Trabalho 2 de OAC

Sobre

Simulador de MIPS

O trabalho consiste de montar um emulador de MIPS em C para realizar ciclo fetch-decode-execute. Esse trabalho tem o objetivo de auxiliar o entendimento de como o processador descobre seus parametros, os coloca em ação e como muda seu fluxo de execução na arquitetura MIPS.

Participante

Aluno	Matrícula	Username Github
Victor André Gris Costa	16/0019311	@victoragcosta

Implementação

Descrição do Problema

Decidimos implementar um simulador do MIPS que pega programas já montados e executa instrução a instrução, simulando o ciclo fetch-decode-execute. Fetch seria capturar a instrução da memória para um registrador especial e incrementar PC de 4 (próxima linha). Decode seria capturar todos os parâmetros possíveis da instrução, podendo ser instruções do tipo-I, tipo-J e tipo-R, simulando como o hardware faria. Execute seria fazer a instrução correta a partir desses parâmetros, alterando os registradores necessários ou acessando a memória.

Funções implementadas

As funções estão em 2 arquivos: run.c e memory.c. Todas as funções envolvendo acesso a memória

isolada do processador estão em memory.c. Todas as funções que acessam registradores internos ao processador estão em run.c, além da lógica de execução de um programa em assembly do MIPS.

- `fetch()` : copia a instrução apontada pelo endereço no registrador *program counter* (pc) para o registrador ri. Logo após incrementa pc de 4 bytes, apontando para a próxima word, que seria a próxima instrução.
- `decode()` : Extrai todos os parâmetros possíveis, simulando como o *hardware* faria, de forma paralela e calculando todas as possibilidades. Os parâmetros de uma instrução seriam *opcode*, *rs*, *rt*, *rd*, *shamnt*, *funct*, imediato de 16 bits (*k16*) e imediato de 26 bits (*k26*).
- `execute()` : Após ter os parâmentros extraídos da instrução, essa função se baseia neles para fazer as alterações nos registros. Nessa função utilizamos um switch para *opcodes* e dentro do case EXT (*opcode* = 0x00) fizemos um switch de *functs*, implementando instruções de acesso a memória (reutilizando do trabalho anterior), instruções de salto incondicional, salto condicional, relativos ou absolutos e instruções aritméticas e lógicas em geral. Além disso implementamos 3 funções de *syscall*: escrever inteiro, escrever string e finalizar programa.
- `step()` : executa um ciclo de `fetch()` , `decode()` e `execute()` .
- `run()` : executa o programa até o fim, considerando fim como o *syscall* para *exit* ou alcançar o endereço 0x1000 (fim do segmento de texto).
- `prepare_run()` : inicializa os registradores e algumas variáveis de controle do simulador.
- `dump_reg(char format)` : escreve todo o conteúdo de todos os registradores, incluindo pc, ri, hi e lo no formato desejado: d para decimal e h para hexadecimal.
- `dump_mem(int start, int end, char format)` : escreve todo o conteúdo da memória no intervalo indicado. Essa função foi parcialmente reciclada do trabalho anterior. O parâmetro *format* escolhe a base: d para decimal e h para hexadecimal.
- `load_text(char *arq_name)` : abre um arquivo binário para leitura contendo o segmento de *text* de um programa MIPS. Ele carrega diretamente na área de memória correspondente ao *text* (0x0000 até 0x0FFC).
- `load_data(char *arq_name)` : abre um arquivo binário para leitura contendo o segmento de *data* de um programa MIPS. Ele carrega diretamente na área de memória correspondente a *data* (0x2000 até 0x3FFC).
- Todas as outras funções foram completamente recicladas do trabalho 1 (corrigindo alguns *asserts* incorretos).

