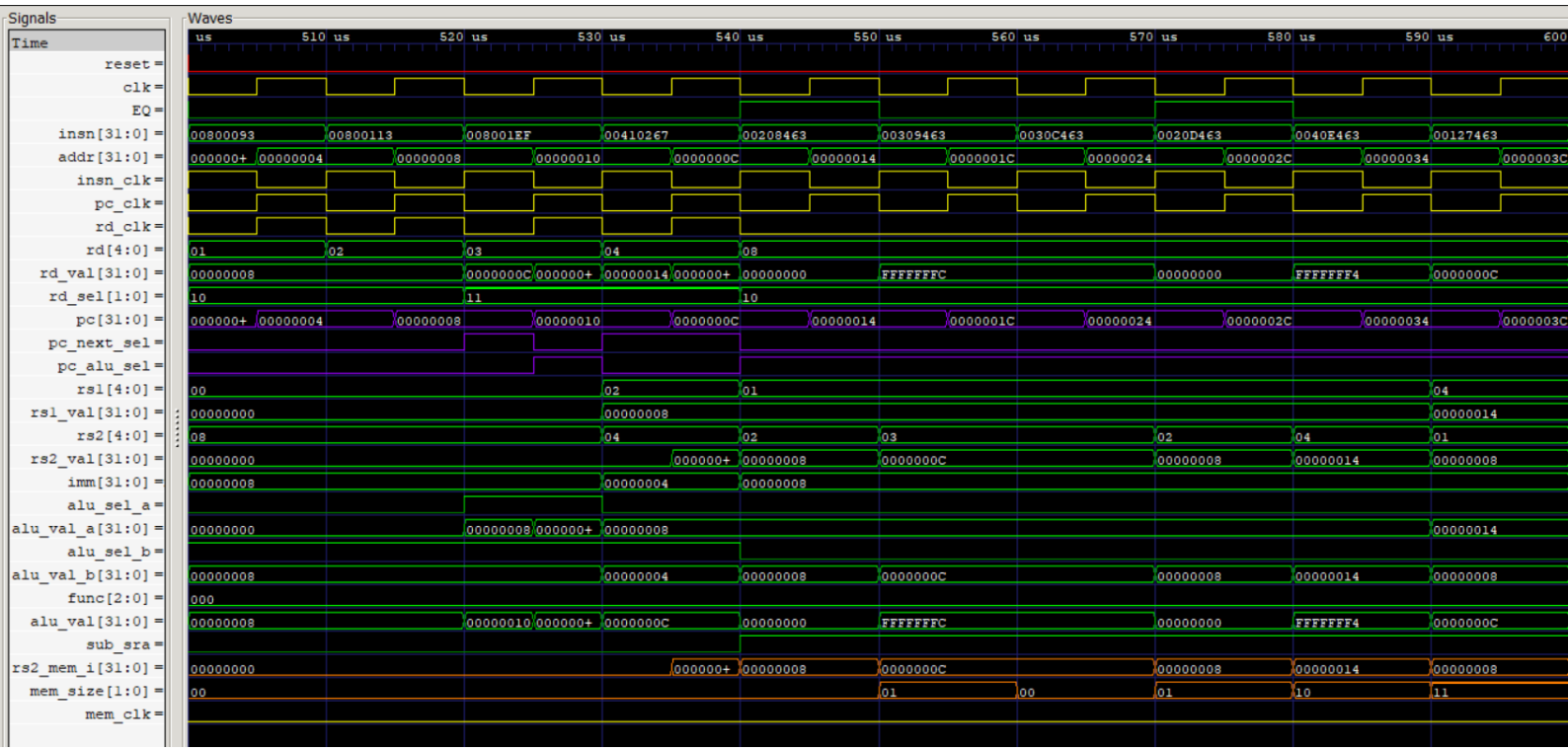
164 40145E33 sra x1C, x08, x01 // x1C = 0xFFFFFF80

168 00F0EEB3 or x1D, x01, x0F // x1D = 0x0F

172 00F0FF33 and x1E, x01, x0F // x1E = 0x00

176 00001F97 auipc x1F, 0x01 // x1F = 176 + 0x01000 = 0x10b0

180 00100073 ebreak // PC = PC + 0 = 180



Para a análise do diagrama temporal, foi necessária a identificação da ação que a instrução realizaria, e em seguida, esta identificação foi comparada com os valores reais entregues pela simulação. Por exemplo, para verificar se a instrução jal x03, 0x08 foi executada corretamente, primeiro foi identificado que esta instrução iria levar o registrador x03 para o valor 0x0c, e que o Program Counter seria atualizado

para 0x10, e de fato, pelo diagrama, é exatamente isto que acontece. Uma instrução de cada tipo teve funcionamento explicado na seção 4.1.

