UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA

CONCEPÇÃO ESTRUTURADA DE CIRCUITOS INTEGRADOS
RELATÓRIO DE ATIVIDADES

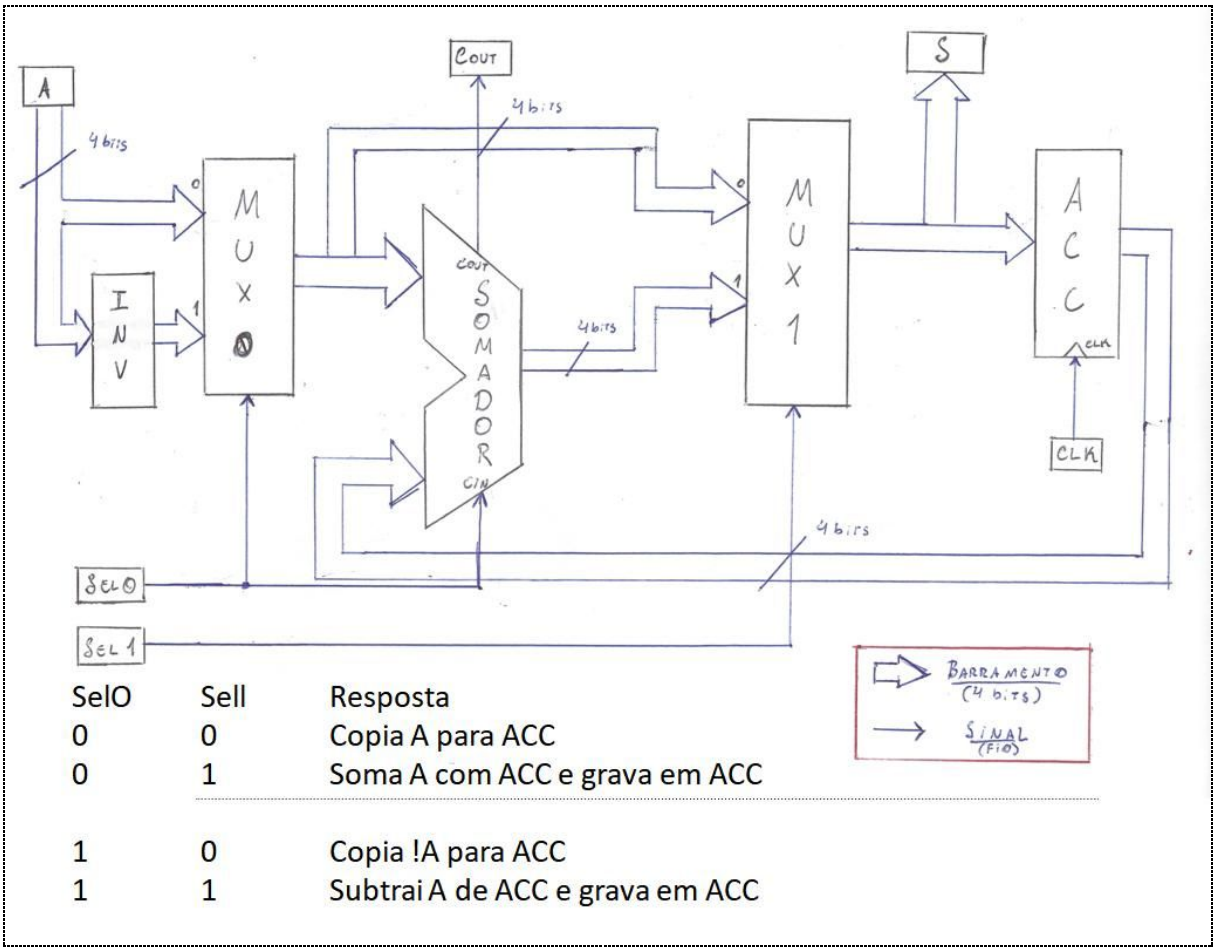
GABRIEL DE OLIVEIRA MOURA SOARES - 20160163998

João Pessoa
NOVEMBRO/2019

Sumário

1. Introdução

Este trabalho é um relatório da implementação do Golden Model ‘Addac’ (Figura 1), para isso será implementado o Golden Model de cada componente lógico e o addac será a combinação destes. Cada Golden Model vai gerar um arquivo no formato ‘.tv’, que contém vetores de testes, onde tem entradas e as respectivas saídas esperadas para aquela entrada.