Testes e Resultados

Testei algumas funções de debug primeiramente a partir do programa proposto pelo professor:

```
Text Hexadecimal:
mem[0] = 20082000
mem[1] = 20092020
mem[2] = 8D290000
mem[3] = 24020004
mem[4] = 20042024
mem[5] = 0000000C
mem[6] = 11200009
mem[7] = 24020001
mem[8] = 8D040000
mem[9] = 0000000C
mem[10] = 24020004
mem[11] = 2004204C
mem[12] = 0000000C
mem[13] = 21080004
mem[14] = 2129FFFF
mem[15] = 08000006
mem[16] = 2402000A
mem[17] = 0000000C
mem[18] = 00000000
mem[19] = 00000000

Text Decimal:
mem[0] = 537403392
mem[1] = 537468960
mem[2] = -1926692864
mem[3] = 604110852
mem[4] = 537141284
mem[5] = 12
mem[6] = 287309833
mem[7] = 604110849
mem[8] = -1929117696
mem[9] = 12
mem[10] = 604110852
mem[11] = 537141324
mem[12] = 12
mem[13] = 554172420
mem[14] = 556400639
mem[15] = 134217734
mem[16] = 604110858
mem[17] = 12
mem[18] = 0
mem[19] = 0

Data Hexadecimal:
mem[2048] = 00000001
mem[2049] = 00000003
mem[2050] = 00000005
```

```
mem[2051] = 00000007
mem[2052] = 0000000B
mem[2053] = 0000000D
mem[2054] = 00000011
mem[2055] = 00000013
mem[2056] = 00000008
mem[2057] = 6F20734F
mem[2058] = 206F7469
mem[2059] = 6D697270
mem[2060] = 6F726965
mem[2061] = 756E2073
mem[2062] = 6F72656D
mem[2063] = 72702073
mem[2064] = 736F6D69
mem[2065] = 6F617320
mem[2066] = 00203A20
mem[2067] = 00000020
mem[2068] = 00000000
mem[2069] = 00000000
mem[2070] = 00000000
mem[2071] = 00000000

Data Decimal:
mem[2048] = 1
mem[2049] = 3
mem[2050] = 5
mem[2051] = 7
mem[2052] = 11
mem[2053] = 13
mem[2054] = 17
mem[2055] = 19
mem[2056] = 8
mem[2057] = 1864397647
mem[2058] = 544175209
mem[2059] = 1835627120
mem[2060] = 1869769061
mem[2061] = 1970151539
mem[2062] = 1869768045
mem[2063] = 1919950963
mem[2064] = 1936682345
mem[2065] = 1868657440
mem[2066] = 2112032
mem[2067] = 32
mem[2068] = 0
mem[2069] = 0
mem[2070] = 0
mem[2071] = 0

Registradores Hexadecimal:
```

```
$0 = 00000000
$1 = 00000000
$2 = 00000000
$3 = 00000000
$4 = 00000000
$5 = 00000000
$6 = 00000000
$7 = 00000000
$8 = 00000000
$9 = 00000000
$10 = 00000000
$11 = 00000000
$12 = 00000000
$13 = 00000000
$14 = 00000000
$15 = 00000000
$16 = 00000000
$17 = 00000000
$18 = 00000000
$19 = 00000000
$20 = 00000000
$21 = 00000000
$22 = 00000000
$23 = 00000000
$24 = 00000000
$25 = 00000000
$26 = 00000000
$27 = 00000000
$28 = 00001800
$29 = 00003FFC
$30 = 00000000
$31 = 00000000
pc = 00000000
ri = 00000000
hi = 00000000
lo = 00000000

Registradores Decimal:
$0 = 0
$1 = 0
$2 = 0
$3 = 0
$4 = 0
$5 = 0
$6 = 0
$7 = 0
$8 = 0
$9 = 0
$10 = 0
```

```
$11 = 0
$12 = 0
$13 = 0
$14 = 0
$15 = 0
$16 = 0
$17 = 0
$18 = 0
$19 = 0
$20 = 0
$21 = 0
$22 = 0
$23 = 0
$24 = 0
$25 = 0
$26 = 0
$27 = 0
$28 = 6144
$29 = 16380
$30 = 0
$31 = 0
pc = 0
ri = 0
hi = 0
lo = 0

Execucao normal do programa:
Os oito primeiros numeros primos sao : 1 3 5 7 11 13 17 19
--------------------
```

Observação: o programa do professor não gera *newline* e aqui no simulador também não gera. Adicionei uma linha nova com um print na main por questões estéticas.

