UNIVERSIDADE DE BRASÍLIA
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

116394 ORGANIZAÇÃO E ARQUITETURA DE COMPUTADORES

Trabalho 2

Simulador RISC-V

17/0009611 | Estevan Alexander de Paula

Prof. Ricardo Pezzuol Jacobi

Turma C

Brasília, 25 de Agosto de 2021.

Índice

1. Apresentação do problema

O trabalho consiste em, implementadas algumas funções básicas de manipulação de memória do RISC-V no trabalho 1, implementar uma série de instruções do RISC-V em 32 bits e um simulador capaz de ler um binário de código assembly gerado pelo RARS, decodificar as instruções ali presentes e executá-las.

A tabela 1 mostra a relação das instruções implementadas no simulador. Em linhas gerais, o simulador utiliza uma memória de 4096 palavras e um banco de 32 registradores, utilizando tipos de inteiro com e sem sinal, a citar: int32_t, uint32_t, uint8_t.

add	addi	and	andi	auipc
beq	bne	bge	bgeu	blt
bltu	jal	jalr	lb	or
lbu	lw	lui	nop	sltu
ori	sb	slli	slt	srai
srli	sub	sw	xor	ecall

Em linhas gerais, o simulador roda a partir da execução de uma função run() (conforme figura 1, que executa em loop a função step() (conforme figura 3), responsável buscar, decodificar e executar a instrução corrente. A função run mantém a execução do loop enquanto o programa não requisita uma saída, o que pode ser feito a partir de um syscall, ou enquanto o programa não atinge o início da área de dados da memória, localizada no endereço 0x2000.

Figura 1. Código da função run

```
void run() {
  pc = 0; ri = 0;
  exitProg = 0;
  sp = 0x3ffc;
  gp = 0x1800;

  while (!exitProg && (pc < 0x2000)) {
    step();
  }
}
```

Figura 2. Código da função step

```
void step() {
  fetch();
  decode();
  execute();
}
```

Para que seja carregado um programa no simulador, entretanto, é necessário utilizar a função load_mem(nome_do_arquivo, endereco_de_inico), para posteriormente utilizar a função run e verificar os resultados do programa.

Figura 3. Função main realizando o carregamento de arquivo e posteriormente rodando o programa

```
int main() {
  load_mem("text.bin", 0);
  load_mem("data.bin", 0x2000);
  run();

  return 0;
}
```

2. Descrição das instruções

A descrição das instruções implementadas segue abaixo:

- add: guarda no registrador de destino a soma do conteúdo dos dois registradores de origem ($rd = rs1 + rs2$)
- addi: guarda no registrador de destino a soma do conteúdo de um registrador de origem e um imediato ($rd = rs1 + imm$)
- and: guarda no registrador de destino o resultado da operação and bitwise entre o conteúdo de dois registradores de origem ($rd = rs1 \& rs2$)
- andi: guarda no registrador de destino o resultado da operação and bitwise entre o conteúdo de um registrador e um imediato ($rd = rs1 \& imm$)
- auipc: guarda no registrador de destino o resultado da soma de um imediato com o pc ($rd = pc + imm$)
- beq: soma ao pc o valor de um imediato caso os conteúdos de ambos os registradores de origem sejam iguais ($rs1 == rs2?pc+ = imm$)
- bne: soma ao pc o valor de um imediato caso os conteúdos dos registradores de origem sejam diferentes ($rs1! = rs2?pc+ = imm$)
- bge: soma ao pc o valor de um imediato caso o conteúdo do primeiro registrador seja maior ou igual que o conteúdo do segundo ($rs1 >= rs2?pc+ = imm$)
- bgeu: soma ao pc o valor de um imediato caso o conteúdo do primeiro registrador seja maior ou igual que o conteúdo do segundo considerando os valores sem sinal ($(uint)rs1 >= (uint)rs2?pc+ = imm$)
- blt: soma ao pc o valor de um imediato caso o conteúdo do primeiro registrador menor que o conteúdo do segundo ($rs1 < rs2?pc+ = imm$)
- bltu: soma ao pc o valor de um imediato caso o conteúdo do primeiro registrador menor que o conteúdo do segundo considerando os valores sem sinal ($rs1 < rs2?pc+ = imm$)
- jal: soma ao pc o valor de um imediato e guarda seu valor anterior no registrador de destino ($rd = pc; pc+ = imm$)
- jalr: soma ao pc o valor de um imediato de 12 bits (tipo de instrução I) e guarda seu valor anterior no registrador de destino ($rd = pc; pc+ = imm$)
- lb: guarda no registrador de destino um byte da memória cujo endereço é dado pela soma do registrador de origem e um imediato ($rd = lb(rs1, imm)$)
- lbu: guarda no registrador de destino um byte da memória cujo endereço é dado pela soma do registrador de origem e um imediato convertendo-o pra um valor sem sinal ($rd = lbu(rs1, imm)$)
- lw: guarda no registrador de destino uma palavra da memória cujo endereço é dado pela soma do registrador de origem e um imediato ($rd = lw(rs1, imm)$)
- lui: guarda no registrador de destino o valor de um imediato ($rd = imm$)
- nop: realiza uma operação sem resultado, que no caso do RISC-V é a operação de adição com um imediato de valor zero com o registrador zero sendo o destino também o o registrador zero ($zero = zero + 0$)
- or: guarda no registrador de destino a operação de or bitwise entre os dois registradores de origem ($rd = rs1|rs2$)
- ori: guarda no registrador de destino a operação de or bitwise entre o registrador de destino e um imediato ($rd = rs1|imm$)
- sb: guarda no endereço de memória dado pela soma do primeiro registrador de origem com um imeditado o byte no segundo registrador de destino (sb rs1, imm, rs2)

