UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ELETRÔNICA E COMPUTAÇÃO
RESIDÊNCIA EM MICROELETRÔNICA

OTTO ÁLAN PINTO DE SOUSA

DELCPG033 - INTRODUÇÃO À VERIFICAÇÃO - PROJETO FINAL

SANTA MARIA
2025

SUMÁRIO

1 INTRODUÇÃO

1.1 Objetivo Geral

Neste projeto serão abordados diferentes metodologias de verificação, a fim de se destacar as diferenças de uso entre cada uma, sua eficiencia e também de promover uma análise dos resultados produzidos por cada uma.

2 METODOLOGIA

2.1 ALU

Para o desenvolvimento deste projeto, foi fornecido o código-fonte da implementação de uma Unidade Lógico Aritmética, do inglês *Arithmetic Logic Unit - ALU*, que implementa as operações de adição, subtração, multiplicação e divisão. O design fornecido possui as seguintes características técnicas:

- O módulo fornecido possui clock;
- O seu pino de *reset* é ativo em alto;
- Dado uma mudança na entrada do módulo, uma mudança da saída só ocorre no próximo ciclo de clock;
- Não há suporte para transações *back-to-back*;
- Apenas as quatro operações básicas já citadas são suportadas;
- A entrada rotulada como A deve ser sempre igual ou maior a entrada rotulada como B.

2.2 Subprojeto 1

Para esta primeira parte do projeto final foram adotadas estratégias diferentes para o desenvolvimento de *testbenchs* para verificação do design fornecido. Como primeira estratégia foi criado um *testbench* simples, que estímula todas as entradas do dispositivo sob teste e se observa a saída resultante para cada estímulo por meio de mensagens de texto ou da observação do comportamento das formas de onda resultantes.

A segunda estratégia utilizada foi a criação de um *testbench* com checagem automática, que além de adotar um funcionamento semelhante ao do *testbench* simples, adiciona passos para checar se os estímulos fornecidos estão produzindo uma saída que atenda as especificações de projeto.

Em complemento as estratégias, a terceira estratégia também realiza checagem automática, porém ao contrário da segunda estratégia, que possui estímulos e verificações limitadas ao que está definido no código-fonte desse *testbench*, esta nova estratégia faz uso do recurso de *testvectors* para realizar as checagens necessárias, facilitando assim a expansão do número de casos de teste a serem realizados sem a necessidade uma intervenção no código-fonte deste *testbench*.

Assim, serão apresentados a seguir *snapshots* dos códigos-fonte criados para atender as estratégias citadas e seus resultados de simulação. Neste projeto, foram utilizados os mesmos estímulos para as três estratégias abordadas.

2.2.1 Testbench Simples

Para este teste, foram criados estímulos de entrada que verificassem o funcionamento de todas as operações presentes na especificação do módulo ALU fornecido. Na Figura 1 é apresentado um pedaço do código-fonte que implementa este teste, é possível observar nessa figura a verifcação da operação de adição com a apresentação das saídas do dispositivo sob teste. O código-fonte utilizado está disponível em https://github.com/pintoXD/ciinovador/blob/development/verification_101/final_project/subproj1/alu_simple_tb.sv.

```
38        // Test Addition
39        A = 8'h9d;
40        B = 8'h60;
41        ALU_Sel = 4'b0000;
42        #10;
43        $display("Addition: A = %d, B = %d, ALU_Out = %d, CarryOut = %b", A, B, ALU_Out, CarryOut);
44
45        A = 8'hb1;
46        B = 8'h9f;
47        ALU_Sel = 4'b0000;
48        #10;
49        $display("Addition: A = %d, B = %d, ALU_Out = %d, CarryOut = %b", A, B, ALU_Out, CarryOut);
50
```

