Universidade de Brasília

Organização e Arquitetura de Computadores

Experimento I

Assembly MIPS e Verilog

---

29 de Abril de 2014

Professor	Marcus Vinícius Lamar - turma A	
Alunos:		
	Danilo Cardoso	11/0010515
	Gustavo Meneses	10/0103766
	Juarez Aires	11/0032829
	Matheus Ramiro	09/0125894

A Assembly MIPS

A.1 Ordenação

A.1.a

A tabela 1 a seguir mostra os resultados obtidos. Os resultados foram obtidos com o código 'sort.s' disponível no moodle e o vetor de entrada V[10]=5,8,3,4,7,6,8,0,1,9. Para se obter a versão decrescente da rotina foi preciso apenas mudar a linha de comparação, trocando o comando **beq** por **bne** no loop for2 da rotina sort.

	Crescente	Decrescente
Instruções tipo R	255(31%)	276(32%)
Instruções tipo I	472(58%)	499(58%)
Instruções tipo J	75(9%)	81(9%)
Total	802	856

Tabela 1

A.1.b

Olhando toda a lista de execução do algoritmo, foram encontrados três exemplos distintos de pseudo-instruções. As instruções *load address*, *load immediate* e *move*. Nenhuma das três existe na linguagem assembly mips, mas o montador substitui da seguinte forma:

Pseudo-instruções	Instruções Reais
la $a0, label	lui $t0, adress
li $t0, imediato	addiu $t0, $zero, imediato
move $t0, $t1	addu $t0, $zero, $t1

Tabela 2: Substituição de pseudo-instruções por instruções reais

A.1.c

O valor 589834 armazenado no endereço 0x10010028 está relacionado com os caracteres ASCII `\n` e `\t` declarados logo após o vetor de entrada no começo do programa. Notamos que 589834 é escrito em binário como 0x0009000a, a concatenação de 0x0009 com 0x000a. Na tabela ASCII os valores para `\n` e `\t` são, respectivamente, 10 e 9. Observando o padrão da tabela, vemos que a posição do dado na memória depende da ordem em que ele foi inserido, elementos inseridos primeiros tem menor índice na memória. O primeiro char inserido, e portanto de menor endereço, foi `\n` = 0x000a. Quando o Mars interpretou a word, no entando, esse byte foi colocado na parte menos significativa. Ou seja, o elemento de menor endereço

é aquele menos significativo. Concluímos então que estamos diante de uma representação Little-endian.

A.1.d

- Sendo I o número de instruções, CPI o número de ciclos de clocks por instruções e T o período de clock, sabemos que o tempo de execução t é dado por:

$$t = I \cdot CPI \cdot T \tag{1}$$

Com CPI = 1, T = 1/f = 1/50M = 2 x 10^{-8}, $I_1 = 802$ e $I_2 = 856$ temos:

$$\begin{aligned} t_1 &= 16.04\mu s \\ t_2 &= 17.12\mu s \end{aligned} \tag{2}$$

- O melhor caso do algoritmo é o caso em que ele não tem que fazer nenhuma troca e o pior caso, o caso em que ele tem que fazer todas as trocas possíveis. Para analisar o custo vamos escrever o algoritmo em pseudo-código:

```
para i = 0 ate i = n-1:
        para j = i-1 ate j = 0
                se v[j] > v[j+1]
                        troca(v[j],v[j+1])
                caso contrario
                        next i
        fim iteracao j
fim iteracao i
```

No melhor caso, simplesmente faremos uma comparação entre cada elemento e seu sucessor. Essa comparação é feita n-1 vezes, do primeiro índice até o penúltimo e a complexidade é O(n).

No pior caso o vetor está completamente desordenado e em cada iteração de i, realizamos trocas em toda iteração j. Temos n iterações em i, e, para cada i, temos i-1 iterações em j. O custo total está relacionado com o produto n(n-1) e portanto a complexidade é O(n^2).

Para verificar os resultados teóricos, foi utilizada a fermenta do mars para fazer a contagem de instruções para o melhor e o pior caso para vários tamanhos do vetor e os resultados são claros, a teoria é sustentada por esses resultados. Vale o comentário que o pior caso para n=50 precisou de 26 mil operações enquanto o melhor caso para n=500 precisou de 17544, a diferença de escala realmente é absurda.

