Departamento de Engenharia Elétrica e de Computação

SEL0606 – Laboratório de Sistemas Digitais

Prof. Dr. Maximiliam Luppe

PRÁTICA Nº11

Aprendizado baseado em problema (PBL)

PBL03 - Circuitos Sequenciais - Máquinas de Estados Finitos

Problema:

Implementar uma Máquina de Estados utilizando linguagem de descrição de hardware VHDL

Equipamentos necessários:

- Kit DE10-Lite

Introdução:

A arquitetura RISC-V define seis formatos de instrução de base (figura 1): Tipo-R para operações de registradores; Tipo-I para valores imediatos short e loads; Tipo-S para stores; Tipo-B para desvios condicionais; Tipo-U para valores imediatos longos; e tipo-J para saltos incondicionais. As instruções do Tipo-R operam sobre três registradores, como `add rd, rs1, rs2`, que realiza a operação [`rd`] = [`rs1`] + [`rs2`], sendo `rd`, `rs1` e `rs2` os registradores do banco de registradores. Instruções do Tipo-I realizam operações que envolvem o uso de valores imediatos (incluídos nas instruções), como `addi rd, rs1, 42`, que realiza a operação [`rd`] = [`rs1`] + `42`. Instruções do Tipo-S e do Tipo-B, por sua similaridade no formato (operam sobre dois registradores e um valor imediato de 12 ou 13 bits), podem ser consideradas só um grupo, e realizam operações de armazenamento (Tipo-S) e de desvio de fluxo (Tipo-B), como `sw a0,4(sp)`, que armazena em M([`sp`] + `4`) o valor de `a0`, ou `beq a0,a1,L1`, que desvia o fluxo para o endereço `L1` se [`a0`] = [`a1`]. Da mesma forma, as instruções do Tipo-U e do Tipo-J também podem ser agrupadas num só grupo

(operam sobre um registrador e um valor imediato de 20 ou 21 bits), como `jal ra,factorial`, que desvia o fluxo para o endereço `factorial` e armazena o endereço de retorno em [`ra`].

Tabela 1 - Formatos de Instruções RV32I

31 30 25	24 21	20 19 15	14 12	11 8	7 6 0	
funct7	rs2	rs1	funct3	rd	opcode	Tipo R
imm[11:0]		rs1	funct3	rd	opcode	Tipo I
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	Tipo S
imm[12] imm[10:5]	rs2	rs1	funct3	imm[4:1] imm[11]	opcode	Tipo B
imm[31:12]				rd	opcode	Tipo U
imm[20]	imm[10:1] imm[11]	imm[19:12]		rd	opcode	Tipo J

Projeto:

O projeto a ser desenvolvido será a implementação de uma unidade de controle para um processador de 16 bits com arquitetura baseada na arquitetura RISC-V. Esta arquitetura terá um conjunto de instruções baseado no conjunto de instruções compactas do RISC-V (RV32C), voltado para aplicações em sistemas embarcados (tabela 2).

Tabela 2 - Formatos de Instruções RV32C

Formato	Significado	15 14 13	12 11	10 9	8 7	6 5	4 3 2	1 0
CR	Registrador	funct4		rd/rs1		rs2		op
CI	Imediato	funct3	imm	rd/rs1		imm		op
CSS	Store relativo a pilha	funct3	imm			rs2		op
CIW	Amplo imediato	funct3	imm				rd′	op
CL	Load	funct3	imm	rs1′		imm	rd′	op
CS	Store	funct3	imm	rs1′		imm	rs2′	op
CB	Desvio	funct3	offset	rs1′		offset		op
CJ	Salto	funct3	jump target					op

Para simplificar a implementação, serão utilizados 5 dos 8 formatos: CR (*Compact-Register*), CI (*Compact-Immediate*), CL (*Compact-Load*), CS (*Compact-Store*) e CB (*Compact-Branch*), com

pequenas modificações nos campos das instruções, criando a arquitetura RV16Cm (RISC-V de 16 bits com conjunto Compacto de instruções modificadas). Na tabela 3 é apresentado o novo formato das instruções.

