Exercício Programa 2
Dizia o Minips que a aritmética...

Arquitetura de Computadores - 2021.Q1
http://professor.ufabc.edu.br/~e.francesquini/2021.q1.ac/

Prof. Emilio Francesquini
e.francesquini@ufabc.edu.br

17 de março de 2021

1 O Projeto

Nesta segunda fase do projeto vamos incrementar ainda mais o MINIPS, nosso mini emulador de MIPS-32. Assim como na fase 1, para tirar a nota máxima neste EP, é condição necessária (mas não suficiente) que seu programa execute corretamente todos os programas de entrada disponibilizados. Eles incluem, naturalmente, aqueles da fase 1 assim como alguns adicionais para a fase 2.

Nesta segunda fase, a principal alteração é a inclusão de operações aritméticas. Elas incluem a multiplicação e divisão de inteiros assim como algumas operações do co-processador matemático, incluindo assim suporte a diversas operações de ponto flutuante. Além disto, nessa segunda fase vamos implementar o *branch delay slot* que faz com que seja possível escrevermos códigos não triviais para o MINIPS usando usando a linguagem C compilada com o GCC, por exemplo.

A dica dada para a implementação da fase 1 continua valendo: implemente o emulador, instrução a instrução, conforme elas forem aparecendo nos programas de teste. Lembre-se: é um MINIPS, não o MIPS completo, então tudo bem se, por enquanto, ele só rodar esses poucos problemas de exemplo.

2 Convenções

Todas as convenções da fase 1 sem mantêm. Além disto:

- Nessa segunda fase incluiremos suporte (ainda que limitado) à execução de programas compilados com o GCC (mais sobre isso abaixo). Por simplicidade, continuaremos a não usar o formato ELF. Contudo, pra fazer isso funcionar, além da seção de dados (`.data`) e de texto (`.text`) que já tínhamos na fase 1, **será necessário incluir o carregamento de uma terceira seção chamada *rodata*[1]** (*read-only data*). Ela pode estar presente ou não e estará armazenada em um arquivo (que segue a mesma convenção de nomes do EP1) com extensão `.rodata`. **Caso o arquivo esteja presente, ele deve ser carregado no endereço `0x00800000`.** Assim como as demais seções, esta nova seção é escrita em palavras de 32 bits, little-endian.

- Nesta segunda fase também deverão ser implementados os registradores de ponto flutuante do coprocessador 1 (`$f0` a `$f31`) além dos registradores específicos `hi` e `lo`. Notem que os registradores específicos não começam com `$`. Isso indica que eles não podem ser escritos ou lidos diretamente (como os registradores de propósito geral) e exigem o uso de instruções específicas como `mflo`, por exemplo. Adicionalmente, mesmo que os registradores do coprocessador 1 sejam de propósito geral, também é preciso usar instruções específicas para acessá-los. Por exemplo: `add $f1, $f2, $f3` não é uma instrução válida. Em tempo, os números dos registradores `$f0` a `$f31` são de 0 a 31 (assim como os registradores comuns). Para diferenciar qual deles deve ser usado por uma instrução específica você deve se basear na documentação da ISA do processador.

  Apesar de não ser necessário para a realização do EP2, assim como existe uma convenção para o uso dos registradores de propósito geral, também existe uma convenção para o uso dos registradores de ponto flutuante. Ela está descrita na tabela abaixo.

Registrador de FP	**Uso**
`$f0` .. `$f2`	Armazena resultados de operações de ponto flutuante
`$f4` .. `$f10`	Registradores para uso temporário.
`$f12` .. `$f14`	Usados para os 2 primeiros parâmetros de chamadas de funções
`$f16` .. `$f18`	Registradores temporários para avaliação de expressões
`$f20` .. `$f30`	Registradores salvos, valor é preservado entre chamadas de funções

[1] A seção `.rodata` guarda dados definidos no seu código que não podem ser modificados (como por exemplo literais).