- slli: guarda no registrador de destino o shift para a esquerda do registrador de origem por uma quantidade de bits dada por um imediato ($rd = rs1 << imm$)
- slt: guarda no registrador a compração entre os registradores de origem por menor que ($rd = rs1 < rs2$)
- sltu: guarda no registrador a compração entre os registradores de origem sem sinal por menor que ($rd = (uint)rs1 < (uint)rs2$)
- srai: guarda no registrador de destino o shift para a direita do registrador de origem por uma quantidade de bits dada por um imediato ($rd = rs1 >> imm$)
- srli: guarda no registrador de destino o shift para a direita do registrador de origem sem sinal por uma quantidade de bits dada por um imediato ($rd = (uint)rs1 >> imm$)
- sub: guarda no registrador de destino o resultado da subtração do conteúdo dos dois registradores de origem ($rd = rs1 - rs2$)
- sw: guarda no endereço de memória dado pela soma do primeiro registrador de origem com um imeditado a palavra no segundo registrador de destino (sw rs1, imm, rs2)
- xor: guarda no registrador de destino a operação de xor bitwise entre os dois registradores de origem ($rd = rs1 \oplus rs2$)
- ecall: pode realizar três tipos de operação, a depender do conteúdo do registrador a0: 1 - imprimir inteiro, 4 - imprimir string, 10 - parar a execução do programa

3. Testes e resultados

3.1. Programa testador

O simulador foi testado utilizando o programa disponibilizado pelo professor, que realiza o teste de 27 das 30 operações implementadas, direta ou indiretamente. O teste realiza uma série de instruções e imprime na tela o número do teste e se o teste foi bem sucedido. Rodando o binário gerado a partir desse programa, o simulador obtém o seguinte resultado:

Figura 4. Output do simulador ao rodar o programa testador.asm fornecido pelo professor

```
trabalho1 on  main [!?]
❯ ls **/*.c | xargs gcc -o riscv_sim # compilando o simulador

trabalho1 on  main [!?]
❯ ./riscv_sim # rodando o simulador
Teste 1 OK
Teste 2 OK
Teste 3 OK
Teste 4 OK
Teste 5 OK
Teste 6 OK
Teste 7 OK
Teste 8 OK
Teste 9 OK
Teste 10 OK
Teste 11 OK
Teste 12 OK
Teste 13 OK
Teste 14 OK
Teste 15 OK
Teste 16 OK
Teste 17 OK
Teste 18 OK
Teste 19 OK
Teste 20 OK
Teste 21 OK
Teste 22 OK
```

