Aluno: Gustavo Pereira Chaves
Matrícula: 19/0014113
Disciplina: Organização e Arquitetura de Computadores
Turma: C

Simulador RISC-V

1. Apresentação

Neste trabalho foi desenvolvido um simulador da arquitetura RV32I na linguagem C, que consiste em interpretar as instruções binárias geradas pelo RARS e executá-las de forma semelhante ao que ocorreria no processador dessa arquitetura. Foram desenvolvidas funções tanto de decodificação quanto de execução, bem como vetores que simulam a memória e o banco de registradores.

Tendo em vista todos esses recursos, torna-se possível compreender de forma aprofundada o funcionamento desse tipo de processador e como as instruções são interpretadas pelo mesmo.

2. Implementação

Para resolver o problema, foram criados três arquivos .c:

main.c → Executa a função run, que roda o programa todo.
riscv.c → Funções que decodificam e executam as instruções
mem.c → Funções de acesso a memória

Para compilar, basta executar: **gcc main.c riscv.c mem.c**

De forma a executar o código gerado pelo RARS, foram necessários os seguintes passos:

- Carregar os arquivos data e code em binário para a memória simulada pelo vetor mem[MEM_SIZE] utilizando a função load_mem().

- Copiar o binário da próxima instrução a ser executada para o ri e incrementar o valor de pc para identificar a próxima instrução utilizando a função fetch().

- Decodificar todos os campos da instrução copiada para RI utilizando a função decode(). Para decodificar os campos, foi necessário fazer shift sucessivos na instrução até a posição desejada e um & (and bit a bit) para selecionar a quantidade de bits extraídos. Fez-se isso para os campos opcode, funct3, funct7, shamt, rd, rs1 e rs2.

- Decodificar o imm utilizando a função geraImm, que através do opcode, identifica a posição do imediato e extrai com uma estratégia semelhante a utilizada para decodificar os campos. No entanto, alguns imediatos podem estar com os bits em posições trocadas, assim foi utilizado o define set_bit para mudá-los para a posição correta.

- Assim, de posse de todos esses campos, basta identificar qual instrução será executada e realizar as operações na memória ou no banco de registradores. Dessa forma, foi feito um switchs sucessivos utilizando o opcode, funct3 e funct7, e realizando a operação.

Foram implementadas então todas as classes de instruções requeridas na função execute():

Para as instruções lógico aritméticas com registradores (Tipo R), bastava executar a operação entre os registradores rs1 e rs2 e salvar no registrador rd.

Para as instruções lógico aritméticas com imediato (Tipo I), era executada a operação entre o registrador rs1 e o imm, salvando o resultado no registrador rd.

Para as instruções de salto condicional (Tipo B), eram comparadas os valores dos registadores rs1 e rs2, e caso a condição fosse verdadeira, soma-se o valor do imediato ao pc da instrução atual (Como PC sempre aponta para a instrução seguinte, foi subtraído 4 do resultado em todas as operações).

Para a instruções de leitura (Tipo I), foram reutilizadas as funções lb, lbu e lw desenvolvidas no trabalho 1, que utilizam o valor de rs1+imediato para encontrar o endereço de memória onde deve-se buscar o dado.

Para as instruções de escrita (Tipo S), também foram reutilizadas as funções sb e sw desenvolvidas no trabalho 1, que também calculam o endereço de memória onde vai ser salvo o dado através do registrador rs1 + imediato.

Para as instruções auipc e lui (Tipo U) o processo foi semelhante. Enquanto que em auipc deve-se somar o pc atual com o imediato deslocado em 12 bits para a esquerda e salvar no registador rd,em lui só é salvo o imediato deslocado.

Para as instrução de salto (Tipo J), é salvo no registrador rd o endereço da próxima instrução e somado ao pc atual o valor do imediato.

Para a instrução jalr (Tipo I), também é salvo em rd o endereço da próxima instrução e alterado o valor de pc para o valor de rs1+imm com o primeiro bit mascarado.

Por fim, o Ecall foi implementando identificando o valor armazenado no registrador A7, para então executar uma das três funções requisitadas: imprimir um inteiro armazenado em a0, imprimir uma string a partir do endereço de a0 ou encerrar o programa.

3. Testes

Para a realização dos testes foi criada uma opção semelhante a do rars, em que é possível executar todo o programa ou passo a passo. Para isso basta alterar o parâmetro da função run, em que 0 executa direto, e 1 instrução por instrução. Com isso, foi possível verificar linha a linha o comportamento do programa em relação ao esperado realizado pelo rars.

Os binários do código e dos dados foram gerados a partir do testador disponibilizado, acrescido de um último teste para verificar a instrução SLTU. Executando o programa C desenvolvido, e realizando passo a passo, pode-se verificar que todas as instruções realizaram as operações desejadas, o que dá um certo grau de confiança com relação a bugs.

Já com relação aos problemas enfrentados, o principal foi encontrar uma forma de reordenar bits de uma variável inteira, solução que foi posteriormente descoberta no arquivo globals.h com a definição de set_bit. Ademais, a função fetch sempre incrementa

o valor de PC para apontar para a próxima instrução, no entanto algumas funções utilizam o valor do pc atual para os cálculos e não o valor seguinte. Foi preciso então subtrair 4 em todas as instruções que realizavam algum tipo de operação usando o PC. Por fim, foi preciso também lembrar da definição do registrador zero, de forma a sempre deixá-lo com todos os bit 0, mesmo que alguma instrução tente modificá-lo.

Com esses bugs resolvidos, o programa funcionou sem grandes problemas, realizando o que foi idealizado pela proposta do trabalho.