Projeto “Processador RISC-V com Pipeline”: Relatório

Participantes

(Equipe 4)

Titho Lívio Duarte Melo	<tldm>
Edivaldo Ambrozio da Silva Filho	<easf>
Gustavo Cravo Teixeira Filho	<gctf>
Heitor de Assis Machado	<ham5>
Guilherme de Carvalho Fabri	<gcf2>

Fork do repositório

(link do projeto): https://github.com/Sencorpion/ProjetoRiscV-ArqComp_2025.2-

Descrição das escolhas do projeto:

O processador original era mínimo. A alu só realizava ADD, AND e EQ. O Controller só reconhecia os opcodes R_TYPE, LW, SW, e BR. O datamemory só suportava LW e SW. O MemtoReg era de 1 bit. Não havia suporte para a maioria das operações aritméticas, lógicas, de shift, de desvio ou de acesso a sub-palavras na memória.

Implementação de Novas Instruções e Reestruturações necessárias:

1. Instruções Aritméticas e Lógicas (R-Type)

- Instruções Adicionadas: SUB, OR, XOR, SLT

- Análise da Implementação: Essas instruções R-Type reutilizam o datapath existente, mas exigem que a alu e seu controlador sejam expandidos.

- Comparativo e Alterações:

alu.sv: A ALU era muito simples. Foi necessário adicionar ‘cases’ para as novas operações: SUB (4‘b0100), OR (4‘b0001), XOR (4‘b0011), e SLT (4‘b0101).

ALUController.sv: O ALUController original era uma série de assigns complexos. Ele foi completamente reestruturado para um formato ‘case’ aninhado, que é muito mais legível e extensível. Dentro do case para ALUOp = 2‘b10 (R-Type), foram adicionados sub-cases para cada funct3 correspondente a SUB, OR, XOR e SLT, gerando os Operation codes corretos para a nova ALU. A lógica para ADD/SUB que usa funct7 também foi implementada aqui.

2. Instruções Aritméticas e Lógicas com Imediato (I-Type)

- Instruções Adicionadas: ADDI, SLTI, SLLI, SRLI, SRAI

- Análise da Implementação: Introduziram a necessidade de um ALUOp para I-Type e expandiram significativamente o imm_Gen.

- Comparativo e Alterações:

imm_Gen.sv: O design antigo não reconhecia o opcode IMM (7‘b0010011). Foi adicionado um ‘case’ para este opcode para extrair o imediato I-Type.

Controller.sv: A lógica foi expandida para reconhecer o opcode IMM. O sinal ALUSrc foi modificado para ser 1 para IMM, e RegWrite também foi ativado. Mais importante, o ALUOp foi modificado para gerar 2‘b11 para instruções I-Type.

alu.sv: Foi expandido com as operações de shift: SLLI (4‘b0110), SRLI (4‘b0111), e SRAI (4‘b1001).

ALUController.sv: Adicionado um ‘case’ principal para ‘ALUOp = 2‘b11’ (I-Type), com sub-cases baseados no funct3 para selecionar as operações de ADDI, SLTI, e as novas operações de shift.

3. Instruções de Desvio Condicional

- Instruções Adicionadas: BNE, BLT, BGE

- Análise da Implementação: Utilizam a infraestrutura do BEQ do design antigo, mas com novas condições de comparação.

- Comparativo e Alterações:

alu.sv: A ALU foi expandida para incluir as operações de comparação BNE (4‘b1100, !=) e BGE (4‘b1010, >=). A SLT (<) é reutilizada para a instrução BLT.

ALUController.sv: A lógica para ALUOp = 2‘b01 (Branch) foi alterada de um simples assign para um ‘case’ aninhado baseado no ‘funct3’. Isso permite decodificar BEQ, BNE, BLT, e BGE e enviar o Operation code correto para a alu.

BranchUnit.sv e Datapath.sv: Nenhuma alteração foi necessária nestes módulos para estas instruções, pois a lógica Branch_Sel = Branch && AluResult[0] genericamente lida com qualquer condição que a ALU possa verificar.

4. Instruções de Acesso à Memória (Sub-palavra)

- Instruções Adicionadas: LB, LH, LBU (Loads) e SB, SH (Stores)

- Análise da Implementação: Exigiu uma reengenharia completa do datamemory para suportar acesso a bytes e meias-palavras.

