UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
LABORATÓRIO DE ORGANIZAÇÃO E ARQUITETURA DE COMPUTADORES

ESTUDANTE: GABRIELA MARQUES DE RANGEL MOREIRA (gmrm)
ESTUDANTE: JEFFERSON PEREIRA DE OLIVEIRA JÚNIOR (jpoj)
ESTUDANTE: LUIZ FELIPE PINTO ÁVILA DE BARROS (lfpab)
ESTUDANTE: THÉO MARCOS DO EGITO MOURA (tmem)

CURSO: ENGENHARIA DA COMPUTAÇÃO
DISCIPLINA CURSADA: LABORATÓRIO DE ORGANIZAÇÃO E ARQUITETURA DE COMPUTADORES

RELATÓRIO DO PROJETO DE RISC-V

Recife - PE
2025

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA

Relatório apresentado como parte projeto do Laboratório de Organização e Arquitetura de Computadores, sendo requisito obrigatório para fins avaliativos da referida disciplina.

Recife - PE
2025

SUMÁRIO

1. APRESENTAÇÃO

Nas páginas seguintes, será relatado o processo de criação do projeto de implementação de instruções RISC-V numa pipeline de cinco estágios, destrinchando seus módulos e visão geral. A atividade utilizou a plataforma ModelSim e linguagem SystemVerilog e é resultado dos aprendizados das disciplinas Organização e Arquitetura de Computadores e Laboratório de Organização e Arquitetura de Computadores.

2. VISÃO GERAL DO PROJETO

Inicialmente, para realizar as implementações das instruções, utilizamos a seguinte tabela para padronizar o código interno das operações no pipeline:

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

Como pode observar, algumas instruções não estão identificadas na imagem acima, como XOR e as de imediato, então nós mesmos definimos seus códigos para implementação, segue abaixo a tabela completa:

Instrução	Tipo	Opcode [6:0]	Funct3 [14:12]	Funct7 [31:25]	ALUOp (2 bits)	ALU Control Line (4 bits)
ADD	R	0110011	000	0000000	10	0010
SUB	R	0110011	000	0100000	10	0110
SLT	R	0110011	010	0000000	10	0111
XOR	R	0110011	100	0000000	10	0011
OR	R	0110011	110	0000000	10	0001
AND	R	0110011	111	0000000	10	0000
ADDI	I	0010011	000	-	10	0010
SLLI	I	0010011	001	0000000	10	0100
SLTI	I	0010011	010	-	10	0111
SRLI	I	0010011	101	0000000	10	0101
SRAI	I	0010011	101	0100000	10	1101
LB	I	0000011	000	-	00	0010
LH	I	0000011	001	-	00	0010
LW	I	0000011	010	-	00	0010
LBU	I	0000011	100	-	00	0010
SB	S	0100011	000	-	00	0010
SH	S	0100011	001	-	00	0010
SW	S	0100011	010	-	00	0010
BEQ	B	1100011	000	-	01	0110
BNE	B	1100011	001	-	01	0110
BLT	B	1100011	100	-	01	0110
BGE	B	1100011	101	-	01	0110
JALR	I	1100111	000	-	XX	0010
JAL	J	1101111	-	-	XX	XXXX

3. INSTRUÇÕES ARITMÉTICAS (ENTREGA 1)

Na Entrega 1, completamos o suporte do processador às instruções aritméticas fundamentais. A partir do código original que continha os comandos ADD e AND, implementamos SUB, OR, XOR e SLT, permitindo manipulações bit a bit e comparações numéricas robustas.

As principais alterações foram realizadas nos módulos ALUController.sv e alu.sv. No controlador, refinamos a decodificação para distinguir a instrução SUB da ADD através da análise do bit 30 no campo funct7, gerando sinais de controle distintos. Na ALU, ajustamos os operadores lógicos para garantir que XOR e OR atuassem em nível de bit e não como operadores booleanos globais.

