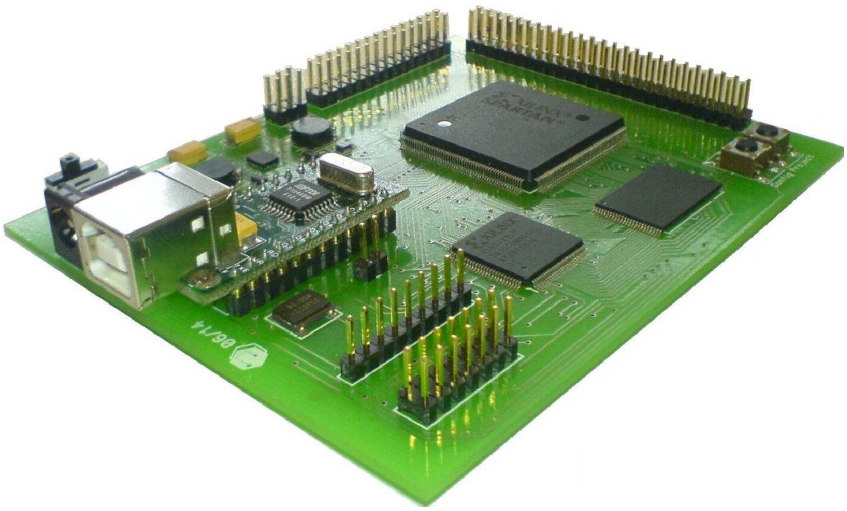
Universidade de Aveiro

Sistemas Digitais Reconfiguráveis (2006/2007)



Relatório de Projecto

Maior Divisor Comum - Algoritmos Iterativos/Recursivos

29 de Dezembro de 2006

Hugo Miguel Leonardo Proença nº 27688

1 Objectivos

O principal objectivo deste trabalho criar um circuito que permite calcular o maior divisor comum utilizando algoritmos iterativos e recursivo, baseados numa máquina de estados finitos hierárquica.

2 Tópicos/Abordagem teórica

O maior divisor comum (*greatest common divisor* - *GCD*) de dois números inteiros, A e B, é o maior inteiro (resultado) que a sua divisão pelos dois números têm resto 0. Apesar do objectivo principal deste trabalho, comparar algoritmos iterativos e recursivos, existem diferentes métodos para calcular o *GCD*, o que levou a que neste trabalho se comparar-se, não só algoritmos iterativos e recursivos, mas também diferentes métodos de cálculo. Assim, foram implementados dois métodos de cálculo, um baseado no algoritmo de *Euclid* e outro no algoritmo de *Dijkstra*. De seguida, apresenta-se os algoritmos recursivos dos dois métodos:

Algoritmo de *Euclid*

$$\text{Para a} \geq b > 0,\ \text{GCD(a,b)} = \begin{cases} a & \text{se b} \leq 0 \\ GCD(b,a) & \text{se } n > m \\ GCD(b, a\%b) & \text{para os restantes casos} \end{cases}$$

Algoritmo de *Dijkstra*

$$\text{Para a} \geq b > 0,\ \text{GCD(a,b)} = \begin{cases} a & \text{se a} = \text{b} \\ GCD(a-b,b) & \text{se } a > b \\ GCD(a,b-a) & \text{se } b < a \end{cases}$$

Os algoritmos iterativos de cada método são bastantes parecidos com o funcionamento dos algoritmos recursivos, visto que, para o caso da função recursiva, quando existe um retorno da última chamada à função, esse será o valor de retorno da primeira chamada à função, o que leva a um simplicidade a nível de implementação, visto que, o valor de retorno é sempre o mesmo. A implementação de uma máquina de estados finitos hierárquica torna-se simples, pois, sabendo quais são quais as condições a verificar e o módulo e estado pretendido, serão geradas as saídas correspondentes.