- Comparativo e Alterações:

datamemory.sv: Este módulo sofreu a mudança mais significativa para estas instruções. O design antigo apenas lia ou escrevia palavras inteiras. O always_ff foi expandido com 'case' statements aninhados:

- Para Loads: A lógica agora usa funct3 para diferenciar LW, LH, LB, LBU. Para LH e LB, um sub-case no a[1:0] seleciona os bytes corretos da Dataout de 32 bits e realiza a extensão de sinal. Para LBU, realiza a extensão com zero.

- Para Stores: A lógica usa funct3 para diferenciar SW, SH, SB. Para SH e SB, um sub-case no a[1:0] gera o sinal Wr de 4 bits. Este sinal Wr é a chave, pois permite habilitar a escrita em bancos de memória de 8 bits individuais dentro do Memoria32Data, efetivamente permitindo a escrita de apenas um byte ou meia-palavra.

5. Instruções de Desvio Incondicional com Link

- Instruções Adicionadas: JAL, JALR

- Análise da Implementação: Esta foi a alteração mais complexa do datapath, introduzindo um novo caminho de dados e nova lógica de controle de desvio.

- Comparativo e Alterações:

Controller.sv: O design antigo não tinha como lidar com o Link. Foi necessário:
- Alargar ‘MemtoReg’ de 1 para 2 bits.
- Criar novos sinais de saída Jump e JumpR.
- Modificar a lógica de MemtoReg para gerar 2‘b10 quando JAL ou JALR são detectados.

RegPack.sv: As structs (id_ex_reg, ex_mem_reg, mem_wb_reg) foram atualizadas para carregar o MemtoReg de 2 bits e os novos sinais Jump e JumpR.

Datapath.sv:
- O mux2 no final do pipeline (resmux) foi substituído por um ‘mux4’. Esta é a mudança de hardware mais crítica, criando um caminho físico do D.Pc_Four para a entrada de dados do RegFile.
- Os novos sinais Jump e JumpR foram adicionados como entradas e conectados ao BranchUnit.

BranchUnit.sv: Foi significativamente melhorado.
- Recebe Jump e JumpR. A lógica de PcSel foi expandida para Branch_Sel || Jump || JumpR.
- A lógica de BrPC tornou-se um case implícito: se Jump, usa PC_Imm, se JumpR, usa o AluResult com LSB zerado.

RISC_V.sv: Os fios e conexões entre Controller e Datapath foram atualizados para os novos sinais.

6. Instrução de Parada

- Instrução Adicionada: HALT

- Análise da Implementação: Implementada como um loop infinito PC = PC.

- Comparativo e Alterações:

Controller.sv: Adicionado um opcode customizado (7‘b1111111) e um novo sinal halt.

BranchUnit.sv: O sinal Halt foi adicionado como entrada e recebeu a maior prioridade na lógica de BrPC e PcSel, forçando um salto para o endereço atual.

Datapath.sv, RegPack.sv, RISC_V.sv: Modificados para rotear o novo sinal Halt do Controller até o BranchUnit.

Descrição dos testes realizados

Teste das instruções aritméticas e de comparação
addi x1, x0, 10 Soma o valor de x0 com o imediato 10 | x1 = 10
addi x2, x0, 5 Soma o valor de x0 com o imediato 5 | x2 = 5
sub x3, x1, x2 Subtrai x2 de x1 | x3 = 5
sub x4, x2, x1 Subtrai x1 de x2 | x4 = -5
addi x5, x3, 15 Soma x3 com o imediato 15 | x5 = 20
addi x6, x5, -1 Soma x5 com o imediato -1 | x6 = 19
slt x7, x2, x1 Verifica se x2 < x1.Como é Verdadeiro, define como 1 | x7 = 1
slt x8, x1, x2 Verifica se x1 < x2.Como é Falso, define como 0 | x8 = 0
slti x9, x4, 5 Verifica se x4 < 5.Como é Verdadeiro, define como 1 | x9 = 1
slti x10, x5, 20 Verifica se x5 < 20. Como é Falso, define como 0 | x10 = 0

