Lab 4: Simulação e Teste

Universidade Estadual de Feira de Santana
Departamento de Tecnologia, Área de Eletrônica e Sistemas
TEC499 MI - Sistemas Digitais

2018.1

Sumário

Introdução

Neste laboratório, você aprenderá como simular seus módulos e testá-los em software antes de descarregá-los em um dispositivo FPGA. Até então, nos laboratórios passados, você vem descarregando seu circuito diretamente na placa FPGA, passando direto pelo processo de síntese, mapeamento e geração do *bitstream* do seu projeto. Isso é viável para circuitos simples que podem ser rapidamente sintetizados e verificados diretamente na placa. Entretanto, esta abordagem não é válida em projetos de larga escala.

Ao longo desta semana, você aprenderá como simular um projeto em hardware e escrever conjuntos de teste (*test benches*). Estas estruturas de teste representam os elementos essenciais no processo de verificação e validação de circuitos grandes e complexos.

Pré-laboratório

Sugerimos fortemente que você inicie a descrição do código Verilog de antemão. Considere que você terá apenas uma semana para se preparar para a apresentação final deste laboratório.

1. Procedimento

Neste laboratório, você descreverá uma ALU típica de processadores MIPS. Você também aprenderá técnicas de simulação e depuração do seu projeto. Estas estratégias são um elemento crítico dentro do fluxo de desenvolvimento e de fundamental importância para o sucesso do seu projeto.

1.1 Pré-laboratório

Certifique-se de completar os itens apresentados nesta seção antes de ir para o laboratório. Você provavelmente não terminará o laboratório durante a sessão se você não completar o pré-lab.

No pré-lab, complete as seguintes tarefas:

1. Leia atentamente este roteiro.
2. Faça o download dos arquivos do laboratório.
3. Escreva os *test benches* em Verilog nos arquivos `ALUTestVectorTestbench.v` e `ALUTestbench.v`.

1.2 Relevância do Projeto

A ALU que você irá implementar neste laboratório corresponde às operações associadas ao conjunto de instruções de um processador MIPS tradicional. Considere com atenção ao padrão do projeto e como a ALU deve funcionar no contexto do processador MIPS. Em particular, é importante verificar a separação entre o *data path* e do controle utilizado neste sistema. Esta e outras características serão melhor exploradas ao longo de todo o curso.

O conjunto de instruções que sua ALU deve implementar é apresentado nas tabelas a seguir. Antes de se assustar com o tamanho das tabelas e a quantidade de instruções, lembre-se de que você implementará apenas a ALU e o decodificador da ALU, não o processador inteiro. Estas tabelas estão aqui apenas para referência. Note ainda que a ALU não precisa fazer nada para instruções de *branch* e *jump* (e.g.: ela pode simplesmente retornar 0 na saída).

1.3 Especificação Funcional

A funcionalidade de cada instrução é apresentada na tabela a seguir. Esteja atento para a descrição RTL correspondente a cada instrução, pois diferenças sutis podem aparecer, especialmente nas **instruções de deslocamento**, em que existe a troca da ordem dos operandos:

- `R[$x]` indica o registrador cujo endereço é `x`
- `SEXT` indica extensão de sinal
- `ZEXT` indica extensão de zero
- `BMEM` indica acesso a memória alinhado por *byte*
- `HMEM` indica acesso a memória alinhado por meia palavra
- `WMEM` indica acesso a memória alinhado por palavra
- `PC` indica o endereço de memória da instrução

Note que, na instrução `LUI`, o operando a ser carregado na parte alta do imediato é ambíguo. Dessa forma, quando estiver implementando sua ALU, por favor, carregue na entrada B o campo do imediato.

