PCS3422 – Organização e Arquitetura de Computadores II

# Execução Fora de Ordem com Algoritmo de Scoreboard


| | |
|---|---|
| Gabriel H. Ribeiro Justino | 9837733 |
| Gabriela von Staa Guedes | 9853248 |
| Luana Vicente Leite | 9281074 |
| Michel Chieregato | 9042790 |


São Paulo

11/2019

# Sumário

# Introdução

Deseja-se desenvolver um algoritmo de scoreboard dentro de um processador Arm_v8. Este algoritmo ajuda a planejar dinamicamente o *pipeline*, permitindo que as instruções possam ser executadas fora de ordem quando não existem conflitos. Todas as dependências de instruções são guardadas e a instrução só segue o curso no *pipeline* se não existe nenhum conflito. Caso haja, o scoreboard monitora o fluxo de execução até que todas as dependências sejam resolvidas e somente depois a instrução é emitida para o pipeline.

Este projeto tinha intenção de ser feito em duas partes principais: o desenvolvimento do scoreboard em um emulador de processador Arm_v8 em C e, posteriormente, a tradução do algoritmo de C para VHDL. Como é explicado no decorrer deste relatório, realizar a tradução parecia menos promissor do que desenvolver o scoreboard do início em VHDL, portanto a estratégia do projeto foi alterada.

Este relatório apresenta, portanto, o nosso processo de desenvolvimento em C, em VHDL e nossa conclusão, que contém uma relação entre os sucessos e as partes incompletas do projeto.

# Desenvolvimento em C

## Decisões primárias

A fim de entender melhor o funcionamento do scoreboard, foi desenvolvido o algoritmo primeiramente em C, a partir de um projeto de processador Arm_v8 previamente desenvolvido por *'omufeed*' e *'mdixon'*, que pode ser encontrado neste repositório do Git.

Sob a orientação do professor, foram definidas duas unidades funcionais, uma que cuida das operações de *Load* e *Store* e uma que irá cuidar das operações de *Add* e *Sub*. Para poder estimar a diferença de tempo que o algoritmo traz para o processo, foi adicionado um *clock* e tempos de execuções arbitrários para as operações. A unidade de *Load* possui um tempo de total de 4 ciclos e a de adição, de 7 ciclos.

Para poder armazenar as informações relevantes, foram adicionados 3 registradores, sendo 2 para as unidades funcionais e 1 para o registrador de *write result*.

Para poder simular o paralelismo do processador, a cada ciclo de *clock* são realizadas 4 tarefas principais:

- Verifica se o *issue* da instrução pode ser realizado dependendo se a unidade funcional em questão está *busy* ou não
- Verifica se a unidade funcional de *load* está *busy*, se sim incrementa o contador da unidade, ao chegar em 4 ciclos realiza a operação (isto pois em software a operação não está limitada por ciclos)
- Realiza o mesmo para a unidade de *add*
- Verifica se instruções acabaram e se o registrador de *Write Back*, assim como os registradores das unidades, deve ser atualizado.

## Pseudocódigo

Inicialmente, foi desenvolvido um pseudocódigo, apresentado abaixo, para poder estruturar o que iria ser feito e em que ordem, dado que o código em si é muito extenso. Este código foi utilizado como base tanto para o desenvolvimento do algoritmo em C quando como base para a lógica do scoreboard desenvolvido em VHDL.

```
contador clock
global _UF1_WB
global _UF2_WB
while(!finish){
    clock++
    if(issueOrStall(instru)){pcCounter ++;} // caso stall nao incrementa
    checkIfSomethingToDoUf1();
    // if acabou de sair de WB n pode fazer issue nesse ciclo
    checkIfSomethingToDoUf2();
```

```
    // if acabou de sair de WB n pode fazer issue nesse ciclo
    CheckWR();
     // verifica se acabou alguma instrucao e atualiza os estados dos reg das
UFs e do WR
}
checkIfSomethingToDoUf(){
    if(!busyBit){return;}
    if((Fj && Rj == Yes) && (Fk && Rk == Yes)){
        // se tiver algo no Fj verifica se ta pronto
        // se tiver algo Fk verifica se ta pronto
        executarInstrucao(); or WB();
    }
    else return ; // nada pra fazer na UF continua
}
issueOrStall(instru){
    if(!decodeInstruction(instru)){
        return false; // se a UF em questao ta busy retorna false e stall
    }
    if( !verifcaWR){
        return false; caso esteja no w result da stall
    }
    if(_UF_WB == true){
        cannot issue
        _UF_WB = false
        return false;
    }
    issue(instru); // se UF nao ta busy faz o issue
}
issue(instru){
    // por os valores no reg de acordo com o espaco q precisava
    reg[16] ou reg[17]
    reg[18] // adiciona no write result o reg q vai escrever
    //como trata WAW nesse tipo de situacao??? pode fazer issue caso já tenha
coisa algo??
}
executarInstrucao(){
    contar o numero de ciclos para dar como terminada operacao
    como nao tem clock de vdd
}
WB(){
    // salva no reg
    if(ciclo WB){setar _UF_WB = true}
```

```
    tira do WR
}
verificaWR(){
    if(reg_write tem algo no write result){return false}
}
```