Figura 1 – *Snapshot Testbench* Simples

Os resultados apresentados na saída textual do emulador de terminal durante a simulação, são ilustradas na Figura 2. Nessa Figura podem ser observadas todos os estímulos de entrada fornecidos, bem como suas respectivas saídas.

```
➜  subproj1 git:(development) ✗ obj_dir/Valu_simple_tb
Addition: A = 157, B =  96, ALU_Out = 253, CarryOut = 0
Addition: A = 177, B = 159, ALU_Out =  80, CarryOut = 1
Addition: A = 151, B =  27, ALU_Out = 178, CarryOut = 0
Addition: A =  39, B =  24, ALU_Out =  63, CarryOut = 0
Addition: A = 137, B =  56, ALU_Out = 193, CarryOut = 0
Subtraction: A =  85, B =  54, ALU_Out =  31, CarryOut = 0
Subtraction: A =  56, B =  16, ALU_Out =  40, CarryOut = 0
Subtraction: A = 234, B =  60, ALU_Out = 174, CarryOut = 1
Subtraction: A = 196, B = 123, ALU_Out =  73, CarryOut = 1
Subtraction: A =  43, B =   6, ALU_Out =  37, CarryOut = 0
Multiplication: A =  14, B =  11, ALU_Out = 154, CarryOut = 0
Multiplication: A =  12, B =   3, ALU_Out =  36, CarryOut = 0
Multiplication: A =  12, B =   3, ALU_Out =  36, CarryOut = 0
Multiplication: A =  15, B =  10, ALU_Out = 150, CarryOut = 0
Multiplication: A =   0, B =   0, ALU_Out =   0, CarryOut = 0
Division: A =  36, B =  34, ALU_Out =   1, CarryOut = 0
Division: A =  29, B =   4, ALU_Out =   7, CarryOut = 0
Division: A =  19, B =  17, ALU_Out =   1, CarryOut = 0
Division: A = 244, B =  47, ALU_Out =   5, CarryOut = 1
Division: A = 183, B =  90, ALU_Out =   2, CarryOut = 1
Default: A = 183, B =  90, ALU_Out = ac, CarryOut = 1
- alu_simple_tb.sv:168: Verilog $finish
```

Figura 2 – Saída Textual *Testbench* Simples

As formas de onda resultantes da execução deste *testbench* são exibidas na Figura 3.

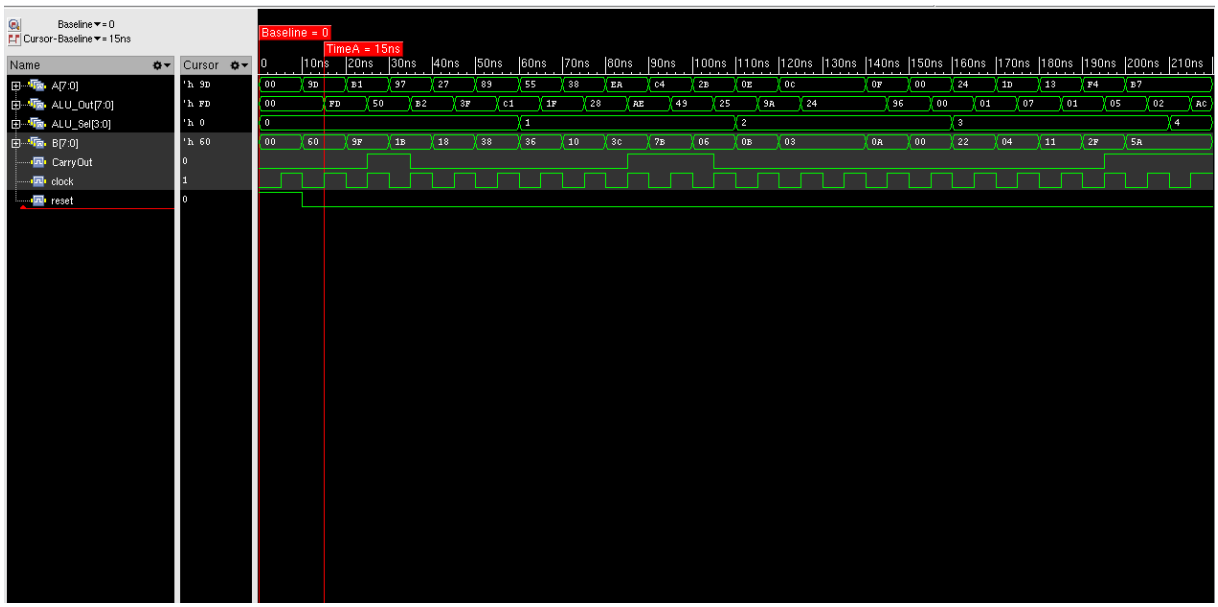

Figura 3 – *Testbench* Simples: Formas de Onda

2.2.2 Testbench Checagem Automática

Para este teste foi utilizando a função `assert` da linguagem System Verilog para realizar as checagens dos valores produzidos, assim caso esses valores não estejam de acordo com o resultado esperado, a execução do teste é interrompida no caso de teste em que ocorreu