Tabela 3 - Formatos de Instruções do RV16Cm

Campos	3-bits	5 bits	3-bits	3-bits	2-bits
Formato CRm	*Funct3*	00000	rd/rs1	rs2	*Opcode*
Formato CIm	*Funct3*	*Imm8[7:3]*	rd/rs1	*Imm8[2:0]*	*Opcode*
Formato CLm	*Funct3*	*Addr8[7:3]*	rd	*Addr8[2:0]*	*Opcode*
Formato CSm	*Funct3*	*Addr8[7:3]*	rs1	*Addr8[2:0]*	*Opcode*
Formato CBm	*Funct3*	*Addr11[7:3]*	*Addr11[10:8]*	*Addr11[2:0]*	*Opcode*

As instruções do Formato CRm utilizam apenas 2 operandos, sendo um deles (rd/rs1) tanto fonte como destino da operação a ser realizada. O outro operando de fonte é o `rs2`. O campo de registradores é de três bits, o que indica que há apenas 8 registradores que podem ser acessados (R0 a R7), todos de 16 bits. As operações que podem ser realizadas são soma (`add`), subtração (`sub`), OU lógico (`or`), E lógico (`and`) e Setar se menor (`slt`). As instruções do Formato CIm possuem dois operandos (`rd`/`rs1` e *Imm8*) e é utilizada para a implementar as instruções de soma com valor imediato (`addi`). As instruções do Formato CLm e CSm realizam o armazenamento (`sw`) e leitura (`lw`) da memória de dados. O campo de *Addr8* é de 8 bits, o que indica que o valor imediato é de 8 bits e que há 256 posições de memória de dados (2^8). As instruções do Formato CBm são de desvio condicional (`bneqz`), que causará o desvio no fluxo da execução das instruções se a última operação realizada pela ULA for diferente de zero. O campo de *Addr11* é de 11 bits, indicando que a memória de programa tem apenas 2048 palavras (2^{11}). Com isso, o contador de programas (PC) será de 16 bits, mas apenas os 11 bits menos significativos serão utilizados para compor o endereço de memória, tanto de dados como de programas. O registrador de instruções (IR) também será de 16 bits. Finalmente, como o campo *Opcode* tem apenas 2 bits, isto significa que o número de instruções que podem ser implementadas seria de apenas 4 (2^2). Mas, considerando o campo *Funct3*, que expande todos os formatos, este número pode ser multiplicado 8 (2^3), ampliando para 32 instruções. O resumo das principais diferenças entre a arquitetura RC32C e a arquitetura RV16Cm está indicado na tabela 4.

Tabela 4 - Principais diferenças entre RV32C e RV16Cm

	RV32C	RV16Cm
Barramento de dados	32 bits	16 bits
Tamanho da instrução	32 bits	16 bits
Barramento Memória Programa	32 bits	11 bits
Tamanho dos Registradores	32 bits	16 bits
Quantidade de registradores	16	8
Instruções tipo R	2 registradores	2 registradores
	rd <- rd op rs	rd <- rd op rs
Instrução LD (*load from mem*)	rt <- [rs + sign_ext(imm)]	rd <- [imm]
Instrução ST (*store mem*)	[rs + sign_ext(imm)] <- rt	[imm] <- rd
Instrução de branch	tipo U/J	tipo J
	bneq rs, rt, label	bneqz label
	se rs ≠ rt, PC <- PC+sign_ext(label)	se ZF=0, PC <- label

Na tabela 5 são apresentadas as instruções que serão implementadas para a arquitetura RV16Cm. As instruções estão agrupadas conforme o **Formato** delas. A coluna **Instrução** mostra a sintaxe da instrução. A coluna **Operação** indica a operação realizada. A coluna **ALUOp** indica a operação a ser realizada pela ULA (ALU_Control). A coluna **Op** mostra o *opcode* da instrução. Podemos ver que todas as instruções do **Formato** CRm tem o mesmo *opcode*, sendo diferenciadas pela coluna **Funct3**, que indica a operação lógico-aritmética a ser realizada. As colunas **rd/rs1** e **rs2** indicam os bits dos registradores de destino e fonte, respectivamente. No caso das instruções `lw`, o campo de 3 bits representa o destino (**rd**), enquanto no caso da instrução `sw`, a fonte (**rs1**). Tanto para o campo **Imm**, como para **Addr**, dependendo da instrução, é necessária uma estrutura que realize a junção e a extensão dos valores para 16 bits. Para as instruções do **Formato** CIm, as colunas **Imm** representam valores de 8 bits em complemento de 2, que deverão ser estendidas em sinal para 16 bits. Para as instruções do **Formato** CLm e CSm, as colunas **Addr** indicam o endereço de 8 bits do dado a ser lido (`lw`), ou armazenado (`sw`), e devem ser estendidas em zero também para 16 bits. Já para as instruções do **Formato** CBm, as colunas **Addr**, indicando o endereço de 11 bits da próxima instrução a ser executada, caso o desvio seja realizado, e devem ser estendidas em zero para 16 bits. As linhas em cinza claro são possíveis instruções que podem ser implementadas, com poucas alterações na arquitetura.