Figura 1: Pseudocódigo do Scoreboard

## Desenvolvimento das instruções

Em seguida, foram separadas as instruções que seriam desenvolvidas para o nosso caso, pois o código em si trata de todos os tipos de instruções existentes, incluindo instruções de multiplicação e divisão, que não serão tratadas neste projeto. Assim, as únicas instruções que foram convertidas para testes foram as instruções de *Load*, *Store*, *Add* e *Sub*. As outras instruções são utilizáveis, porém não estão respeitando os ciclos de *clocks* determinados previamente.

Em seguida, é descrito em mais detalhes a ordem das instruções e como foram implementadas:

1. A cada ciclo incrementa-se o *clock* do sistema
2. A instrução que o PC está apontando é decodificada e verifica se a Unidade Funcional que ela necessita está livre ou não:
    a. Caso esteja livre: Realiza o *issue* e incrementa o PC
    b. Caso não esteja livre: Continua, não incrementando o PC
3. Verifica se a UF1 está *busy*:
    a. Se está *busy*: chama a função correspondente (*Add* ou *Sub*) e incrementa o número de *clocks* interno. Caso tenha atingido o total de 7 ciclos, realiza a operação e o WB, e retira do WR o registrador usado, além de zerar o valor da UF1
    b. Se não está *busy*: continua sem realizar nada
4. Verifica se a UF2 está *busy*:
    a. Se está *busy*: chama a função correspondente (*Load* ou *Store*) e incrementa o número de *clocks* interno. Caso tenha atingido o total de 4 ciclos, realiza a operação e o WB, e retira do WR o registrador usado, além de zerar o valor da UF2
    b. Se não está *busy*: continua sem realizar nada
5. Verifica se ocorreu *Write Back* e se houve mudança no registrador de *Write Result*:
    a. Caso sim: atualiza nas UFs os estados dos registradores Rj e Rk que estão em espera de resultado
    b. Caso não: continua sem realizar nada

## Código Teste sem Scoreboard

O objetivo da implementação do algoritmo de scoreboard é a melhoria na eficiência do processador. Dado este fato, devemos demonstrar portanto que um mesmo código de teste, ao ser rodado em um processador simples, sem scoreboard, terá seu tempo total de processamento maior

que um processador com scoreboard. Para isso, criamos o código a seguir, que foi rodado nos dois processadores.

- LD r2,r3,#4 => 11111000010 000000100 00 00011 00010
- ADD r2,r0,r1 => 10001011000 00001 000000 00000 00010
- LD r4,r3,#4 => 11111000010 000000100 00 00011 00100
- ADD r2,r0,r1 => 10001011000 00001 000000 00000 00010
- LD r5,r3,#4 => 11111000010 000000100 00 00011 00101
- ADD r0,r5,r2 => 10001011000 00010 000000 00101 00000
- SUB r4,r5,r2 => 11001011000 00010 000000 00101 00100

Os valores em binário foram baseados no arquivo Green Card, disponibilizado no Moodle da Disciplina. Também reproduzimos aqui, na figura a seguir, os formatos das instruções R e D, dado que são as únicas implementadas neste projeto. As instruções *Add* e *Sub* são do tipo R, enquanto *Load* e *Store* são do tipo D.

