UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA

CONCEPÇÃO ESTRUTURADA DE CIRCUITOS INTEGRADOS
RELATÓRIO DE ATIVIDADES

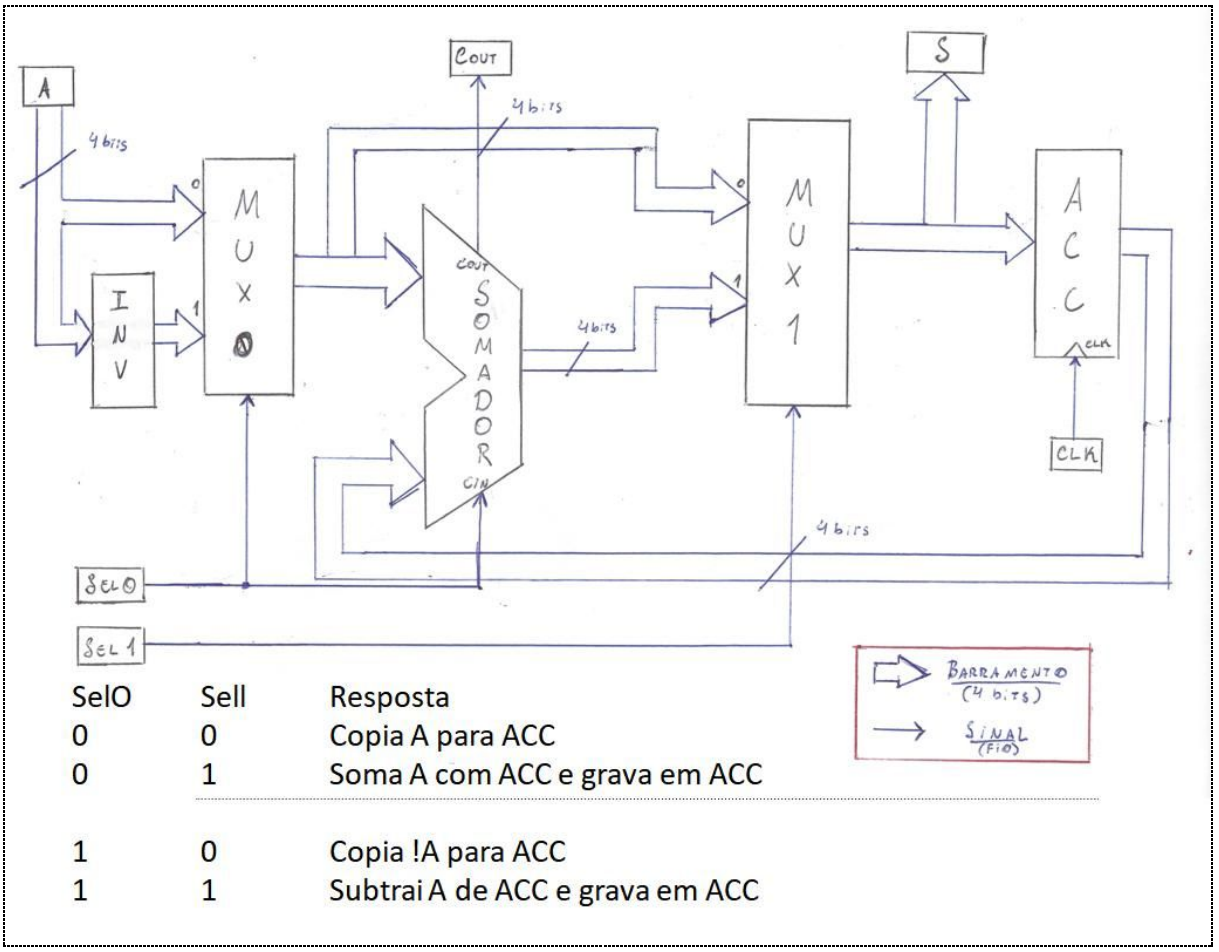
GABRIEL DE OLIVEIRA MOURA SOARES - 20160163998

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

No estado da classe wire é armazenado o último dado que foi enviado para aquele wire, que é definido com um inteiro, onde o(s) bit(s) do inteiro é o dado que está passando pelo wire.