Tabela 5 - Instruções da arquitetura RV16Cm

Formato	Instrução	Operação	ALUOp	Funct3								rd/rs1			rs2			Op	
CRm	add rd, rs2	rd <- rd + rs2	.000	0	0	0	0	0	0	0	0	d	d	d	s	s	s	0	0
	sub rd, rs2	rd <- rd - rs2	.001	0	0	1	0	0	0	0	0	d	d	d	s	s	s	0	0
	and rd, rs2	rd <- rd AND rs2	.010	0	1	0	0	0	0	0	0	d	d	d	s	s	s	0	0
	or rd, rs2	rd <- rd OR rs2	.011	0	1	1	0	0	0	0	0	d	d	d	s	s	s	0	0
	slt rd, rs2	rd <- 1 se rd < rs, 0 caso contrário	.101	1	0	1	0	0	0	0	0	d	d	d	s	s	s	0	0

Formato	Instrução	Operação	ALUOp	Funct3			Imm[7:3]					rd/rs1			Imm[2:0]			Op	
CIm	addi rd, Imm	rd <- rd + s_ext(Imm)	.000	0	0	0	i	i	i	i	i	d	d	d	i	i	i	0	1
	subi rd, Imm	rd <- rd – s_ext(Imm)	.001	0	0	1	i	i	i	i	i	d	d	d	i	i	i	0	1
	andi rd, Imm	rd <- rd AND s_ext(Imm)	.010	0	1	0	i	i	i	i	i	d	d	d	i	i	i	0	1
	ori rd, Imm	rd <- rd OR s_ext(Imm)	.011	0	1	1	i	i	i	i	i	d	d	d	i	i	i	0	1

Formato	Instrução	Operação	ALUOp	Funct3			Addr[7:3]					rd			Addr[2:0]			Op	
CLm	lw rd, Addr	rd <- [Addr]	.xxx	0	0	0	a	a	a	a	a	d	d	d	a	a	a	1	0

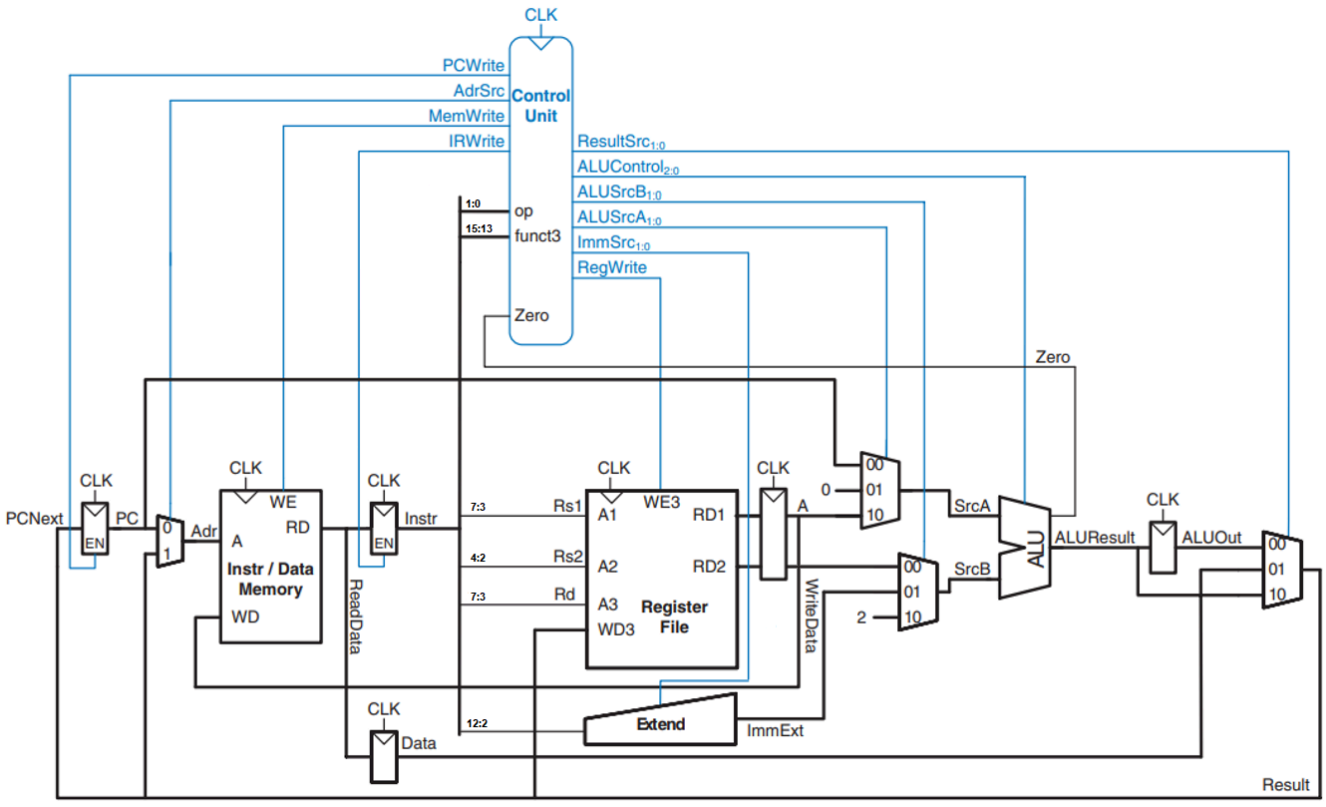
Formato	Instrução	Operação	ALUOp	Funct3			Addr[7:3]					rs1			Addr[2:0]			Op	
CSm	sw rs1, Addr	[Addr] <- rs1	.xxx	0	0	1	a	a	a	a	a	s	s	s	a	a	a	1	0

