EXPLORAÇÃO DO ESPAÇO DE PROJETO EM ARQUITETURAS PARA CÁLCULO DA RAIZ QUADRADA INTEIRA

Thaísa L. da Silva, Gustavo P. Mateus, Rita Cristina G. Berardi, Érico K. Sawabe, Ezequiel C. Blasco, José Luís A. Güntzel, Luciano V. Agostini

Universidade Federal de Pelotas – Departamento de Informática
Grupo de Arquiteturas e Circuitos Integrados – GACI
Caixa Postal 354 – CEP. 96010-900 – Pelotas/RS – Brasil

{thleal, gpmateus, ritacgb, sawabe, yshkael, guntzel, agostini}@ufpel.edu.br

RESUMO

Este trabalho tem como objetivo explorar o espaço de projeto do cálculo da raiz quadrada inteira, através de três implementações distintas para realizar este cálculo: serial sem uso de *pipeline*, serial com uso de *pipeline* e paralela. Estas três versões foram desenvolvidas para dados de entrada de 8 e 16 bits. As arquiteturas foram descritas em VHDL e sintetizadas para um FPGA da família Flex10KE da Altera. Os resultados de síntese indicaram que a implementação com *pipeline* utilizou menos recursos do que a implementação serial e atingiu um desempenho até 4,8 vezes superior. A implementação paralela, obteve um desempenho até 160 vezes superior que a implementação serial, com impacto máximo de 10 vezes no consumo de LCs.

1. INTRODUÇÃO

A busca por um projeto eficiente, com alto desempenho, mínima utilização de recursos e reduzido consumo de potência leva os projetistas de circuitos integrados a implementarem seus projetos de diferentes formas, até encontrarem a solução mais adequada.

Projetos de circuitos integrados caracterizam-se hoje por seu crescente refinamento e eficiência, tais características são indispensáveis para a colocação de um produto de qualidade no mercado. Objetivando desenvolver projetos cada vez mais precisos, os projetistas de chips, consideram todas as possíveis implementações de seu projeto, avaliando, principalmente, o poder de processamento e o gasto de energia resultantes de cada implementação [1].

Este artigo propõe a exploração do espaço de projeto do cálculo da raiz quadrada inteira. Tal exploração foi feita através da realização de três implementações do cálculo da raiz quadrada inteira, sendo que duas delas são sincronizadas pelo *clock* e a outra é realizada em lógica combinacional pura. As três implementações foram descritas em VHDL [2] e, posteriormente, sintetizadas para um FPGA da família FLEX10KE da Altera [3]. Para validar as arquiteturas desenvolvidas foram realizadas diversas simulações, e através dos resultados obtidos, espera-se demonstrar a importância da avaliação de um projeto sob todos os seus ângulos, para a escolha da solução mais vantajosa. Além disso, através da análise dos resultados será possível verificar os impactos em termos desempenho e uso de recursos quando da utilização de cada um dos cálculos desenvolvidos em um projeto mais robusto.

2. DESENVOLVIMENTO DO CÁLCULO DA RAIZ QUADRADA INTEIRA

O cálculo da raiz quadrada de um número inteiro foi desenvolvido de três formas distintas: serial com e sem a utilização de técnicas de *pipeline*, considerando todas as operações sincronizadas pelo sinal de *clock*, e paralela, considerando todas as operações realizadas em lógica combinacional. Foram desenvolvidas duas versões para cada uma das três implementações citadas acima, uma com 8 bits de entrada e outra com 16 bits de entrada. Em todas as versões desenvolvidas são considerados apenas dados de entrada positivos e as entradas de dados possuem valor maior ou igual a 4. Nas versões de 8 bits, o maior valor de entrada é 255 e, nas versões de 16 bits, o maior valor de entrada é 65535.