3.2. Instruções não testadas pelo programa testador

3.2.1 jalr

Programa utilizado para testar a instrução jalr:

```
.data
word:    .word 0xFAFEF1F0

TAB:     .asciz "\t"
NL:      .asciz  "\n"
Label:   .asciz "JalrT"
l_ok:    .asciz  " OK"
l_fail:  .asciz  " FAIL"

.text    # pula para OK se jalr funcionar corretamente
         li t0, 8
         jalr a0, t0, 4
         j FAIL
         j OK

FAIL:
         la a0, Label
         li a7, 4
         ecall
         la a0, l_fail
         li a7, 4
         ecall
         la a0, NL
         ecall
         li a7, 10
         ecall

OK:
         la a0, Label
         li a7, 4
         ecall
         la a0, l_ok
         li a7, 4
         ecall
         la a0, NL
         ecall
         li a7, 10
         ecall
```

Output:

```
trabalho1 on  main [!?]
❯ ./riscv_sim # rodando o simulador para o programa de teste de jalr
JalrT OK

trabalho1 on  main [!?]
❯
```

3.2.2 nop

Para testar nop, simplesmente fizemos um programa que roda a instrução para ver se o simulador realmente realizaria uma instrução addi zero, zero, 0. Programa utilizado para testar a instrução nop:

```
.data
word:   .word 0xFAFEF1F0

TAB:    .asciz "\t"
NL:     .asciz  "\n"
Label:  .asciz "Nop T"
l_ok:   .asciz  " OK"
l_fail: .asciz  " FAIL"

.text
        nop
        j OK

FAIL:
        la a0, Label
        li a7, 4
        ecall
        la a0, l_fail
        li a7, 4
        ecall
        la a0, NL
        ecall
        li a7, 10
        ecall

OK:
        la a0, Label
        li a7, 4
        ecall
        la a0, l_ok
        li a7, 4
        ecall
        la a0, NL
        ecall
        li a7, 10
        ecall
```

Para que pudéssemos ver que o nop realmente realizava uma addi zero, zero, 0, fizemos uma alteração no código para mostrar o conteúdo dos registradores antes e depois de um addi rd, 0, 0 rodar, caso o conteúdo de todos os registradores permanecesse 0 após o nop, significa que o simulador executou a instrução correta, conforme a imagem abaixo:

```
case OPCODE_I_1:
  switch(funct3) {
    case 0b000: // addi
      if(breg[rs1] == 0 && imm12_i == 0)
        dump_reg('h');
      breg[rd] = breg[rs1] + imm12_i;
      if(breg[rs1] == 0 && imm12_i == 0)
        dump_reg('h');
      break;
```

Output:

```
trabalho1 on  main [!?]
❯ ls **/*.c | xargs gcc -o riscv_sim # compilando o simulador com mudanças

trabalho1 on  main [!?]
❯ ./riscv_sim # rodando o simulador para o programa de teste de nop
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0

0
```

```
0

0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0

Nop T OK

trabalho1 on  main [!?]
❯
```

3.2.3 sltu

Programa utilizado para testar a instrução sltu:

```
.data
word:   .word 0xFAFEF1F0

TAB:    .asciz "\t"
NL:     .asciz  "\n"
Label:  .asciz "SltuT"
l_ok:   .asciz  " OK"
l_fail: .asciz  " FAIL"

.text
        li t0, -1
        li t1, 5
        sltu t3, t0, t1
        beqz t3, OK
        j FAIL

FAIL:
        la a0, Label
        li a7, 4
        ecall
        la a0, l_fail
        li a7, 4
        ecall
        la a0, NL
        ecall
        li a7, 10
        ecall

OK:
        la a0, Label
        li a7, 4
        ecall
        la a0, l_ok
        li a7, 4
        ecall
        la a0, NL
        ecall
        li a7, 10
        ecall
```

Output:

```
trabalho1 on  main [✘!?]
❯ ./riscv_sim # rodando o simulador para o programa de teste de sltu
SltuT OK

trabalho1 on  main [!?]
❯
```