Formato	Instrução	Operação	ALUOp	Funct3			Addr[7:3]					Addr[10:8]			Addr[2:0]			Op	
CBm	bneqz Addr	PC <- Addr se ALUResult ≠ 0	.xxx	0	0	0	a	a	a	a	a	a	a	a	a	a	a	1	1
	beqz Addr	PC <- Addr se ALUResult = 0	.xxx	0	0	1	a	a	a	a	a	a	a	a	a	a	a	1	1

A arquitetura RISC-V possui pelo menos duas formas de implementação: Ciclo Único e Pipeline. Na implementação em Ciclo Único, todas as estruturas da arquitetura são ativadas ao mesmo temo, de acordo com a instrução a ser executada. Já na implementação Pipeline, a ativação das

estruturas segue uma sequência ordenada, e os dados vão fluindo pela arquitetura de forma controlada, por meio de registradores (*pipes*). Uma implementação intermediária entre estas duas é a Multiciclo. Nesta implementação, as estruturas da arquitetura são controladas por uma máquina de estados finita que gera, de forma ordenada, os sinais de controle que ativam cada uma das estruturas, de acordo com a instrução a ser executada, controlando o fluxo dos dados. Na figura 1 está a representação completa da arquitetura RV16C utilizando o conceito de Multiciclo. Ele é capaz de executar as instruções descritas na tabela 5: `add`, `sub`, `and`, `or`, `slt`, `lw`, `sw`, `addi` e `bneqz`. A arquitetura Multiciclo é dividida em três unidades: controle (formado pela *Control Unit* e lógica adicional), memóri (formado pela *Instruction/Data Memory*), e *datapath* (demais circuitos). Observe que a unidade de memória é responsável pelo armazenamento tanto das instruções como dos dados.