Um foco específico foi dado ao tratamento de dados com sinal na instrução SLT. Introduzir a lógica de casting ($signed) nas entradas da unidade aritmética, assegurando que a comparação entre números em completo de dois fosse avaliada corretamente, corrigindo o comportamento padrão de comparação unsigned do SystemVerilog.

Por fim, a validação foi feita por meio de simulações no ModelSim com sequências de instruções disponibilizadas no material didático, cobrindo os test cases tanto de operações aritméticas usuais quanto as com números negativos. Os resultados obtidos coincidiram com o esperado e confirmaram a funcionalidade da entrega.

4. INSTRUÇÕES DE MEMÓRIA (ENTREGA 2)

Na Entrega 2, estendemos o suporte do processador às instruções de acesso à memória. A partir de um código base que já implementava apenas LW e SW, adicionamos as instruções LB, LH, LBU, SB e SH, permitindo operações com bytes e halfwords, com e sem extensão de sinal.

As principais alterações foram realizadas no módulo de memória de dados (datamemory.sv) e na unidade de controle. Em datamemory.sv, implementamos a seleção do byte ou halfword correto a partir dos bits menos significativos do endereço e a geração de masks de escrita, de forma que SB altere apenas um byte e SH altere dois bytes da palavra, preservando os demais bits. Para loads, introduzimos lógica de extensão de sinal ou de zero, controlada por sinais derivados do campo funct3 da instrução.

Na unidade de controle, estendemos a decodificação dos opcodes de load/store para gerar novos sinais indicando o tamanho do acesso (byte, halfword ou word) e o tipo de extensão a ser aplicado ao dado lido. O datapath do pipeline foi mantido, concentrando as mudanças no estágio MEM e no caminho de write back para os registradores.

A validação foi feita por meio de programas de teste em assembly, que exercitaram leituras e escritas de bytes e halfwords em diferentes posições dentro da palavra, além da diferença entre LB e LBU. Os resultados do nosso processador foram comparados com um simulador de referência RISC-V, confirmando a correção da implementação.

5. INSTRUÇÕES DE IMEDIATO (ENTREGA 3)

Na Entrega 3, a arquitetura do processador foi expandida para suportar operações fundamentais do tipo I (Imediato). O foco foi a implementação das instruções aritméticas ADDI e SLTI, bem como o conjunto completo de instruções de deslocamento: SLLI (lógico à esquerda), SRLI (lógico à direita) e SRAI(aritmético à direita).

No módulo ALUController, a decodificação exigiu um tratamento específico para distinguir os tipos de deslocamento à direita. Enquanto ADDI, SLTI e SLLI são identificadas apenas pelo campo *funct3*, as instruções SRLI e SRAI compartilham o mesmo código neste campo. A diferenciação foi realizada através da análise do bit 30 do campo *funct7*, permitindo ao controlador sinalizar corretamente à ALU quando preservar o sinal (aritmético) ou preencher com zeros (lógico).

No caminho de dados (alu.sv), a implementação priorizou o tratamento correto de sinais. Para a instrução SRAI, foi empregado o operador de deslocamento aritmético em conjunto com o *casting* para *signed*, garantindo que números negativos mantivessem sua integridade durante o deslocamento. Similarmente, a instrução SLTI reutilizou a lógica de comparação com sinal, assegurando que a avaliação entre o registrador fonte e o imediato refletisse corretamente a magnitude em complemento de dois.

A validação funcional foi conduzida via simulação no ModelSim. Os testes confirmaram que a instrução ADDO realizou somas corretas com imediatos negativos e que a instrução SLTI avaliou precisão nas comparações. Além disso, verificou-se que SRAI propagou corretamente o bit de sinal, diferindo do comportamento de SRLI, validando a robustez da decodificação e execução deste conjunto de instruções.