4.2.2 Teste de programas em C

Para ser possível confirmar o funcionamento do processador em um contexto mais prático, o processador também foi simulado com alguns simples programas em C, pois a verificação é mais simples, e garante a funcionalidade do processador com ferramentas utilizadas na realidade.

Para realizar a compilação dos programas em C, foi necessário a instalação de um Toolchain especial para RISC-V 32 bits, disponível em <https://github.com/riscv-collab/riscv-gnu-toolchain> . Com o Toolchain instalado, basta compilar o programa utilizando (riscv32-unknown-elf-gcc -march=rv32i -mabi=ilp32 -ffreestanding -fno-pic -nostdlib -Wl,-Ttext=0x0 -Wl,--no-relax main.c -o main), que irá gerar um executável, em seguida, basta realizar um dump deste executável utilizando (riscv32-unknown-elf-objdump -Mnumeric,no-aliases -dr main > main.dump), este arquivo .dump gerado irá conter o assembly do programa, junto com o código hexadecimal de cada instrução, e a sua posição na memória. Com este arquivo em mãos, basta inserir cada instrução em um novo .hex, carrega-lo na memória, e executar a simulação do processador.

Além disso também é importante lembrar que o RISC-V utiliza primariamente *little-endian*, isto é, os bits mais significativos de uma palavra estão nos endereços posteriores, fato que é de extrema importância para a realização das análises de memória.

Também vale notar que o C faz uso do registrador x02 como *Stack pointer*, que nada mais é que é um registrador que armazena o endereço onde foi armazenado o último elemento da memória, desta forma, ele serve como referência para onde o processador pode começar a armazenar dados. Para os testes, existirá uma linha de *in-line assembly* no começo de cada programa, inicializando o *Stack pointer* para um valor que seja suficiente para a execução do programa. Além disso também foi inserido mais um comando em *in-line assembly* no fim da execução do programa, para ser possível iniciar uma instrução *ebreak* para pararmos a execução da simulação.

4.2.2.1 Teste: Variável

O primeiro programa que foi testado, foi um simples programa que cria uma variável, a inicializa, e em seguida a incrementa em 1.

```
/* Arquivo para testar funcionalidade básica: alocar uma variável */
/* inicia o stack pointer em 1000, para realizar os testes */
asm(
  "addi sp, zero,1000\n\t"
);
int main () {
  int a;
  a = 1;
  a = a + 1;
  /* ebreak para parar a execução do processador assim que o programa for finalizado */
  asm(
    "ebreak\n\t"
  );
}
```

Antes de executar o programa, é válido analisar o arquivo .dump gerado, já que ele nos permite interpretar o que “criar uma variável” em C realmente reflete no hardware:

```
main:     file format elf32-littleriscv


Disassembly of section .text:

00000000 <main-0x4>:
   0:    3e800113              addi    x2,x0,1000

00000004 <main>:
   4:    fe010113              addi    x2,x2,-32
```

```
 8:  00812e23        sw    x8,28(x2)
 c:  02010413        addi  x8,x2,32
10:  00100793        addi  x15,x0,1
14:  fef42623        sw    x15,-20(x8)
18:  fec42783        lw    x15,-20(x8)
1c:  00178793        addi  x15,x15,1
20:  fef42623        sw    x15,-20(x8)
24:  00100073        ebreak
28:  00000013        addi  x0,x0,0
2c:  00078513        addi  x10,x15,0
30:  01c12403        lw    x8,28(x2)
34:  02010113        addi  x2,x2,32
38:  00008067        jalr  x0,0(x1)
```

Interpretando o assembly, podemos verificar que o programa inicia realizando operações com o *stack pointer*, que neste caso está sendo utilizado, juntamente em conjunto com o registrador x08 ou *frame pointer* para definir o endereço onde devemos armazenar a variável. Em seguida, podemos ver que há uma instrução que soma o registrador x0 ou *zero* com o imediato 1, e logo em seguida este valor é armazenado na memória RAM utilizando uma instrução de *Store*. Quando a linha em que a variável recebe o incremento, podemos verificar que é utilizada uma instrução de *Load* para recuperar a variável ao registrador x15, ele é incrementado em 1, e depois é armazenado novamente para a memória RAM. Desta forma podemos visualizar o que uma variável realmente representa em hardware.

Agora, ao executar a simulação e analisar o arquivo de memória, obtemos o resultado:

```
0000F40  78 78 0D 0A 78 78 0D 0A 78 78 0D 0A 78 78 0D 0A   xx..xx..xx..xx..
0000F50  30 32 0D 0A 30 30 0D 0A 30 30 0D 0A 30 30 0D 0A   02..00..00..00..
0000F60  78 78 0D 0A 78 78 0D 0A 78 78 0D 0A 78 78 0D 0A   xx..xx..xx..xx..
```

Lembrando que o processador utiliza *little-endian p*odemos verificar que o número dois realmente está na memória.