Figura 1 - Processador RISC-V multiciclo



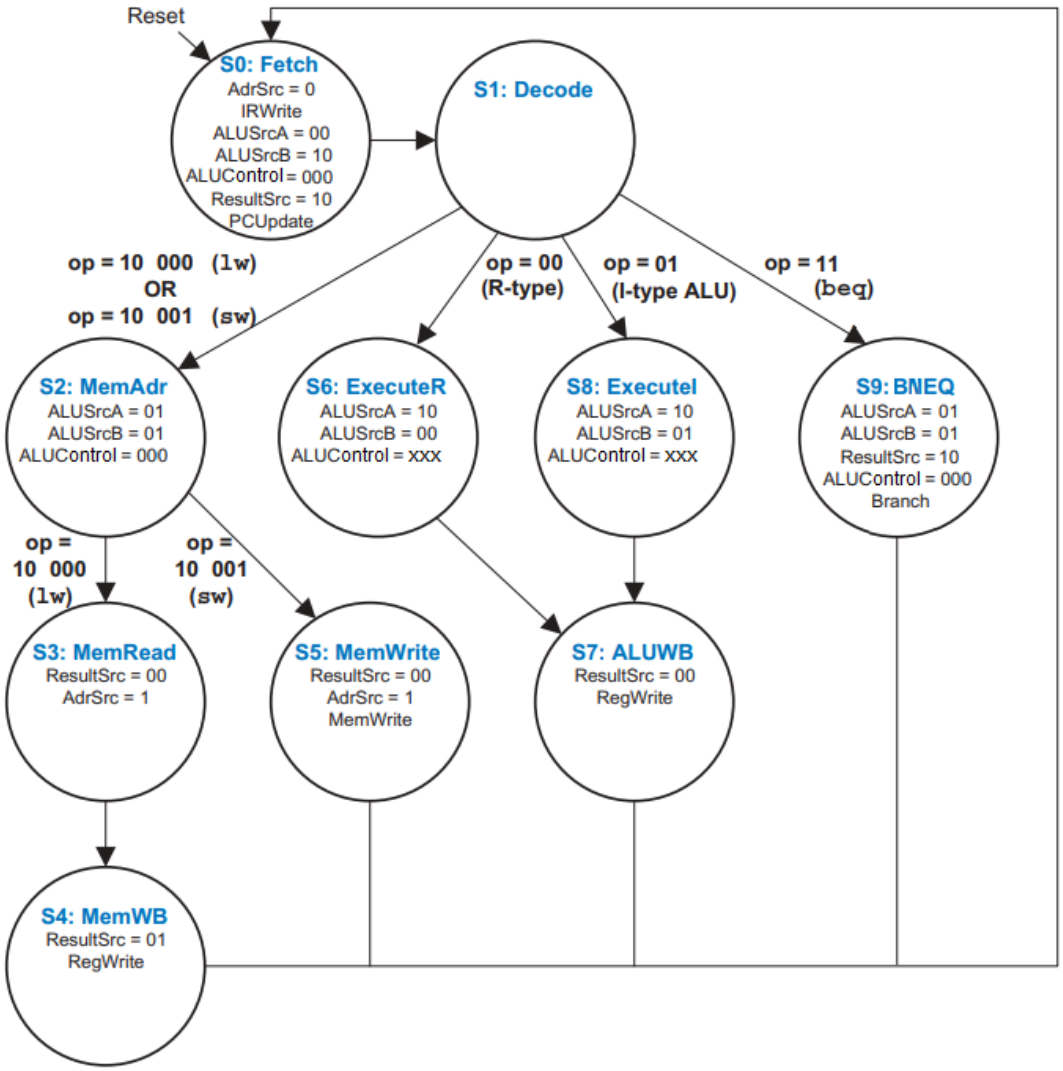
Fonte: Adaptado de Digital Design and Computer Architecture - RISC-V Edition

https://doi.org/10.1016/C2019-0-00213-0

Implementação da Unidade de Controle

Nesta prática será projetada e implementada a Unidade de Controle Multiciclo da arquitetura RV16C utilizando uma máquina de estados finitos (*Finite-State Machine* - FSM).

Figura 2 – Diagrama de Estados da Unidade de Controle do Processador RV16Cm multiciclo



O diagrama de estados desta FSM está descrito na figura 2. Cada estado está associado a uma instrução e define quais são os sinais de controle que deverão ser ativados para a correta execução da instrução pela arquitetura. Vale a pena ressaltar que há dois tipos de sinais de controle: de seleção e de

ativação. Os sinais de seleção são utilizados para selecionar o caminho que os dados percorrerão pela arquitetura. Eles acionam principalmente multiplexadores e a ULA, e devem manter seus valores nos estados seguintes, até serem novamente modificados. Eles são identificados pela atribuição de um valor a eles nos estados. Já os sinais de ativação controlam principalmente registradores e deverão ser ativados apenas nos estados indicados, permanecendo desativados nos estados seguintes. Observe que estes sinais não recebem valores dentro dos estados, sendo sua ativação em nível lógico alto.