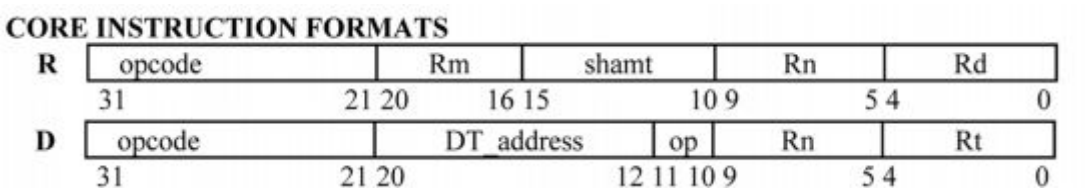


Figura 2: Formatos das instruções do tipo R e D

Como este código de teste faz cálculos com números da memória e dos registradores, precisamos inserir alguns valores diferentes de zero. As imagens abaixo demonstram quais valores foram inseridos nestas variáveis.

```
registers[0]=0x3;
registers[1]=0x7;
registers[2]=0x1;
registers[3]=0x82;
registers[4]=0x9;
registers[5]=0x2;
```

Figura 3: Valores iniciais dos registradores

```
memory[130]=0x8;
memory[134]=0x4;
memory[138]=0x5;
```

Figura 4: Valores iniciais da memória

Ao rodar esse código no processador sem scoreboard, como tem 3 instruções *Load* e 4 instruções *Add/Sub*, a duração esperada é de:

$$4 \times 7 + 3 \times 4 = 28 + 12 = 40\ ciclos$$

Dados os valores iniciais, espera-se como estado final dos registradores, após a realização destas instruções:

```
Registers
-R0: 15 -R1: 7 -R2: 10 -R3: 142 -R4: 5 -R5: 5 -R6: 0 -R7: 0 -R8: 0 -R9: 0 -R10: 0 -R11: 0 -R12: 0 -R13: 100 -R14: 0 -R15
: 63
---------- 40 Cycles, end of instruction 6----------
```

Figura 5: Valores finais dos registradores após rodar o código sem Scoreboard e número de ciclos de duração

Como pode-se observar na Figura 4, o número de ciclos de duração do processamento foi igual ao calculado teoricamente.

## Código de Teste com Scoreboard

Para calcular quantos ciclos o processador com scoreboard demoraria para realizar esse código, foi desenvolvida a tabela de *issues* que simulam os ciclos em que cada instrução será realizada.

Tabela 1: Status das instruções do Scoreboard

| Instruções | Issue | Read Operators | Execution | Write Back |
|---|---|---|---|---|
| **LD r2,r3,#4** | 1 | 2 | 3 | 4 |
| **ADD r2,r0,r1** | 5 | 6 | 10 | 11 |
| **LD r4,r3,#4** | 6 | 7 | 8 | 9 |
| **ADD r2,r0,r1** | 12 | 13 | 17 | 18 |

| | | | | |
|---|---|---|---|---|
| **LD r5,r3,#4** | 13 | 14 | 15 | 16 |
| **ADD r0,r5,r2** | 19 | 20 | 24 | 25 |
| **SUB r4,r5,r2** | 26 | 27 | 31 | 32 |

Portanto, espera-se que o processamento desse código em um processador com scoreboard seja finalizado com 32 ciclos de *clock*.

Ao rodar o código implementado, espera-se encontrar o mesmo resultado da Figura 4, mas em 32 ciclos, como era esperado pelo cálculo teórico.

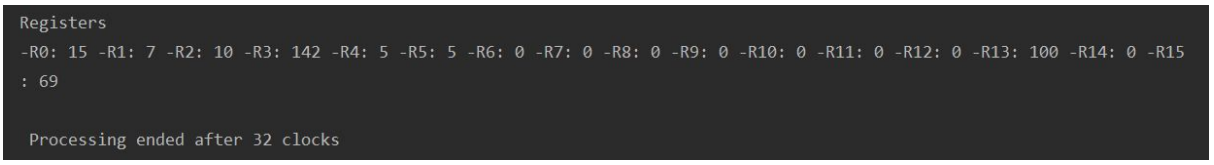


Figura 6: Valores finais dos registradores após rodar o código com scoreboard

## Melhoria na Performance

Dado que o processador sem scoreboard processou o código em 40 ciclos e o com Scoreboard o processou em 32 ciclos, ou uma melhoria neste código de:

$$\frac{40-32}{40} = 20\%$$

## Dependências de Escrita

Como pode ser percebido na tabela dos *issues*, existe uma dependência WAW entre as duas primeiras instruções. Assim, não é realizado o *issue* da segunda instrução, já que ela não pode ser adicionada ao registrador *WriteResult*. Assim, mesmo que a unidade funcional de *Add/Sub* esteja livre, espera-se o final da primeira instrução antes de realizar o *issue*.