3 Implementação

A implementação deste projecto foi orientada para a sua utilização na placa DETIUA-S3, usada nas aulas práticas de *SDR*. Contudo, quase todos os módulos construídos para este projecto foram construídos em *VHDL*, apenas com exepecção do componente que efectua a divisão (*core_divider*), o que permite uma fácil adaptação deste projecto para outro tipo de plataforma.
Para calcular o *GCD* foi criado um componente, denominado *GCD_Divider* para o caso do algoritmo de *Euclid*, ou *GCD_Subtract* para o caso do algoritmo de *Dijkstra* , que, além de calcular o maior divisor comum entre os operandos *a* e *b*, permite fornecer informação sobre o estado do resultado (válido ou não), e também se ocorreu algum erro (erro possível na *HFSM*). Permite também selecionar o tipo de algoritmo, iterativo ou recursivo. É neste componente que é feita a interligação entre as operações matemáticas com a máquina de estados finitos hierárquica, permitindo assim o cálculo do *GCD*. A figura 1 representa os componentes de topo utilizados neste componente (o componente *Divider* não é utilizado no caso do *GCD_Subtract*).

3.1 *RegN* - registo parametrizavel

O componente *RegN* não é mais do que um registo em que o seu tamanho é genérico, possuído um sinal de *reset* e um sinal de activação (*clkEnb*).

3.2 *Divider* - divisão

O algoritmo de *Euclid* envolve a execução da operação divisão, operação esta que na ferrameta *ISE* da *Xilinx* não é directamente sintetizavél. Então, para efectuar a divisão entre dois números foi utilizado um componente *IP Core*, o *core_divider*, que calcula o quociente e resto da divisão entre um dividendo e um divisor. Para a utilização deste componente, o tamanho dos operados têm que ser fixo, sendo que, para este projecto, foi escolhido

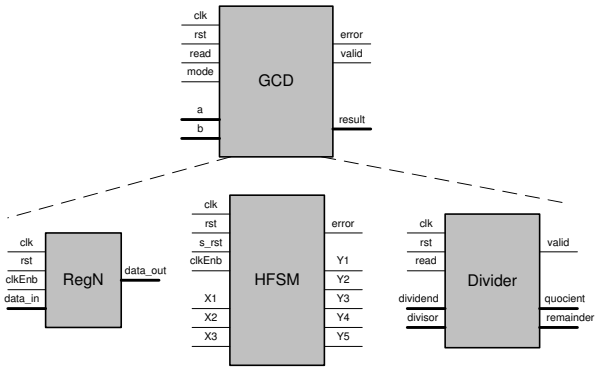


Figura 1: Componentes utilizados para a construção do *GCD*

um tamanho de 32 bits. Este componente introduz um atraso máximo(informação retirada do *datasheet* do componente), para a configuração utilizada, de:

$$\text{Atraso maximo} = \text{Tamanho dos operandos} + 2 = 34 \text{ (ciclos de relógio)}$$

O componente *Divider* foi criado para facilitar a utilização do componente *core_divider*, de modo que, este componente fornece a informação da validade da operação, ou seja, quando se pretende realizar uma divisão deve-se esperar pelo sinal *valid* após a deactivação do sinal *read*. Este componente introduz um atraso fixo de 34 ciclos de relógio por divisão, e o tamanho dos operandos pode variar de 1 até 32 bits.

3.3 *HFSM* - Máquina de estados finitos hierárquica

Para implementar a *HFSM* seguiu-se o modelo fornecido pela referência [1]XXXYYYYYXXXXX, em que as condições de entrada e as saídas da máquina estão representadas na figura 2.

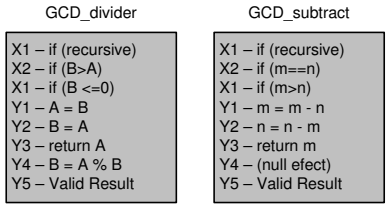


Figura 2: Condições da *HFSM*

A máquina implementa possuí 3 módulos, em que o módulo *Z0* representa a escolha do tipo de algoritmo, o módulo *Z1* representa a execução do algoritmo recursivo, e o módulo *Z2* representa a execução do algoritmo iterativo. Nas figuras 3, 4 e 5 estão representados o funcionamento de cada módulo, para os dois métodos utilizados.