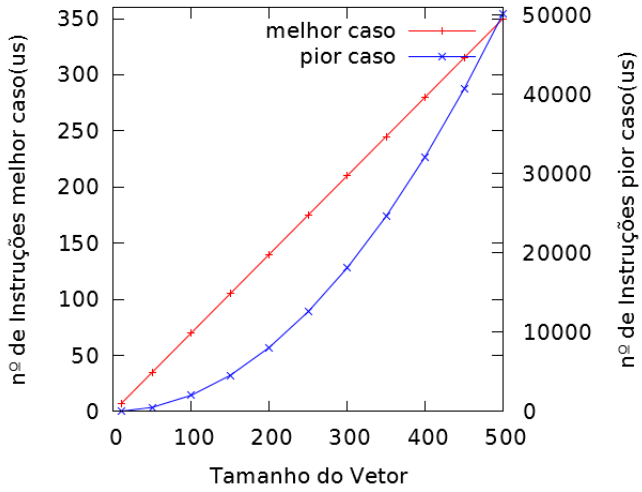

Figura 1: Tempo de execução para o melhor e o pior caso

A.2 Compilador MIPS GCC

A.2.a

Os resultados dos testes realizados para comparação são mostrados na tabela a seguir.

Tipo de Instrução	Assembly	C	C, -O1	C -O2	C -O3
Tipo R	255 (31%)	731 (33%)	199 (27%)	191(27%)	*226 (41%)
Tipo I	472 (58%)	1456 (66%)	503 (70%)	504 (72%)	*312 (57%)
Tipo J	75 (9%)	12 (0%)	10 (1%)	1 (0%)	*7 (1%)
Total	802	2199	712	696	*545

Tabela 3: Quantidade e tipos de instruções para cada versão do bubblesort

Vários problemas foram encontrados para fazer com que o código assembly gerado pelo cross-compiler fosse rodado no MARS:

- É necessário retirar todas as diretivas para o montador que não sejam reconhecidas pelo MARS. O tedioso processo é feito apertando o botão *assembly* do MARS e comentando uma a uma as diretivas não reconhecidas informadas na janela de log do programa.
- j $31 final não funciona, o programa monta, mas na execução não fecha de forma limpa. É necessário retirar essa linha e usar um syscall ($v0=10) para fechar o programa de forma limpa.
- Trocar o .rdata por .data

- Apagar os “j putchar” e “j printf” que o mars não reconhece
- Substituir estruturas do tipo

```
lui $at,%hi(LABEL)
addiu $T0,$AT,%lo(LABEL)
```

  por:

```
la $t0, label
```

- Adicionar a diretiva .data se durante a otimização essa diretiva for apagada e houver uma string/caractere a ser escrito na tela.
- Adicionar a diretiva .text se alguma otimização retirá-la.
- É necessário prestar atenção sempre que houver alguma instrução após um jump para trocar o jump com essa instrução logo abaixo dela, para que essa instrução seja executada.
- Em uma das estruturas para o sort, é necessário prestar atenção quando o sort dá problema, pois possivelmente é na ordem de duas instruções. Um caso relativamente grave que ocorreu foi a inversão de um comando de jump com um comando de aumento do contador que resultou em um contador que só conta quando faz uma troca.
- Não usar printf (“%d “) ou coisas do tipo, separar em imprimir o numero e depois em um comando separado imprimir o texto, pois assim quando o compilador passar para C ele vai separar em printf + putchar e fica mais fácil de ajudar o mars.
- Ao fazer um scanf usando o syscall, é necessário salvar na pilha após pegar o valor.
- Às vezes o slt tenta usar um número diretamente no lugar do terceiro argumento, quando isso acontece basta trocar por slti.

A.2.b

A tabela na imagem a seguir mostra os resultados esperados dos diferentes parâmetros de compilação

gcc -O option flag

Set the compiler's optimization level.

option	optimization level	execution time	code size	memory usage	compile time
-O0	optimization for compilation time (default)	+	+	-	-
-O1 or -O	optimization for code size and execution time	-	-	+	+
-O2	optimization more for code size and execution time	--		+	++
-O3	optimization more for code size and execution time	---		+	+++
-Os	optimization for code size		--		++
-Ofast	O3 with fast none accurate math calculations	---		+	+++

+increase ++increase more +++increase even more -reduce --reduce more ---reduce even more

Figura 2: otimizações do gcc

Das informações da tabela temos acesso apenas ao tamanho do código e da quantidade de instruções, que permite estimar o tempo de execução. As informações estão na tabela a seguir. Vemos que as diferenças devem estar no tempo de compilação, uso de memória, tempo de execução e no tamanho do código produzido.