# Desenvolvimento em VHDL

## Tradução do Scoreboard em C

Após o término da elaboração do código em C, como foi recomendado pelo professor, demos início ao processo de tradução do algoritmo para VHDL. Porém, com o decorrer da tradução, nos deparamos com imprevistos que dificultaram a elaboração de um projeto funcional.

A primeira dificuldade encontrada foi retirar do algoritmo em C somente os fragmentos significativos para o scoreboard, visto que nesta implementação não existe uma separação dos componentes do processador em arquivos diferentes. A existência de funções também pode ser considerada uma adversidade para a criação do VHDL, o uso repetido delas não é simples de ser repassado ao nosso projeto. Por fim, como se trata de uma linguagem de simulação de hardware, não é possível fazer uso do recurso existente em C que possibilita criar uma variável com mesmo nome para diferentes ocasiões.

Tendo os pontos ressaltados em vista, optamos por tentar elaborar um projeto em VHDL do início, com base no pseudocódigo apresentado previamente neste relatório, ao invés de tentar apenas traduzir. Permanecemos nesta alternativa por alguns dias e, por fim, comparamos a evolução do projeto de tradução com o de desenvolvimento direto em VHDL, para definir com qual seguiríamos. Escolhemos o de desenvolvimento direto, tanto pelas justificativas apresentadas anteriormente quanto pela maior facilidade em encontrar falhas (algo mais complicado de se fazer quando está traduzindo um código não planejado para aquela linguagem).

## Variáveis Básicas

Com base na mesma lógica que implementamos em C, precisamos de três variáveis principais, que representam as tabelas do processo de scoreboard, ou seja, duas para as UFs e uma para o *Write Result*.

```
type UF is array (9 downto 0) of std_logic_vector(4 downto 0);
signal UF_Load, UF_Add : UF;

type resultStatus is array (31 downto 0) of std_logic_vector(1 downto 0);
signal regResult : resultStatus;
```

Figura 7: Variáveis principais do Scoreboard

Abaixo, fizemos uma tabela com os valores possíveis de cada campo da variável que representa as UFs e da variável do *Write Result*, para servir como referência na leitura do código, dado que os valores não são totalmente autoexplicativos. Como o tamanho de cada campo deve ser o mesmo para todos, tivemos que colocar o maior tamanho utilizado, que no caso é 5 bits, que representa os registradores (que são gravados em Fi, Fj e Fk). Os campos que não necessitavam de 5 bits os utilizaram do mesmo modo, normalmente com extremos do tipo 00000 e 11111.

Tabela 2: Valores possíveis de cada campo das variáveis das UFs

| Campo na Variável | Campo das Tabelas | Valores Possíveis | Explicação do Valor |
|---|---|---|---|
| UF(0) | Busy | 00000 | Não |
| | | 11111 | Sim |
| UF(1) | Op | UF_Add: 00001 | Add |
| | | UF_Add: 00010 | Sub |
| | | UF_Load: 00001 | Load |
| | | UF_Load: 00010 | Store |
| UF(2) | Fi: Reg. de Destino | Bits 4-0 da instrução | Reg. de Destino |
| UF(3) | Fj: Reg. de Origem 1 | 00000 | Para Load/Store (só tem 1 reg. de origem) |
| | | Bits 9-5 da instrução | Para Add/Sub |
| UF(4) | Fk: Reg. de Origem 2 | Bits 20-16 da instrução | Para Load/Store |
| | | Bits 9-5 da instrução | Para Add/Sub |
| UF(5) | Qj: UF produzindo Fj | 00011 | UF Load/Store |
| | | 00010 | UF Add/Sub |
| UF(6) | Qk: UF produzindo Fk | 00011 | UF Load/Store |
| | | 00010 | UF Add/Sub |
| UF(7) | Rj: Flag de Fj pronto | 11111 | Não |
| | | 00000 | Sim |
| UF(8) | Rk: Flag de Fk pronto | 11111 | Não |
| | | 00000 | Sim |
| UF(9) | Estágio da instrução dessa UF | 00000 | Finalizou WR |
| | | 00010 | Read Operators |
| | | 00011 | Execution |
| | | 00100 | Write Result |