As duas implementações seriais para o cálculo da raiz quadrada inteira utilizaram o algoritmo apresentado na Fig.1, onde “X” é a entrada e “r” é o resultado [4]. Este algoritmo, apesar de funcionar corretamente, não é considerado o mais eficiente para a solução do cálculo da raiz quadrada, pois o tempo de processamento de um cálculo completo é dependente do valor de entrada e, portanto, não é um tempo constante para qualquer cálculo. No algoritmo esta dependência se expressa através do laço que é controlado, mesmo que indiretamente, pela variável “X”, que é o próprio valor de entrada sob o qual deseja-se extrair a raiz. Com um valor alto para “X”, o laço se repetirá mais vezes do que com um valor baixo e, deste

modo, o tempo de processamento será maior no primeiro caso. Já a implementação paralela para realizar o cálculo da raiz quadrada inteira prevê um circuito completamente combinacional, isto é, sem nenhum registrador e, por conseqüência, sem *clock*.

```
r = 1;
d = 2;
s = 4;
while t = 0
        r = r +1;
        d = d+2;
        s = s + d + 1;
        t = signal(X – s);
```

Figura 1 – Algoritmo para o cálculo da raiz quadrada inteira

Na implementação paralela, o algoritmo apresentado na Fig. 1 foi adaptado para que o laço de repetições fosse eliminado e, após essa adaptação, tal algoritmo foi simplificado, substituindo-se todas as somas de duas constantes por uma outra constante igual a soma das duas primeiras. Esta regra foi aplicada até que houvessem apenas somas com variáveis. Deste modo, na implementação paralela foi possível eliminar as variáveis "d" e "s" do algoritmo original. Neste caso, eliminou-se significativamente o número de operadores e, o que é mais interessante, todos os cálculos foram realizados em paralelo, aumentado a velocidade de processamento. O algoritmo simplificado é mostrado na Fig. 2.

```
r1 = 2; r2 = 3; r3 = 4; r4 = 5;  ...

        t1 = signal(X – 9);
        t2 = signal(X – 16);
        t3 = signal(X – 25);
        t4 = signal(X – 36);
        t5 = signal(X – 49);
                  ⋮
        if (t1 = 1) then
            r = r1;
        else if (t2 = 1) then
                  r = r2;
              else if (t3 = 1) then
                        r = r3;
                    else if (t4 = 1) then
                              r = r4;
                                  ⋮
```

Figura 2 - Algoritmo modificado para a implementação paralela

Neste algoritmo paralelo, diferentemente do que ocorre no algoritmo serial, o tempo de processamento de cálculos completos é constante, não dependendo do valor de entrada, pois não há repetições no algoritmo paralelo, como pode ser observado na Fig. 2. Esta é uma grande vantagem que o algoritmo paralelo possui sob sua versão serial com repetições.

Nos próximos itens do artigo são apresentadas mais detalhadamente as três implementações desenvolvidas, sendo que cada uma delas foi projetada de duas formas: uma para manipular dados de 8bits e outra para dados de 16 bits.

3. IMPLEMENTAÇÃO SERIAL DO CÁLCULO DA RAIZ QUADRADA

O desenvolvimento do cálculo da raiz quadrada inteira com base no algoritmo apresentado na Fig. 1 utilizou uma estrutura hierárquica. A descrição de mais alto nível dessa implementação, conforme está apresentado na Fig. 3, é composta apenas de uma parte operativa e uma parte de controle, possuindo como principais sinais uma entrada "X" (número do qual se deseja extrair a raiz), "Clk" (sinal ativo na borda de subida), e "RST" (reseta o sistema e é um sinal ativo alto) e as saídas "R" (contendo o valor da raiz quadrada de "X") e "Ok" (indicando se o resultado está pronto).

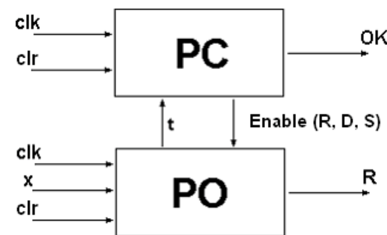

Figura 3 – Principais blocos formadores do cálculo da raiz quadrada com implementação serial sem *pipeline*

A parte operativa é formada por três blocos: R, D e S, sendo que cada um deles é formado por um registrador e um somador. O bloco D, que realiza a operação $D = D + 2$, gera uma saída para o bloco S, responsável pela operação $S = S + D + 1$. A saída do bloco S é ligada a um comparador, que recebe como entradas "X" e a saída do bloco S. Enquanto a saída do comparador é igual a zero (t = "0"), o cálculo da raiz continua em andamento até que a saída do comparador seja igual a um (t = "1"). Nesse momento, a saída do bloco R, responsável pela operação $R = R + 1$, está estável com o valor da raiz que foi calculada.