Teste das instruções lógicas
addi x1, x0, 547 Soma o valor de x0 com o imediato 547 | x1 = 547
addi x2, x0, -2 Soma o valor de x0 com o imediato -2 | x2 = -2
addi x3, x0, 2047 Soma o valor de x0 com o imediato 2047 | x3 = 2047
or x4, x1, x3 Operação OR bit a bit entre x1 e x3 | x4 = 2047
slli x5, x4, 4 Desloca logicamente x4 à esquerda 4 bits | x5 = 32752
srli x6, x2, 4 Desloca logicamente x2 à direita 4 bits | x6 = 268435455
srai x7, x2, 4 Desloca aritmeticamente x2 à direita 4 bits | x7 = -1
slli x8, x1, 0 Desloca logicamente x1 à esquerda 0 bits | x8 = 547

Teste das instruções de acesso à memória
addi x1, x0, 1547 Soma o valor de x0 com o imediato 1547 | x1 = 1547
sh x1, 0(x0) Armazena a meia palavra de x1 na memória endereço 0 | Mem[0] = 1547
sb x1, 4(x0) Armazena o primeiro byte de x1 na memória endereço 4 | Mem[4] = 11
lh x2, 0(x0) Carrega meia palavra da memória 0 para x2 | x2 = 1547
lb x3, 0(x0) Carrega um byte da memória 0 para x3 | x3 = 11
lh x4, 4(x0) Carrega meia palavra da memória 4 para x4 | x4 = 11
lb x5, 4(x0) Carrega um byte da memória 4 para x5 | x5 = 11
addi x6, x0, -10 Soma o valor de x0 com o imediato -10 | x6 = -10
sw x6, 8(x0) Armazena a palavra completa de x6 na memória endereço 8 | Mem[8] = -10
lbu x7, 8(x0) Carrega um byte sem sinal (unsigned) da memória 8 para x7 | x7 = 246

Teste das instruções de desvio

addi x1, x0, 10 Soma o valor de x0 com o imediato 10 | x1 = 10
addi x2, x0, 20 Soma o valor de x0 com o imediato 20 | x2 = 20
addi x3, x0, 10 Soma o valor de x0 com o imediato 10 | x3 = 10
bne x1, x2, 8 10 ≠ 20 (Verdadeiro). Salta 8 bytes | Salto realizado
addi x4, x0, 999 Instrução pulada | (sem alteração)
addi x4, x0, 1 Executada após o salto. Define x4 como 1 | x4 = 1
bne x1, x3, -24 10 ≠ 10 (Falso). Não salta | Salto não realizado
addi x5, x0, 1 Execução normal. Define x5 como 1 | x5 = 1
blt x1, x2, 8 10 < 20 (Verdadeiro). Salta 8 bytes | Salto realizado
addi x6, x0, 999 Instrução pulada | (sem alteração)
addi x6, x0, 1 Executada após o salto. Define x6 como 1 | x6 = 1
bge x1, x2, -44 10 ≥ 20 (Falso). Não salta | Salto não realizado
addi x7, x0, 1 Execução normal. Define x7 como 1 | x7 = 1
jal x15, 12 Salta 12 bytes (pula 2 instr.). Salva retorno em x15 | Salto realizado
addi x8, x0, 999 Instrução pulada | (sem alteração)
addi x21, x0, 1 Instrução pulada (x21 continua 0) | x21 = 0
addi x8, x0, 1 Aterrissagem do JAL (define x8) | x8 = 1
beq x21, x4, 12 0 == 1 (Falso). Não salta | Salto não realizado
jalr x16, x15, 4 Pula para x15+4 (Endereço do addi x21). | Salto realizado
addi x21, x0, 1 Executada agora via JALR | x21 = 1
addi x8, x0, 1 Re-executada | x8 = 1
beq x21, x4, 12 1 == 1 (Verdadeiro). Salta 12 bytes | Salto realizado
jalr x16, x15, 4 Instrução pulada (devido ao BEQ) | (sem alteração)
addi x9, x0, 999 Instrução pulada (devido ao BEQ) | (sem alteração)
addi x20, x0, -1 Aterrissagem final do BEQ | x20 = -1

Teste do halt

addi x1, x0, 10 Soma o valor de x0 com o imediato 10 | x1 = 10
addi x2, x0, 20 Soma o valor de x0 com o imediato 20 | x2 = 20
add x3, x1, x2 Soma o valor de x1 com x2 (10 + 20) | x3 = 30
halt Interrompe a execução do processador | (Execução terminada)
addi x5, x0, 999 Instrução não executada (processador parado) | (sem alteração)
add x6, x5, x5 Instrução não executada (processador parado) | (sem alteração)