Tabela 3: Valores possíveis dos campos de regResult

| Campo | Valores Possíveis | Explicação do Valor |
|---|---|---|
| regResult(x) | 00 | Vazio |
| | 01 | UF Add |
| | 10 | UF Load |

Além das variáveis para guardar as informações do scoreboard, criamos sinais de saída da Unidade de Controle que devem ser passados para o Fluxo de Dados e então entrar nos registradores que ficam entre os estágios, para agirem como um *enable*. Assim, o scoreboard iria definir quando os estágios do *pipeline* deveriam funcionar e passar os resultados para o próximo registrador de estágio.

## Estágio 1: Issue

Seguindo a referência (1)[1], temos uma orientação do que fazer em cada estágio. No de issue, temos:

- Condições de espera:
  - UF deve estar livre
  - Nenhuma outra instrução escreve no mesmo registrador de destino (para evitar WAW)
- Ações:
  - Faz issue da instrução, inserindo valores na UF
  - Atualiza valores do Scoreboard

Assim, essa parte do código é envolta por um IF que confere se o sinal *issueInstruc* é 1, ou seja, se foi definido que deve fazer a issue da próxima instrução, e também confere qual é o op da instrução atual (só é válido para Add, Sub, Load e Store).

Dentro deste IF, há outro IF que confere o campo UF(0), para definir se a UF em questão está ou não ocupada. Se não estiver, há outro IF que confere se o registrador de destino já está sendo calculado por outra UF, ao conferir a variável regResult. Por fim, caso o regResult não esteja preenchido, faz-se a issue: define-se *issueInstruc* como 1, para poder tentar fazer a *issue* da próxima instrução, além de preencher os valores da UF e do regResult. Caso uma das conferências não seja correta, o sinal *issueInstruc* é definido como 0, para que o pipeline não permita a leitura de uma nova instrução. Assim, a instrução do Scoreboard permanece a mesma até que seja feita a *issue* dela, como manda o algoritmo.

## Estágio 2: Read Operators

Este estágio é mais simples, como visto em (1).

- Condições de espera: todos os operandos de origem estão disponíveis

[1] Tradução livre do texto lido nesta referência.

- Ações: lê os operandos e começa execução no próximo ciclo

Deste modo, é conferido apenas se o estágio da UF (UF(9)) é igual a 00010, ou seja, está no estágio 2, e se as flags dos registradores de origem (UF(7) e UF(8)) estão em 00000, que significa que estão prontos.

Caso isso seja verdadeiro, o estágio é passado para 3 (00011) e o sinal readOper, que sai da UC, é definido como 1. Além disso, também são passados os readReg1 e readReg2 para fora da UC, pois são os endereços de registradores a serem lidos no estágio 2 do pipeline.

## Estágio 3: Execution

Este estágio tem apenas as ações de:

- A UF começa execução
- Quando resultado está pronto, o Scoreboard é notificado

Para isso, criamos 4 novos sinais para a UC: duas entradas (resultReady1 e resultReady2) e duas saídas (execute1 e execute2). Há dois sinais de cada pois há duas UFs, que teoricamente são relacionadas a duas ULAs no fluxo de dados, então devem ser controladas separadamente.

A única conferência para que os sinais de execução sejam ativados é de que a UF está no estágio 3 (00011).

## Estágio 4: Write Result

Apesar da simplicidade das tarefas, vistas a seguir, a escrita do código foi mais complicada.

- Condição de espera: nenhuma outra UF vai ler o registrador de destino da instrução
- Ação: escreve no registrador de destino e atualiza o Scoreboard

Caso o estágio da UF seja 4 (00100) e o resultReady (1 ou 2, dependendo da UF tratada) seja 1, compara-se os registradores de origem das UFs e as suas respectivas flags. Caso a flag esteja como Sim (ou seja, o valor de origem já está pronto) e o registrador de origem é igual ao registrador de destino da UF que atualmente está no estágio de WR, não é escrito ainda o valor calculado no registrador de destino.

Caso isso não ocorra e algum registrador de origem é igual ao registrador de destino atual, a sua flag é Não e o regResult correspondente é a UF atual, o valor calculado é escrito e a flag é atualizada para Sim.