Os valores que são armazenados nos registradores dos blocos R, D e S quando o sistema é resetado não são zeros, mas sim os valores de inicialização que estão apresentados nas primeiras linhas da Fig. 1.

A parte de controle, por sua vez, foi desenvolvida através de uma máquina de estados finitos [4], que está apresentada na Fig. 4. Esta máquina de estados funciona da seguinte forma: enquanto o sinal RST está ativado,

nenhum registrador está habilitado para a escrita. Quando RST é desativado, a máquina de estados passa a testar o valor de "t" (saída do comparador). Enquanto t = "1", os registradores vão sendo habilitados em seqüência nos seus respectivos estados. Após sair do estado 4, a máquina de estados passa por uma bolha, na qual todos os registradores estão desabilitados, então a máquina volta para o estado de teste e, quando t = "0" o resultado estará pronto.

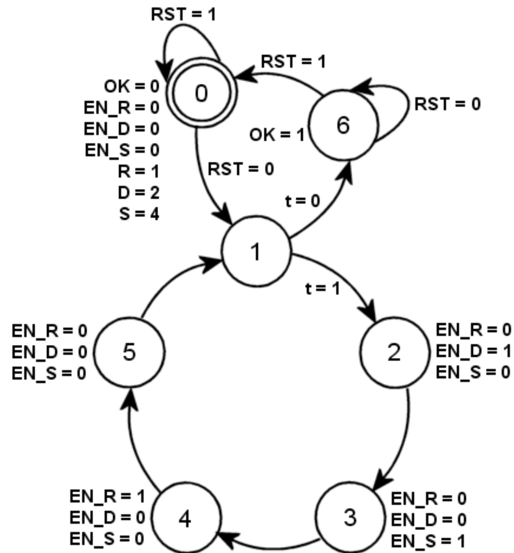

Figura 4 – Máquina de Estados Finitos

O cálculo da raiz quadrada inteira serial foi completamente descrito em VHDL [2] no ambiente Quartus II da Altera [3].

A síntese do mesmo também foi realizada com auxílio da ferramenta QuartusII [3] e foi direcionada a um FPGA da família FLEX 10KE da Altera, mais especificamente o dispositivo EPF10K30ETC144-1 [5].

Os resultados extraídos da síntese do cálculo projetado para manipular dados de 8 bits indicaram a utilização de 51 células lógicas e 15 pinos do dispositivo. Com relação ao desempenho, a descrição do cálculo atingiu um período mínimo de 7,5ns, correspondendo a uma freqüência máxima de operação de 133MHz.

Já os resultados obtidos a partir da síntese da implementação para manipular dados de 16 bits, apresentaram um total de 99 células lógicas e 27 pinos utilizados para o mesmo dispositivo. A freqüência máxima atingida foi de 76,3MHz e o período máximo foi de 13,1ns.

Comparando-se os resultados encontrados em ambas as implementações (8 e 16 bits) pode-se perceber, como esperado, que a freqüência com 16 bits diminuiu muito devido ao maior número de bits que estão sendo manipulados pelos somadores.

Como o número de iterações do laço do algoritmo apresentado na Fig. 1 depende do valor "X" de entrada, existe um tempo diferenciado para a execução do menor e do maior valor de entrada. Para a implementação com 8 bits, este tempo varia de 52,5 a 540ns, enquanto que na versão com 16 bits, esta variação é ainda maior, com tempos entre 91,7ns e 16,7µs.

Para realizar a validação das implementações de 8 e 16 bits foram realizadas várias simulações, utilizando valores de entrada especialmente selecionados para gerar estímulos representativos para as arquiteturas desenvolvidas.

A Fig. 5 mostra um exemplo de simulação da arquitetura com 8 bits de entrada. Na Fig. 5, a ativação do sinal "RST" inicializa todas as variáveis. Em seguida, com a desativação do sinal "RST", é iniciado o processo de cálculo. A entrada "X" (número que se quer extrair a raiz quadrada) recebe o valor "50", apenas como exemplo.

A saída "R" mostra todas as iterações do laço, inclusive aquelas com valores parciais, até que a saída "R" estabilizada com o valor correto, o que é indicado com a ativação do sinal "Ok". Para um novo cálculo ter início, é necessário ativar novamente o sinal "RST" e inserir um novo valor na entrada "X".