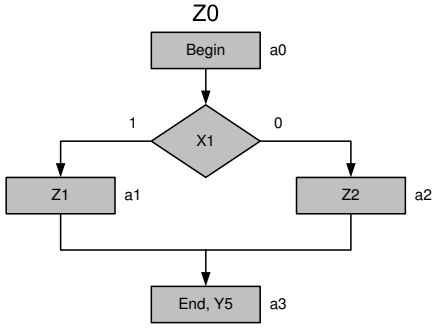


Figura 3: *HFSM* - Módulo Z0

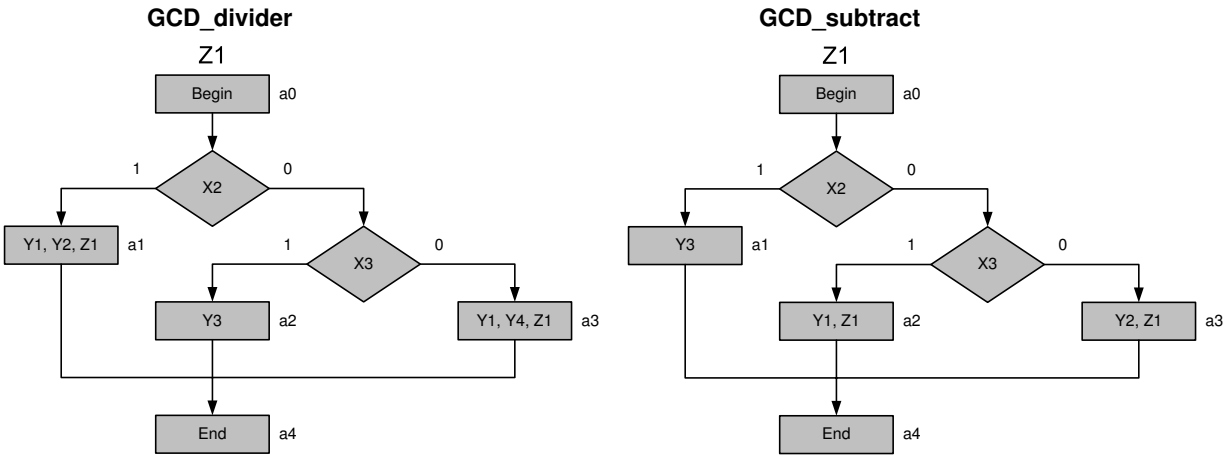


Figura 4: *HFSM* - Módulo Z1

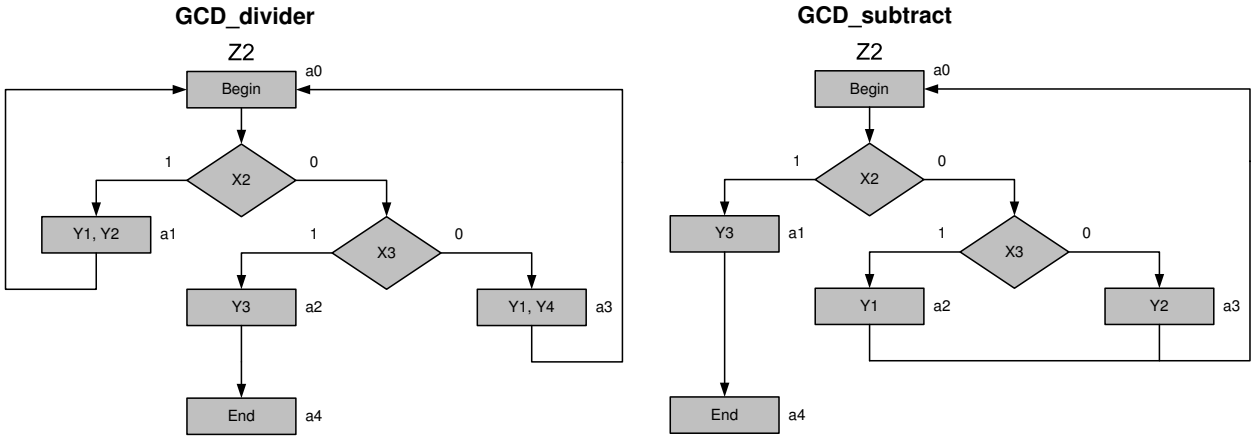


Figura 5: *HFSM* - Módulo Z2

3.4 *GCD* - Interligação

O cálculo do *GCD* através de *HFSM* envolve várias fases/etapas que necessitam de ser realizadas sequencialmente. Deste modo, foi considerado 3 diferentes níveis de actuação:

- Colocação dos valores iniciais no sistema;
- Comutação dos valores dos operadores;
- Cálculo dos novos valores dos operadores e das condições da *HFSM*.