4.2.2.2 Teste: Vetor

O segundo teste executado, foi um teste envolvendo a alocação de vetores, onde foi alocado um pequeno vetor, e feito algumas operações entre seus elementos:

```
/* Teste para funcionalidade básica: uso e alocamento de vetores*/
/* inicia o stack pointer em 1000, para realizar os testes */
asm(
  "addi sp, zero,1000\n\t"
);
int main () {
  int v[6];
  v[0] = 1;
  v[1] = v[0] + 1;
  v[2] = 3;
  v[3] = 890;
  v[4] = v[3] - v[2];
  v[5] = -90;
  /* ebreak para parar a execução do processador assim que o programa for finalizado */
  asm(
    "ebreak\n\t"
  );
}
```

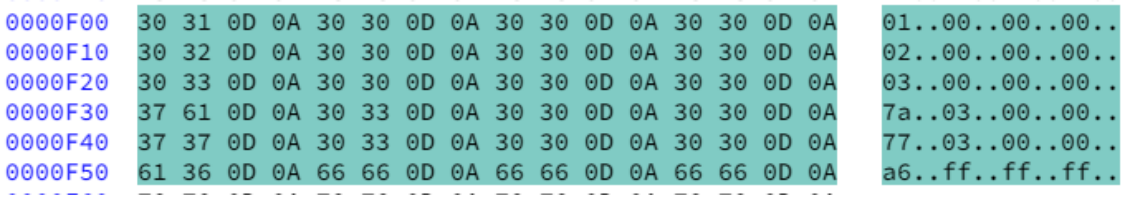
Novamente vale a pena analisar o arquivo .dump, para interpretarmos como é realizada a alocação de um vetor:

```
main:     file format elf32-littleriscv


Disassembly of section .text:

00000000 <main-0x4>:
   0:    3e800113              addi    x2,x0,1000
```

```
00000004 <main>:
   4:   fd010113          addi  x2,x2,-48
   8:   02812623          sw    x8,44(x2)
   c:   03010413          addi  x8,x2,48
  10:   00100793          addi  x15,x0,1
  14:   fcf42c23          sw    x15,-40(x8)
  18:   fd842783          lw    x15,-40(x8)
  1c:   00178793          addi  x15,x15,1
  20:   fcf42e23          sw    x15,-36(x8)
  24:   00300793          addi  x15,x0,3
  28:   fef42023          sw    x15,-32(x8)
  2c:   37a00793          addi  x15,x0,890
  30:   fef42223          sw    x15,-28(x8)
  34:   fe442703          lw    x14,-28(x8)
  38:   fe042783          lw    x15,-32(x8)
  3c:   40f707b3          sub   x15,x14,x15
  40:   fef42423          sw    x15,-24(x8)
  44:   fa600793          addi  x15,x0,-90
  48:   fef42623          sw    x15,-20(x8)
  4c:   00100073          ebreak
  50:   00000013          addi  x0,x0,0
  54:   00078513          addi  x10,x15,0
  58:   02c12403          lw    x8,44(x2)
  5c:   03010113          addi  x2,x2,48
  60:   00008067          jalr  x0,0(x1)
```

Como é possível perceber nas instruções, os *stores* estão sendo feitos sequencialmente, iniciando de um endereço mais baixo, e armazenando em sequência todos os elementos, isso indica que um vetor nada mais é que o valor contido em uma sequência de endereços.

Analisando o arquivo de memória para confirmar a previsão:



```
0000F00  30 31 0D 0A 30 30 0D 0A 30 30 0D 0A 30 30 0D 0A   01..00..00..00..
0000F10  30 32 0D 0A 30 30 0D 0A 30 30 0D 0A 30 30 0D 0A   02..00..00..00..
0000F20  30 33 0D 0A 30 30 0D 0A 30 30 0D 0A 30 30 0D 0A   03..00..00..00..
0000F30  37 61 0D 0A 30 33 0D 0A 30 30 0D 0A 30 30 0D 0A   7a..03..00..00..
0000F40  37 37 0D 0A 30 33 0D 0A 30 30 0D 0A 30 30 0D 0A   77..03..00..00..
0000F50  61 36 0D 0A 66 66 0D 0A 66 66 0D 0A 66 66 0D 0A   a6..ff..ff..ff..
```

Podemos confirmar então, que em C, um vetor nada mais é que uma sequência de variáveis.