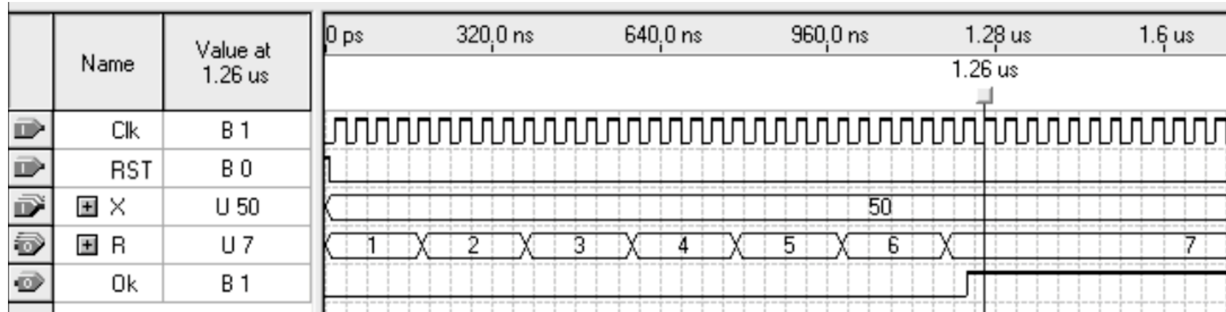

Figura 5 – Exemplo de simulação do cálculo da raiz quadrada realizado pela solução serial sem *pipeline*

4. IMPLEMENTAÇÃO SERIAL COM UTILIZAÇÃO DE *PIPELINE*

A implementação do cálculo da raiz quadrada inteira com utilização de técnicas de *pipeline* também utilizou uma estrutura hierárquica, definindo blocos com funções específicas que, integrados, geraram um único bloco de hardware. A idéia explorada nesta implementação é permitir que os operadores utilizados estejam sempre realizando alguma operação. Deste modo, os operadores foram dispostos em um *pipeline* que será melhor apresentado a seguir.

Na descrição de mais alto nível desse bloco, apresentada na Fig. 6, pode-se observar que, semelhantemente à implementação anterior, a arquitetura é composta de uma parte operativa e outra de controle.

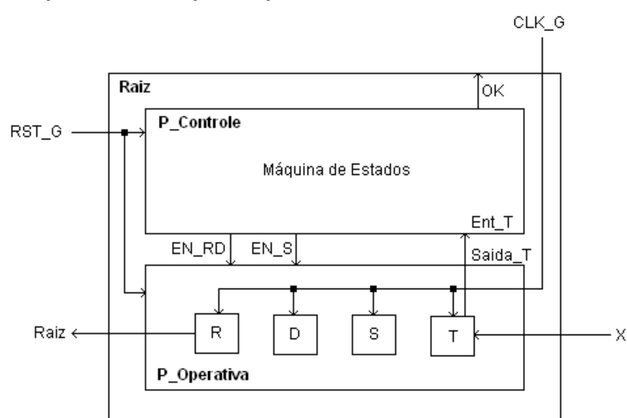

Figura 6 – Principais blocos formadores do cálculo da raiz quadrada com implementação serial com *pipeline*

Ainda na Fig. 6, pode-se perceber que o grande bloco (*Raiz*) possui a entrada "X" (número do qual se deseja extrair a raiz), e os sinais "CLOCK", e "RST' (reseta o sistema e é ativo baixo), e como saída "R" (contendo o valor da raiz quadrada de "X").

A parte operativa, por sua vez, recebe, além dos sinais de "Clk" e "RST", os sinais de habilitação "EN_RD" e "EN_S" oriundos da parte de controle, e devolve para esta, o sinal "Saida_T", que é um flag que será usado pela parte de controle para definir quando um cálculo completo foi finalizado.

A parte de controle foi desenvolvida com uma máquina de estados finitos que, através dos sinais de "Clk" e "RST", controla e sincroniza todas as operações realizadas na parte operativa, habilitando e desabilitando tais operações, conforme o estado em que se encontra.

A parte operativa é formada por quatro blocos principais: R, D, S e T.

O bloco R é responsável por gerar o cálculo $R = R+1$ do algoritmo apresentado na Fig. 1. O bloco R está apresentado na Fig. 7 e é composto por um somador e por três registradores. Estes registradores são utilizados para viabilizar o uso de *pipeline*, armazenando os valores que serão usados em cálculos futuros e liberando o somador para realizar um novo cálculo.