essa incosistência. Um *snapshot* do código-fonte desse teste é ilustrado na Figura 4. O código-fonte utilizado está disponível em https://github.com/pintoXD/ciinovador/blob/development/verification_101/final_project/subproj1/alu_auto_tb.sv.

```
45        // Test Addition
46        A = 8'h9d;
47        B = 8'h60;
48        ALU_Sel = 4'b0000;
49        #10;
50        // $display("Addition: A = %d, B = %d, ALU_Out = %d, CarryOut = %b", A, B, ALU_Out, CarryOut);
51        assert(ALU_Out == 8'hfd && CarryOut == 1'b0) else $fatal(1,"Test Case 1 failed");
52
53        A = 8'hb1;
54        B = 8'h9f;
55        ALU_Sel = 4'b0000;
56        #10;
57        // $display("Addition: A = %d, B = %d, ALU_Out = %d, CarryOut = %b", A, B, ALU_Out, CarryOut);
58        assert(ALU_Out == 8'h50 && CarryOut == 1'b1) else $fatal(1,"Test Case 2 failed");
59
```

Figura 4 – *Snapshot Testbench* com Checagem Automática

A saída textual da execução deste teste é ilustrado na Figura 5.

```
Loading snapshot worklib.alu_auto_tb:sv ..................... Done
xcelium> source /home/tools/cadence/installs/XCELIUM2209/tools/xcelium/files/xmsimrc
xcelium> run
Addition: A = 151, B =  27, ALU_Out = 178, CarryOut = 0
Addition: A =  39, B =  24, ALU_Out =  63, CarryOut = 0
Addition: A = 137, B =  56, ALU_Out = 193, CarryOut = 0
Subtraction: A =  85, B =  54, ALU_Out =  31, CarryOut = 0
Subtraction: A =  56, B =  16, ALU_Out =  40, CarryOut = 0
Subtraction: A = 234, B =  60, ALU_Out = 174, CarryOut = 1
Subtraction: A = 196, B = 123, ALU_Out =  73, CarryOut = 1
Subtraction: A =  43, B =   6, ALU_Out =  37, CarryOut = 0
Multiplication: A =  14, B =  11, ALU_Out = 154, CarryOut = 0
Multiplication: A =  12, B =   3, ALU_Out =  36, CarryOut = 0
Multiplication: A =  12, B =   3, ALU_Out =  36, CarryOut = 0
Multiplication: A =  15, B =  10, ALU_Out = 150, CarryOut = 0
Multiplication: A =   0, B =   0, ALU_Out =   0, CarryOut = 0
Division: A =  36, B =  34, ALU_Out =   1, CarryOut = 0
Division: A =  29, B =   4, ALU_Out =   7, CarryOut = 0
Division: A =  19, B =  17, ALU_Out =   1, CarryOut = 0
Division: A = 244, B =  47, ALU_Out =   5, CarryOut = 1
Division: A = 183, B =  90, ALU_Out =   2, CarryOut = 1
Default: A = 183, B =  90, ALU_Out = ac, CarryOut = 1
Simulation complete via $finish(1) at time 240 NS + 0
./alu_auto_tb.sv:196        $finish;
xcelium> exit
TOOL:   xrun(64)        24.09-s001: Exiting on Feb 22, 2025 at 10:57:34 -03  (total: 00:00:11)
```

Figura 5 – Saída Textual *Testbench* Automatizado

As formas de onda resultantes da execução deste *testbench* são exibidas na Figura 6