Para melhor descrever estes níveis, na figura 6 etá representado um digrama dos resgistos envolvidos em cada nível
Posteriormente, o último nível foi divido em dois, para uma maior facilidade na verificação do correcto funcionamento.

3.5 Implementação - Simulação

Resumido o que foi acima referido, o componente *GCD* efectua a interacção das diferentes operações necessárias. Esta interacção é controlada por uma máquina de estados, representada na figura 7, em que cada estado controla um nível de interacção difirente.
Para testar os vários componentes, e para obter resultados para uma futura comparação, foram criadas várias *Test Benchs*, que permite efectuar a simulação de cada componente.

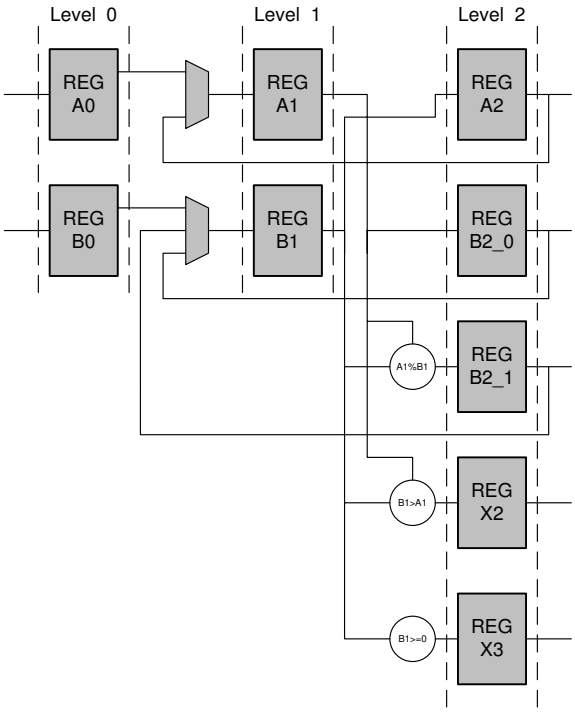


Figura 6: Representação dos níveis de actuação

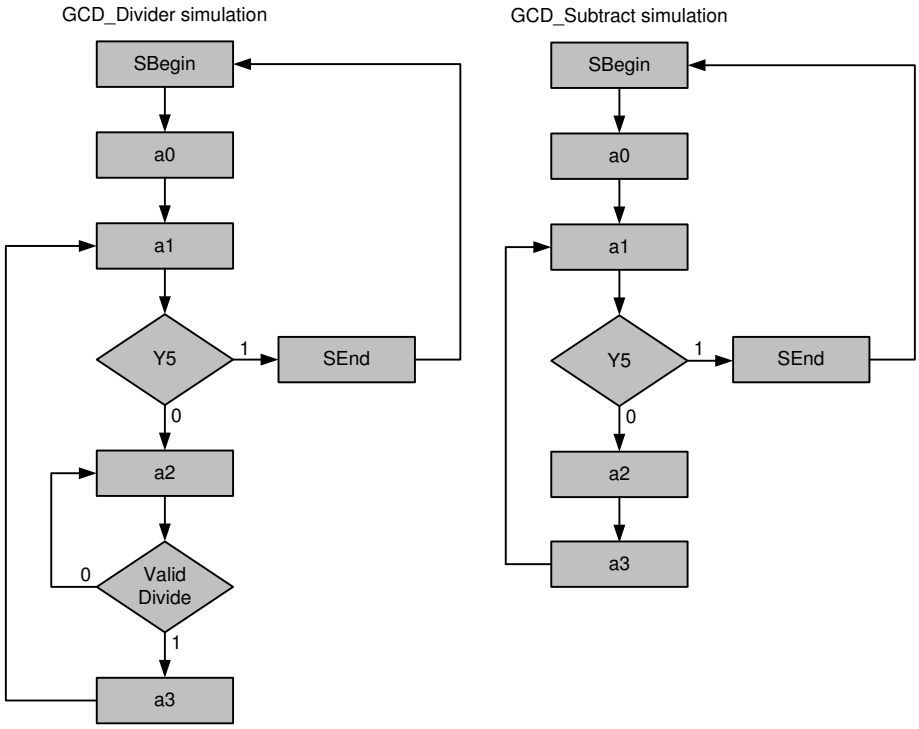


Figura 7: Máquina de estados - simulação

4 Implementação - *Placa DETIUA-S3*

A implementação do componente *GCD* na placa *DETIUA-S3* não deve ser feita directamente, pois, deste modo não é possível obter qualquer de resultados. De modo a verificar o funcionamento de componente na placa, construiu-se um circuito que permite visualizar num monitor VGA o cálculo do maior divisor comum de dois operandos, operandos estes que podem ser introduzidos pelo teclado. Este circuito está limiatado a utilização

de operandos de 8 bits.