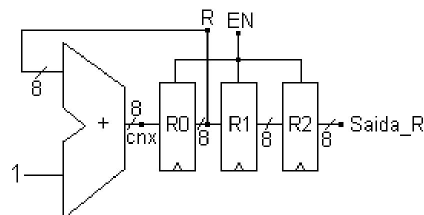

Figura 7 – Bloco R (projeto de 8 bits)

O bloco D, apresentado na Fig. 8, é responsável pelo cálculo $D = D + 2$ do algoritmo da Fig. 1. O bloco D possui um somador e um registrador, para acumular o resultado da soma.

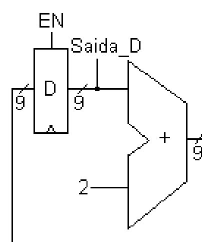

Figura 8 – Bloco D (projeto de 8 bits)

O bloco S está apresentado na Fig. 9 e executa a operação $S = S + D + 1$ do algoritmo da Fig. 1. Este bloco utiliza um registrador para acumular o resultado e um somador. Este somador possui como entradas a saída do bloco D, o valor armazenado no registrador "S" e o *carry* de entrada fixo em "1"

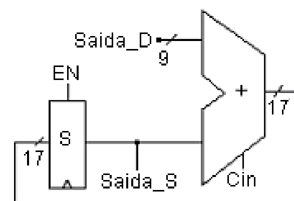

Figura 9 – Bloco S (projeto de 8 bits)

O bloco T, apresentado na Fig. 10, contém apenas um subtrator que é utilizado como um comparador. Este subtrator realiza a operação $X - S$ e entrega apenas o bit de sinal do resultado, chamado de "t". Este bit é um flag gerado pela parte operativa para a parte de controle identificar se o cálculo da raiz foi finalizado ou não. Se t = "0", então "X" é maior que "S", indicando que o cálculo ainda não terminou. Se t = "1", então "X" é menor do que "S", indicando que o cálculo terminou e que a saída "R" contém o valor correto da raiz quadrada.

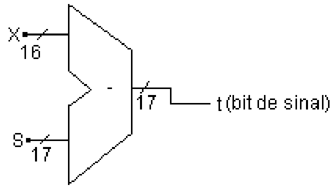

Figura 10 – Bloco T (projeto de 8 bits)

A máquina de estados finitos da parte de controle está apresentada na Fig. 11. No estado inicial é testado o valor do Reset (RST). Se RST = "0", os blocos R e D são habilitados para o primeiro cálculo e o bloco S é desabilitado. Simultaneamente, os registradores dos blocos R, D e S são inicializados com os valores apresentados nas primeiras linhas do algoritmo da Fig. 1. No caso do bloco R, esse valor inicial será armazenado no primeiro dos três registradores utilizados neste bloco (R0 na Fig. 7). Ainda neste estado, a saída "OK" é inicializada com "0", indicando que o resultado ainda não é válido. Quando RST for colocado em "1" e ocorre uma borda ascendente de clock, a máquina de estados vai para o segundo estado.

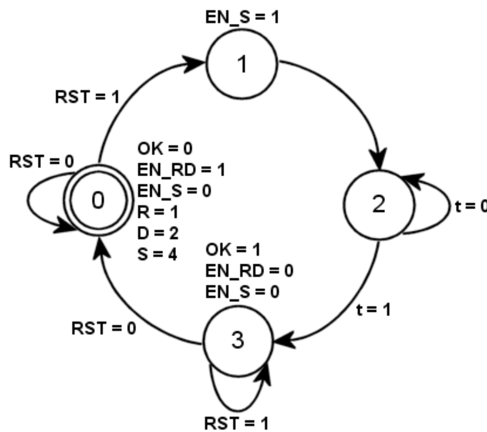

Figura 11 – Máquina de Estados Finitos

No segundo estado, o bloco S é habilitado e os blocos R e D continuam habilitados. O valor de R calculado neste estado (Fig. 7) é armazenado em "R0" e o valor anterior "R0" é copiado para "R1".

No terceiro estado, os blocos R, D e S continuam habilitados, o novo valor de R calculado neste estado (Fig. 7) é armazenado em "R0", o valor anterior "R0" é copiado para "R1" e o valor anterior de "R1" é copiado em "R2". O registrador "R2" é a própria saída do bloco R e, então, somente no terceiro estado do controle é que existirá um valor válido na saída do bloco R.