Notas:

- registradores vêm aos pares, cada um com 32 bits
- doubles só podem ser armazenados em registradores pares (pois ele usa, na verdade, o registrador e o próximo)
- floats, por sua vez, podem ser armazenados em qualquer registrador
- Instruções de ponto flutuante têm seu próprio formato. Verifique o Greencard assim como a documentação da ISA do MIPS (link abaixo)

3 Syscalls

Nesta fase, além das syscalls do EP1, será necessário implementar:

Valor de `$v0`	**Descrição da Syscall**
2	Imprime o float contido em `$f12`
3	Imprime o double contido em `$f12` / `$f13`
6	Lê um float e o coloca em `$f0`
7	Lê um double e o coloca em `$f0` / `$f1`

4 Comentários sobre novas instruções (e não tão novas assim)

Como vamos implementar a funcionalidade de *branch delay slot*, uma instrução que era virtualmente desnecessária antes agora passa a ser útil: `nop`. A instrução `nop` pode ser vista como sendo do tipo R com opcode `0x00` e funct `0x00`. Ou seja, pode se confundir com a instrução `sll` que compartilha o mesmo opcode/funct. Como saber qual é qual? A diferença é que no caso do `nop`, todos os campos são zerados, então a instrução codificada em hexa seria `0x00000000`. Em tempo, se você interpretar isso como `sll`, tudo continua funcionando, é apenas um shift para a esquerda de 0 posição no registrador `$zero`. Ou seja, sem efeitos. Cuidado para implementar a saída da função `decode` corretamente.

Nos códigos implementados em C e compilados com o GCC, também vai aparecer a instrução `break` (opcode `0x00` funct `0x0d`) . Não é preciso implementá-la (mas é preciso saber decodificá-la!). Durante a execução trate-a como um `nop`.

4.1 Branch delay slot

Devido a estrutura em pipeline do MIPS, as instruções de *branch* como (`j, jal, jalr, jr, beq, bgez, blez, bne`, ...) têm, na verdade, um comportamento um pouco diferente do que tínhamos na fase 1. Por exemplo:

```
jal PARA_LONGE
ins1 # branch delay slot
ins2
...
...
...
PARA_LONGE:
jr $ra
ins3 #branch delay slot
...
...
...
```

Vai executar a seguinte sequência de instruções:

1. `jal PARA_LONGE`
2. `ins1`
3. `jr $ra`
4. `ins3`
5. `ins2`

As instruções `ins1` e `ins3` estão no *branch delay slot*. Caso haja uma instrução de *branch* no *branch delay slot*, o resultado é indefinido. Pode ser que o computador crie dentes e coma o seu gato. Nestes casos o correto seria colocar um `nop` no *slot*.

Em resumo, isso significa duas coisas ficaram diferentes do EP1:

- `$ra` recebe `PC + 8` e não `+4` como anteriormente (para instruções que alteram esse registrador)
- Todas as instruções colocadas após um *branch* vão ser executadas, independente do seu resultado (caso seja condicional).

Os programas de entrada da fase 1 foram alterados (corrigidos?) para ter esse comportamento. Você pode ficar surpreso em ver que nem todas as entradas atualizadas rodam corretamente no seu emulador antigo. 😕 Contudo, basta acertar o *branch delay slot* que deve dar tudo certo! 🎉

O emulador MARS tem suporte a execução com *branch delay slots*. Para ativá-la, vá em *"Settings"* e escolha *"Delayed branching"*.

Atenção: assegure-se de usar tanto a versão nova da implementação de referência quanto das entradas para o EP2.

5 Recursos e cuidados

Continuam os mesmos da fase 1. Além disto, o emulador de referência implementado pelo professor foi atualizado. Você pode baixar a nova versão nos links abaixo:

- Linux (x86-64): minips
- Windows (x86-64): minips.exe

Além dos recursos já citados (Greencard, MARS, Godbolt, etc) você vai provavelmente precisar usar o manual "de verdade" do MIPS: MIPS32 Architecture For Programmers Volume II: The MIPS32 Instruction Set. Esse documento contém a lista completa das instruções, incluindo algumas que serão necessárias mas que não constam no Greencard.

5.1 Compilando códigos em C para o MINIPS

Junto com as entradas (atualizadas) que já estavam presentes no EP 1 (1 a 8), teremos também 8 entradas adicionais (9 a 16). Algumas dessas novas entradas vêm no mesmo pacote que lhes é familiar: `.data`, `.text`, opcionalmente `.rodata` e `.asm`. Outras entradas, contudo, foram geradas diretamente a partir de um programa em C. Neste caso estão incluídos `.data`, `.text`, `.rodata` e `.c`.