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, que inverte o bit da entrada e envia para a saída. A classe ‘Inverter’ recebe um ‘Wire’ em seu construtor que representa a entrada do inversor, e tem apenas uma saída ‘y que é a entrada invertida’. Ela herda todo o comportamento do ‘LogicBlock’ e portanto basta apenas sobrecarregar o método ‘operation’ onde ele recebe a entrada x, faz a inversão do bit, e retorna a saída y

```python
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 & 1
        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’

```python
def create_inverter_tvs():
    a = Wire()
    inverter = Inverter(a)
    file = open('inverter.tv', 'w')
    file.write("# a_y\n")
    for i in [0, 1]:
        a.set(i)
        file.write("{:1b}_{:1b}\n".format(a, inverter.y))
    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 os 2 casos de teste possíveis, que é com entrada 0 e com entrada 1, 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
0_1
1_0
```

Figura 7 - Conteúdo 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, d0: Wire, d1: Wire, sel: Wire):
        inputs = {
            'd0': d0,
            'd1': d1,
            'sel': sel,
        }
        super().__init__(inputs, ['y'])

    def operation(self, inputs: Dict[str, int]) -> Dict[str, int]:
        if inputs['sel'] == 0:
            return {'y': inputs['d0']}
        else:
            return {'y': inputs['d1']}
```

Figura 8 - Classe Mux

Também foi criado uma função para gerar os vetores de teste, gerando todas as possíveis entradas, que nesse caso foram $2^3 = 8$ pois são 3 bits de entrada.

```
def create_mux_tvs():
    d0 = Wire()
    d1 = Wire()
    sel = Wire()
    mux = Mux(d0=d0, d1=d1, sel=sel)

    file = open('mux.tv', 'w')
    file.write("# d0 d1 sel y\n")
    for i0 in [0, 1]:
        d0.set(i0)
        for i1 in [0, 1]:
            d1.set(i1)
            for s in [0, 1]:
                sel.set(s)
                file.write("{:1b}_{:1b}_{:1b}_{:1b}\n".format(d0, d1, sel, mux.y))
    file.close()
```

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

```
# d0 d1 sel y
0 0 0 0
0 0 1 0
0 1 0 0
0 1 1 1
1 0 0 1
1 0 1 0
1 1 0 1
1 1 1 1
```

Figura 10 - Conteúdo do ‘mux.tv’

2.4. Somador

O somador tem 3 entradas: a e b e cin; e 2 saídas: cout que é o bit mais significativo da soma e y que é o bit menos significativo da soma. A implementação do somador foi feita seguindo as operações realizadas por um circuito ‘Full Adder’, como se tivesse as portas lógicas.

```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]:
        a, b, cin = inputs['a'], inputs['b'], inputs['cin']

        p = a ^ b
        sum_ = p ^ cin
        cout = a & b | (p & cin)

        return {
            'y': sum_,
            'cout': cout
        }
```

Figura 11 - Classe Adder

Também foram gerados todos os possíveis vetores de teste, da mesma forma que foi feito com o mux.

```
def create_adder_tvs():
    a = Wire()
    b = Wire()
    cin = Wire()
    adder = Adder(a=a, b=b, cin=cin)

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

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

```
# a b cin cout y
0 0 0 0 0
0 0 1 0 1
0 1 0 0 1
0 1 1 1 0
1 0 0 0 1
1 0 1 1 0
1 1 0 1 0
1 1 1 1 1
```

Figura 13 - Conteúdo 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.

```
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 15 vetores de teste para o acumulador. Para cada vetor de teste, era sorteado se invertia o dado de entrada ou o clock, em relação ao vetor anterior com ⅓ de chance de inverter o block e ⅔ de chance de inverter o dado de entrada.

```
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")

    for i in range(15):  # gera 15 casos de teste
        if randint(0, 2) == 0:  # inverte clock 1/3 de chance
            clock = 1 - clk.data
            clk.set(clock)
        else:  # inverte entrada 2/3 de chance
            data = 1 - x.data
            x.set(data)

        file.write("{:1b}_{:1b}_{:1b}\n".format(x, clk, acc.y))

    file.close()
```

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

```
# x_clk_y
0_1_0
1_1_0
0_1_0
1_1_0
```

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, os dados são de 4 bits, para chegar nesse circuito, iremos criar uma versão de 1 bit que será instanciada 4 vezes. Como todos os componentes têm entradas e saídas como objetos do tipo ‘Wire’, basta conectar corretamente esses objetos que o circuito é montado.

```python
class Addac:
    def __init__(self, a: Wire, sel0: Wire, sel1: Wire, clk: Wire, cin: Wire = None):
        inv = Inverter(a)
        mux0 = Mux(d0=a, d1=inv.y, sel=sel0)
        adder_cin = sel0 if cin is None else cin
        adder = Adder(a=mux0.y, cin=adder_cin, b=Wire())
        mux1 = Mux(d0=mux0.y, d1=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
```

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.

Observe que a classe tem um parâmetro opcional ‘cin’, quando ele é passado, ele é inserido no cin do somador, quando ele não é passado, o cin do somador é o sel0, como no esquema da Figura 1. Isso vai ser útil na criação do addac de 4 bits.