No terceiro estado é realizado um teste com o valor de "t" (gerado no bloco T – Fig 10). Se "t" for igual a "1", então o cálculo terminou e o controle vai para o último estado. Se "t" for igual a "0", então o cálculo não foi finalizado ainda e a máquina fica nesse estado testando o valor de "t" a cada ciclo de clock, até que este seja igual a "1", conforme mostra a Fig. 11.

No último estado, a saída "OK" recebe "1" indicando que a raiz foi encontrada e que o seu resultado válido está disponível na saída "R". Os blocos R, D e S são desabilitados e a máquina de controle é mantida neste estado até que um novo reset aconteça. Quando o sinal RST (reset) for novamente habilitado (valor 0), então o controle volta ao estado inicial e um novo cálculo é iniciado.

Buscando otimizar o projeto, optou-se por realizar em paralelo os cálculos de R e D apresentados no algoritmo da Fig. 1. Como não há nenhum tipo de dependência de dados nestas duas operações, elas puderam ser realizadas em paralelo.

Do ponto de vista do *pipeline*, optou-se por manter os blocos R, D e S habilitados simultaneamente no terceiro estado e desabilitá-los somente quando "t" for igual a "1", no quarto estado. Deste modo, os dois primeiros estados são estados de inicialização e preenchimento do *pipeline*, enquanto que todo o laço de repetição fica embutido no terceiro estado. Assim, a cada ciclo de clock, os valores dos blocos R, D e S vão sendo calculados em paralelo no terceiro estágio.

Na Fig. 12 está apresentado um trecho do diagrama temporal do *pipeline*. Nesta figura, estão indicados os blocos através das letras R, D, S e T, e o dado que está sendo processado, representado pelo seu respectivo número. Ao final do primeiro ciclo, R0 e D0 estão prontos para serem usados no cálculo de S0 durante o segundo ciclo. Simultaneamente, no segundo ciclo, também são calculados R1 e D1. No terceiro ciclo o valor de T0 é calculado a partir de S0 e, ao mesmo tempo, estão sendo calculados os valores de R2, D2 e S1, e assim por diante. Se, por exemplo, na Fig. 12, ao final do ciclo em que T2 é calculado o resultado indicar que a raiz quadrada está pronta, então o valor correto desta raiz foi calculado em R2 e os valores de R3, D3, S3, R4 e D4 são descartados. É por isso que existem três registradores no bloco R, pois o valor calculado em R2 não terá sido perdido quando os cálculos R3 e R4 acontecerem.

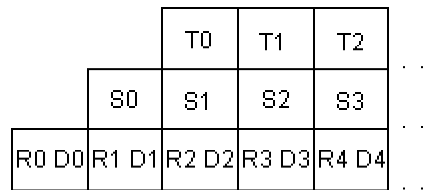

Figura 12 – Trecho do diagrama temporal do *pipeline*

Os somadores desenvolvidos neste projeto foram todos dedicados, para minimizar o consumo de recursos [6]. Estes somadores utilizam operações específicas para cada caso, adaptando os cálculos de saída e *carry out* de cada um dos somadores, conforme as suas entradas.

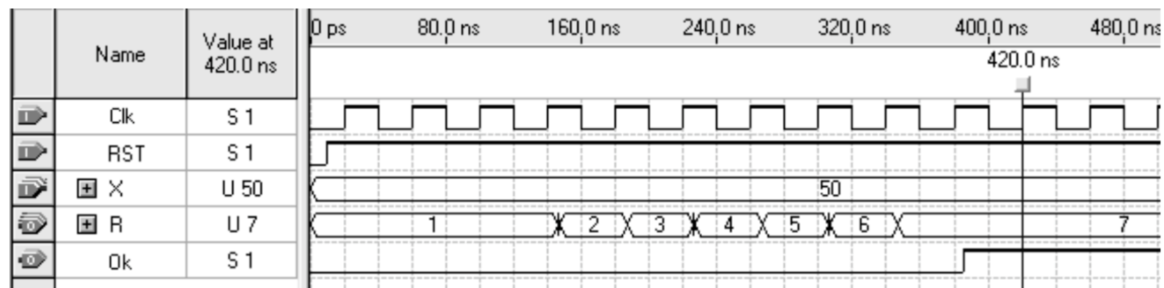

Figura 13 – Exemplo de simulação do cálculo da raiz quadrada serial com utilização de *pipeline*

A síntese de ambos os projetos (8 e 16 bits) foi realizada com auxílio da ferramenta QuartusII [3] e foi direcionada a um FPGA da família FLEX 10KE da Altera [3], o mesmo dispositivo utilizado na implementação sem *pipeline*.