Caso tenha curiosidade em como os arquivos `.data`, `.text`, `.rodata` foram gerados a partir do `.c`, dê uma olhada no Makefile que acompanha o pacote. Você pode, inclusive, criar os seus próprios programas em C e rodar dentro do seu MINIPS! `\o/`

Além disto, pode ser interessante usar a ferramenta `objdump` nos `.elf` de cada uma das entradas dadas (ou das suas próprias) para ver o código descompilado. **Atenção:** cuidado para usar a versão do `objdump` para MIPS little-endian, ou você pode acabar vendo coisas sem sentido. Na maior parte

das distribuições Linux o nome do pacote é algo como `mipsel-linux-gnu-gcc` e `mipsel-linux-gnu-binutils` [2].

Rodando em um Manjaro em x86 o comando fica:

```
$ mipsel-linux-gnu-objdump -D 13.arit.elf
13.arit.elf:     file format elf32-tradlittlemips

Disassembly of section .text:

00400000 <__start>:
  400000:       27bdffe0        addiu   sp,sp,-32
  400004:       afbf001c        sw      ra,28(sp)
  400008:       0c100194        jal     400650 <main>
  40000c:       00000000        nop

00400010 <p_int>:
  400010:       24020001        li      v0,1
  400014:       00802025        move    a0,a0
  400018:       0000000c        syscall
  40001c:       03e00008        jr      ra
  400020:       00000000        nop

00400024 <p_float>:
  400024:       44036000        mfc1    v1,$f12
  400028:       24020002        li      v0,2
  40002c:       0000000c        syscall
  400030:       03e00008        jr      ra
  400034:       00000000        nop
...
...
```

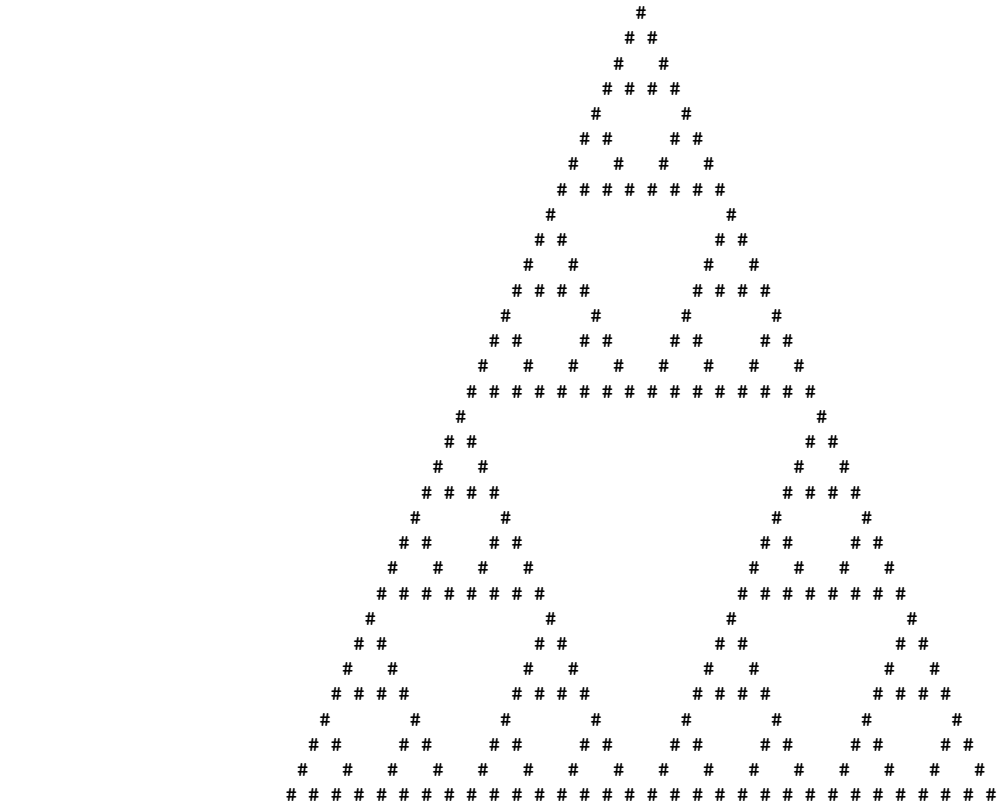
6 Entradas e saídas esperadas

O seu programa deve se comportar como o programa de exemplo fornecido oferecendo duas opções de execução. Uma que decodifica o binário e imprime o assembly e a outra que efetivamente executa o código.