6. INSTRUÇÕES DE SALTO (ENTREGA 4)

Na Entrega 4, completamos o suporte do processador a controle de fluxo, implementando as instruções de branch BNE, BLT e BGE, além das instruções de salto JAL e JALR, sobre uma base que já possuía BEQ. Isso permitiu representar laços, condicionais e chamadas de função de forma compatível com o conjunto RV32I.

As principais modificações envolveram a unidade de controle, o gerador de imediatos e a lógica de atualização do PC. A unidade de controle passou a decodificar os novos opcodes e campos funct3, produzindo sinais que parametrizam um comparador de branch (igual, diferente, menor e maior/igual, com sinal) e um seletor de destino de salto (PC relativo ou baseado em rs1 + imediato). O módulo de geração de imediatos foi extendido para suportar os formatos B-type e J-type, usados em branches e em JAL, além do imediato I-type empregado em JALR.

Na lógica do PC, foi adicionado um multiplexador que seleciona entre três fontes: PC + 4 (fluxo sequencial), PC + imm (branches e JAL) e (rs1 + imm) & ~1 (destino alinhado de JALR). A decisão de branch/jump é tomada no estágio EX do pipeline; quando o desvio é tomado, o PC é atualizado com o novo alvo e instruções especulativamente buscadas são descartadas, garantindo a correção da execução mesmo sem predição de desvio.

Testamos essas extensões com programas que implementam laços de soma, comparações condicionais (if/else) e chamadas/retornos de funções usando JAL e JALR. Em todos os casos, os valores dos registradores e o fluxo de controle observados no nosso processador coincidiram com os resultados obtidos no simulador de referência, validando a implementação.

7. PSEUDO-INSTRUÇÃO HALT (ENTREGA 5)

Nesta fase final de verificação e testes, foi introduzido o suporte à pseudo-instrução HALT, um mecanismo essencial para delimitar o término da execução de programas em um ambiente sem sistema operacional. Diferentemente das etapas anteriores, esta implementação não exigiu alterações físicas no *datapath* ou na Unidade de Controle, demonstrando a flexibilidade do conjunto de instruções RISC-V já implementado.

A estratégia adotada consistiu em mapear o comando HALT para a instrução nativa JAL, configurada com o registrador de destino x0 e um deslocamento de zero. Esta configuração instrui o processador a calcular o próximo endereço de execução como sendo igual ao endereço atual, criando um laço infinito em nível de instrução. Como o registrador x0 é zero, a tentativa de salvar o endereço de retorno (PC + 4) é ignorada pelo hardware, sem causar efeitos colaterais no estado do banco de registradores.

Esta abordagem permitiu "congelar" o estado do processador de forma eficaz, mantendo o sinal de *clock* ativo e o pipeline cheio, mas impedindo o avanço do PC para regiões de memória não inicializadas. Isso eliminou a necessidade de implementar lógica complexa ou sinais de controle dedicados apenas para interromper a simulação.

A validação foi realizada através da análise dos logs de transição no ModelSim. Confirmou-se o funcionamento correto ao observar que, após a execução do último comando válido do programa de teste, o processador entrou em um estado estável onde o endereço do PC permaneceu constante. O *transcript* da simulação registrou tentativas repetitivas de escrita no registrador x0 no mesmo ciclo de tempo relativo, comprovando que o processador havia entrado no laço infinito intencional, validando assim o mecanismo de parada.

8. CONCLUSÃO

Em conclusão, o projeto alcançou êxito no desenvolvimento de um processador RISC-V funcional, sendo capaz de executar 24 instruções que cobrem operações aritméticas, lógicas, de acesso à memória e de controle de fluxo, além da a pseudo-instrução HALT. Por meio da sincronização entre a unidade de controle e o caminho de dados e, após rigorosas etapas de teste no ModelSim, o sistema se demonstrou seguro e funcional. Por fim, a equipe obteve aprendizados práticos sobre o funcionamento de uma CPU já com comandos de gerenciamento, um avanço em relação a CPU feita na disciplina Sistemas Digitais anteriormente.