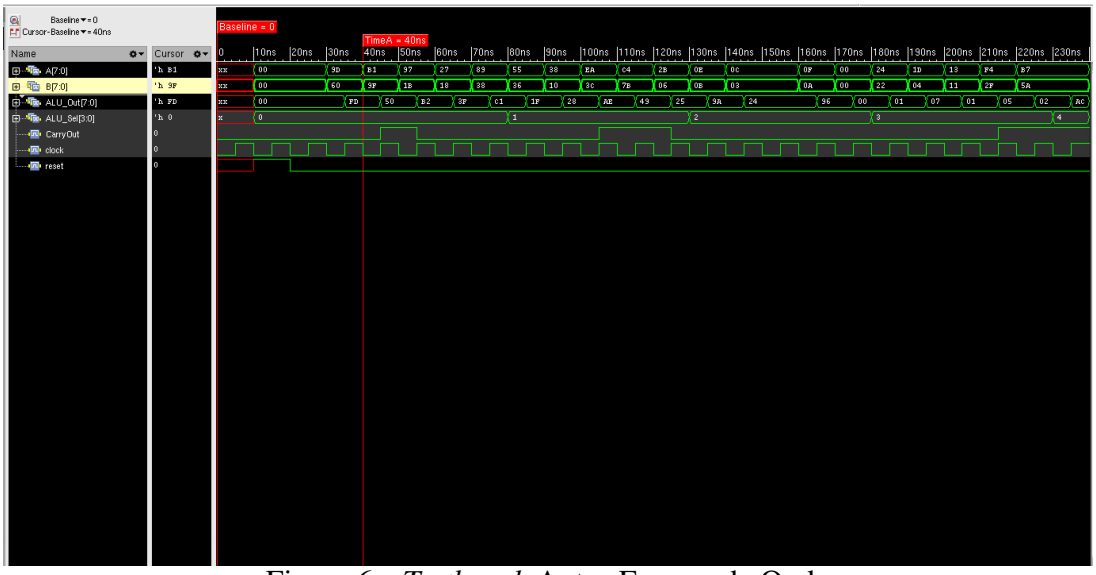

Figura 6 – *Testbench* Auto: Formas de Onda

2.2.3 Testbench Checagem Automática com testvectors

Neste teste foram utilizados *testvectors* com os estímulos que a serem fornecidos ao dispositivo sob teste, bem como as saídas esperadas para cada um desses estímulos. A Figura 7 apresenta o snapshot do arquivo de vetores de testes utilizado, que é organizado como segue:

- Coluna 1: Estímulo de entrada A;
- Coluna 2: Estímulo de entrada B;
- Coluna 3: Estímulo de entrada ALUSel;
- Coluna 4: Saída ALUOut esperada;
- Coluna 5: Saída CaryOut esperada;

Todos os valores presentes nesse arquivo de vetores de teste estão no formato hexadecimal. O código-fonte desse vetor está presente em https://github.com/pintoXD/ciinovador/blob/development/verification_101/final_project/subproj1/data2.tv.

```
1   9d 60 0 fd 0
2   b1 9f 0 50 1
3   97 1b 0 b2 0
4   27 18 0 3f 0
5   89 38 0 c1 0
6   55 36 1 1f 0
```

Figura 7 – *Snapshot* dos vetores de teste usados

Para este teste, foram criados rotinas de leitura das linhas dos vetores de teste, de forma que a cada novo ciclo de clock, uma nova linha é lida e a verificação das saídas é iniciada na borda de descida desse novo ciclo de clock. A próxima linha de vetores só é lida quando a verificação da linha atual é completamente realizada. O código-fonte utilizado está disponível em https://github.com/pintoXD/ciinovador/blob/development/verification_101/final_project/subproj1/alu_testvectors_tb.sv.

A Figura 8 apresenta as saída textual o código-fonte deste teste. Diferentemente dos testes anteriores, não há exibição dos valores que estão sendo testados no momento, apenas uma mensagem indicando o fim da execução. Contudo, caso um erro seja encontrado, o teste é interrompido e uma mensagem de erro é exibida informando qual vetor produziu o comportamento inesperado.

```
xcelium> source /home/tools/cadence/installs/XCELIUM2209/tools/xcelium/files/xmsimrc
xcelium> run
Finished testing all vectors.          0 errors found.
Simulation complete via $finish(1) at time 2021 NS + 0
./alu_testvectors_tb.sv:57             $finish;
xcelium> exit
TOOL:   xrun(64)        24.09-s001: Exiting on Feb 22, 2025 at 11:19:19 -03  (total: 00:00:10)
```

Figura 8 – Saída Textual Checagem com *testvectors*

Um recorte das formas de onda produzidas durante este teste é ilustrado na Figura 9.