SelO	Sell	Resposta
0	0	Copia A para ACC
0	1	Soma A com ACC e grava em ACC
1	0	Copia !A para ACC
1	1	Subtrai A de ACC e grava em ACC

Figura 1 - Esquema do Addac

2. Construção dos Golden Model's

Os goldens models serão implementados na linguagem Python. Antes de vermos a implementação dos componentes lógicos em si, veremos algumas estruturas que foram criadas para dar suporte a criação dos componentes lógicos:

2.1. Estruturas de suporte

As primeiras delas é a 'Wire' que representa a entidade onde os dados (bits) serão inseridos e lidos e a interface WireListener.

```python
class Wire:
    def __init__(self):
        self.data: int = 0
        self.listeners: List[WireListener] = []

    def listen(self, listener: WireListener):
        listener.on_wire_change()
        self.listeners.append(listener)

    def remove_listener(self, listener: WireListener):
        self.listeners.remove(listener)

    def notify_listeners(self):
        for listener in self.listeners:
            listener.on_wire_change()

    def set(self, data: int):
        self.data = data
        self.notify_listeners()

    def __str__(self):
        return format(self.data, '04b')

    def __format__(self, format_spec):
        return format(self.data, format_spec)
```

Figura 2 - Classe Wire

```python
class WireListener(ABC):
    @abstractmethod
    def on_wire_change(self):
        pass
```

Figura 3 - Classe WireListener

Um objeto que implementa a interface ‘WireListener’ pode se registrar em um ‘Wire’ para ser notificado quando houver uma mudança nos dados do Wire. Assim, toda vez que alguém inserir um dado no wire (através do método ‘set’) todos aqueles que se registraram serão notificados da mudança, através do método ‘on_wire_change’ definido na interface.

Outra classe que foi criada para dar suporte a criação dos blocos lógicos, foi uma classe abstrata “LogicBlock” que implementa algumas funcionalidades que serão comuns a todos os blocos lógicos.

```python
class LogicBlock(WireListener):
    def __init__(self, inputs: Dict[str, Wire], outputs: List[str]):
        self.inputs = inputs
        self.outputs: Dict[str, Wire] = {x: Wire() for x in outputs}
        self.y = self.outputs[outputs[0]]  # first element in outputs is the main output
        for inp in inputs.values():
            inp.listen(self)

    def on_wire_change(self):
        values: Dict[str, int] = {k: v.data for k, v in self.inputs.items()}
        result = self.operation(values)
        for k, v in result.items():
            self.outputs[k].set(v)

    def add_input(self, name: str, a: Wire):
        if name in self.inputs:
            self.inputs[name].remove_listener(self)

        self.inputs[name] = a
        a.listen(self)

    @abstractmethod
    def operation(self, inputs: Dict[str, int]) -> Dict[str, int]:
        pass
```

Figura 4 - Classe LogicBlock

Ela implementa a interface WireListener e se registra para escutar as mudanças de todas as suas entradas. Sempre que alguma entrada muda ele executa o método ‘operation’ passando as entradas atuais, aplicando a operação do bloco lógico e pega as saídas e atualizam nas saídas que também são ‘wires’. Dessa forma, qualquer bloco lógico pode herdar as funcionalidades dessa classe, bastando apenas implementar o método ‘operation’ que define o comportamento do bloco lógico.

2.2. Inversor

Primeiro componente lógico criado foi o inversor, ele foi criado como sendo um inversor de 4 bits, que faz a inversão bit a bit. A classe ‘Inverter’ recebe um ‘Wire’ em seu construtor que representa a entrada do inversor, e tem apenas uma saída ‘y’. Ela herda todo o comportamento do ‘LogicBlock’ e portanto basta apenas sobrecarregar o método ‘operation’ onde ele recebe a entrada x, faz a inversão de seus bits, e retorna a saída y

```
class Inverter(LogicBlock):
    def __init__(self, input_: Wire):
        super().__init__({'x': input_}, ['y'])

    def operation(self, inputs: Dict[str, int]) -> Dict[str, int]:
        x = inputs['x']
        y = ~x & 0b1111
        return {'y': y}
```

Figura 5 - Classe Inverter

Além disso foi criado a função para criar os vetores de teste que foram salvos no arquivo ‘inverter.tv’

```
def create_inverter_tvs():
    inp = Wire()
    inverter = Inverter(inp)
    file = open('inverter.tv', 'w')
    file.write("# a_y\n")
    for i in range(0, 2**4):
        inp.set(i)
        file.write("{:04b}_{:04b}\n".format(i, inverter.y.data))
    file.close()
```

Figura 6 - Função para criação dos vetores de teste do inversor

Nele é criado um ‘Wire’ que é passado como entrada para o inversor; No for é gerado todos os casos de teste possíveis, que são todos os valore de 4 bits (0 a 15), e cada entrada é inserida no Wire de entrada do inversor e é lido então a saída dele e são gravados no arquivo.

```
# a_y
0000_1111
0001_1110
0010_1101
0011_1100
```