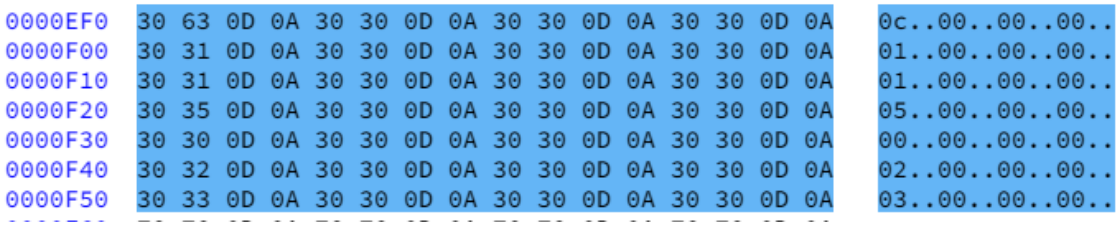
4.2.2.3 Teste: Operações

O terceiro teste realizado, foi um teste realizando operações aritméticas, entretanto, as operações como multiplicação, divisão e resto não estão disponíveis para RISC-V32I, sendo necessário uma implementação especial delas.

```c
/* Teste para funcionalidade básica: operações aritméticas*/
/* inicia o stack pointer em 1000, para realizar os testes */
asm(
  "addi sp, zero,1000\n\t"
);
int main () {
  int a, b, c, d, e, f, g, h;
  a = 3;
  b = 2;
  c = 0;

  d = a + b; /* d = 5*/
  e = a - b;  /* e = 1*/
  f = a >> (b-1); /* a = 011 >> 1 -> a = 001*/
  g = a << (b); /* a = 011 << 2 -> a = 1100 */

  /*
  funções não disponíveis / devem ser implementada de forma diferente:
  a = a * b;
  b = b / b;
  c = b / c;
  a = a * b * c;
```

```
 */
 /* ebreak para parar a execução do processador assim que o programa for
finalizado */
 asm(
   "ebreak\n\t"
 );
 return 0;
}
```

Neste caso não é agregativo verificar o arquivo .dump, pois todas estas operações possuem instruções próprias, e não há nada para descobrir. Analisando a memória para verificar se de fato os valores estão corretos:



Analisando este arquivo, além de perceber que as variáveis não estão armazenadas de forma sequencial na qual aparecem no programa, é possível identificar que todos os valores estão presentes.

4.2.2.4 Teste: Funções

O quarto teste realizado, foi a execução de um programa contendo uma função, novamente, para verificarmos se este aspecto está funcional no processador, e também para descobrir como se comporta uma função em hardware:

```
/* Teste para funcionalidade básica: Funções */
/* inicia o stack pointer em 1000, para realizar os testes */
asm(
  "addi sp,zero,1000\n\t"
);
int funcaoTeste(int, int);

int main(){
  int a, b, c;
```

```
  a = 4;
  b = 2;
  c = funcaoTeste(a, b);
  /* ebreak para parar a execução do processador assim que o programa for
finalizado */
  asm(
    "ebreak\n\t"
  );
}

int funcaoTeste(int a, int b){
  int c;
  c = 5;
  return a + b + c;
}
```

Desta vez vale a pena analisar o arquivo .dump, para verificarmos como uma função será interpretada:

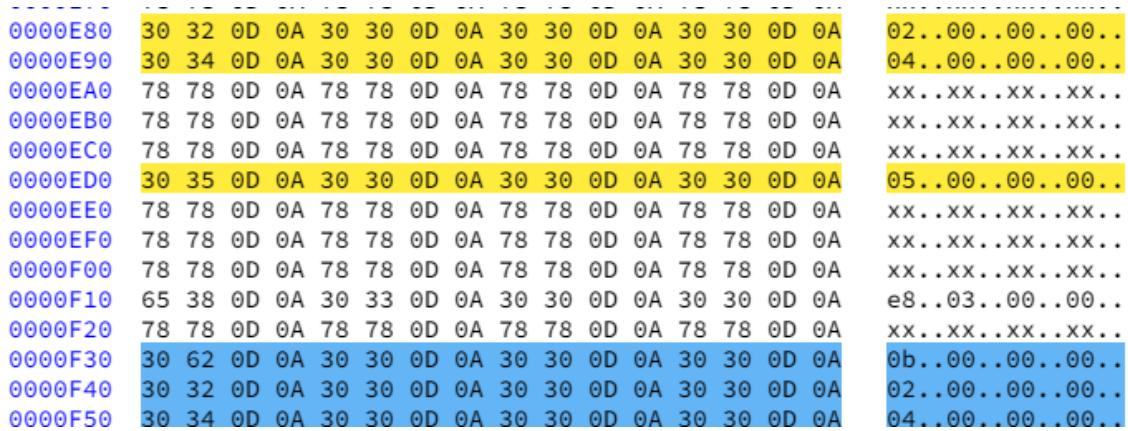
```
main:     file format elf32-littleriscv


Disassembly of section .text:

00000000 <main-0x4>:
   0:   3e800113                addi    x2,x0,1000

00000004 <main>:
   4:   fe010113                addi    x2,x2,-32
   8:   00112e23                sw      x1,28(x2)
   c:   00812c23                sw      x8,24(x2)
  10:   02010413                addi    x8,x2,32
  14:   00400793                addi    x15,x0,4
  18:   fef42623                sw      x15,-20(x8)
```

```
 1c:   00200793          addi   x15,x0,2
 20:   fef42423          sw     x15,-24(x8)
 24:   fe842583          lw     x11,-24(x8)
 28:   fec42503          lw     x10,-20(x8)
 2c:   00000097          auipc  x1,0x0
 30:   028080e7          jalr   x1,40(x1) # 54 <funcaoTeste>
 34:   fea42223          sw     x10,-28(x8)
 38:   00100073          ebreak
 3c:   00000013          addi   x0,x0,0
 40:   00078513          addi   x10,x15,0
 44:   01c12083          lw     x1,28(x2)
 48:   01812403          lw     x8,24(x2)
 4c:   02010113          addi   x2,x2,32
 50:   00008067          jalr   x0,0(x1)

00000054 <funcaoTeste>:
 54:   fd010113          addi   x2,x2,-48
 58:   02812623          sw     x8,44(x2)
 5c:   03010413          addi   x8,x2,48
 60:   fca42e23          sw     x10,-36(x8)
 64:   fcb42c23          sw     x11,-40(x8)
 68:   00500793          addi   x15,x0,5
 6c:   fef42623          sw     x15,-20(x8)
 70:   fdc42703          lw     x14,-36(x8)
 74:   fd842783          lw     x15,-40(x8)
 78:   00f70733          add    x14,x14,x15
 7c:   fec42783          lw     x15,-20(x8)
 80:   00f707b3          add    x15,x14,x15
 84:   00078513          addi   x10,x15,0
 88:   02c12403          lw     x8,44(x2)
 8c:   03010113          addi   x2,x2,48
 90:   00008067          jalr   x0,0(x1)
```

Observando o assembly, descobrimos que uma função nada mais é que um endereço na memória onde se inicia algum código, e utilizamos uma instrução do tipo

jump and link para ir para esta função, e ao completar a execução, utilizamos outro *jump and link* para voltarmos no programa principal.

Analisando a memória, podemos verificar as variáveis locais da função em amarelo, e as variáveis da função principal em azul:



4.2.2.5 Teste: Recursão

O quinto programa executado, foi um programa contendo uma recursão, novamente, para podermos explorar e descobrir como uma recursão é executada em hardware:

```
/* Teste para funcionalidade básica: Recursão */
/* inicia o stack pointer em 1000, para realizar os testes */
asm(
  "addi sp,zero,1000\n\t"
);
int multiplicainc(int, int);

int main(){
  int a, b;
  a = 5;
  b = 3;
  multiplicainc(a, b);
  /* ebreak para parar a execução do processador assim que o programa for finalizado */
  asm(
    "ebreak\n\t"
```

```
  );
}

int multiplicainc (int a, int b){
  int c = 2;
  if (a == 0)
    return c;
  return b + multiplicainc(a-1, b) + c;
}
```

Neste caso também vale a pena visualizar o arquivo .dump, para tentarmos visualizar como uma recursão é interpretada:

```
main:     file format elf32-littleriscv


Disassembly of section .text:

00000000 <main-0x4>:
   0:   3e800113                addi    x2,x0,1000

00000004 <main>:
   4:   fe010113                addi    x2,x2,-32
   8:   00112e23                sw      x1,28(x2)
   c:   00812c23                sw      x8,24(x2)
  10:   02010413                addi    x8,x2,32
  14:   00500793                addi    x15,x0,5
  18:   fef42623                sw      x15,-20(x8)
  1c:   00300793                addi    x15,x0,3
  20:   fef42423                sw      x15,-24(x8)
  24:   fe842583                lw      x11,-24(x8)
  28:   fec42503                lw      x10,-20(x8)
  2c:   00000097                auipc   x1,0x0
```