Figura 9 – *Testbench* Testvectors: Formas de Onda

2.3 Subprojeto 2

Um metodologia diferente de testes foi introduzida nesta etapa. Para verificar o correto funcionamento do módulo ALU fornecido, foi fornecido também uma implementação de uma rotina de verificações utilizando a *Universal Verification Methodology*. Nessa metodologia, são construídos componentes reutilizáveis que realizam funções de geração de estímulo, monitoramento de entradas e saídas e verificação dos resultados obtidos de maneira autônoma.

A estrutura UVM utilizada nesta etapa é semelhante ilustrada na Figura 10. A descrição de cada um desses componenstes é apresentada nas Subseções a seguir. Os resultados da execução da verificação utilizando essa metodologia são brevemente apresentados na Subseção 2.3.8.

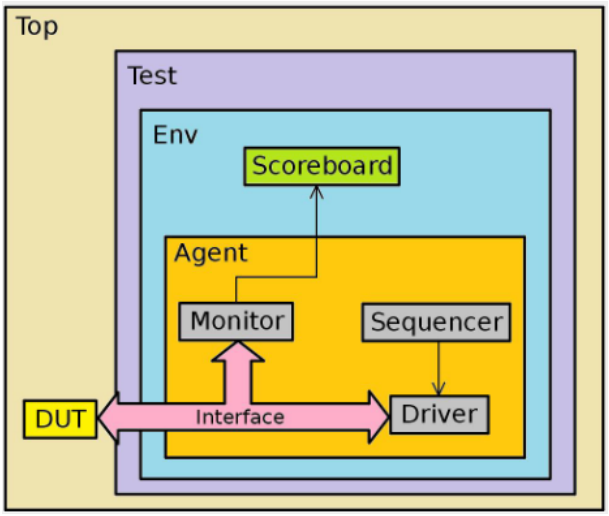

Figura 10 – Estrutura UVM: Genérica

2.3.1 Sequencer

É neste módulo em que são gerados os estímulos para a realização dos passos de verificação. Os estímulos criados neste módulo são enviados via transações UVM para o módulo *Driver*.

Neste projeto foram utilizadas três classes para compor a criação do *Sequencer* UVM. A primeira classe, `alu_sequence_item`, cria transações com conteúdo personalizado e que consiste de dois números inteiros aleatórios em que o primeiro número, chamado `a`, está no intervalo fechado de 10 a 20, enquanto o segundo número, chamado `b`, está em um intervalor fechado de 1 a 10.

A classe `alu_test_sequence` por sua vez instancia classe `alu_sequence_item` para assim criar transações UVM que são utilizadas então na classe `alu_sequencer` para enviá-las aos demais componentes interessados.

2.3.2 Driver

A partir deste módulo são enviado as sequências de estímulos geradas pelo módulo *Sequencer*, como descrito em 2.3.1, por meio de uma interface para o dispositivo sob teste, aonde serão devidamente processadas.

2.3.3 Monitor

Neste módulo são realizadas as capturas de atividades que ocorram nas entradas e saídas do dispositivo sob teste, traduzindo esses dados para objetos de transações para que possam ser enviados para outros componentes.

2.3.4 Scoreboard

Módulo que recebe os valores produzidos pelo dispositivo sob testee e verifica se esses valores foram gerados corretamente, com base nas especificações do projeto e também dos estímulos enviados ao dispostivo sob teste.

Neste projeto, este módulo realiza as quatro operações matemáticas definidas nas especificações do design forneceido e compara o valor obtido com o valor recebido da saída o dispostivo sob teste.

2.3.5 Agente

É neste módulo que é realizado a integração entre os módulos *Driver, Monitor e Sequencer*, produzindo assim uma instância única que conecta todos esses componentes internamente.

2.3.6 Environment

Realiza a integração entre os módulos *Agent e Scoreboard*, podendo conter tambéms demais componentes UVM que são necessários para configuração do aplicação. Para este projeto, somente os componentes já citados fazem parte do *Environment* criado.