## Testbench e Waveform

Como é explicado ainda neste relatório, o projeto em VHDL não foi completamente finalizado. Por isso, nosso *testbench* foi simplificado, com a intenção de apenas testar as variáveis das UFs, do registrador de status do Write Result e os sinais de saída. Ou seja, fizemos um testbench específico para a UC com scoreboard, sem considerar o processador completo.

No arquivo do *testbench*, definimos as instruções sequencialmente de modo simples, sendo que elas só são passadas caso o *issueInstruc* (*issueOUT*, pois é a saída - *issueInstruc* é o sinal interno da UC) esteja ativado. Além disso, colocamos que os sinais de resultReady (1 e 2) são sempre 1, para facilitar.

```
process(issueOUT,clk)
begin
	if(reset = '1') then
		ordemInstruc <= 0;
	end if;
	if(falling_edge(clk) and issueOUT = '1') then
		if(ordemInstruc = 0) then
			instruc <= "11111000010" & "000000100" & "00" & "00011" &
"00010"; --LDUR R2,R3,#4
			ordemInstruc <= ordemInstruc + 1;
		elsif(ordemInstruc = 1) then

			instruc <= "10001011000" & "00001" & "000000" & "00000" &
"00010"; --ADD R2,R0,R1
			ordemInstruc <= ordemInstruc + 1;
		elsif(ordemInstruc = 2) then
			instruc <= "11111000010" & "000000100" & "00" & "00011" &
"00100"; --LDUR R4,R3,#4
			ordemInstruc <= ordemInstruc + 1;
		elsif(ordemInstruc = 3) then
			instruc <= "10001011000" & "00001" & "000000" & "00000" &
"00010"; --ADD R2,R0,R1
			ordemInstruc <= ordemInstruc + 1;
		elsif(ordemInstruc = 4) then
			instruc <= "11111000010" & "000000100" & "00" & "00011" &
"00101"; --LDUR R5,R3,#4
			ordemInstruc <= ordemInstruc + 1;
		elsif(ordemInstruc = 5) then
			instruc <= "10001011000" & "00010" & "000000" & "00101" &
"00000"; --ADD R0,R5,R2
			ordemInstruc <= ordemInstruc + 1;
		elsif(ordemInstruc = 6) then
			instruc <= "11001011000" & "00010" & "000000" & "00101" &
"00100"; --SUB R4,R5,R2
			ordemInstruc <= ordemInstruc + 1;
		end if;
	end if;
end process;
```

Figura 8: Lógica para passar os valores das instruções no testbench do scoreboard

Com as instruções definidas, rodamos o *testbench* e obtemos o *waveform* mostrado abaixo. Para facilitar a leitura, editamos o *waveform* para destacar os números de ciclos e os valores de cada sinal em cada ciclo, além de separar em várias imagens.

Alguns valores mudam na borda de descida do *clock*, para que seja possível pegar os valores corretos na hora de considerar o *issue*, que é na borda de subida. Por isso, os valores apresentados nas imagens são considerados depois da borda de subida.