Mnemônico	**Descrição RTL**	**Nota**
LB	R[$rt] = SEXT(BMEM[R[$rs]+SEXT(imm)) [31:0]])	delayed
LH	R[$rt] = SEXT(HMEM[R[$rs]+SEXT(imm)) [31:1]])	delayed
LW	R[$rt] = WMEM[(R[$rs]+SEXT(imm)) [31:2]]	delayed
LBU	R[$rt] = ZEXT(BMEM[R[$rs]+SEXT(imm)) [31:0]])	delayed
LHU	R[$rt] = ZEXT(HMEM[R[$rs]+SEXT(imm)) [31:1]])	delayed
SB	BMEM[R[$rs]+SEXT(imm)[31:0]] = R[$rt][7:0]	
SH	HMEM[R[$rs]+SEXT(imm)[31:1]] = R[$rt][15:0]	
SW	WMEM[R[$rs]+SEXT(imm)[31:2]] = R[$rt]	
ADDIU	R[$rt] = R[$rs] + SEXT(imm)	
SLTI	R[$rt] = R[$rs] < SEXT(imm)	
SLTIU	R[$rt] = R[$rs] < SEXT(imm)	unsigned compare
ANDI	R[$rt] = R[$rs] & ZEXT(imm)	
ORI	R[$rt] = R[$rs] | ZEXT(imm)	
XORI	R[$rt] = R[$rs] ^ ZEXT(imm)	
LUI	R[$rt] = imm,16'b0	
SLL	R[$rd] = R[$rt] << shamt	
SRL	R[$rd] = R[$rt] >> shamt	
SRA	R[$rd] = R[$rt] >>> shamt	
SLLV	R[$rd] = R[$rt] << R[$rs]	
SRLV	R[$rd] = R[$rt] >> R[$rs]	
SRAV	R[$rd] = R[$rt] >>> R[$rs]	
ADDU	R[$rd] = R[$rs] + R[$rt]	
SUBU	R[$rd] = R[$rs] - R[$rt]	
AND	R[$rd] = R[$rs] & R[$rt]	
OR	R[$rd] = R[$rs] | R[$rt]	
XOR	R[$rd] = R[$rs] ^ R[$rt]	
NOR	R[$rd] = ~R[$rs] & ~R[$rt]	
SLT	R[$rd] = R[$rs] < R[$rt]	
SLTU	R[$rd] = R[$rs] < R[$rt]	unsigned compare
J	PC = PC[31:28], target, 2'b0	delayed
JAL	R[31] = PC + 8; PC = PC[31:28], target, 2'b0	delayed
JR	PC = R[$rs]	delayed
JALR	R[$rd] = PC + 8; PC = R[$rs]	delayed
BEQ	PC = PC + 4 + (R[$rs] == R[$rt] ? SEXT(imm) << 2 : 0)	delayed
BNE	PC = PC + 4 + (R[$rs] != R[$rt] ? SEXT(imm) << 2 : 0)	delayed
BLEZ	PC = PC + 4 + (R[$rs] <= 0 ? SEXT(imm) << 2 : 0)	delayed
BGTZ	PC = PC + 4 + (R[$rs] > 0 ? SEXT(imm) << 2 : 0)	delayed
BLTZ	PC = PC + 4 + (R[$rs] < 0 ? SEXT(imm) << 2 : 0)	delayed
BGEZ	PC = PC + 4 + (R[$rs] >= 0 ? SEXT(imm) << 2 : 0)	delayed

1.4 Codificação ISA

31 26	25 21	20 16	15 11	10 6	5 0	
opcode	rs	rt	rd	shamt	funct	R-type
opcode	rs	rt	immediate			I-type
opcode	target					J-type
Load and Store Instructions						
100000	base	dest	signed offset			LB rt, offset(rs)
100001	base	dest	signed offset			LH rt, offset(rs)
100011	base	dest	signed offset			LW rt, offset(rs)
100100	base	dest	signed offset			LBU rt, offset(rs)
100101	base	dest	signed offset			LHU rt, offset(rs)
101000	base	dest	signed offset			SB rt, offset(rs)
101001	base	dest	signed offset			SH rt, offset(rs)
101011	base	dest	signed offset			SW rt, offset(rs)
I-Type Computational Instructions						
001001	src	dest	signed immediate			ADDIU rt, rs, signed-imm.
001010	src	dest	signed immediate			SLTI rt, rs, signed-imm.
001011	src	dest	signed immediate			SLTIU rt, rs, signed-imm.
001100	src	dest	zero-ext. immediate			ANDI rt, rs, zero-ext-imm.
001101	src	dest	zero-ext. immediate			ORI rt, rs, zero-ext-imm.
001110	src	dest	zero-ext. immediate			XORI rt, rs, zero-ext-imm.
001111	00000	dest	zero-ext. immediate			LUI rt, zero-ext-imm.
R-Type Computational Instructions						
000000	00000	src	dest	shamt	000000	SLL rd, rt, shamt
000000	00000	src	dest	shamt	000010	SRL rd, rt, shamt
000000	00000	src	dest	shamt	000011	SRA rd, rt, shamt
000000	rshamt	src	dest	00000	000100	SLLV rd, rt, rs
000000	rshamt	src	dest	00000	000110	SRLV rd, rt, rs
000000	rshamt	src	dest	00000	000111	SRAV rd, rt, rs
000000	src1	src2	dest	00000	100001	ADDU rd, rs, rt
000000	src1	src2	dest	00000	100011	SUBU rd, rs, rt
000000	src1	src2	dest	00000	100100	AND rd, rs, rt
000000	src1	src2	dest	00000	100101	OR rd, rs, rt
000000	src1	src2	dest	00000	100110	XOR rd, rs, rt
000000	src1	src2	dest	00000	100111	NOR rd, rs, rt
000000	src1	src2	dest	00000	101010	SLT rd, rs, rt
000000	src1	src2	dest	00000	101011	SLTU rd, rs, rt
Jump and Branch Instructions						
000010	target					J target
000011	target					JAL target
000000	src	00000	00000	00000	001000	JR rs
000000	src	00000	dest	00000	001001	JALR rd, rs
000100	src1	src2	signed offset			BEQ rs, rt, offset
000101	src1	src2	signed offset			BNE rs, rt, offset
000110	src	00000	signed offset			BLEZ rs, offset
000111	src	00000	signed offset			BGTZ rs, offset
000001	src	00000	signed offset			BLTZ rs, offset
000001	src	00001	signed offset			BGEZ rs, offset