Após uma primeira tentativa de implementação deste circuito, verificou-se que este não funcionava se fosse implementado com uma máquina de estados representada na figura 7. Deste modo, foi construída uma nova máquina de estados (figura 8), que divide cada estado da máquina anterior em vários de modo a garantir o correcto funcionamento do circuito.

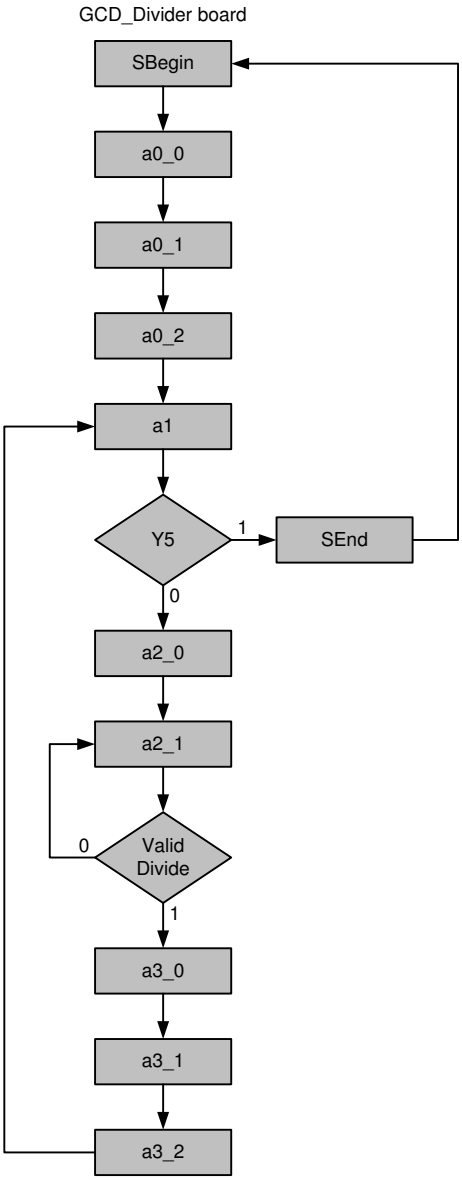


Figura 8: Máquina de estados - Placa *DETIUA-S3*

5 Apresentação e discussão de resultados

Os testes efectuados aos vários componentes tiveram com principais objectivos, além verificar o correcto funcionamento deste, a comparação entre os algoritmos recursivos e iterativos, e a comparação entre os dois métodos de cálculo implementados. Estes testes foram efectuados no simulador do programa *ISE* da *Xilinx*, tendo como base *Test Benchs* realizadas para este fim. Os testes efectuados a nível funcional foram bem sucedidos. Para melhor analisar os resultados, as unidades utilizadas foram ciclos de relógio (*ClockCycles*), para a execução temporal, e o número de ciclos de execução dos níveis (*LevelCycles*), ou seja, o número de vezes que os 3 (ou 4) diferentes níveis de interacção são executados.

5.1 Algoritmo Recursivo VS Iterativo

Para comparar os algoritmos recursivos e iterativos contruíu-se uma tabela (tabela 1) que representa o número de *ClockCycles* em função do número de *LevelCycles* .

	GCD_Divider			*GCD_Subtract*	
LevelCycles	*Recursive ClockCycles*	*Iterative ClockCycles*		*Recursive ClockCycles*	*Iterative ClockCycles*
1	226	226		22	22
2	374	300		34	28
3	522	374		46	34
4	670	448		58	40
5	818	522		70	46
...	...	...		...	...

