

Trabalho 3 - OAC - Simulador RISC-V

Kálley Wilkerson - 170038050

Resumo

Este arquivo é a documentação de um simulador da arquitetura **RV321**. Aqui serão documentadas as funções do programa, mostrados alguns dos testes feitos e também instruções de como usar. O objetivo é fazer um simulador que consiga executar as principais intruções da arquitetura.

1 Plataforma utilizada

Este projeto foi feito utilizando como compilador o **gcc 7.3.0** na plataforma linux **Ubuntu 18.04** e feito em um editor de texto e não em uma *IDE*, dessa forma para executar o programa basta compilá-lo diretamente pelo terminal.

2 Como utilizar

Para começar que você esteja no terminal dentro da pasta **src** deste projeto. Para rodar o programa digite **./main** no terminal. Se o executável *main* não existir, compile-o digitando **make main** ou **make all** que irá compilar também os testes.

Para limpar todos os arquivos **.o** e executáveis para recompilar tudo de novo utilize o comando **make clean**.

Para compilar os testes utilize o comando **make run_tests** ou **make all**. Para rodar os testes digite **./run_tests**.

Caso queira mudar o programa rodado por **./main** abra o arquivo **main.c** e modifique no local indicado no arquivo.

3 Funções implementadas

As funções de acesso a memória não serão documentadas neste projeto pois sua documentação já foi feita no trabalho 2.

3.1 `init_simulator()`

Esta função inicializa algumas variáveis como os registradores **pc** e **ri** entre outros, também zera a memória e o banco de registradores.

3.2 `read_mem()`

Ler o conteúdo de um arquivo binário e guarda ele em um vetor que representa a memória do computador.

Argumentos:

- **mem** - vetor que é a memória simulada
- **filename** - *string* que contém o caminho até o arquivo a ser aberto.

3.3 load_data()

Copia o conteúdo lido de um binário que está guardado em um vetor chamado **data_mem** para a área de memória *RAM* do vetor **memory**.

3.4 fetch()

Carrega a palavra apontada por **pc** da memória para o registro **ri**. E então passa o **pc** para a próxima instrução.

3.5 decode()

Decodifica a instrução presente no registro **ri** em vários campos relevantes como imediatos, *opcodes* e *funct*. Ela utiliza as funções **getField** e **setField** várias vezes para separar as partes da palavra presente em **ri**.

3.6 execute()

Esta é finalmente a função que executa a instrução presente em **ri**. Ela utiliza os campos separados pela função **decode** para diferenciar entre as diversas instruções possíveis cobertas por este simulador. Sua estrutura é feita basicamente de estruturas **switch-case** para fazer as diferenciações.

3.7 dump_mem()

Imprime na tela os valores da memória que estão em um intervalo dado pelos seus argumentos em um certo formato fornecido, podendo ser esse formato hexadecimal ou decimal.

Argumentos:

- **start** - endereço do *byte* indicando por onde começar a imprimir
- **end** - endereço do *byte* de até onde imprimir
- **format** - formato da impressão ('h' ou 'd')

3.8 dump_reg()

Imprime na tela os valores de todos os registradores, podendo ser essa impressão em formato hexadecimal ou decimal, dado pelo argumento **formato** da função.

3.9 step()

Executa a sequência **fetch** - **decode** - **execute**.

3.10 run()

Executa uma sequência de **step**'s que só para quando a *flag* de fim de execução é ativada ou quando o limite do arquivo de texto é atingido.

4 Testes

Este programa foi feito utilizando um desenvolvimento orientado a testes bem simples, para executar os testes feitos para várias partes do programa execute o executável **run_tests** na pasta **src**, caso necessário, recompila o programa com **make all**.

Um dos testes feito foi um programa que calcula os primeiros valores da sequência de Fibonacci. A Figura 1 mostra a saída mostrada no simulador *RARS* tomado como base. A Figura 2 mostra o resultado na execução do arquivo binário no simulador desse projeto, podemos observar que o resultado é o mesmo. O código original deste teste está na pasta **progs** no arquivo **fibonacci.asm**.