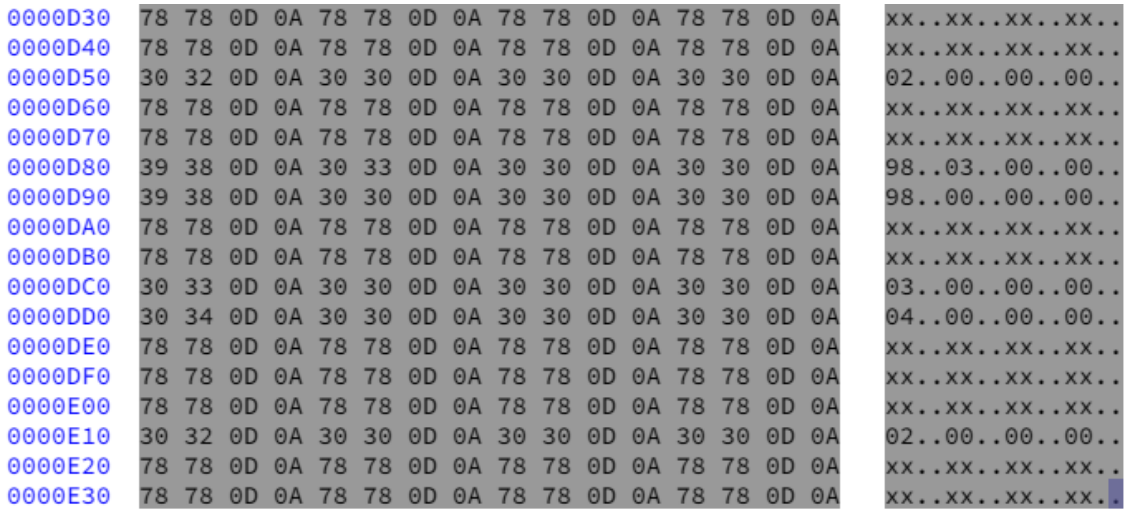
```
 30:   024080e7          jalr    x1,36(x1) # 50 <multiplicainc>
 34:   00100073          ebreak
 38:   00000013          addi    x0,x0,0
 3c:   00078513          addi    x10,x15,0
 40:   01c12083          lw      x1,28(x2)
 44:   01812403          lw      x8,24(x2)
 48:   02010113          addi    x2,x2,32
 4c:   00008067          jalr    x0,0(x1)

00000050 <multiplicainc>:
 50:   fd010113          addi    x2,x2,-48
 54:   02112623          sw      x1,44(x2)
 58:   02812423          sw      x8,40(x2)
 5c:   03010413          addi    x8,x2,48
 60:   fca42e23          sw      x10,-36(x8)
 64:   fcb42c23          sw      x11,-40(x8)
 68:   00200793          addi    x15,x0,2
 6c:   fef42623          sw      x15,-20(x8)
 70:   fdc42783          lw      x15,-36(x8)
 74:   00079663          bne     x15,x0,80 <multiplicainc+0x30>
 78:   fec42783          lw      x15,-20(x8)
 7c:   0300006f          jal     x0,ac <multiplicainc+0x5c>
 80:   fdc42783          lw      x15,-36(x8)
 84:   fff78793          addi    x15,x15,-1
 88:   fd842583          lw      x11,-40(x8)
 8c:   00078513          addi    x10,x15,0
 90:   00000097          auipc   x1,0x0
 94:   fc0080e7          jalr    x1,-64(x1) # 50 <multiplicainc>
 98:   00050713          addi    x14,x10,0
 9c:   fd842783          lw      x15,-40(x8)
 a0:   00f70733          add     x14,x14,x15
 a4:   fec42783          lw      x15,-20(x8)
 a8:   00f707b3          add     x15,x14,x15
```

```
ac:   00078513        addi   x10,x15,0
b0:   02c12083        lw     x1,44(x2)
b4:   02812403        lw     x8,40(x2)
b8:   03010113        addi   x2,x2,48
bc:   00008067        jalr   x0,0(x1)
```

Analisando o programa, podemos perceber que a cada chamada da função, ele apenas vai afastando o *stack pointer*, desta forma, a cada vez que a recursão é chamada, ele reserva o espaço para as variáveis declaradas dentro da função e também para o endereço de retorno, já que como qualquer função, devemos continuar a execução de onde paramos. Desta forma, uma chamada recursiva funciona primordialmente como qualquer chamada de função. Desta forma, também podemos concluir que, como o processador consegue executar funções, também não deve ter problemas com uma recursão.

Analisando a memória:

É possível identificar todas as variáveis locais da primeira chamada da recursão, e ainda o endereço de retorno na posição 0000E50, analisando ainda mais a fundo a memória:



Podemos identificar novamente a variável local da função, mas desta vez com 4 ao invés de 5, provando que a chamada recursiva realmente aconteceu, acompanhado do valor de retorno.

