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

116394 ORGANIZAÇÃO E ARQUITETURA DE COMPUTADORES

Trabalho III: Simulador RISC-V

OBJETIVO

Este trabalho consiste na implementação de um simulador da arquitetura RV32I em linguagem de alto nível (C/C++/Java). O simulador deve implementar as funções de busca da instrução (*fetch()*), decodificação da instrução (*decode()*) e execução da instrução (*execute()*). O programa binário a ser executado deve ser gerado a partir do montador RARS, juntamente com os respectivos dados. O simulador deve ler arquivos binários contendo o segmento de código e o segmento de dados para sua memória e executá-lo.

DESCRIÇÃO

Geração dos arquivos

As instruções e dados de um programa RV32I para este trabalho devem vir necessariamente de arquivos montados pelo RARS. Para ilustrar o procedimento, considere o exemplo a seguir, um programa que imprime na console os 8 primeiros números primos:

```
.data
primos:      .word 1, 3, 5, 7, 11, 13, 17, 19
size:        .word 8
msg:         .asciz "Os oito primeiros numeros primos sao : "
space:       .ascii " "

.text
       la t0, primos       # carrega endereço inicial do array
       la t1, size         # carrega endereço de size
       lw t1, 0(t1)        # carrega size em t1
       li a7, 4            # imprime mensagem inicial
       la a0, msg
       ecall

loop:  beq t1, zero, exit  # se processou todo o array, encerra
       li a7, 1            # serviço de impressão de inteiros
       lw a0, 0(t0)        # inteiro a ser exibido
       ecall
       li a7, 4            # imprime separador
       la a0, space
       ecall
       addi t0, t0, 4      # incrementa indice array
       addi t1, t1, -1     # decrementa contador
       j loop              # novo loop
exit:  li a7, 10
       ecall
```

Montagem do programa

Antes de montar o programa deve-se configurar o RARS através da opção:

Settings->Memory Configuration, opção Compact, Text at Address 0

Ao montar o programa (F3), o RARS exibe na aba “Execute” os segmentos *Text* e *Data*. O segmento de código (*Text*) deste programa começa no endereço `0x00000000` de

memória e se encerra no endereço `0x00000054`, que contém a instrução *ecall*. O segmento de dados começa na posição `0x00002000` e termina na posição `0x000204c`. Verifique a ordem dos caracteres da mensagem *msg* no segmento de dados usando a opção ASCII de visualização.

O armazenamento destas informações em arquivo é obtido com a opção:

File -> Dump Memory...

As opções de salvamento devem ser:

Código:

.text (0x00000000 - 0x00000054) - que é o valor *default* para este exemplo

Dump Format: binary

Dados:

.data (0x00002000 - 0x00002ffc) - área entre *data* e *heap*.

Dump Format: binary

Gere os arquivos com nomes text.bin e data.bin.

Leitura do código e dos dados

O código e os dados contidos nos arquivos devem ser lidos para a memória do simulador.

A memória deve ser modelada como um arranjo de inteiros:

```
#define MEM_SIZE 4096
int32_t mem[MEM_SIZE];
```

Ou seja, a memória é um arranjo de 8KWords, ou 32KBytes.

Acesso à Memória

Reutilizar as funções desenvolvidas no trabalho anterior adaptadas ao contexto do RISC-V:

```
int32_t lb(uint32_t address, int32_t kte);
int32_t lh(uint32_t address, int32_t kte);
int32_t lw(uint32_t address, int32_t kte);
int32_t lbu(uint32_t address, int32_t kte);
int32_t lhu(uint32_t address, int32_t kte);
void sb(uint32_t address, int32_t kte, int8_t dado);
void sh(uint32_t address, int32_t kte, int16_t dado);
void sw(uint32_t address, int32_t kte, int32_t dado);
```

Os endereços são todos de *byte*. A operação de leitura de *byte* retorna um inteiro com o *byte* lido na posição menos significativa. A escrita de um *byte* deve colocá-lo na posição correta dentro da palavra de memória.

Registradores

Os registradores *pc, sp, gp* e *ri*, e também os campos da instrução (*opcode*, *rs*, *rt*, *rd*, *shamt*, *funct*) devem ser definidos como variáveis globais. *pc* e *ri* podem ser do tipo *unsigned int (uint32_t)*, visto que não armazenam dados, apenas instruções, assim como *sp e gp*, que armazenam endereços de memória - nunca são negativos.

Valores iniciais dos registradores: (modelo compacto)

- `pc = 0x00000000`
- `ri = 0x00000000`
- `sp = 0x00003ffc`
- `gp = 0x00001800`

obs: sp é necessário para funções recursivas, iniciado no final da memória de 8KB.