2.3.7 Test

Podendo instanciar múltiplos componentes UVM, é neste módulo em que é realizado o controle dos componentes criados anteriormente para este teste. Diferentes casos de teste podem ser criados neste módulo, sem a necessidade de escrever o mesmo código para esses diferentes casos.

2.3.8 Testes Realizados

Foram realizados testes com a implementação UVM fornecida na plataforma EDA Playground e alguns resultados são apresentado a seguir. A implementação produz uma saída textual bastante extensa, porém a Figura 11 apresenta um recorte dessa saída. A Figura 12 também apresenta um recorte das formas de onda produzidas durante o processo de verificação.

```
UVM_INFO scoreboard.sv(105) @ 1153: uvm_test_top.env.scb [COMPARE] Transaction Passed! ACT=  3, EXP=  3
UVM_INFO scoreboard.sv(105) @ 1161: uvm_test_top.env.scb [COMPARE] Transaction Passed! ACT=  3, EXP=  3
UVM_INFO scoreboard.sv(105) @ 1169: uvm_test_top.env.scb [COMPARE] Transaction Passed! ACT= 24, EXP= 24
UVM_INFO scoreboard.sv(105) @ 1177: uvm_test_top.env.scb [COMPARE] Transaction Passed! ACT=  4, EXP=  4
UVM_INFO scoreboard.sv(105) @ 1185: uvm_test_top.env.scb [COMPARE] Transaction Passed! ACT=  4, EXP=  4
UVM_INFO scoreboard.sv(105) @ 1193: uvm_test_top.env.scb [COMPARE] Transaction Passed! ACT= 17, EXP= 17
UVM_INFO scoreboard.sv(105) @ 1201: uvm_test_top.env.scb [COMPARE] Transaction Passed! ACT=110, EXP=110
UVM_INFO scoreboard.sv(105) @ 1209: uvm_test_top.env.scb [COMPARE] Transaction Passed! ACT=110, EXP=110
UVM_INFO /xcelium23.09/tools/methodology/UVM/CDNS-1.2/sv/src/base/uvm_objection.svh(1271) @ 1215: report
UVM_INFO /xcelium23.09/tools/methodology/UVM/CDNS-1.2/sv/src/base/uvm_report_server.svh(847) @ 1215: rep
--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :  152
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
```