Tabela 1: Representação *ClockCycles* em função de *LevelCycles*

Pela análise da tabela 1, verifica-se que existe uma relação linear entre os *ClockCycles* à medida que o número de *LevelCycles* aumenta. Verifica-se assim as seguintes relações:

GCD_Divider

$$\text{Recursive} \qquad ClockCycles = 226 + 2.(LevelCycles - 1).74$$

$$\text{Iterative} \qquad ClockCycles = 226 + (LevelCycles - 1).74$$

GCD_Subtract

$$\text{Recursive} \qquad ClockCycles = 22 + 2.(LevelCycles - 1).6$$

$$\text{Iterative} \qquad ClockCycles = 22 + (LevelCycles - 1).6$$

Calculada a regressão linear de cada modo, torna-se fácil de representar graficamente os 2 modos. Assim, no gráfico da figura 9, está representada a comparação entre os dois tipos de algoritmos, recursivo e iterativo.

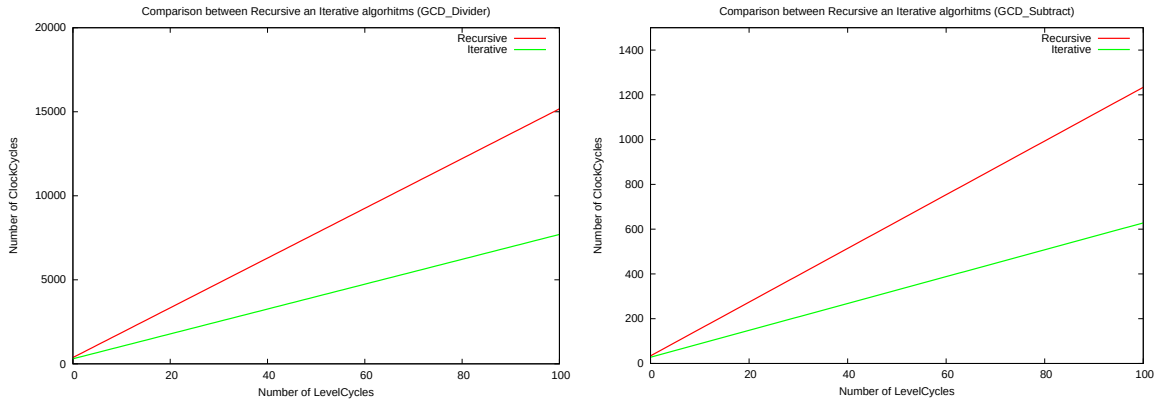


Figura 9: Comparação entre algoritmos recursivos e iterativos

Tanto pelas expressões anteriores com pelo gráfico da figura 9, verifica-se que para este problema (cálculo do *GCD*) é muito mais adequado a utilização de um algoritmo iterativo. Para valores elevados de *LevelCycles*, o algoritmo iterativo é duas vezes mais rápido que o algoritmo recursivo.

5.2 Algoritmo *Euclid* VS *Dijkstra*

De modo a comparar os algoritmos de *Euclid* e de *Dijkstra*, foi construída uma tabela, tabela 2, que mostra alguns exemplos de cálculo do *GCD* para diferentes valores dos operados. Os testes foram realizados utilizando os algoritmos recursivos de cada método.

				GCD_Divider			*GCD_Subtract*	
a	*b*	*result*		*LevelCycles*	*ClockCycles*		*LevelCycles*	*ClockCycles*
128	28	4		5	818		9	118
28	128	4		6	966		9	118
30	3	3		2	374		10	130
3	30	3		3	522		10	130
127	9	1		3	522		23	286
127	90	1		6	966		13	166
2700	92	4		5	818		39	478
2700	9	9		2	374		300	3610
2700	90000	900		4	670		36	442

Tabela 2: Comparação entre o algoritmo de *Euclid* VS *Dijkstra*

Se a única maneira de comparar os dois métodos fosse efectuar um comparação a nível dos *ClockCycles* em função dos *LevelCycles*, o algoritmo de *Dijkstra* certamente seria vencedor. Contudo, a comparação entre os dois algoritmos deve ter em conta o objectivo final, e aí, o algoritmo de *Euclid* consegue realiza-lo em muito menos *LevelCycles*, que se pode dizer que, a este nível, este algoritmo é mais eficiente, mas, cada *LevelCycle* demorar muito mais tempo que o algoritmo de *Dijkstra*, o que leva em que haja casos em que este algoritmo seja mais eficiente. Pode-se dizer que, quando a diferença entre os operandos é relativamente grande, o algoritmo de *Euclid* é mais adequado, verificando-se o contrário quando a diferença não é grande.