Figura 7 - Primeiras linhas 2.4. do ‘inverter.tv’

2.3. Mux

O Mux segue a mesma linha de criação do inversor: Recebe as suas entradas, que no caso dele é dois dados e um seletor, e define uma saída ‘y’; E também herda o comportamento da classe LogicBlock definindo apenas a operação do multiplexador.

```
class Mux(LogicBlock):
    def __init__(self, data1: Wire, data2: Wire, sel: Wire):
        inputs = {
            'd1': data1,
            'd2': data2,
            'sel': sel,
        }
        super().__init__(inputs, ['y'])

    def operation(self, inputs: Dict[str, int]) -> Dict[str, int]:
        if inputs['sel'] == 0:
            return {'y': inputs['d1']}
        else:
            return {'y': inputs['d2']}
```

Figura 8 - Classe Mux

Também foi criado a função que gera os vetores de teste. Diferente do inversor, não foi gerado todos os vetores possíveis, pois seriam 512 casos, e a princípio não é necessário uma quantidade tão grande. Então foi selecionado 6 possíveis entradas e nas entradas do mux foram inseridas todos os pares possíveis desses elementos (inclusive o mesmo elemento nas duas entradas) e para cada par, foi selecionado um por vez.

```
def create_mux_tvs():
    d1 = Wire()
    d2 = Wire()
    sel = Wire()
    mux = Mux(data1=d1, data2=d2, sel=sel)

    data_test = [0b0000, 0b1111, 0b1010, 0b0101, 0b0011, 0b1100]

    file = open('mux.tv', 'w')
    file.write("# d1_d2_sel_y\n")
    for i1 in data_test:
        d1.set(i1)
        for i2 in data_test:
            d2.set(i2)
            for s in [0, 1]:
                sel.set(s)
                file.write("{:04b}_{:04b}_{:1b}_{:04b}\n".format(i1, i2, s, mux.y.data))
    file.close()
```

Figura 9 - Função para criação dos vetores de teste do multiplexador

```
# d1_d2_sel_y
0000_0000_0_0000
0000_0000_1_0000
0000_1111_0_0000
0000_1111_1_1111
```

Figura 10 - Primeiras linhas do ‘mux.tv’

2.4. Somador

O somador tem 3 entradas: a e b de 4 bits e o cin de 1 bit; e 2 saídas: y, que é o resultado da soma com 4 bits, e o cout que seria o quinto bit da soma. A implementação do somador segue os mesmos passos dos outros blocos lógicos.

```python
class Adder(LogicBlock):
    def __init__(self, a: Wire, b: Wire, cin: Wire):
        inputs = {
            'a': a,
            'b': b,
            'cin': cin
        }
        super().__init__(inputs, ['y', 'cout'])
        self.cout = self.outputs['cout']

    def operation(self, inputs: Dict[str, int]) -> Dict[str, int]:
        sum_ = inputs['a'] + inputs['b'] + inputs['cin']
        cout = sum_ >> 4
        y = sum_ & 0b1111
        return {
            'y': y,
            'cout': cout
        }
```

Figura 11 - Classe Adder

Para gerar os vetores de testes, foram selecionados alguns números a serem somados, e foi variado o cin para cada par escolhido, da mesma forma que foi feita no mux, e os vetores foram salvos no arquivo ‘adder.tv’

```python
def create_adder_tvs():
    a = Wire()
    b = Wire()
    cin = Wire()
    adder = Adder(a=a, b=b, cin=cin)
    data_test = [0b0000, 0b1111, 0b1010, 0b0101, 0b0011, 0b1100]

    file = open('adder.tv', 'w')
    file.write("# a_b_cin_y_cout\n")
    for i1 in data_test:
        a.set(i1)
        for i2 in data_test:
            b.set(i2)
            for c in [0, 1]:
                cin.set(c)
                file.write("{:04b}_{:04b}_{:1b}_{:04b}_{:1b}\n".format(i1, i2, c, adder.y, adder.cout))
    file.close()
```

Figura 12 - Função para criação dos vetores de teste do multiplexador

```
# a_b_cin_y_cout
0000_0000_0_0000_0
0000_0000_1_0001_0
0000_1111_0_1111_0
0000_1111_1_0000_1
```

Figura 13 - Primeiras linhas do ‘adder.tv’