Figura 9: Valores da UF Add para os 9 primeiros ciclos

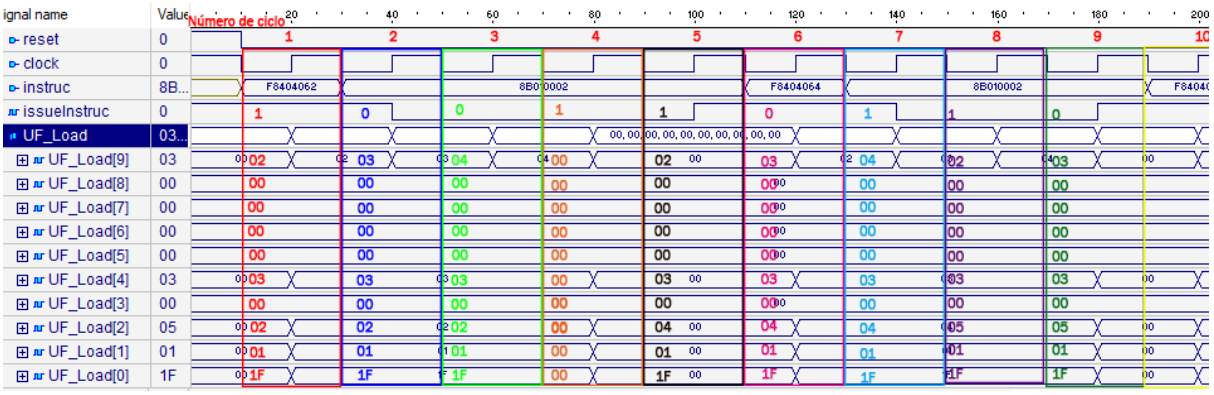


Figura 10: Valores da UF Load para os 9 primeiros ciclos

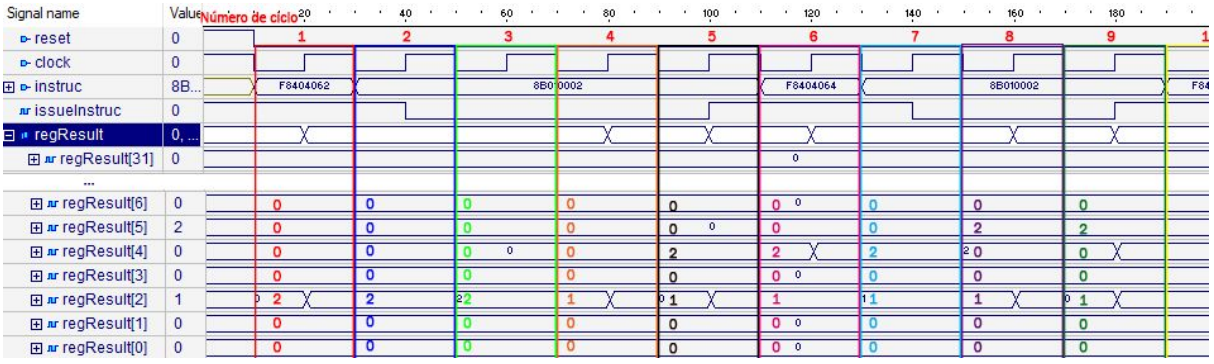


Figura 11: Valores relevantes do regResult para os 9 primeiros ciclos

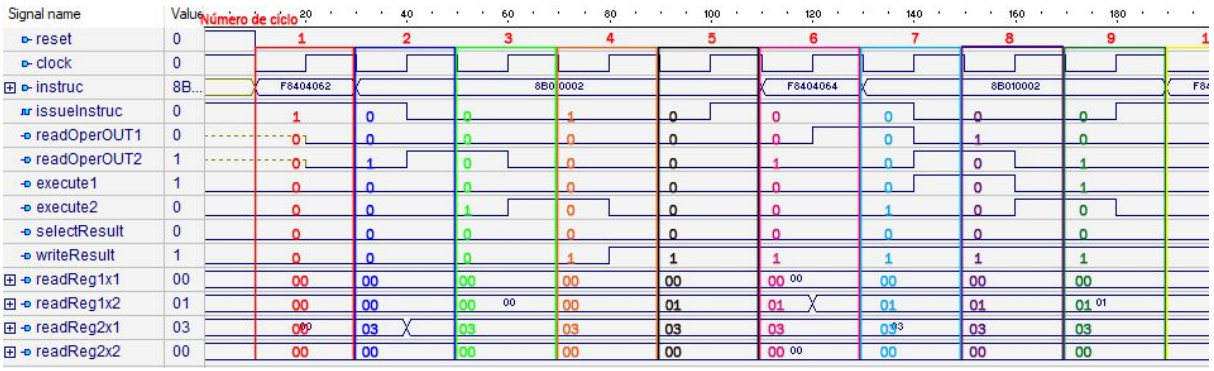


Figura 12: Valores dos sinais do scoreboard nos 9 primeiros ciclos

Levando em consideração as imagens apresentadas, podemos fazer uma interpretação da corretude dos valores a partir da lógica do algoritmo.

## Ciclo 1

Abaixo, fazemos uma explicação sinal a sinal do ciclo 1 - os outros ciclos tem lógica análoga, então nos limitamos a um ciclo.