Resultados obtidos

Resultado do teste das instruções aritméticas e de comparação

```
 45: Register [ 1] written with value: [0000000a] | [        10]
 55: Register [ 2] written with value: [00000005] | [         5]
 65: Register [ 3] written with value: [00000005] | [         5]
 75: Register [ 4] written with value: [fffffffb] | [4294967291]
 85: Register [ 5] written with value: [00000014] | [        20]
 95: Register [ 6] written with value: [00000013] | [        19]
105: Register [ 7] written with value: [00000001] | [         1]
115: Register [ 8] written with value: [00000000] | [         0]
125: Register [ 9] written with value: [00000001] | [         1]
135: Register [10] written with value: [00000000] | [         0]
```

Resultado do teste das instruções lógicas

```
 45: Register [ 1] written with value: [00000223] | [       547]
 55: Register [ 2] written with value: [fffffffe] | [4294967294]
 65: Register [ 3] written with value: [000007ff] | [      2047]
 75: Register [ 4] written with value: [000007ff] | [      2047]
 85: Register [ 5] written with value: [00007ff0] | [     32752]
 95: Register [ 6] written with value: [0fffffff] | [ 268435455]
105: Register [ 7] written with value: [ffffffff] | [4294967295]
115: Register [ 8] written with value: [00000223] | [       547]
```

Resultado do teste das instruções de acesso à memória

```
 45: Memory [  0] written with value: [0000060b] | [      1547]
 45: Register [ 1] written with value: [0000060b] | [      1547]
 55: Memory [  4] written with value: [0000060b] | [      1547]
 65: Memory [  0] read with value: [xxxxxxxx] | [         x]
 65: Memory [  0] read with value: [00000000] | [         0]
 70: Memory [  0] read with value: [0000060b] | [      1547]
 75: Register [ 2] written with value: [0000060b] | [      1547]
 75: Memory [  0] read with value: [0000000b] | [        11]
 85: Memory [  4] read with value: [0000000b] | [        11]
 85: Register [ 3] written with value: [0000000b] | [        11]
 85: Memory [  4] read with value: [0000060b] | [      1547]
 90: Memory [  4] read with value: [0000000b] | [        11]
 95: Register [ 4] written with value: [0000000b] | [        11]
105: Memory [  4] read with value: [0000000b] | [        11]
115: Register [ 5] written with value: [0000000b] | [        11]
125: Memory [  8] written with value: [fffffff6] | [4294967286]
125: Register [ 6] written with value: [fffffff6] | [4294967286]
135: Memory [  8] read with value: [0000000b] | [        11]
135: Memory [  8] read with value: [00000000] | [         0]
140: Memory [  8] read with value: [000000f6] | [       246]
145: Register [ 7] written with value: [000000f6] | [       246]
```

Resultado do teste das instruções de desvio

```
  45: Register [ 1] written with value: [0000000a] | [        10]
  55: Register [ 2] written with value: [00000014] | [        20]
  65: Register [ 3] written with value: [0000000a] | [        10]
 105: Register [ 4] written with value: [00000001] | [         1]
 125: Register [ 5] written with value: [00000001] | [         1]
 165: Register [ 6] written with value: [00000001] | [         1]
 185: Register [ 7] written with value: [00000001] | [         1]
 195: Register [15] written with value: [00000038] | [        56]
 225: Register [21] written with value: [00000001] | [         1]
 235: Register [ 8] written with value: [00000001] | [         1]
 275: Register [20] written with value: [ffffffff] | [4294967295]
```

Resultado do teste do halt

```
  45: Register [ 1] written with value: [0000000a] | [        10]
  55: Register [ 2] written with value: [00000014] | [        20]
  65: Register [ 3] written with value: [0000001e] | [        30]
```

Dificuldades encontradas

Implementação da Lógica de Link (JAL e JALR)

- Dificuldade: A maior dificuldade conceitual foi a implementação da operação “Link” (rd = PC + 4) das instruções JAL e JALR. O design original não possuía um caminho de dados que permitisse que o valor PC + 4 (calculado no início do pipeline) chegasse até a porta de escrita do banco de registradores no estágio final de Write-Back.

- Análise do Problema: Percebeu-se que o multiplexador final (resmux) só podia escolher entre o resultado da ALU e os dados da memória, controlado por um sinal MemtoReg de apenas 1 bit. Era impossível selecionar uma terceira fonte de dados.