	Escrito em Assembly	C, -O0	C, -O1	C, -O2	C,-O3
Nº de instruções	802	2199	712	696	*545
Tamanho	1.19 KB	2.94 KB	1.95 KB	1.89 KB	3.41 KB

Tabela 4

Até onde podemos ver, os resultados concordam com a tabela. O único resultado não previsto pela tabela é que a otimização O3 criou um programa grande em tamanho ocupado na memória.

A.2.c

O código iterativo é mostrado a seguir:

```
.data
str1:   .asciiz "Fibonacci("
str2:   .asciiz ")= "

.text
.globl __start
__start:

main:
        #Recebendo o valor n do usuario e colocando no reg S1
        li $v0, 5
        syscall
        move $s1, $v0
```

```
        #Chamando a subrotina que recebe a0=n e retorna v0 = Fib(n)
        move $a0, $s1
        jal Fib

        #Chamando a subrotina que recebe a0=n,a1=Fib(n) e imprime na tela
        la $a0,($s1)   #passando o n
        la $a1,($v0)   #passando o fib(n)
        jal show

        #Retornando para o S.O.
        li $v0,10
        syscall

#Para a subrotina Fib, a0=n e eu quero retornar v0=Fib(n)
Fib:
        move $t0, $a0
        #se n<0, va para o caso s1 onde retornamos Fib(n)=0
        bltz $t0, Fib_s2

        #Fib(0) = Fib(1) = 1
        beq $t0, 0, Fib_s3
        beq $t0, 1, Fib_s3

        #Fib(n>2), #t0=n, v0=Fib(i), t1=Fib(i-1), t2=Fib(i-2), t3=i
        li $v0, 2
        li $t1, 1
        li $t2, 1
        li $t3, 2

Fib_L1: beq $t3, $t0, Fib_s1 #while (i!=n)
        move $t2, $t1  #Passando o fibonacci(i-1) para fibonacci(i-2)
        move $t1, $v0  #Passando o fibonacci(i) para fibonacci(i-1)
        add  $v0, $t1, $t2     # Calculando o novofib(i)
        addi $t3, $t3, 1       #i++
        j Fib_L1

Fib_s1: jr $ra

Fib_s2: move $v0, $zero
        jr $ra

Fib_s3: li $v0, 1
        jr $ra


#Para a subrotina show, vai chegar A0 o numero n e A1 o Fib(n)
show:  move $t0,$a0          #tenho que liberar o A0 o syscall e por isso estou
     salvando em t0 = n e A1 = Fib(n)
        move $t1,$a1

        #Imprimindo o primeiro pedaco da string
        li $v0, 4
        la $a0, str1
        syscall

        #Imprimindo o n
        li $v0, 1
        la $a0, ($t0)
        syscall

        #Imprimindo o segundo pedaco da string
```

```
        li $v0, 4
        la $a0, str2
        syscall

        #Imprimindo o fib(n)
        li $v0, 1
        la $a0, ($t1)
        syscall

        jr $ra
```