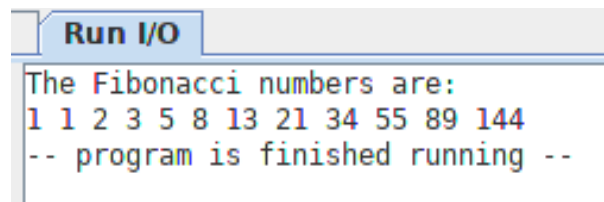


Figura 1: Fibonacci, saída mostrada no RARS

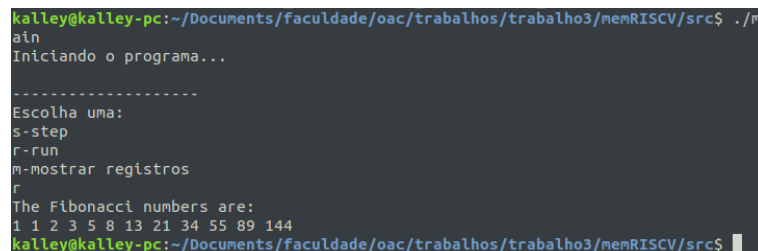
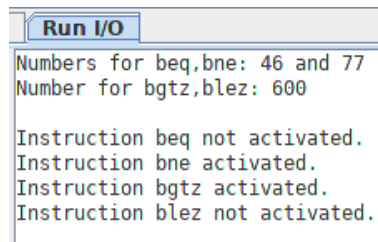


Figura 2: Execução do programa utilizando o simulador aqui documentado

Outro teste feito foi um que testa a execução de branches e *jumps*. O código está na pasta **progs** no arquivo **IJ_BRANCH_JUMP.asm**. A Figura 3 mostra o resultado da execução do código no *RARS*.

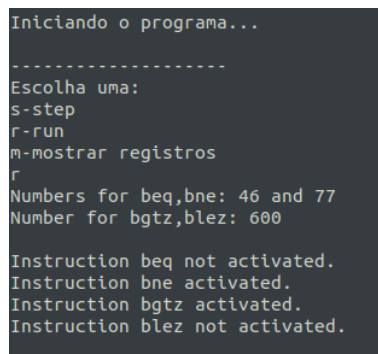
Outro teste feito foi o que está no arquivo **decbin.asm** dentro da pasta **progs**. A Figura 5 mostra o resultado da execução deste arquivo no *RARS* e a Figura 6 mostra o resultado da execução neste simulador, como pode ser visto, o resultado deste teste também foi satisfatório.



```
Run I/O
Numbers for beq,bne: 46 and 77
Number for bgtz,blez: 600

Instruction beq not activated.
Instruction bne activated.
Instruction bgtz activated.
Instruction blez not activated.
```

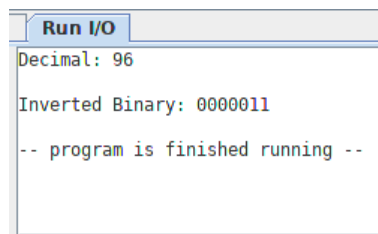
Figura 3: Execução do teste de branches e jumps no RARS



```
Iniciando o programa...
-----
Escolha uma:
s-step
r-run
m-mostrar registros
r
Numbers for beq,bne: 46 and 77
Number for bgtz,blez: 600

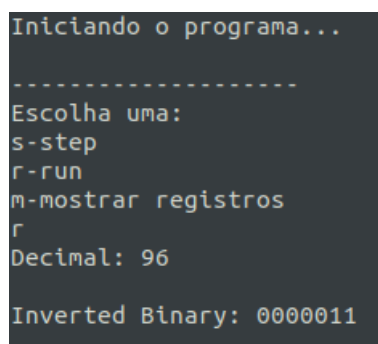
Instruction beq not activated.
Instruction bne activated.
Instruction bgtz activated.
Instruction blez not activated.
```

Figura 4: Execução do teste de jumps e branches neste simulador



```
Run I/O
Decimal: 96
Inverted Binary: 0000011
-- program is finished running --
```

Figura 5: Execução de decbin no RARS



```
Iniciando o programa...
-----
Escolha uma:
s-step
r-run
m-Mostrar registros
r
Decimal: 96
Inverted Binary: 0000011
```

Figura 6: Execução de decbin neste simulador