- Solução: A solução exigiu uma alteração estrutural significativa no datapath:

1. Alargamento de MemtoReg: O sinal MemtoReg foi estendido para 2 bits no Controller e em todos os registradores do pipeline (RegPack.sv), criando um novo estado (2‘b10) para representar a fonte “PC+4”.

2. Substituição do Mux: O mux2 no estágio de Write-Back foi substituído por um mux4, e a saída D.Pc_Four do último registrador do pipeline foi conectada a uma das novas entradas desse multiplexador.

Diferenciação e Otimização entre JAL e JALR

- Dificuldade: Inicialmente, houve a tentação de tratar JAL e JALR de forma idêntica, usando a ALU principal para ambos os cálculos de endereço de desvio. Isso teria funcionado, mas não seria eficiente.

- Análise do Problema: Uma análise mais profunda do BranchUnit revelou que ele já continha um somador que calculava PC + Imm para as instruções de branch. Usar a ALU principal para o JAL seria redundante e um desperdício de recursos.

- Solução: A solução adotada foi mais elegante:

1. Para JAL: Reutilizou-se o somador interno do BranchUnit. Um novo sinal Jump foi criado para comandar o BranchUnit a usar seu próprio resultado PC_Imm.

2. Para JAL: A ALU principal foi utilizada, pois a operação (rs1 + imm) exigia acesso ao banco de registradores (rs1), algo que o BranchUnit não possui. Um sinal JumpR foi criado para comandar o BranchUnit a usar o AluResult como fonte para o desvio.

Implementação de uma Instrução HALT Robusta

- Dificuldade: A primeira tentativa de implementar a instrução HALT foi através de um "stall" do Program Counter.

- Análise do Problema: Testes e análises teóricas mostraram que paralisar o PC apenas enquanto a instrução HALT estivesse no estágio de Decode era uma solução falha. Assim que a instrução avançasse no pipeline, o PC seria liberado e o programa continuaria a execução indevidamente.

- Solução: A solução correta e muito mais robusta foi criar um "auto-loop".
  1. A BranchUnit foi modificada para que, ao receber o sinal Halt, ela forçasse a seleção do PC (PcSel=1) e definisse o endereço de desvio como o próprio endereço da instrução HALT (BrPC = PC).
  2. Isso prende o processador em um loop PC = PC, onde ele busca e executa a mesma instrução HALT indefinidamente, parando efetivamente o avanço do programa.

Gerenciamento de Sinais de Controle Através do Pipeline

- Dificuldade: Conforme novas instruções (JAL, JALR, HALT) foram adicionadas, múltiplos sinais de controle (Jump, JumpR, Halt, MemtoReg de 2 bits) precisaram ser gerados no estágio de Decode e utilizados em estágios posteriores (Execute ou Write-Back). No início, houve erros de compilação devido a conexões ausentes ou fios com larguras incorretas.

- Análise do Problema: Os erros apontavam para portas de módulo não conectadas ou com larguras de bit incompatíveis entre o Controller, Datapath, e o RISC_V de topo.

- Solução: Foi necessário um trabalho meticuloso de "encanamento" do pipeline:
  1. Adicionar os novos sinais às structs dos registradores no RegPack.sv.
  2. Garantir que cada always block do Datapath que controla um registrador de pipeline estivesse corretamente salvando (latcheando) e limpando (em caso de flush) esses novos sinais.
  3. Atualizar as declarações de porta e as instanciações de módulo no Datapath.sv e RISC_V.sv para refletir os novos sinais.

Conclusão

A execução deste projeto resultou na evolução bem-sucedida de um núcleo RISC-V minimalista para um processador robusto, agora capaz de suportar um set de instruções expandido, incluindo operações lógicas, aritméticas e de acesso à memória em nível de sub-palavra.

O principal desafio técnico superado foi a implementação das instruções de controle de fluxo (JAL, JALR e HALT). A solução exigiu alterações estruturais críticas no datapath, especificamente a substituição do multiplexador final por um mux4 e o alargamento dos sinais de controle (MemtoReg) para permitir o link correto no estágio de Write-Back. Além disso, a estratégia de "auto-loop" para o HALT garantiu uma parada de execução segura e previsível.

Por fim, a bateria de testes realizada validou a integridade das modificações, confirmando que o gerenciamento de sinais através dos estágios do pipeline foi implementado corretamente, sem comprometer a eficiência do design original.