2. Recursos

Para obter o conteúdo do laboratório, faça o download dos arquivos no seu quadro do Trello. Ao extrair o arquivo, certifique-se de que você consegue visualizar os diretórios `/src` e `/sim` dentro da pasta `lab4`. Note que não há `Makefile` para síntese e implementação do circuito, uma vez que não iremos sintetizar nosso projeto ou mesmo descarregá-lo na placa. Neste laboratório, iremos apenas verificar se o circuito funciona, em nível de simulação funcional, utilizando o **ModelSim**. Nos próximos laboratórios aprenderemos sobre outra técnica de simulação caracterizada por levar em consideração as características físicas da estrutura interna do dispositivo.

2.1 Testando o Projeto

Antes de iniciar a descrição de qualquer um dos módulos, você deve primeiro escrever os testes. Dessa forma, poderá ser capaz de testar seus módulos a medida em que o for descrevendo. Outra razão pela qual você deve escrever seus testes antes é que se você precisar modificar o seu módulo, você sempre poderá executá-los no sentido de garantir que o circuito ainda funciona. Mais importante, você também deve entender a funcionalidade esperada de cada módulo antes de escrever tanto o código quanto os testes.

Existem diversas técnicas que podem ser empregadas para testar seu projeto. Neste laboratório, você testará apenas dois módulos, e por isso fará uso de testes de unidade. Para os seus futuros projetos, é esperado que você escreva testes de unidade para qualquer módulo, assim como implemente também testes de integração.

2.2 Test bench em Verilog

Uma das maneiras mais simples de testar códigos Verilog é a partir de arquivos *test bench*. O esqueleto de um *test bench* foi fornecido para você no arquivo `ALUTestbench.v`. Algumas partes importantes deste arquivo precisam ser analisadas:

1. `` `timescale 1ns / 1ps `` - Esta expressão especifica a referência de unidade de tempo e precisão associada. Isso significa que qualquer atraso no *test bench* leva 1ns e a simulação deve operar com uma precisão de no máximo 1ps. Modifique esses valores caso ache necessário durante a depuração.

2. A geração do clock é feita a partir do código abaixo. Uma vez que a ALU é puramente combinacional, esta parte não é necessária. Você pode tratar este trecho como referência quando estiver escrevendo um *test bench* para circuitos sequenciais.

```
parameter Halfcycle = 5; // half period is 5ns
localparam Cycle = 2*Halfcycle;
reg Clock;
// Clock Signal generation:
initial Clock = 0;
always
  #(Halfcycle) Clock = ~Clock;
```

   (a) O bloco `initial` define o valor do clock para `0` no inicio da simulação. Você deve inicializar o clock em `0`, caso contrário você estará tentando modificar as entradas ao

mesmo tempo em que o clock muda e isso pode resultar em um mal comportamento do seu circuito.

(b) Você deve usar um bloco `always` sem lista de sensibilidade para fazer com que o clock modifique por si só

3. `localparam loops = 25;` - quantidade de iterações que o seu *test bench* deve realizar. Lembre-se de modificar este valor para aumentar a cobertura do seu teste.

4. `task checkOutput;` - esta rotina encapsula alguns artifícios da linguagem que você poderá replicar quantas vezes for necessário. Note que isso **não** é equivalente à uma função (uma vez que Verilog também possui o ambiente `function`).

5. `{$random} & 31'h7FFFFFFF` - o comando `$random` produz um inteiro de 32 bits pseudor-randômico. Usaremos uma máscara para o resultado no sentido de obter a faixa de valores apropriada.

Para estes dois módulos, as entradas e saídas que você deve considerar são: `opcode, funct, A, B` e `Out`. Dessa forma, para testar seu projeto de modo a garantir a eficácia da sua implementação, você deve realizar todas as possíveis combinações entre `opcode` e `funct`. Além disso, deve verificar que a saída correta (`Out`) é gerada a partir dos valores de `A` e `B` que você introduziu.

O *test bench* fornecido junto com este laboratório produz valores randômicos para `A` e `B` e calcula `REFout = A + B`. Ele também possui chamadas à *task* `checkOutput` para carga e descarga de instruções, a partir das quais a ALU deve realizar somas. Esta rotina é reponsável por verificar a corretude do resultado produzido por sua ALU e interromper a simulação em caso de falhas. Você deve escrever os testes para as combinações restantes entre `opcode` e `funct`.