Exemplo:

[2] Fico me perguntando porque é el e não le, de *little endian*. Será que é porque está escrito em *little endian*? 😉

```
$ ./minips run 16.automato
Digite a regra que deseja rodar (tente 18, 60, 30, 94): 18
Iteracoes: 31
```



```
Execution finished successfully
--------------------------
Instruction count: 95539 (R: 37656 I: 55284 J: 2599 FR: 0)
Simulation Time: 1.65 sec.
Average IPS: 57886.54


Simulated execution times for:
--------------------------
Monocycle
        Cycles: 95539
        Frequency: 8.4672 MHz
        Estimated execution time: 0.0113 sec.
        IPC: 1.00
        MIPS: 8.47
Pipelined
        Cycles: 95543
        Frequency: 33.8688 MHz
        Estimated execution time: 0.0028 sec.
        IPC: 1.00
        MIPS: 33.87
Speedup Monocycle/Pipeline: 4.00x
```

Em caso de dúvidas, envie para o Discord da disciplina.

As entradas de exemplo podem ser baixadas aqui (entradas.zip) e contêm:

Entrada	Descrição
`01.soma`	Soma 3 + 4 e imprime o resultado
`02.hello`	Hello World!
`03.input`	Lê um inteiro e imprime na saída
`04.branches`	Testes simples com instruções de branches **(agora com delay slots!)**
`05.fibo`	Imprime a sequência de Fibonacci
`06.collatz`	Conta quantos elementos tem a sequência de Collatz
`07.loadstore`	Testes simples com instruções de load/store
`08.sort`	QuickSort em um vetor de elementos
`09.contador`	Seu emulador é uma arquitetura de von Neumann?
`10.hello++`	Testa a instrução `jalr`
`11.surpresinha`	Mas, é de von Neumann mesmo?
`12.branch_delay_slot`	Auto explicativo.
`13.arit`	Testes aritméticos em (C)
`14.flutuantes`	Testes aritméticos em ponto flutuante
`15.pi`	Calcula π usando Madhava–Leibniz (C)
`16.automato`	Autômato celular simples (C)

Note que o programa de referência lê automaticamente os arquivos com extensão `.text`, `.data` e `.rodata` (caso exista) quando é fornecido o prefixo do teste desejado.

6.1 Extra

Apesar de não ser obrigatório para a fase 2 do projeto, você pode imprimir as estatísticas tal qual é feito pelo programa de referência. O hardware que "inspirou" essas estatísticas é o MIPS que era usado pelo Playstation 1. A palavra inspirou foi escolhida deliberadamente. As estatísticas atuais não levam em conta tempo de acesso à memória, utilização das caches, etc. Então nada mais são do que um chute grosseiro. Nós vamos melhorar isso na fase 3.

Por enquanto, caso aceitem fazer essa tarefa extra (que vai virar obrigatória na fase 3) assumam que o processador funciona a uma frequência de 33.8688 MHz, ou seja, um período de 29525,6991686 picossegundos com um pipeline de 5 estágios, tal qual visto em aula. Assumam também que a versão monociclo funciona com uma frequência 4 vezes menor.

6.2 Bônus

Você terá 1 ponto adicional na nota se você achar um bug na implementação de referência feita pelo professor. Não contam como bugs: passar instruções inválidas, reclamar de mensagens de erros toscas, passar parâmetros inválidos... Basicamente, é um programa que tem garantia vitalícia limitada! Máximo de 2 pontos por cliente. 😜

7 Entrega

Junto com o seu código você deve entregar um brevíssimo relatório (máximo de 2 páginas) contendo obrigatoriamente:

- Nome completo
- RA
- Usuário do GitHub
- Link para vídeo no YouTube ou qualquer outro lugar acessível pelo professor.

O seu vídeo deve ter duração máxima de 5 minutos, deve mostrar o seu código executando e deve destacar os pontos fortes da sua implementação assim como as eventuais limitações e dificuldades que você teve durante a implementação.

A entrega do código e do relatório deve ser feita pelo GitHub Classroom através do link `https://classroom.github.com/a/-MJ_AOaV`. Será considerado como entrega o último *commit* (não esqueça de dar *push*) no repositório até a **data limite de 11/04/2021**.

Projetos entregues com atraso sofrerão descontos seguindo a seguinte tabela:

Dias em atraso	Nota máxima
1 dia	7
2 dias	6
3 dias	5
>3 dias	0

Para discussões, dúvidas e comentários utilize o Discord em https://discord.gg/9RtRcx3.