Função fetch()

A função `void fetch()` lê uma instrução da memória e coloca-a em *ri*, atualizando o *pc* para apontar para a próxima instrução (soma 4).

Função decode()

Deve extrair todos os campos da instrução:

- *opcode*: código da operação
- *rs1*: índice do primeiro registrador fonte
- *rs2*: índice do segundo registrador fonte
- *rd*: índice do registrador destino, que recebe o resultado da operação
- *shamt*: quantidade de deslocamento em instruções *shift* e *rotate*
- *funct3:* código auxiliar de 3 bits para determinar a instrução a ser executada
- *funct7:* código auxiliar de 7 bits para determinar a instrução a ser executada
- *imm12_i*: constante de 12 bits, valor imediato em instruções tipo I
- *imm12_s*: constante de 12 bits, valor imediato em instruções tipo S
- *imm13*: constante de 13 bits, valor imediato em instruções tipo SB, bit 0 é sempre 0
- *imm20_u*: constante de 20 bits mais significativos, 31 a 12
- *imm21*: constante de 21 bits para saltos relativos, bit 0 é sempre 0

Todos os valores imediatos tem o sinal estendido.

Função execute()

A função `void execute()` executa a instrução que foi lida pela função `fetch()` e decodificada por `decode()`.

Função step()

A função `step()` executa uma instrução do MIPS:

step() => fecth(), decode(), execute()

Função run()

A função `run()` executa o programa até encontrar uma chamada de sistema para encerramento, ou até o *pc* ultrapassar o limite do segmento de código (2k *words*).

Função dump_mem(int start, int end, char format)

Imprime o conteúdo da memória a partir do endereço *start* até o endereço *end*. O formato pode ser em hexa (‘h’) ou decimal (‘d’).

Função dump_reg(char format)

Imprime o conteúdo dos registradores do MIPS, incluindo o banco de registradores e os registradores *pc*, *hi* e *lo*. O formato pode ser em hexa (‘h’) ou decimal (‘d’).

Instruções a serem implementadas:

```
enum OPCODES {  // lembrem que so sao considerados os 7 primeiros bits dessas constantes
       LUI = 0x37,      AUIPC = 0x17,          // atribui 20 bits mais significativos
       ILType = 0x03,                          // Load type
       BType = 0x63,                           // branch condicional
       JAL = 0x6F,      JALR = 0x67,           // jumps - JAL formato UJ, JALR formato I
       StoreType = 0x23,                       // store
       ILAType = 0x13,                         // logico-aritmeticas com imediato
       RegType = 0x33,                         // operacoes LA com registradores
       ECALL = 0x73                            // chamada do sistema - formato I
};

enum FUNCT3 { // campo auxiliar de 3 bits
       BEQ3=0,  BNE3=01, BLT3=04,  BGE3=05,  BLTU3=0x06,  BGEU3=07,     // branches
       LB3=0,   LH3=01,  LW3=02,  LBU3=04,  LHU3=05,                    // loads
       SB3=0,   SH3=01, SW3=02,                                         // stores
       ADDSUB3=0,  SLL3=01,  SLT3=02,  SLTU3=03,                        // LA com
       XOR3=04,  SR3=05,         OR3=06,         AND3=07,               // registradores
       ADDI3=0,  ORI3=06,  SLTI3=02,  XORI3=04,  ANDI3=07,              // LA com
       SLTIU3=03,  SLLI3=01,  SRI3=05                                   // imediatos

};

enum FUNCT7 {

// campo auxiliar de 7 bits para as instrucoes SRLI/SRAI, ADD/SUB, SRL/SRA

       ADD7=0, SUB7=0x20, SRA7=0x20, SRL7=0, SRLI7=0x00, SRAI7=0x20
};
```

Syscall: implementar as chamadas para (ver *help* do MARS)

- imprimir inteiro
- imprimir string
- encerrar programa

Verificação do Simulador

Para verificar se o simulador está funcionando corretamente deve-se utilizar o RARS para geração de códigos de teste, que incluam código executável e dados. Os testes devem verificar todas as instruções implementadas no simulador.

Atentar para uso de pseudo-instruções. No RARS, elas são traduzidas para instruções nativas do RISC-V. Se utilizar pseudo-instruções, verificar se, depois da montagem, o RARS gera instruções aceitas pelo simulador.

Serão fornecidos códigos de teste para esta tarefa.

Entrega

Entregar:
- relatório da implementação:
    - descrição do problema
    - descrição sucinta das funções implementadas
    - testes e resultados
- o código fonte do simulador, com a indicação da plataforma utilizada:
    - qual compilador empregado
    - sistema operacional
    - IDE (Eclipse, XCode, etc)

Entregar no Moodle em um arquivo compactado, com o número de matrícula do aluno para identificar o arquivo.