Se necessário, lembre-se de restringir `A` e `B` a valores razoáveis (e.g. utilize máscaras, ou certifique-se de que eles não são iguais a zero) no sentido de garantir que uma função seja efetivamente testada. Escreva também testes onde as entradas `A`, `B` e a saída `Out` sejam *hardcoded*. Isso quer dizer que, na região do código indicada, você deve definir valores prefixados para as entradas do módulo testado. Testes *hardcoded* são importantes para analisar casos de teste específicos, como no exemplo a seguir. Neste caso, queremos verificar especificamente o resultado da soma entre os valores definidos em `A` e `B`.

```
//
// Hardcoded test example
//
opcode = `RTYPE;
funct  = `ADDU;
A = 32'b10111000000000001011100101111011; // problematic input for A
B = 32'b00100000000000001010111011001010; // problematic input for B
REFout = A + B; // expected result
#1;
checkOutput(opcode, funct);
```

Utilize o exemplo acima para produzir seus próprios testes *hardcoded*. Considere aindaq que você deverá produzir todas as combinações possíveis de `opcode` e `funct`.

2.3 Test bench por Vetor de Teste

Uma forma alternativa de testar o seu circuito é utilizando um vetor de teste. Este tipo de estrutura de teste, é formada a partir de um conjunto de vetores de bits mapeados para as entradas e saídas

do seu módulo. Todas as entradas podem ser introduzidas de uma só vez se você estiver testando um circuito de lógica combinacional, como neste laboratório, ou ao longo de intervalos de tempo para circuitos de lógica sequencial (e.g.: uma FSM).

Você escreverá um *test bench* em Verilog que lê partes de um vetor de bits que correspondem às entradas do circuito, as introduz no módulo a ser testado e compara a saída com os bits de saída também presentes no vetor de teste. O seu vetor de teste deve adotar o formato a seguir.

```
[107:102] = opcode
[101:96]  = funct
[95:64]   = A
[63:32]   = B
[31:0]    = REFout
```

Abra o esqueleto fornecido a você no arquivo `ALUTestVectorTestbench.v`. Você precisa completar o módulo utilizando o comando `$readmemb` para ler o arquivo que contem os vetores de teste (`testvectors.input`), localizado no diretório `sim/tests`. Além disso, você deverá incluir alguns comandos `assign` para atribuir as partes do vetor de teste a variáveis do tipo *registers*. Finalmente, escreva um laço `for` para iterar sobre todos os vetores de teste. Lembre-se de modificar o valor do parâmetro `testcases`, de modo que este corresponda ao total de vetores de testes presentes no arquivo `testvectors.input`.

A sintaxe do laço `for` pode ser encontrada no arquivo `ALUTestbench.v`. O comando `$readmemb` possui como argumentos o nome do arquivo e um *register* bidimensional, e.g.:

```
reg [5:0] simpson [0:20];
$readmemb("homer.input", simpson);
```

No exemplo acima, estamos declarando um vetor de 6 posições, cada uma delas composta por palavras de 21 bits.

2.4 Escrevendo Vetores de Teste

Adicionalmente, você também terá que gerar vetores de teste reais para serem usados no seu *test bench*. Um vetor de teste pode ser tanto gerado em Verilog (da mesma forma como geramos `A` e `B` usando um gerador de números randômicos de forma iterativa em torno das combinações possíveis entre `opcode` e `funct`), ou usando alguma linguagem de *script*. Uma vez que nós já escrevemos um *test bench* em Verilog para a nossa ALU e o seu respectivo decodificador, nós iremos escrever alguns testes na mão.

Vetores de teste possuem o seguinte formado da esquerda para a direita (MSB no final):

```
0:5 = opcode
6:11 = funct
12:43 = A
44:75 = B
76:107 = REFout
```

Este formato é o mesmo utilizado para o *test bench* (eles devem ser compatíveis, caso contrário não irá funcionar!). ATENÇÃO: Verilog indexa os bits na ordem inversa daquela que foi definida acima (MSB no início).

Abra o arquivo `sim/tests/testvectors.input` e adicione vetores de teste para as seguintes instruções no final (inclua manualmente os 108 zeros e uns requeridos para cada vetor de teste):

- SLT
- SLTU
- SRA
- SRL

Nós fornecemos também um *script* gerador de vetores de teste escrito em Python. Nós usamos este gerador para produzir os vetores de teste fornecidos a você. Se está curioso, você pode ler o próximo parágrafo e entender um pouco sobre o conteúdo do arquivo. Caso contrário, sinta-se livre para pular para a próxima seção.

O *script* `ALUTestGen.py` pode ser encontrado no diretório `sim/tests`. Execute-o para gerar os vetores de teste na pasta `/sim/tests`. Todos os métodos para gerar os vetores de teste estão nos dois dicionários Python `opcodes` e `functs`. As funções lambda presentes nestes dicionários (separados por vírgulas) correspondem, respectivamente: a função que a operação deve realizar, uma função para restringir a entrada `A` dentro de um intervalo específico, e uma função para restringir a entrada `B` dentro de um intervalo específico.

