CIn / Universidade Federal de Pernambuco

Relatório do projeto Risc-V Pipeline

Laboratório de OAC

André Castro Polo Norte Filho <acpnf>
Ikelvys Kauê Vitorino Costa <ikvc>
João Pedro Lima de Araujo <jpla>
Pablo Nunes de Oliveira <pno>

Sumário:

Introdução

Este projeto teve como objetivo implementar um processador RISC-V (RV32I) em System Verilog baseado na arquitetura pipeline, utilizando conceitos de Organização e Arquitetura de Computadores apresentados em sala de aula.

O desenvolvimento permitiu compreender na prática as principais estruturas internas de um processador RISC, incluindo o fluxo de execução de instruções,unidade de controle, ALU, banco de registradores, estágios de pipeline, entre outros.

Este relatório descreve a metodologia utilizada, a implementação dos módulos, os testes realizados e os resultados obtidos, finalizando com uma discussão sobre limitações e possíveis melhorias.

Implementação

As implementações seguiram a ordem definida nas parciais. Em cada etapa, acrescentamos o código necessário para suportar um conjunto de instruções ou um novo comportamento no pipeline.

Parcial 1 – Operações aritméticas e lógicas

Nesta etapa, implementamos as seguintes operações:

- SUB, OR, XOR, SLT

Para realizar essas implementações, tivemos que incrementar/modificar os módulos ALU e ALUController.

- Na implementação do SUB, o ALUController precisa enviar o sinal de operação correto para a ALU. O sinal designado para a SUB é o 0110.
- Seguimos a mesma ideia para os demais comandos, o sinal para OR é 0001, o do XOR é 0011, e o de SLT 1100.

Parcial 2 – Acesso à memória

Nesta entrega, implementamos corretamente as instruções de load e store responsavel pelo acesso e escrita na memória:

- LH, LB, LBU
- SH, SB

Para isso, ajustamos:

- LW e SW já vieram implementados no código base.
- Para as instruções do tipo Load, modificamos o Data Memory. As instruções utilizam um registrador de base e um valor imediato como deslocamento para criar o endereço de leitura.
- Já para as instruções do tipo Store, também alteramos o datamemory, selecionando se salvaremos metade da palavra (SH) ou apenas o byte (SB)
- Os sinais MemRead e MemWrite (vem da unidade de controle, para determinar o tipo de acesso à memória)
- a passagem do endereço da ALU para a memória. (comandos de Read Data(rd) e Write Data(wd))

Parcial 3 – Instruções imediatas e Deslocamento

Nessa parcial implementamos essas instruções:

- ADDI, SLTI, SLLI, SRAI, SRLI

Para isso:

- Modificamos o imm_Gen, para extrair e estender corretamente os valores dos imediatos de cada tipo de instrução.
- A unidade de controle,
- A ALU para aceitar operações com imediato.

No Imm_Gen, implementamos tipos de :

- Tipo I (Imediatos - loads, JALR, e aritmética) :

  O Imediato dessas operações geralmente estão nos primeiros 12 bits mais significativos [31:20], A extensão de sinal é aplicada de acordo com o bit mais significativo da instrução, aplicando nos últimos 20 bits do imediato, para também garantir operações negativas.

- Tipo S (Stores):

  Os 12 bits do imediato das operações de stores são divididos em duas partes, [31:25] e [11:7]. O Imm_Gen concatena os bits e aplica a extensão de sinal.

- Tipo B (Branch):

  Para o tipo B, o imediato fica embaralhado na instrução, e tem um bit a mais formando 13 bits, no qual o bit menos significativo é 0, então o módulo destrincha os bits e concatena corretamente. Os bits são organizados da seguinte forma:
  Imm[12] = inst[31]; imm[10:5] = inst[30:25]; imm[4:1] = instr[11:8], imm[11] = inst[7]; imm[0] = bit 0 fixo.