Figura 11 – UVM Saída Textual: Recorte

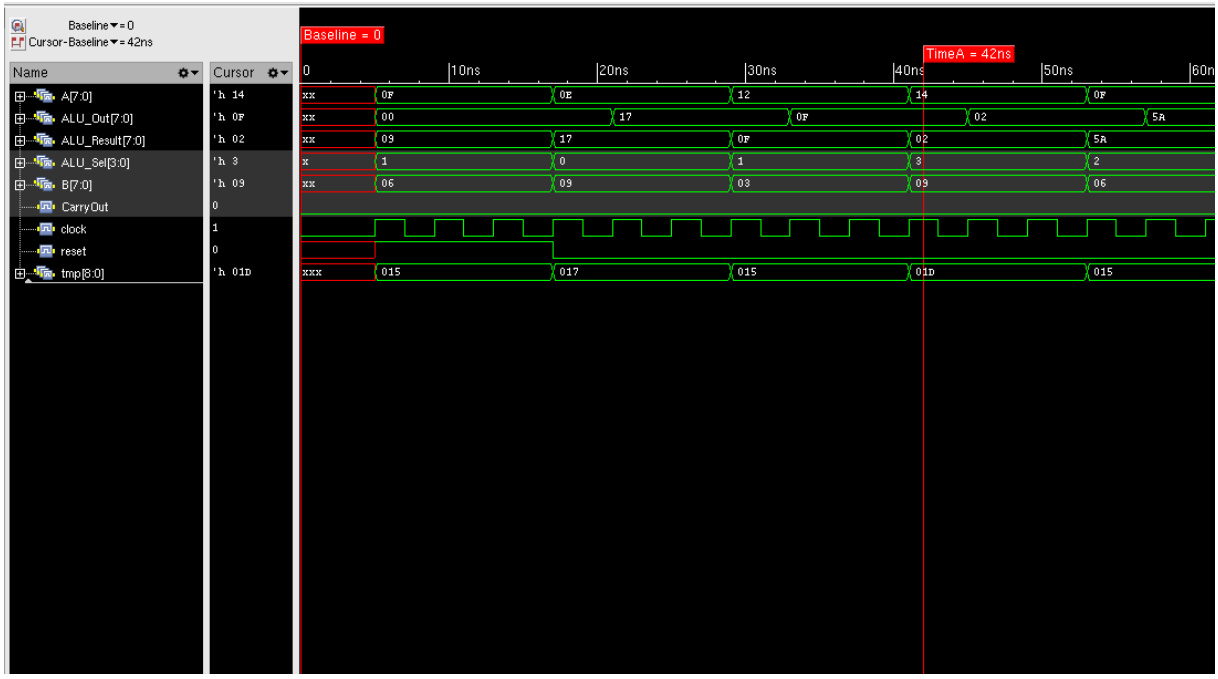

Figura 12 – Formas de Onda UVM

3 RESULTADOS

3.1 Resultado de Simuações

Durante a realização deste projeto, não foram notadas inconsistências entre as especificações definidas para o *design* da unidade lógica aritmética fornecida e os resultados obtidos por meio dos testes realizados. Alguns dos resultados dessas simulações, que atestam essas afirmações, podem ser verificados nas figuras presentes ao longo deste trabalho.

3.2 Considerações sobre as Metodologias Utilizadas

Neste projeto foram realizados verificações da um módulo ALU fornecido utilizando diferentes metodologias de verificação. As metodologias de verificação simples, com a geração de estímulos e verificação manual das respostas, e verificação automatizada com e sem o uso de vetores de teste, até certo ponto, conseguiram verificar a funcionalidade do design fornecido.

Contudo, ao usar essas técnicas, houve uma extensa repetição de código entre os testes simples e os testes automatizados. Além disso a grande quantidade de casos de teste resultou em *scripts* muito extensos, dificultando o entendimento, manutenção e até mesmo expansão dos casos de teste já existente.

Algumas dessas dificuldades puderam ser contornadas a adição de vetores de teste, em que estímulos e saídas esperadas são fornecidas em um único local, facilitando a expansão de casos de teste. Porém, nesse tipo de abordagem, quaisquer modificações no formato do vetor de teste, demanda uma atualização do script de verificação que faz uso desses vetores de teste.

A abordagem de verificação que faz uso da metodologia UVM, se mostra mais adequada quando do reuso de código, dispensando a necessidadde de repetição de código. Contudo, essa metodologia possui uma complexidade de desenvolvimento, adequação e execução maior do que as abordagens anteriores, sendo necessário ponderar seu uso.

Por fim, a escolha da metodologia de verificação a ser utilizada depende do nível de complexidade exigido para avaliar o dispositivo sob teste. Para verificações simples, geralmente utilizadas em fases inciais de desenvolvimento, as verificações por testes simples podem ser adequadadas, contudo, a medida que há um aumento da complexidade de projeto, as demais metodologias ganham mais ênfase.

4 CONCLUSÃO

Neste projeto foram utilizados diferentes metodologias de verificação para verificar as funcionalidades de um módulo ALU fornecido. Apesar de diferentes, todas demonstraram o correto funcionamento desse módulo. Porém, o nível de complexidade de implementação de cada uma, evidenciou seus pontos fortes e fracos.

Apesar de serem de mais difícil manutenção e expansão, o teste simples pode verificar as funcionalidades do módulo ALU sem maiores dificuldades. Da mesma forma que o teste automtizado também o pode, porém transferindo a responsabilidade de atestar a corretude da saída do verificador para o código em si.

Em adição a esses testes, os testes com vetores de teste, possibilitam uma expansão dos casos de testes mais facilmente, porém podem dificultar a manutenção desses testes uma vez que a estrutura dos vetores é fortemente ligada a estrutura do teste.

A metodologia de verificação UVM, por sua vez, se mostrou mais eficiente no tocante a resudo de código, facilidade de expansão e manutenção. Apesar de uma maior complexidade de implementação, há um controle maior sobre os estímulos criados, bem como sobre os casos testes, permitindo a realização de um processo de verificação mais eficiente.