Se você modificar o *script* Python, execute o gerador novamente para produzir novos vetores de teste. Este procedimento irá sobrescrever o arquivo `testvectors.input`, portanto não faça isso se você já tiver escrito à mão os seus vetores de teste no arquivo!

```
% python ALUTestGen.py
```

O comando acima escreverá os vetores de teste dentro do arquivo `testvectors.input`. Use este arquivo como fonte de vetores de teste ao carregá-lo com o comando `$readmemb`.

3. Descrevendo os Módulos Verilog

Neste laboratório, nós fornecemos as interfaces dos módulos para você. Eles foram logicamente divididos em um elemento de controle (`ALUdec.v`) e um *data path* (`ALU.v`). O *data path* representa as unidades funcionais, enquanto o controle implementa a lógica necessária para alimentar o *data path*. Você será responsável por implementar ambos os módulos. Descrições sobre o que cada uma das entradas e saídas de cada módulo significa podem ser encontradas nas primeiras linhas dos respectivos arquivos fonte.

A ALU deve receber um `ALUop`, além de duas entradas `A` e `B`, e fornecer uma saída de acordo com o código representado por `ALUop`. As operações que ela deve suportar são listadas na **Especificação Funcional**. Não se preocupe com a extensão de sinal, pois ela deve ser realizada fora da ALU. O decodificador da ALU (`ALUdec`) usa os sinais `opcode` e `funct` para determinar o valor de `ALUop` que a ALU deve realizar. Neste sentido, é razoável que você considere utilizar comando `case` do Verilog para sua descrição. A sintaxe do `case` é resumida no trecho de código a seguir.

```
always@(*) begin
  case(foo)
    2'b00: // something happens here
    2'b01: // something else happens here
    2'b10, 2'b11: // you can have more than one case do the same thing
  endcase
end
```

Para tornar o seu trabalho mais fácil, este laboratório conta com dois arquivos Verilog de cabeçalho (`Opcode.vh` e `ALUop.vh`). Eles fornecem, respectivamente, macros para os `opcodes` e `functs` presentes na ISA do MIPS150, e para as operações realizadas dentro da ALU.

Você pode modificar o arquivo `ALUop.vh` no sentido de otimizar a codificação do `ALUop`, mas se modificar o arquivo `Opcode.vh` irá tornar o projeto incompatível com o esqueleto do *test bench* fornecido. Você pode utilizar essas macros colocando uma crase antes do seu nome, por exemplo:

```
case(opcode)
      `ADDIU:
```

é equivalente a:

```
case(opcode)
      6'b001001:
```

4. Usando o ModelSim

Uma vez que você tenha finalizado seus *test benches* e (preferencialmente) concluído a descrição dos módulos Verilog, você pode agora simular seu projeto. Neste laboratório, você usará o `ModelSim`, uma ferramenta de EDA largamente utilizada para simulação e depuração de projetos de sistemas digitais. A equipe de apoio da disciplina reuniu as funcionalidades necessárias para o uso eficiente do `ModelSim` em um `Makefile`.

Para simular o seu projeto, você deve primeiro compilá-lo e corrigir qualquer eventual erro de sintaxe. Para isso execute o comando a seguir:

```
% cd ~/lab4/sim
% make compile
```

O alvo `compile`, verifica a sanidade do seu código, ao mesmo tempo em que o interpreta, com o intuito de preparar o seu ambiente de simulação. Uma vez que seu projeto tenha sido compilado sem falhas, você precisa executar alguns casos de teste. O sistema de compilação necessário para a execução dos casos de teste está no interior do diretório `tests`. Um caso de teste é representado por um arquivo de extensão `.do`. Cada arquivo corresponde a um *script* em Tcl, uma linguagem largamente utilizada por diversas ferramentas de EDA. Para a maior parte do trabalho, você não deve se preocupar com os detalhes do Tcl; você utilizará estes *scripts* apenas quando enviar comandos direto para o `ModelSim`. O *script* Tcl a seguir é responsável por executar o `ALUTestbench`.

```
set MODULE ALUTestbench
start $MODULE
add wave $MODULE/*
add wave $MODULE/DUT1/*
add wave $MODULE/DUT2/*
run 100us
```

A primeira linha atribui o valor `ALUTestbench` para a variável `MODULE`. Seu valor é referenciado ao longo do resto do *script* como `$MODULE`. O comando `start` diz ao `ModelSim` qual modulo Verilog ele deve simular. O comando `add` é particularmente interessante. Por padrão, o `ModelSim` não coleta nenhuma informação de *waveform* da simulação. A string `'*'` é um atalho para “qualquer coisa”, e

portando este comando faz com que o ModelSim grave os sinais oriundos das transições produzidas a partir do *test bench*, assim como quaisquer dos sinais oriundos de DUT1 e DUT2. Uma vez que você comece a trabalhar em projetos de maior complexidade, você pode querer olhar os sinais dentro de um dado sub-módulo. Para adicionar estes sinais, simplesmente edite o arquivo .do, adicionando o novo comando ‘‘add wave <alvo>’’; por exemplo, se DUT1 e DUT2 possuírem um módulo chamado my_submodule:

```
add wave $MODULE/DUT1/my_submodule/*
add wave $MODULE/DUT2/my_submodule/*
```