2.5. Acumulador

Na classe do acumulador, é armazenado em seu estado o último clock recebido. Quando alguma entrada é alterada, ele verifica se o último clock foi 0 e o clock atual é 1, em caso positivo, ele copia a entrada para a saída, em qualquer outro caso, nada é feito. Sua implementação é feita da mesma forma dos outros blocos.

```python
class Acc(LogicBlock):
    def __init__(self, x: Wire, clk: Wire):
        self.lastClk: int = 0
        super().__init__({'x': x, 'clk': clk}, ['y'])

    def operation(self, inputs: Dict[str, int]) -> Dict[str, int]:
        output: Dict = {'y': inputs['x']} if inputs['clk'] == 1 and self.lastClk == 0 else {}
        self.lastClk = inputs['clk']
        return output
```

Figura 14 - Classe Acc

Foram gerados 50 vetores de teste para o acumulador. Para cada vetor de teste, era feito um sorteio para escolher se ia mudar a entrada ou o clock em relação ao vetor anterior com 1/3 de chance de ser o clock e 2/3 de chance de ser o dado. Se o clock fosse escolhido, o clock era invertido, se o dado era escolhido, era gerado um valor aleatório entre todos os possíveis.

```python
def create_acc_tvs():
    x = Wire()
    clk = Wire()
    acc = Acc(x=x, clk=clk)

    file = open('acc.tv', 'w')
    file.write("# x_clk_y\n")

    clock: int = 0
    data: int = 0

    for i in range(50):  # gera 50 casos de teste
        if randint(0, 1) == 0:  # muda dado
            data = randint(0, 15)
            x.set(data)
        else:  # inverte clock
            clock = 1 - clock
            clk.set(clock)

        file.write("{:04b}_{:1b}_{:04b}\n".format(data, clock, acc.y))

    file.close()
```

Figura 15 - Função para criação dos vetores de teste do acumulador

```
# x_clk_y
0000_1_0000
0000_0_0000
0101_0_0000
0101_1_0101
```

Figura 16 - Primeiras linhas do ‘acc.tv’

2.6. Addac

O addac é simplesmente a composição dos componentes criados, da forma que é mostrada na Figura 1. Como todos os componentes têm entradas e saídas como objetos do tipo ‘Wire’, basta conectar corretamente esses objetos que o circuito é montado.

```
class Addac:
    def __init__(self, a: Wire, sel0: Wire, sel1: Wire, clk: Wire):
        inv = Inverter(a)
        mux0 = Mux(data1=a, data2=inv.y, sel=sel0)
        adder = Adder(a=mux0.y, cin=sel0, b=Wire())
        mux1 = Mux(data1=mux0.y, data2=adder.y, sel=sel1)
        acc = Acc(x=mux1.y, clk=clk)
        adder.add_input('b', acc.y)

        self.s = mux1.y
        self.cout = adder.cout
        self.acc = acc.y
```

Figura 17 - Classe Addac

No momento de criação do somador, ainda não tinha sido criado o acumulador, que era sua segunda entrada, por isso foi inserido um fio ‘dummy’ que depois foi substituído pelo acumulador na linha imediatamente abaixo da sua criação.Figura 12 - Primeiras linhas do ‘acc.tv’.

Para cada função (cada par possível de sel0 e sel1) foram criados 50 vetores de teste com o mesmo critério que foi criado os vetores de teste do acumulador.

```
def create_addac_tvs():
    a = Wire()
    sel0 = Wire()
    sel1 = Wire()
    clk = Wire()
    addac = Addac(a=a, sel0=sel0, sel1=sel1, clk=clk)

    file = open('addac.tv', 'w')
    file.write("# a_sel0_sel1_clk_cout_s\n")

    for s0 in [0, 1]:
        sel0.set(s0)
        for s1 in [0, 1]:
            sel1.set(s1)
            for i in range(50):  # 50 casos de teste para cada função
                if randint(0, 2) == 0:  # inverte clock 1/3 de chance
                    clock = 1 - clk.data
                    clk.set(clock)
                else:  # muda entrada 2/3 de chance
                    data = randint(0, 15)
                    a.set(data)

                file.write('{:04b}_{:1b}_{:1b}_{:1b}_{:1b}_{:04b}\n'
                           .format(a, sel0, sel1, clk, addac.cout, addac.s))
    file.close()
    return
```

Figura 18 - Função para criação dos vetores de teste do addac

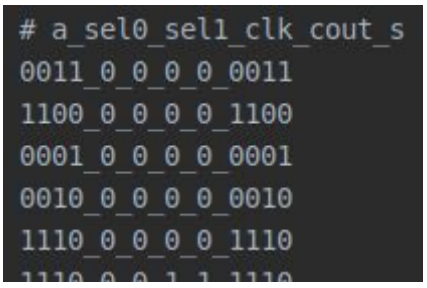


Figura 19 - Primeiras linhas do ‘addac.tv’