5.3 Relatório de síntese

Apresenta-se na figura 10, o sumário apresentado pelo programa *ISE* da *Xilinx*, para o circuito implementado na placa *DETIUA-S3*.

GCD_PLACA-DETIUA-S3 Project Status

Project File:	GCD_Placa-DETIUA-S3.ise	**Current State:**	Programming File Generated
Module Name:	GCD_top	• **Errors:**	No Errors
Target Device:	xc3s400-4pq208	• **Warnings:**	82 Warnings (74 new)
Product Version:	ISE 8.2.02i	• **Updated:**	sáb 30. Dez 21:30:50 2006

GCD_PLACA-DETIUA-S3 Partition Summary

No partition information was found.

Device Utilization Summary

Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	1,827	7,168	25%	
Number used as Flip Flops	1,810			
Number used as Latches	17			
Number of 4 input LUTs	2,473	7,168	34%	
Logic Distribution				
Number of occupied Slices	1,726	3,584	48%	
Number of Slices containing *only* related logic	1,726	1,726	100%	
Number of Slices containing unrelated logic	0	1,726	0%	
Total Number 4 input LUTs	2,609	7,168	36%	
Number used as logic	2,473			
Number used as a route-thru	90			
Number used as Shift registers	46			
Number of bonded IOBs	11	141	7%	
IOB Flip Flops	3			
Number of Block RAMs	1	16	6%	
Number of MULT18X18s	1	16	6%	
Number of GCLKs	4	8	50%	
Number of DCMs	1	4	25%	
Total equivalent gate count for design	116,865			
Additional JTAG gate count for IOBs	528			

Performance Summary

Final Timing Score:	0	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints Met		

Detailed Reports

Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	sáb 30. Dez 21:25:24 2006	0	66 Warnings (66 new)	83 Infos (83 new)
Translation Report	Current	sáb 30. Dez 21:25:36 2006	0	0	1 Info (1 new)
Map Report	Current	sáb 30. Dez 21:25:52 2006	0	10 Warnings (6 new)	4 Infos (3 new)
Place and Route Report	Current	sáb 30. Dez 21:26:48 2006	0	2 Warnings (2 new)	1 Info (1 new)
Static Timing Report	Current	sáb 30. Dez 21:26:58 2006	0	0	1 Info (1 new)
Bitgen Report	Current	sáb 30. Dez 21:30:50 2006	0	4 Warnings (0 new)	1 Info (0 new)

Secondary Reports

Report Name	**Status**	**Generated**
Xplorer Report		

Figura 10: Sumário do circuito para placa *DETIUA-S3*

6 Conclusão

Para o problema do cálculo do maior divisor comum de dois inteiros positivos, um algoritmo iterativo é mais adequado que um recursivo, devido em grande parte, ao valor de retorno das funções recursivas ser constante, o que permite uma optimização a nível do algoritmo iterativo. A escolha de qual o método de cálculo para este problema revela-se algo incerto, visto que, têm que existir um compromisso entre o desempenho, o algoritmo e procedimento.

7 Melhoramentos

Possíveis fontes de melhoramento do projecto:

- Desenvolver mecanismos de medição de tempos no circuito implementado na placa *DETIUA-S3*;
- Desenvolver métodos de escolha automática do tipo de algoritmos (*Euclid* ou *Dijkstra*) ;

Referências

[1] : Sklyarov, V.; Skliarova, I; DETI Universidade de Aveiro, 2006 SDR - Tutorial 10.

[2] : Goldman, Kenneth J.; 1996 http://www.cs.wustl.edu/ Ekjg/cse131/Notes/DataAbstraction/recursive.html.