Para cada função (cada par possível de sel0 e sel1) foram criados 15 vetores de teste com o mesmo critério que foi criado os vetores de teste do acumulador.

```python
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("# sel0_sel1_a_clk_cout_s\n")

    for s0 in [0, 1]:
        sel0.set(s0)
        for s1 in [0, 1]:
            sel1.set(s1)
            for i in range(15):  # 15 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:  # inverte entrada 2/3 de chance
                    data = 1 - a.data
                    a.set(data)

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

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

```
# sel0 sel1 a clk cout s
0 0 1 0 0 1
0 0 0 0 0 0
0 0 0 1 0 0
0 0 1 1 0 1
0 0 0 1 0 0
0 0 1 1 0 1
0 0 0 1 0 0
0 0 1 1 0 1
```

Figura 19 - Primeiras linhas do ‘addac.tv’

2.7. Addac4

Para implementar o Addac de 4 bits (Addac4) iremos utilizar 4 instâncias do addac de 1 bit, fazendo a conexão de acordo com o diagrama abaixo.

Para ajudar na implementação, foram criadas mais duas estruturas auxiliares: a WireSplitter que recebe 1 wire com dados de 4 bits, e transforma em 4 wires com dados de 1 bit cada; e a WireJoiner que faz o inverso, recebendo 4 wires com dados de 1 bit, e transformando em 1 wire com 4 bits de dados.

```python
class WireSplitter(LogicBlock):
    def __init__(self, a: Wire):
        bits_keys = ['bit{}'.format(i) for i in range(4)]
        super(WireSplitter, self).__init__({'a': a}, bits_keys)
        self.bits = [self.outputs[k] for k in bits_keys]

    def operation(self, inputs: Dict[str, int]) -> Dict[str, int]:
        a_bits = [(inputs['a'] >> x) & 1 for x in range(4)]
        return {'bit{}'.format(i): a_bits[i] for i in range(4)}
```

Figura 20 - Classe WireSplitter

```python
class WireJoiner(LogicBlock):
    def __init__(self, bits: List[Wire]):
        super(WireJoiner, self).__init__({'bit{}'.format(i): bits[i] for i in range(4)}, ['y'])

    def operation(self, inputs: Dict[str, int]) -> Dict[str, int]:
        bits = [inputs['bit{}'.format(i)] for i in range(4)]
        out = 0
        for i in range(4):
            out += (bits[i] << i)

        return {'y': out}
```

Figura 21 - Wire Joiner

No Addac4 como entrada ‘a’ recebemos 1 wire de 4 bits, então fazemos o split para colocar nas entradas dos 4 addacs de 1 bit. Depois pegamos as saídas dos 4 addacs é fazemos um join, para transformar a saída em 1 wire de 4 bits.

```
class Addac4:
    def __init__(self, a: Wire, sel0: Wire, sel1: Wire, clk: Wire):
        # separa bits em 4 wires
        splitted = WireSplitter(a)

        addac0 = Addac(a=splitted.bits[0], sel0=sel0, sel1=sel1, clk=clk, cin=sel0)
        addac1 = Addac(a=splitted.bits[1], sel0=sel0, sel1=sel1, clk=clk, cin=addac0.cout)
        addac2 = Addac(a=splitted.bits[2], sel0=sel0, sel1=sel1, clk=clk, cin=addac1.cout)
        addac3 = Addac(a=splitted.bits[3], sel0=sel0, sel1=sel1, clk=clk, cin=addac2.cout)

        # junta os bits em 1 wire
        s_bits = [addac0.s, addac1.s, addac2.s, addac3.s]
        joiner = WireJoiner(s_bits)

        # saidas
        self.s = joiner.y
        self.cout = addac3.cout
```

Figura 22 - Classe Addac4

Para gerar os vetores de teste para o addac de 4 bits foi usado o mesmo critério que o addac de 1 bit, porém a entrada no lugar de ser invertida, é sorteada entre os valores possíveis (qualquer valor de 4 bits), e como há uma maior possibilidade de entradas, foram geradas 60 entradas para cada função do Addac4.

```python
def create_addac4_tvs():
    a = Wire()
    sel0 = Wire()
    sel1 = Wire()
    clk = Wire()
    addac4 = Addac4(a=a, sel0=sel0, sel1=sel1, clk=clk)

    file = open('../simulation_modelsim/addac4.tv', 'w')
    file.write("# sel0_sel1_a_clk_cout_s\n")

    for s0 in [0, 1]:
        sel0.set(s0)
        for s1 in [0, 1]:
            sel1.set(s1)
            for i in range(60):  # 60 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:  # inverte entrada 2/3 de chance
                    data = randint(0, 15)
                    a.set(data)

                file.write('{:1b}_{:1b}_{:04b}_{:1b}_{:1b}_{:04b}\n'
                           .format(sel0, sel1, a, clk, addac4.cout, addac4.s))
    file.close()
```

Figura 23 - Função para criação dos vetores de teste do Addac4

```
# sel0_sel1_a_clk_cout_s
0_0_0000_1_0_0000
0_0_1101_1_0_1101
0_0_1001_1_0_1001
0_0_1001_0_0_1001
0_0_1001_1_1_1001
0_0_1101_1_1_1101
0_0_0100_1_0_0100
0_0_1010_1_1_1010
```

Figura 24 - Primeiras linhas do Addac4.tv