Finalmente, o comando run executa a simulação. Ele recebe a quantidade de tempo como um argumento, neste caso 100us (100 microssegundos). Outras unidades (ns, ms, s) também são aceitas. A simulação será realizada ao longo deste intervalo de tempo. Na maioria dos casos, isso servirá como um *timeout*, uma vez que seu *test bench* pode sair da simulação (usando a chamada de sistema $finish() em Verilog) quando esta for finalizada.

Vamos tentar executar uma simulação. Para executar todos os casos no diretório de testes execute:

```
% make
```

Este comando irá primeiro recompilar seu projeto, se necessário, e então executar a simulação. A seguir outros comandos que podem lhe ser úteis:

- make clean: algumas vezes você pode acidentalmente cancelar uma simulação, ou em outros casos fazer com que o make acredite que seus dados de simulação estão atualizados quando, na verdade, não estão. Se está em dúvida, execute este comando antes de realizar um make.
- make results/<testcasename>.transcript: Quando você possuir múltiplos *test benches* em seu projeto e só quiser executar um deles.

Ao executar seus testes, se tudo correr como esperado, você deve visualizar a saída da simulação impressa no seu terminal. Você deverá se deparar uma das seguintes linhas na saída:

```
# FAIL: Incorrect result for opcode 000000, funct: 100011:
# A: 0xdbfa08fd, B: 0x318c32a8, DUTout: 0xaa6dd655, REFout:
```

ou

```
# ALL TESTS PASSED!
```

As saídas para todos os testes também são escritas no arquivo results/<testcasename>.transcript, caso ache necessário depurar fora do ambiente do console.

5. Visualizando Waveforms

Após completada a simulação, você pode visualizar o *waverform* para os sinais que você adicionou ao seu *script* de caso de teste. O banco de dados do *waveform* é armazenado em arquivo `.wlf` localizado dentro do diretório `results`. Para visualizá-los utilize o *script* `viewwave` incluso no diretório `sim`. Lembre-se que pode ser necessário torná-lo executável no seu ambiente Linux. Para isso, execute o comando a seguir.

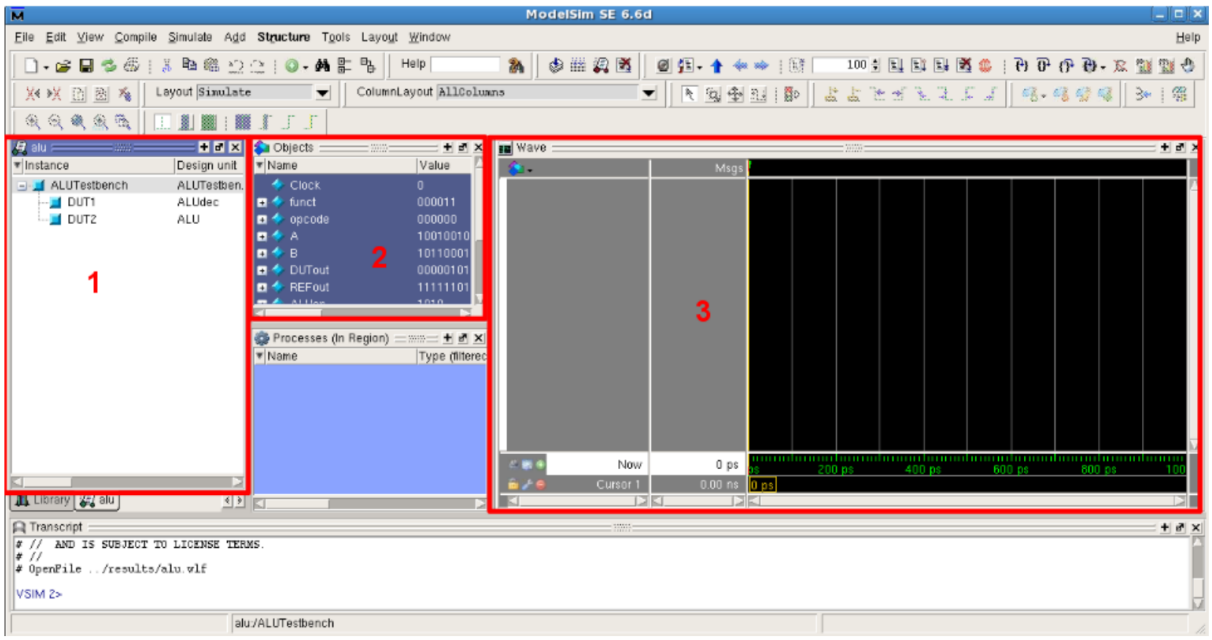
```
% chmod a+x viewwave
```

O código a seguir apresenta um exemplo de chamada ao *script* `viewwave`.

```
% ./viewwave results/alu.wlf
```

Este comando abrirá uma janela do `ModelSim` que lhe apresentará uma perspectiva hierárquica dos sinais que a sua simulação capturou.