- Ação: Issue da instrução 1 - LD R2,R3,#4
- UF_Add: sem alteração
- UF_Load:
  - Fica ocupada: (0) recebe 1F
  - Instrução é um Load: (1) recebe 01
  - Registrador de destino é R2: (2) recebe 02
  - Como é Load, só tem um registrador de origem: (3) recebe 00
  - Registrador de origem é R3: (4) recebe 03
  - Nenhuma UF está produzindo R3: (5) e (6) recebem 00
  - As origens já estão prontas: (7) e (8) recebem 00
  - Está pronto para ir para ir para o próximo estágio: (9) recebe 02
- regResult:
  - Registrador de destino é R2: (2) recebe 02, porque é a UF_Load que está calculando
- Sinais:
  - issueInstruc: fica em 1, pois fez issue da instrução atual, então pode ler a próxima
  - readOper1: é da UF_Add, então não se altera
  - readOper2: recebe 1, pois todas as origens da instrução em UF_Load estão prontas
  - execute1: é da UF_Add, então não se altera
  - execute2: fica em 0, pois o próximo estágio é de leitura de operandos
  - selectResult: fica em 0, pois o próximo estágio é de leitura de operandos
  - readReg1x1: é da UF_Add, então não se altera
  - readReg1x2: é da UF_Add, então não se altera
  - readReg2x1: recebe 03, que é o registrador de origem da instrução
  - readReg2x2: recebe 00, pois Load só tem um registrador de origem

### Demais ciclos

Como podemos ver nas imagens 9 e 10, é feito o *issue* da segunda instrução no ciclo 5, como esperado. A issue da instrução 3 também é feita no ciclo 6, de acordo com a tabela 1. Desse ciclo em diante, há uma ligeira discrepância, dado que na tabela 1 levou-se em consideração que as instruções Add e Sub levavam 7 ciclos para serem completamente executadas, algo que não foi levado em conta neste *testbench*.

Depois do ciclo 14, é identificado que a UF_Add, com a instrução ADD R0,R5,R2, fica travada no estágio de leitura de operandos, com a flag (7) em Não - ou seja, (3) (que é R5) é dito que não está pronto ainda. O que não está correto, pois no ciclo anterior a instrução LDUR R5,R3,#4 fez o WR e o regResult(5) foi atualizado para 0. Assim, conclui-se que há alguma falha na lógica de issue. Porém, não tivemos tempo para descobrir com precisão o erro e corrigi-lo.

## Partes Incompletas

Mesmo com a elaboração do código, algumas partes do projeto permaneceram incompletas. Apesar de todos os esforços dos integrantes do grupo, não conseguimos produzir um algoritmo que atendesse a todos os requisitos, a exemplo da, como já foi mencionado anteriormente, paralisação da UF_ADD a partir do décimo quarto ciclo.

Além disso, não fizemos a integração da nossa Unidade de Controle com o processador completo desenvolvido pelo outro grupo da disciplina, dado que o projeto deles não estava completo quando os contactamos. Isso limitou nosso desenvolvimento, pois, após a integração, deveríamos modificar o processador para conter duas ULAs, além de maior número de saídas no banco de registradores e fazer a inserção dos sinais de controle nos registradores entre os estágios do *pipeline*.

Além dessas alterações, a unidade de controle do projeto entregue consiste de um modelo adaptado da que foi criada no início da disciplinas, portanto ela possui somente instruções aritméticas, de *Load* e de *Store*. Se a integração tivesse sido feita, teríamos um processador capaz de processar todas as instruções.

# Conclusão

FAZER A PARTE DE C AQUI NO COMEÇO

No que diz respeito à porção de VHDL do projeto, encontramos adversidades com as quais não contávamos e que, no fim, não fomos capazes de contornar. Apesar desses problemas, geramos um código que simula parcialmente um algoritmo de *Dynamic Scheduling*, demonstrando, pelo menos nos ciclos iniciais, a emissão dos *issues* e as próximas operações realizadas pelo scoreboard.

Por fim, mesmo com um processador em VHDL não capaz de executar todas as tarefas requisitadas, o grupo está contente com o produto final obtido, visto que conseguimos produzir o algoritmo de scoreboard completamente funcional em C e conseguimos avançar em grande parte do projeto em VHDL. Dada a complexidade da tarefa, tanto pela dificuldade do algoritmo quanto pela duplicidade de projetos completos a serem desenvolvidos, em C e em VHDL, acreditamos que completamos grande parte do que nos foi proposto e estamos satisfeitos.

# Bibliografia


(1) http://users.utcluj.ro/~sebestyen/_Word_docs/Cursuri/SSC_course_5_Scoreboard_ex.pdf
(2) https://people.eecs.berkeley.edu/~pattrsn/252S98/Lec04-tomosulo.pdf