Antes de começar a implementação do processador RV16Cm multiciclo, é necessário determinar os corretos sinais de controle para cada estado no Diagrama de Transição de Estados do processador multiciclo, apresentado na Tabela 6. Estes sinais serão as saídas da máquina de estados, que terá como entradas os sinais de clock (CLK), instrução (Op) e função (Funct3).

Tabela 6 - Sinais de saída da Unidade de Controle

Estado	PCWrite	AdrSrc	MemWrite	IRWrite	ResultSrc[1..0]	ALUControl[2..0]	ALUSrcB[1..0]	ALUSrcA[1..0]	ImmSrc[1..0]	RegWrite
S0: Fetch										
S1: Decode										
S2: MemAdr										
S3: MemRead										
S4: MemWB										
S5: MemWrite										
S6: ExecuteR										
S7: ALUWB										
S8: ExecuteI										
S9: BNEQZ										

Complete os dados de decodificação de saída do Decodificador Principal na Tabela 6. Baseados nos conhecimentos adquiridos na disciplina de Sistemas Digitais, implemente uma máquina de estados

finitos de acordo com a Figura 2. Tenha cuidado com cada passo. Demora muito mais tempo para depurar um circuito errado do que projetá-lo corretamente na primeira vez.

O relatório deve descrever, de forma sucinta, conceitos de Máquinas de Estados Finitos.

Procedimento Experimental:

Apresentar a implementação da Máquina de Estados Finitos (FSM), com entradas de CLK, CLR, OP, FUNCT3 e ZERO, e saídas PCWrite, AdrSrc, MemWrite, IRWrite, ResultSrc, ALUControl, ALUSrcA, ALUSrcB, ImmSrcB e RegWrite, baseada na arquitetura RV16Cm (figura 1), de acordo com o Diagrama de Estados da figura 2, utilizando a linguagem de descrição de hardware VHDL.

Criar uma pasta denominada `DE10_LITE_FSM`, com as subpastas `docs`, `modelsim`, `quartus` e `src`, e criar um projeto na pasta `quartus`, também denominado DE10_LITE_FSM.

Implementar a Máquina de Estados Finitos utilizando VHDL, denominado `FSM.v`, armazenando o código na pasta `src`. Incorporar o código da Máquina de Estados Finitos ao projeto principal (DE10_LITE_RegBank), ligando as chaves SW(1 downto 0), SW(4 downto 2) e SW(5) às entradas OP, FUNCT3 e ZERO, os push-buttons KEY(0) e KEY(1) às entradas CLK e CLR, as saídas aos LEDs e/ou displays de 7 segmentos, e executar o projeto no kit DE10_LITE.

Apresentar código VHDL, circuito RTL, número de células lógicas utilizadas e foto do kit com o circuito funcionando.

Exemplo de Máquina de 4 estados do tipo Moore:

```
-- A Moore machine's outputs are dependent only on the current state.
-- The output is written only when the state changes.  (State
-- transitions are synchronous.)

entity four_state_moore_state_machine is
      port(
            clk         : in bit;
            input       : in bit;
            reset       : in bit;
            output      : out bit_vector(1 downto 0)
      );
end entity;

architecture rtl of four_state_moore_state_machine is

      -- Build an enumerated type for the state machine
      type state_type is (s0, s1, s2, s3);

      -- Register to hold the current state
      signal state, next_state   : state_type;

begin
      -- Logic to advance to the next state
      process (clk, reset)
      begin
            if reset = '1' then
                  state <= s0;
            elsif (clk'event and clk = '1') then
                  state <= next_state;
            end if;
      end process;

      -- Logic to define the next state
      process (state)
            case state is
                  when s0=>
                        if input = '1' then
                              next_state <= s1;
                        else
                              next_state <= s0;
                        end if;
                  when s1=>
                        if input = '1' then
                              next_state <= s2;
                        else
                              next_state <= s1;
                        end if;
                  when s2=>
                        if input = '1' then
```

```
                        next_state <= s3;
                    else
                        next_state <= s2;
                    end if;
                when s3 =>
                    if input = '1' then
                        next_state <= s0;
                    else
                        next_state <= s3;
                    end if;
            end case;
        end process;

        -- Output depends solely on the current state
        process (state)
        begin
            case state is
                when s0 =>
                    output <= "00";
                when s1 =>
                    output <= "01";
                when s2 =>
                    output <= "10";
                when s3 =>
                    output <= "11";
            end case;
        end process;

end rtl;
```