Nota: O `ModelSim` é seu AMIGO! Ao longo de todo o curso, o `ModelSim` será a sua ferramenta principal para depuração dos seus projetos. É extremamente importante que você gaste um tempo entendendo como executar testes e usar o `ModelSim` para visualizar os resultados.



A figura acima é uma captura de tela do `ModelSim` assim que aberto. As caixas numeradas são:

1. Lista dos módulos envolvidos no *test bench*. Você pode selecionar um deles para visualizar seus sinais na janela `Objects`.
2. Janela Object - lista todos os *wires* e *registers* no seu módulo. Você pode adicionar os sinais ao *waveform* selecionando com clique direito do mouse e fazendo `Add > To Wave > Selected Signals`.

3. Visualizador de *waveform* - Os sinais que você adicionar a partir da janela de objetos são apresentados aqui. Você pode navegar através das formas de onda à procura de valores específicos, ou mesmo seguir adiante e retroceder uma transição por vez.

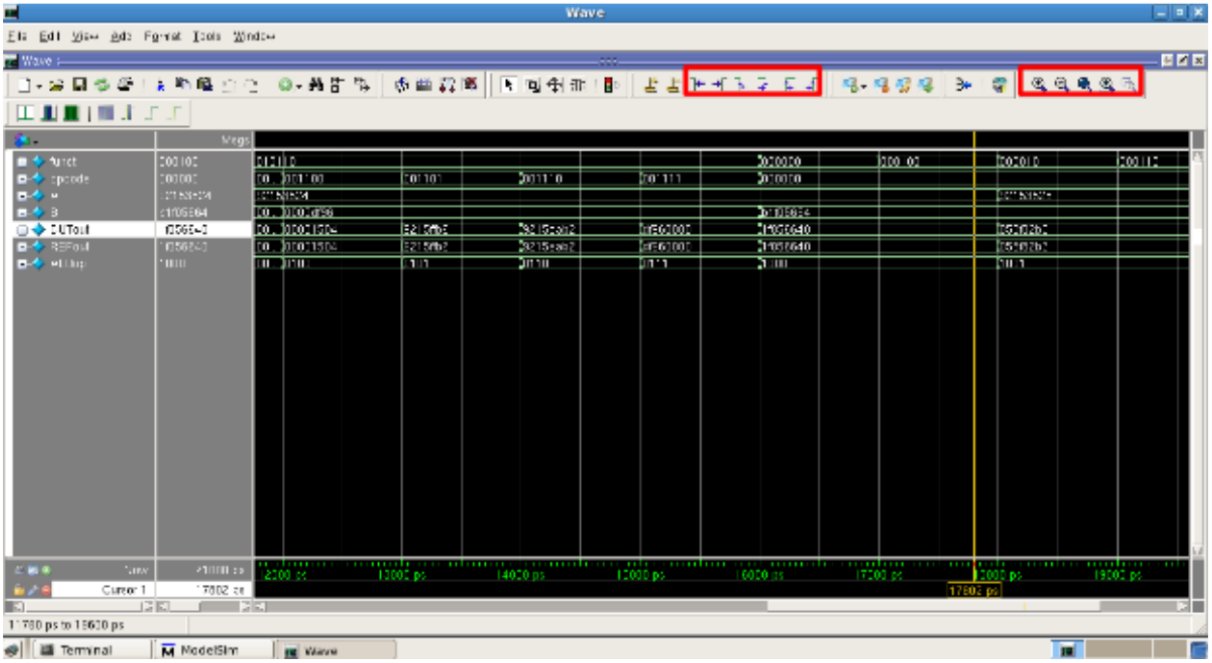
4. Janela de Comandos - Nesta região do programa, você poderá introduzir comandos Tcl ou visualizá-los ao executar uma ação através da interface gráfica do ModelSim. Veremos a seguir o quão útil é ficar de olho no que acontece nesta janela e como ela pode ser usada para automatizar nosso trabalho de depuração.