O código para calcular Fibonacci recursivamente é apresentado a seguir:

```
        .data
str1:   .asciiz "Fibonacci("
str2:   .asciiz ")= "

.text
.globl __start
__start:

main:
        #Recebendo o valor n do usuario e colocando no reg S1
        li $v0, 5
        syscall
        move $s1, $v0

        #Chamando a subrotina que recebe a0=n e retorna v0 = Fib(n)
        move $a0, $s1
        jal Fib

        #Chamando a subrotina que recebe a0=n,a1=Fib(n) e imprime na tela
        la $a0,($s1)   #passando o n
        la $a1,($v0)   #passando o fib(n)
        jal show

        #Retornando para o S.O.
        li $v0,10
        syscall


#Para a subrotina Fib, a0=n e eu quero retornar v0=Fib(n)
Fib:
        slti $t0, $a0, 2
        beq $t0, $zero, cont  # case 1: n>2
        bltz $a0, Fib_s1      # case 2: n<0
        j Fib_s2              # case 3: n={0,1}

Fib_s1: li $v0, 0             #if n<0      return 0
        jr $ra

Fib_s2: addi $v0, $zero, 1    # if n={0, 1} return 1
        jr $ra

cont: # empilhando
        addi $sp, $sp, -16          # salvar espaco na pilha para 4 coisas: ra, n
            fib (n-1) e fib (n-2)
        sw $ra, 0($sp)
        sw $a0, 4($sp)
```

```
        addi $a0, $a0, -1           # calcular fib(n - 1) e salva na pilha
        jal Fib
        sw $v0, 8($sp)

        lw $a0, 4($sp)              # calcular fib(n - 2) e salva na pilha
        addi $a0, $a0, -2
        jal Fib
        sw $v0, 12($sp)

        #desempilhando
        lw $ra, 0($sp)
        lw $t0, 8($sp)
        lw $t1, 12($sp)
        addi $sp, $sp, 16

        add $v0, $t0, $t1           # fib(n - 1) + fib(n - 2)

        jr $ra


#Para a subrotina show, vai chegar A0 o numero n e A1 o Fib(n)
show:  move $t0,$a0          #tenho que liberar o A0 o syscall e por isso estou
     salvando em t0 = n e A1 = Fib(n)
        move $t1,$a1

        #Imprimindo o primeiro pedaco da string
        li $v0, 4
        la $a0, str1
        syscall

        #Imprimindo o n
        li $v0, 1
        la $a0, ($t0)
        syscall

        #Imprimindo o segundo pedaco da string
        li $v0, 4
        la $a0, str2
        syscall

        #Imprimindo o fib(n)
        li $v0, 1
        la $a0, ($t1)
        syscall

        jr $ra
```

A.2.d

O código iterativo em C é mostrado a seguir:

```
#include <stdlib.h>
#include <stdio.h>

int main (){
        int v1, v2, n;
        int aux, i;

        scanf ("%d", &n);
```

```
        if (n<0){
                v1=0;
                goto saida;
        }

        if (n < 2){
                v1=1;
                goto saida;
        }

        for (i=1, v1=1, v2=1; i<n; i++) {
                aux=v1;
                v1 += v2;
                v2=aux;
        }

        saida:
        printf ("Fibonacci(");
        printf ("%d", n);
        printf (") = ");
        printf ("%d", v1);
        return 0;}
}
```

O código recursivo em C é mostrado a seguir:

```
#include <stdlib.h>
#include <stdio.h>

int Fib (int n){

        if (n<0)        return 0;
        if (n<2)        return 1;

        return Fib(n-1)+ Fib (n-2);
}

int main (){

        int n, v1;

        scanf("%d", &n);

        v1=Fib(n);

        printf ("Fibonacci(");
        printf ("%d", n);
        printf (") = ");
        printf ("%d", v1);
        return 0;
}
```

A.2.e

Os códigos da etapa foram cross-compilados para assembly mips com as diretivas requeridas. Temos então 10 códigos em assembly:

- 2 códigos escritos diretamente por nós, 1 recursivo e outro iterativo
- 4 códigos produzidos com o cross-compiler para fibonacci iterativo
- 4 códigos produzidos com o cross-compiler para fibonacci recursivo

Os códigos produzidos pelo coss-compiler foram debugados para serem rodados no MARS. Não consegui-se, no entanto, debugar o código recursivo gerado com -O3. O código foi debugado o suficiente para rodar no MARS, mas a resposta calculada não condiz com a esperada(o código calcula alguma coisa e termina sem erro, mas a sequência não corresponde a de Fibonacci). Todos os outros foram debugados e funcionam. Os códigos foram então testados para n $\in \{1, 5, 10, 15, 20\}$ e obteve-se o gráfico a seguir:

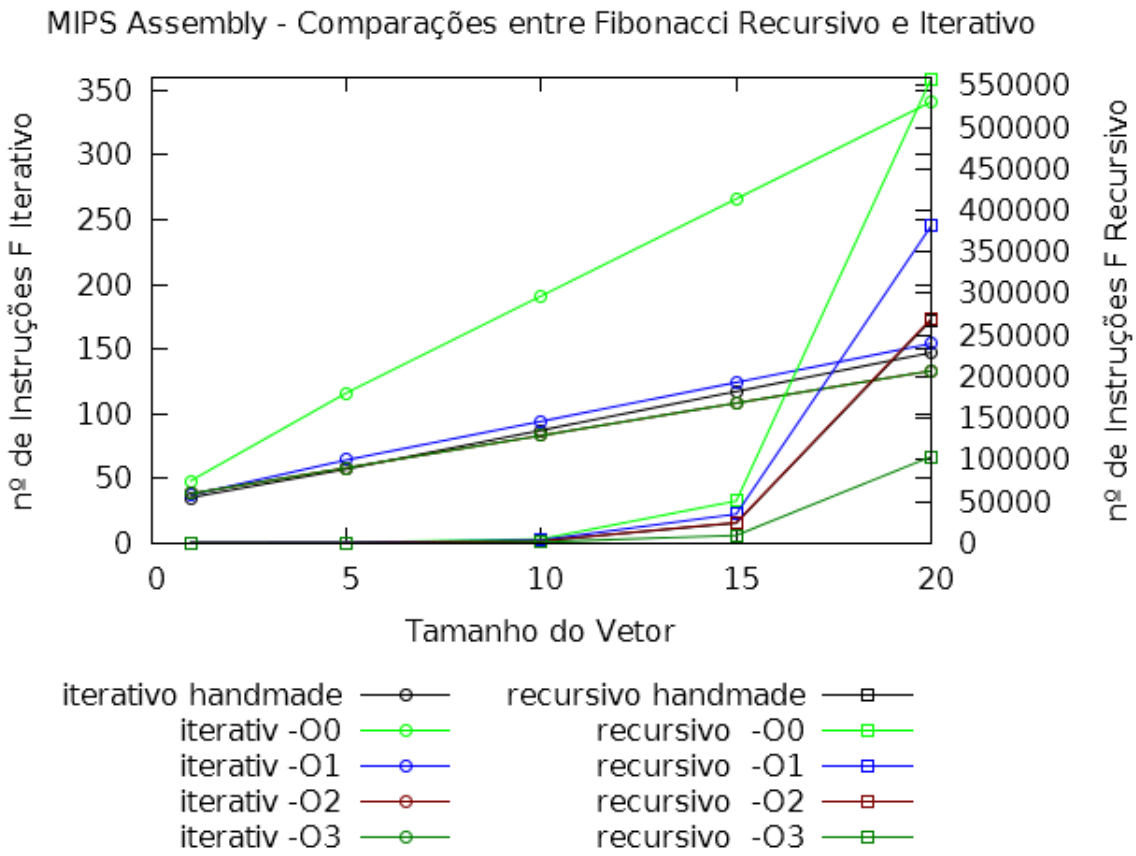

Figura 3: Comparação dos algoritmos desenvolvidos

Note a diferença na escala dos eixos. Observa-se que a escala de contagem de iterações para o algoritmo recursivo é muito maior que a do algoritmo

iterativo. Damos um zoom na escala para avaliar o comportamento para os menores n:

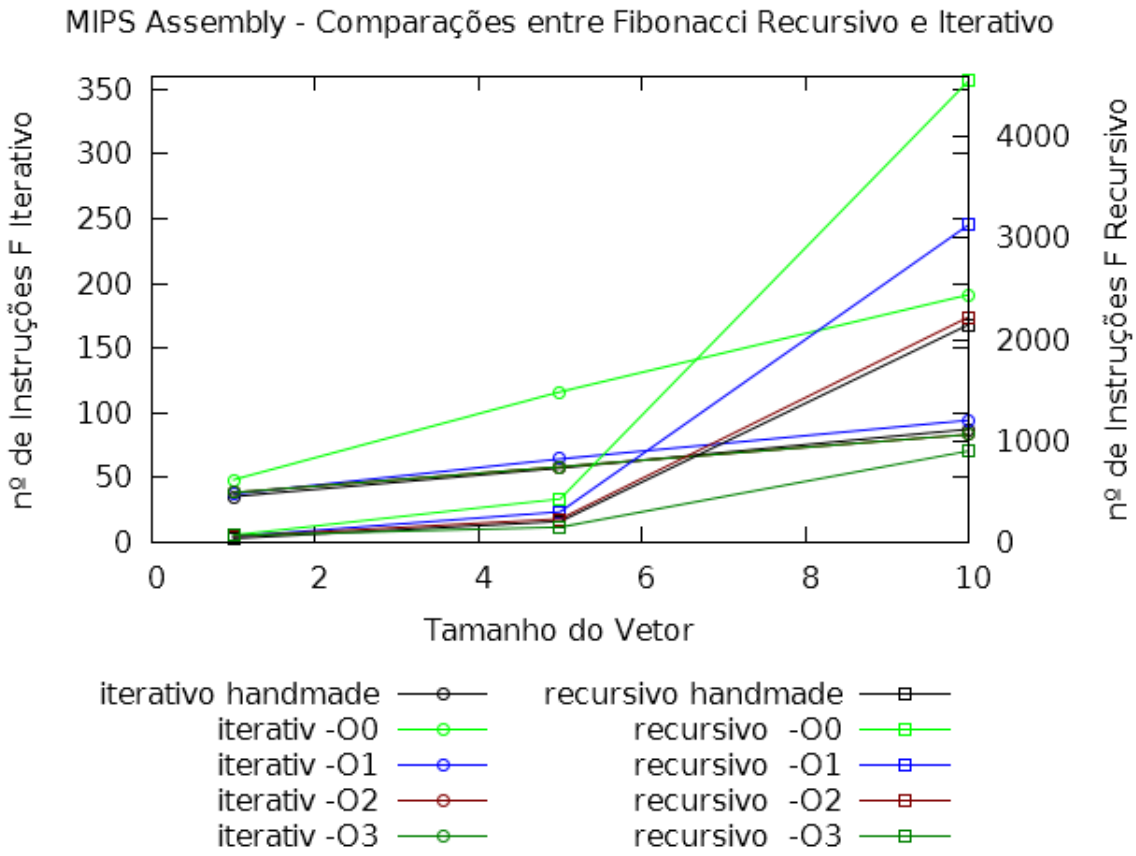

Figura 4: Comparação dos algoritmos desenvolvidos(zoom)

Notamos que o comportamento do algoritmo iterativo parece linear, e o recursivo deve ser parabólico, se não exponencial. Por fim, notamos que a melhor otimização é aquela feita com a diretiva -O3. No entanto, não conseguimos fazer a otimização -O3 funcionar e não podemos escolher esta como a melhor. Para prosseguirmos, escolheríamos a melhor otimização que funciona: -O2. No entanto, para os dois algoritmos essa otimização foi somente um pouco melhor que o código escrito a mão. Por simplicidade, escolhemos o código feito a mão como aquele com melhor custo/benefício em relação ao trabalho que dá fazer ele funcionar e o resultado obtido.

A.2.f

A.2.g

B Verilog

B.1 Implementação de Driver de 7 Segmentos

O teste da placa pode ser visto em: `http://www.youtube.com/watch?v=zjfapP_vkDw`

B.2 Implementação de um Somador de 4 bits

B.2.1 Desenho Esquemático

(a) Os blocos desenvolvidos são mostrados a seguir:

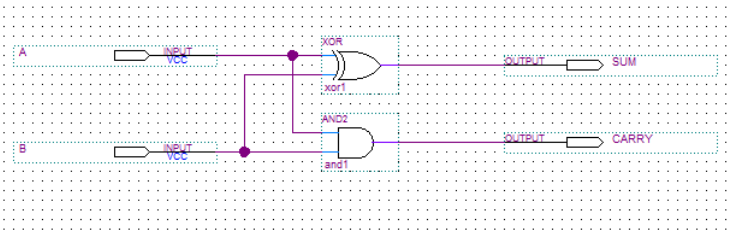

Figura 5: Meio Somador

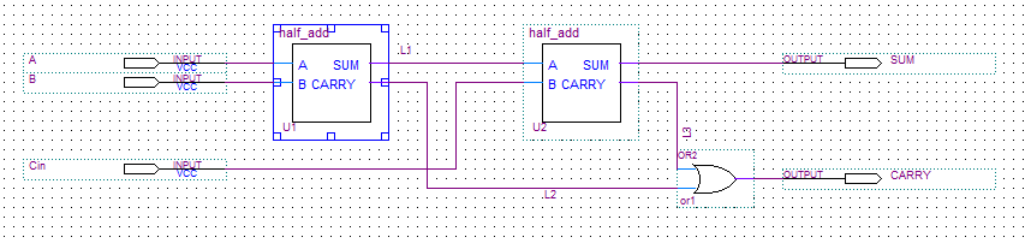

Figura 6: Somador Completo

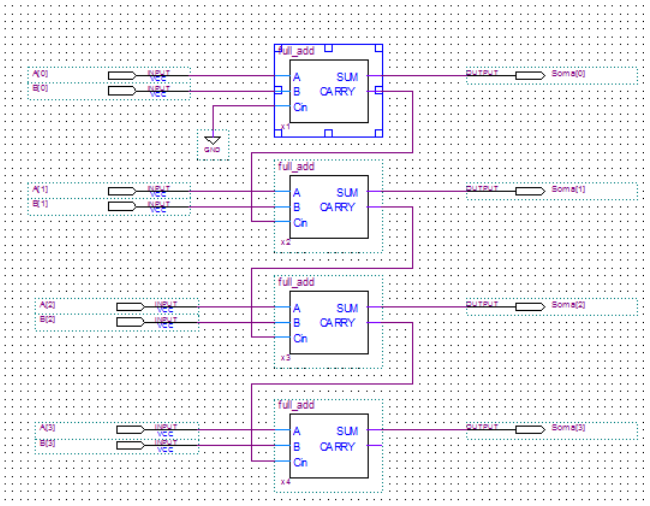

Figura 7: Somador de 4 bits

(b) O resultado da simulação temporal com T = 10ns é mostrado a seguir:

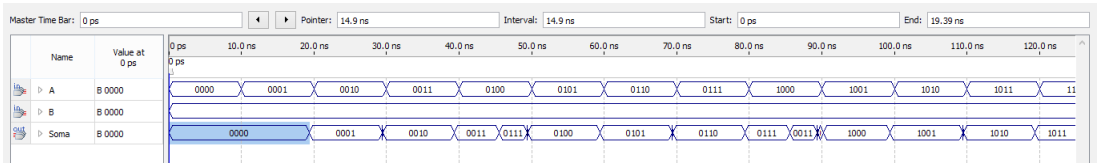

Figura 8: Somador de 4 bits

Esta simulação ficou mais condizente com os resultado dos tempos dos caminhos completos feitos por cada implementação, este tinha o tempo de atraso de 9,39ns, que foi o maior tanto na simulação temporal quanto no maior tempo de atraso de propagação.

O resultado da simulação funcional é mostrado a seguir. O resultado é o mesmo para esse e os dois itens subsequentes, por isso será mostrado apenas uma vez.

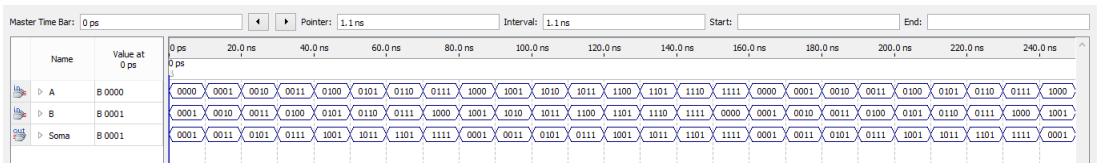

Figura 9: Simulação Funcional

(c) O link para o vídeo no youtube mostrando o resultado pode ser visto em:

`http://www.youtube.com/watch?v=ZspIA5N1yYg`

B.2.2 Nível de Portas Lógicas

(a) Os blocos desenvolvidos são mostrados a seguir;

- meio somador

```
/* Meio somador */

module half_addv (sum,carry,A,B); //Define entradas e saidas do modulo

output sum, carry; //Define as saidas

input A,B; //Define as entradas

xor(sum,A,B); //comportamento da saida sum de acordo com as entradas

and(carry,A,B); //comportamento da saida carry de acordo com as entradas

endmodule
```

- somador completo

```
/* Somador completo */

module full_addv (sum,carry,A,B,Cin); //Define entrada e saida do modulo

output sum, carry; // Define as saidas

input A, B, Cin; // Define as entradas

wire L1,L2,L3; // Fios que ligam dois meio somadores

half_addv(L1,L2,A,B); // Primeiro meio somador

half_addv(sum,L3,L1,Cin); // Segundo meio somador

or(carry,L3,L2); // Porta OU

endmodule
```

- somador 4bits

```
/* Somador completo de 4 bits */

module add4v (soma,A,B); //Define as entradas e saidas do somador

output [3:0]soma; // Vetor de saida dos somadores

input A[3:0], B[3:0]; // Vetor de entrada dos 4 bits para cada entrada

wire s1,s2,s3; // Fios para passar o carry de um somador para outro

full_addv(soma[0],s1,A[0],B[0],0); // Primeiro somador da cascata
```

```
full_addv(soma[1],s2,A[1],B[1],s1); // Segundo somador

full_addv(soma[2],s3,A[2],B[2],s2); // Terceiro somador

full_addv(soma[3],X,A[3],B[3],s3); // Quarto somador

endmodule
```

(b) O resultado da simulação temporal com T = 10ns é mostrado a seguir:

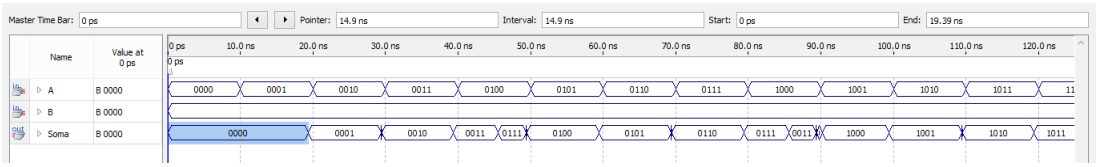

Figura 10: Somador de 4 bits

Nessa simulação também houve o mesmo problema que esperava-se, o da propagação de atrasos, mas mesmo assim ainda nota-se que o atraso da saída que foi de 5,24ns, o que não foi esperado pelo fato de que a implementação comportamental, tinha o tempo de execução menor que o de portas, e agora a de portas mostra-se com um tempo te atraso de propagação menor, conclui-se então que uma implementação que tenha um caminho com maior tempo de atraso, não necessariamente terá o maior atraso de propagação.

O resultado da simulação funcional é o mesmo do item a.

(c) O link para o vídeo no youtube mostrando o resultado pode ser visto em:

`http://www.youtube.com/watch?v=ZspIA5N1yYg` [1]

B.2.3 Descrição Comportamental

(a) O bloco desenvolvido é mostrado a seguir;

```
/* Somador completo de 4 bits */

module add4cv (A, B, Cin, S, Cout);
       input [3:0] A, B;

       input Cin;

       output[3:0]S;
```

[1] O link é o mesmo. Já que os resultados obtidos foram os mesmos, filmou-se apenas uma vez

```
        output Cout;

        assign {Cout, S} = A + B + Cin;

endmodule
```

(b) O resultado da simulação temporal com T = 10ns é mostrado a seguir:

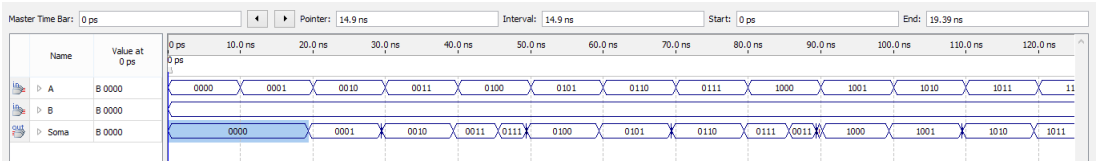

Figura 11: Somador de 4 bits

O quadro acima mostra o resultado da soma dos valores de A com os valores de B, mostrando ainda os atrasos, o atraso da primeira mudança de chave foi 6,28ns, mas conforme a propagação dos atrasos vai aumentando começam a aparecer alguns valores que imprevisíveis por momentos muito pequenos, mas ainda assim o valor correto aparece por mais tempo no estudo analítico da simulação temporal do esquemático comportamental, então fica um tipo de ciclo que os valores aparecem por mais e menos tempo.

O resultado da simulação funcional é o mesmo do item a.

(c) O link para o vídeo no youtube mostrando o resultado pode ser visto em:

`http://www.youtube.com/watch?v=ZspIA5N1yYg` [2]

B.2.4

(a) O número de elementos lógicos utilizados pelas três formas de implementação é igual, pelo fato de que o Quartus interpreta as três de um jeito semelhante.

(b) Para a implementação esquemática o maior TPD foi para o bit menos significativo da entrada A para o bit mais significativo da soma:

$$A[0] \rightarrow Soma[3] = 11.020ns \tag{3}$$

[2] O link é o mesmo. Já que os resultados obtidos foram os mesmos, filmou-se apenas uma vez

Para a implementação a nível de portas lógicas o maior TPD foi para o segundo bit menos significativo da entrada A para o bit mais significativo da soma:

$$A[1] \rightarrow Soma[3] = 10.756ns \tag{4}$$

Para a implementação a nível comportamental o maior TPD foi do Carry in para o Carry out:

$$Cin \rightarrow Cout = 10.679ns \tag{5}$$

Apesar de termos o mesmo resultado ao fim das três implementações, podemos perceber que a nível comportamental temos uma otimização de desempenho, pois o maior TPD foi o menor dentre os três.

Referências

[1] Patterson, D.A.; Hennessy, J.L. *Computer Organization and Design* 5th ed. Elsevier, 2014.