4.2.2.6 Teste: Heapsort

O programa mais notável que foi testado, foi um algoritmo de Heapsort, pois o mesmo faz grande uso de várias instruções, como jumps, branches, e operações aritméticas.

```
/* Teste refinado para teste de maior parte das funções: heapsort*/
/* inicia o stack pointer em 2000, para realizar os testes */
asm(
  "addi sp,zero,2000\n\t"
);

void heapfica (int v[], int n);
void heapsort (int v[], int n);
void rebaixa(int v[], int i, int n);
void sobe (int v[], int i, int n);
void troca (int v[], int i, int j);

int main(){
  int n;
  int v[10];
  n = 10;
```

```
  v[0] = 13;
  v[1] = -30;
  v[2] = 56;
  v[3] = -92;
  v[4] = 10;
  v[5] = 26;
  v[6] = 2;
  v[7] = -9;
  v[8] = 0;
  v[9] = 0;

  heapsort(v, n);
  /* ebreak para parar a execução do processador assim que o heapsort for
finalizado */
  asm(
    "ebreak\n\t"
  );
}

void troca (int v[], int i, int j){
  int aux = v[i];
  v[i] = v[j];
  v[j] = aux;
}

void heapfica (int v[], int n){
  int i;
  for (i = ((n-2) >> 1); i >= 0; i--)
    rebaixa(v, i, n);
}

void heapsort (int v[], int n){
  int i;
```

```
  heapfica(v,n);
  for (i=n-1; i>0; i--){
    troca (v, 0, i);
    rebaixa(v, 0, i);
  }
}

void rebaixa(int v[], int i, int n){
  int pai = i, filho = i + i + 1;

  while (filho < n){
    if (filho+1 < n && v[filho+1] > v[filho])
      filho = filho + 1;
    if (v[pai] > v[filho])
      break;
    troca (v, pai, filho);
    pai = filho;
    filho = 2 * pai + 1;
  }
}

void sobe (int v[], int i, int n){
  int filho = i, pai = (i-1) >> 1;
  while (filho > 0){
    if (v[pai] < v[filho]){
      troca (v,pai,filho);
      filho = pai;
      pai = (filho-1) >> 1;
    }
    else break;
  }
}
```

Neste caso também não interessante analisar o .dump, pois além de ser um programa extenso, não possui nada que já não foi descoberto.

Neste programa, utilizamos Heapsort para ordenar o seguinte vetor:

13	-30	56	-92	10	26	2	-9	0	0

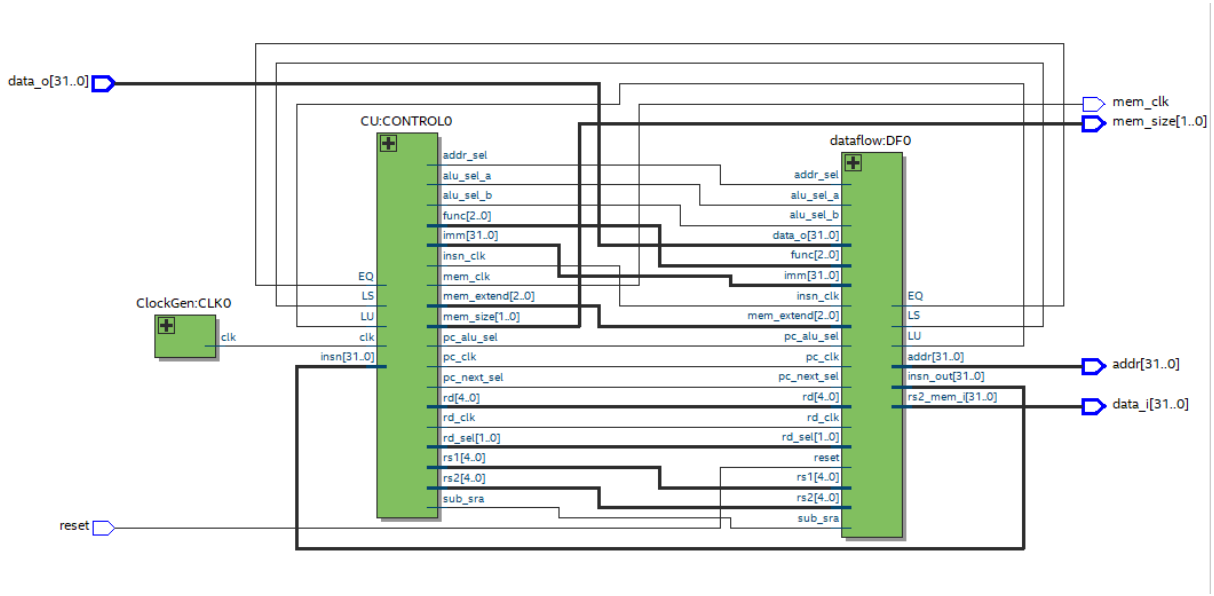
E novamente, para realizar a verificação, utilizamos o arquivo que contém o último estado da memória RAM, e o analisamos. Como visto anteriormente no programa contendo vetores, sabemos que nesta estrutura de dados, todos os valores estão em sequência na memória.

```
0001DF0  30 31 0D 0A 30 30 0D 0A 30 30 0D 0A 30 30 0D 0A   01..00..00..00..
0001E00  78 78 0D 0A 78 78 0D 0A 78 78 0D 0A 78 78 0D 0A   xx..xx..xx..xx..
0001E10  78 78 0D 0A 78 78 0D 0A 78 78 0D 0A 78 78 0D 0A   xx..xx..xx..xx..
0001E20  64 30 0D 0A 30 37 0D 0A 30 30 0D 0A 30 30 0D 0A   d0..07..00..00..
0001E30  37 38 0D 0A 30 30 0D 0A 30 30 0D 0A 30 30 0D 0A   78..00..00..00..
0001E40  78 78 0D 0A 78 78 0D 0A 78 78 0D 0A 78 78 0D 0A   xx..xx..xx..xx..
0001E50  61 34 0D 0A 66 66 0D 0A 66 66 0D 0A 66 66 0D 0A   a4..ff..ff..ff..
0001E60  65 32 0D 0A 66 66 0D 0A 66 66 0D 0A 66 66 0D 0A   e2..ff..ff..ff..
0001E70  66 37 0D 0A 66 66 0D 0A 66 66 0D 0A 66 66 0D 0A   f7..ff..ff..ff..
0001E80  30 30 0D 0A 30 30 0D 0A 30 30 0D 0A 30 30 0D 0A   00..00..00..00..
0001E90  30 30 0D 0A 30 30 0D 0A 30 30 0D 0A 30 30 0D 0A   00..00..00..00..
0001EA0  30 32 0D 0A 30 30 0D 0A 30 30 0D 0A 30 30 0D 0A   02..00..00..00..
0001EB0  30 61 0D 0A 30 30 0D 0A 30 30 0D 0A 30 30 0D 0A   0a..00..00..00..
0001EC0  30 64 0D 0A 30 30 0D 0A 30 30 0D 0A 30 30 0D 0A   0d..00..00..00..
0001ED0  31 61 0D 0A 30 30 0D 0A 30 30 0D 0A 30 30 0D 0A   1a..00..00..00..
0001EE0  33 38 0D 0A 30 30 0D 0A 30 30 0D 0A 30 30 0D 0A   38..00..00..00..
0001EF0  30 61 0D 0A 30 30 0D 0A 30 30 0D 0A 30 30 0D 0A   0a..00..00..00..
0001F00  78 78 0D 0A 78 78 0D 0A 78 78 0D 0A 78 78 0D 0A   xx..xx..xx..xx..
```

E de fato, analisando a memória, e lembrando que os valores estão armazenados em little-endian, isto é, os bytes mais significativos estão nos endereços mais altos, podemos perceber que os 10 números em sequência estão de fato ordenados.

5. FINALIZAÇÃO

O processador RISC-V 32I desse projeto foi inteiramente descrito em verilog. As sínteses do circuito foram feitas usando o programa quartus, com simulações feitas pelo Questa Intel FGPA. Seu design foi feito considerando a ISA do RV-32I, a qual estabelece um conjunto de instruções (e suas codificações) que o processador deve saber executar. O circuito completo sintetizado é apresentado a seguir:



Onde os inputs e outputs do processador se conectam com a memória. O clk gerado pelo bloco “ClockGen” foi deixado nos arquivos .v em um valor suficiente para fazer a análise dos testes executados. Pela análise de tempo feita no quartus, no relatório “clocks”, o processador por atingir um clk de 1000MHz. Para o modelo “slow 1100mV 85C”, a frequência máxima é de 27.81 MHz. O projeto possui espaço para melhorias, podendo ter alguns circuitos desenvolvidos de forma mais eficientes, como o regfile ou circuitos que não foram profundamente planejados como os shifter, extensor de memória e decodificador do imediato.

Em funcionamento, o circuito acima executa todas as funções previstas pela ISA, podendo executar programas escritos na linguagem C. Desse fato que é possível entender o poder de um processador, pois por ser programável ele pode executar diversas funções/ algoritmos que realizam tarefas úteis. Um dos exemplos testados foi o algoritmo de ordenação Heapsort, realizado para ordenar números presentes num vetor. Expansões podem ser feitas futuramente, como a de pontos flutuantes por exemplo, aumentando a quantidade de operações e, consequentemente, o poder do circuito de resolver problemas executando programas.