Como exemplo de uso do visualizador de *waveform*, suponha que você receba a seguinte saída quando executar o `ALUTestbench`:

```
# PASS: opcode 000000, funct 000110
#
A: 0x92153525, B: 0xb1f05664, DUTout: 0x058f82b3,
REFout: 0x058f82b3
# FAIL: Incorrect result for opcode 000000, funct: 000011:
#
A: 0x92153525, B: 0xb1f05664, DUTout: 0x058f82b3,
REFout: 0xfd8f82b3
```

Neste caso, o comando `$display()` já dirá tudo o que você precisa saber para corrigir sua falha, mas você descobrirá que isso nem sempre será suficiente para identificar uma falha. Por exemplo, se você tem uma FSM que precisa ser analisada ao longo de múltiplos intervalos de tempo, o visualizador de *waveform* apresentará os dados em um formato muito mais evidente. Se o seu projeto possuir mais de um domínio de clock, também seria praticamente impossível dizer o que está acontecendo de errado apenas com comandos `$display()`. De qualquer forma, esta prática é importante, pois você poderá adquirir prática no que diz respeito ao manuseio do `ModelSim`.

Se você acrescentar todos os sinais de `ALUTestbench` ao visualizador de *waveform*, verá uma janela semelhante à que segue:



As duas caixas em destaque contêm as ferramentas para navegação e zoom. Você pode navegar ao longo dos botões, ou consultar o manual da ferramenta, para entender o significado e o propósito de cada um deles. No Visualizador de *waveform*, você pode encontrar a localização (tempo) onde o *test bench* falhou da seguinte forma:

1. Selecionando `DUTout`
2. Clicando em `Edit > Wave Signal Search > Search for Signal Value > 0x058f82b3`

Agora você pode examinar todos os outros valores de sinais neste intervalo de tempo. Você verá que `REFout` possui o valor `0xfd8f82b3`. A partir do `opcode` e a `funct`, você sabe que esta deve ser uma instrução SRA e parece que sua ALU realizou um SRL. Talvez você tenha escrito:

```
Out = B >>> A[4:0];
```

Analisando superficialmente isso parece funcionar, mas não! Acontece que você precisa dizer ao Verilog para tratar `B` como um número com sinal para que a instrução SRA funcione como esperado. Ciente disso, você modifica a linha de código correspondente para:

```
Out = $signed(B) >>> A[4:0];
```

Após fazer esta mudança, você executa o teste novamente e cruza os dedos. Com sorte você verá a linha `# ALL TESTS PASSED!` Caso contrário, você precisará depurar seu módulo até que todos os testes do arquivo de vetor de testes e os casos de teste *hardcoded* sejam bem sucedidos.

Note que, cada operação realizada através da interface gráfica produzirá um comando Tcl correspondente na Janela de Comandos. Estas diretivas de código são importantes caso desejemos encerrar os trabalhos e para retomá-los em outro momento. Por padrão, o ModelSim carrega o *waveform* desconsiderando a análise dos sinais internos dos módulos. Para tornar este processo mais conveniente, você pode exportar os comandos gerados pela ferramenta em um arquivo `.tcl` ou `.do`. Para isso, siga até o menu `File` → `Save Transcript As...`, e escolha um nome para o seu arquivo de comandos. Para retornar ao ponto em que parou, execute o comando `viewwave` e em seguida, na Janela de Comandos introduza a seguinte linha de código:

```
VSIM> do <command_file_name>.tcl
```

Note que você pode editar o seu *script* Tcl, antes de executá-lo, no sentido de remover ou introduzir novos comandos, de acordo com a sua conveniência (ou experiência).

Agora digamos que você tenha identificado um erro na simulação e em seguida identifique uma nova falha. Note que ao executar o comando `make`, um novo arquivo `wlf`. Caso você deseje atualizar a visualização da sua base dados, você deve executar o seguinte comando Tcl na Janela de Comandos, onde `alu` corresponde ao dataset utilizado pelo ModelSim.

```
VSIM> dataset reload -f alu
```

Estes são apenas alguns exemplos de como a ferramenta pode ser utilizada para facilitar o seu trabalho. Além destas, o ModelSim possui uma série de funcionalidades que podem ser úteis em certos momentos; tantas que não é possível detalhá-las aqui. Se você precisa fazer alguma coisa, e acredita que esta funcionalidade pode já existir, Google *it*. Ou pergunte ao seu professor. Mas tente o Google primeiro. Se descobrir alguma coisa útil, compartilhe suas descobertas!

6. Acompanhamento

Parabéns! Você descreveu e testou de verdade um elemento chave dos processadores MIPS e deve agora estar familiarizado com testes de módulos Verilog. Agora responda às seguintes questões de acompanhamento:

1. No `ALUTestbench`, as entradas da ALU foram geradas randomicamente. Quando seria preferível realizar um teste exaustivo, no lugar de um teste randômico?
2. Quais falhas, se existiram, seu *test bench* ajudou a corrigir?
    - Para uma de suas falhas, descreva um pequeno programa em Assembly que teria falhado se ela não fosse identificada.

Além disso, esteja preparado para mostrar seus arquivos de *test bench* e explicar os casos *hardcoded* que você projetou.

Você também deve ser capaz de demonstrar que ambos os testes referentes aos vetores de teste gerados pelo *script* Python e seus vetores de teste *hardcoded* funcionam.