Comparando ao binário gerado pelo MARS podemos ver que o carregamento do arquivo para a memória está funcionando muito bem e que o dump_mem também funciona bem. Comparando os registradores inicializados no MARS com o do simulador, estão iguais, sendo assim, as funções de dump_reg e prepare_run estão bem implementadas.

Comparando a execução do MARS com a do simulador, podemos notar que tem o mesmo resultado e portando deve estar funcionando muito bem.

Testei também o programa de testes liberado pelo professor no sábado(?) para conferir se as instruções estão sendo executadas corretamente.

```
Execucao normal do programa de testes:
Teste1 OK
Teste2 OK
Teste3 OK
Teste4 OK
Teste5 OK
Teste6 OK
Teste7 OK
Teste8 OK
Teste9 OK
Teste10 OK
Teste11 OK
Teste12 OK
Teste13 OK
Teste14 OK
Teste15 OK
Teste16 OK
Teste17 OK
Teste18 OK
Teste19 OK
Teste20 OK
Teste21 OK
Teste22 OK
Teste23 OK
Teste24 OK
Teste25 OK
Teste26 OK
Teste27 OK
Teste28 OK
Teste29 OK
Teste30 OK
Teste31 OK

--------------------
```

Pode-se observar que gerou a mesma saída que o MARS e portanto deve ser um programa bem robusto. Eu encontrei dois comandos usados pelo MARS em pseudo instruções que não foram pedidas para ser implementadas no simulador como `addu` e `addiu` que considerando que não temos exceções de overflow no simulador são iguais a `add` e `addi` eu apenas fiz elas serem equivalentes. Os testes ajudaram a encontrar alguns erros com algumas instruções, como LUI (implementei completamente errado), ANDI, XORI, ORI (ocorreu uma extensão de sinal quando não deveria) e DIV (esqueci do resto no hi), ressaltando a importância de testes.

Sobre o código

Sistema Operacional

O programa foi desenvolvido em um *Windows Subsystem for Linux* de Ubuntu. Basicamente é um Ubuntu embutido no Windows 10, então o código é garantido de funcionar em Ubuntu, porém não se tem tanta certeza quanto a Windows.

Compilador

O programa é compilado utilizando o g++ (gcc version 4.8.4 (Ubuntu 4.8.4-2ubuntu1~14.04.4)).

IDE

O programa foi desenvolvido no editor de texto Visual Studio Code. Esse editor de texto é praticamente uma IDE com as extensões corretas.

Uso

Para compilar o programa vá até a pasta src e execute o comando `make` no seu linux. O binário será gerado na raiz do projeto com o nome de sim_mips. Alternativamente você pode executar os comandos:

```
g++ -c -o obj/memory.o memory.c -Wall -g -I ../include
g++ -c -o obj/sim_mips.o sim_mips.c -Wall -g -I ../include
g++ -c -o obj/run.o run.c -Wall -g -I ../include
g++ -o ../sim_mips obj/memory.o obj/sim_mips.o obj/run.o -lm -Wall -g -I ../include
```

ou até:

```
gcc -c -o obj/memory.o memory.c -Wall -g -I ../include
gcc -c -o obj/sim_mips.o sim_mips.c -Wall -g -I ../include
gcc -c -o obj/run.o run.c -Wall -g -I ../include
gcc -o ../sim_mips obj/memory.o obj/sim_mips.o obj/run.o -lm -Wall -g -I ../include
```

Para executar precisa ir para a pasta do executável e chamar `./sim_mips` (o programa utiliza

caminho relativo para achar os arquivos e estou supondo a raiz como caminho atual).

Github

Este projeto se encontra no Github no perfil do aluno neste repositório (https://github.com/victoragcosta/SimuladorMIPS.git)