- Tipo J (Jumps):

  Os imediatos do tipo J também tem bits espalhados na instrução, porém o imediato possui 21 bits, sendo organizado da seguinte maneira;
  imm[20] = instr[31]; imm[10:1] = instr[30:21]; imm[11] = instr[20]; imm[19:12] = instr[19:12]; imm[0] com o bit 0 fixo.

Parcial 4 – Desvios e jumps

- Esta foi a etapa mais desafiadora do projeto para o grupo. Tivemos dificuldades iniciais com a lógica de controle e enfrentamos diversos erros de simulação, mas conseguimos implementar com sucesso o funcionamento das instruções designadas: BNE, BLT, BGE, JAL e JALR.

- A lógica geral de um processador executa sequencialmente, onde o PC é igual ao PC mais 4. Mas, as instruções de jump e desvios quebram essa regra, forçando o multiplexador de busca a escolher um novo valor calculado. Para viabilizar isso, a implementação envolveu alterações em três módulos principais: o Controller, a Branch Unit e o Datapath.

- Primeiramente, no módulo Controller, precisamos ensinar o processador a separar os novos opcodes. Definimos as constantes para JAL e JALR e criamos sinais de controle específicos que não existiam no projeto original. Implementamos os sinais jmp e jmpr para indicar, respectivamente, instruções de salto incondicional direto e indireto. Além disso, criamos o sinal jmp_sel. Este sinal é fundamental para a função de link, pois ele informa ao estágio de escrita, que o dado a ser gravado no registrador de destino não é o resultado da ALU nem da memória, mas sim o endereço de retorno, ou seja, PC mais 4.

- Depois, realizamos modificações na Branch unit. A principal dificuldade lógica aqui foi diferenciar o cálculo do endereço de destino. Para instruções de Branch e JAL, o endereço é calculado somando o PC atual ao valor imediato. Já para o JALR, o endereço de destino é calculado somando o valor de um registrador base com o imediato, operação que ocorre na ALU principal. Para resolver isso, configuramos um multiplexador dentro da Branch Unit. Se o sinal jmpr estiver ativo, a unidade seleciona o resultado vindo da ALU. Se for um Branch tomado ou um JAL, ela seleciona o cálculo do PC mais imediato.

- Por fim, integramos tudo isso no Datapath. O desafio foi garantir que os sinais de controle viajassem corretamente pelos registradores de pipeline até o momento de uso. No estágio de busca, configuramos o multiplexador do PC para aceitar o endereço calculado pela Branch Unit sempre que um salto fosse validado. No estágio final de Write Back, adicionamos um novo multiplexador chamado wrsmux. Esse componente utiliza o sinal jmp_sel para selecionar entre o dado processado e o valor de PC mais 4, garantindo que, ao realizar uma chamada de função, o processador salve corretamente o endereço de retorno no registrador especificado.

Com essas alterações no fluxo de dados e controle, validamos que todas as instruções de desvio condicional e incondicional estão operando corretamente.

Parcial 5 – Instrução Halt

Por fim, implementamos :

O sinal para HALT

- Para realizar isso, primeiramente mudamos o módulo Controller adicionando um novo sinal de controle denominado halt_com. Este sinal foi configurado para assumir nível lógico 1 exclusivamente quando opcode da instrução atual corresponder ao código de HALT.

- No Datapath e nos registradores de Pipeline, tivemos que garantir a propagação desse sinal. Modificamos os parâmetros do registrador B no módulo RegPack para que ele pudesse armazenar e transportar o sinal Halt_com até os estágios posteriores de execução.

- A lógica de travamento foi implementada efetivamente na Branch unit. Passamos o sinal de Halt como um dos inputs desta unidade e configuramos a seleção do próximo PC de forma específica. Quando o sinal halt_com é detectado, a Branch Unit seleciona como próximo endereço o valor de PC mais 4 menos 4. O resultado dessa operação matemática é o retorno ao próprio endereço da instrução atual. Isso cria um loop infinito intencional onde somente a instrução de HALT é executada repetidamente, impedindo que o restante do programa seja processado

---

5. Integração final

Após implementar todas as partes, o grupo se reuniu para fazer uma simulação completa, para verificar o funcionamento e integridade geral.