Os resultados de síntese obtidos no projeto que manipula dados de 8 bits indicaram a utilização de 47 células lógicas e 15 pinos do dispositivo. Em termos de desempenho, o cálculo atingiu um período mínimo de 8,3 ns, correspondendo a uma freqüência máxima de operação de 120,5 MHz. Já os resultados de síntese do projeto de 16 bits indicaram que foram utilizadas 91 células lógicas e 27 pinos do dispositivo, alcançando, uma freqüência de 77,5 MHz e um período de 12,9ns.

A diferença de tempo para a execução do menor e do maior valor de entrada para a implementação com *pipeline* foi de 49,8 a 157,7ns para 8 bits e de 77,4ns e 3,5µs para 16 bits. Estas diferenças de tempo, mesmo sendo ainda muito significativas, diminuíram sensivelmente em relação a implementação sem o uso de *pipeline*, especialmente para os maiores valores.

A validação das implementações com 8 e 16 bits que utilizaram *pipeline* foi realizada através de várias simulações. A Fig. 13 mostra um exemplo de simulação, onde o sinal de clock (Clk) sincroniza as operações que estão sendo realizadas, o sinal reset (RST) inicia a simulação ativo (ativo em nível baixo) e, antes da segunda borda de subida de clock, é desativado. Apenas como exemplo, na Fig. 13 a entrada “Ent_X” (número que se quer extrair a raiz quadrada) recebe o valor “50”. A saída “R” mostra todas as iterações, inclusive aquelas com valores inválidos, até que seja estabilizada com o valor correto. A saída “OK” inicialmente possui o valor “0”, indicando que a raiz do número dado pela entrada “X” ainda não é válido. Somente quando a saída “R” se torna estável e correta é que o sinal “OK” recebe o valor “1”.

5. IMPLEMENTAÇÃO PARALELA

A implementação paralela do cálculo da raiz quadrada inteira compõe um circuito completamente combinacional, não possuindo nenhum registrador.

A descrição de mais alto nível desta solução está apresentada na Fig. 14, onde pode-se observar que foram usados subtratores, todos em paralelo, para a subtração da variável “X” por todas as constantes geradas a partir das antigas variáveis “s” e “d”. Tais subtratores foram simplificados, de modo que as suas saídas possuem apenas um bit, relativo ao sinal do resultado da operação realizada.

Além disso, foi usado um circuito de chaveamento, que decide qual dos resultados possíveis deve ser colocado na saída, considerando, para tanto, as variáveis “tn”, onde $1 \leq n \leq 254$ (no projeto de 16 bits, por exemplo).

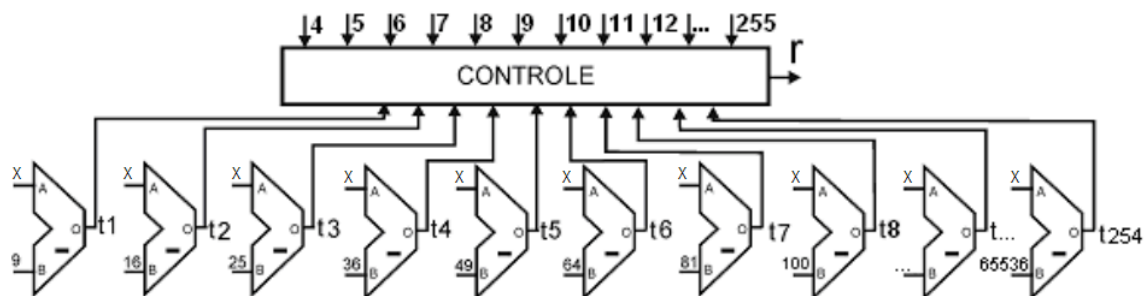

Figura 14 – Descrição de Alto Nível do Cálculo da Raiz Quadrada Inteira com implementação paralela

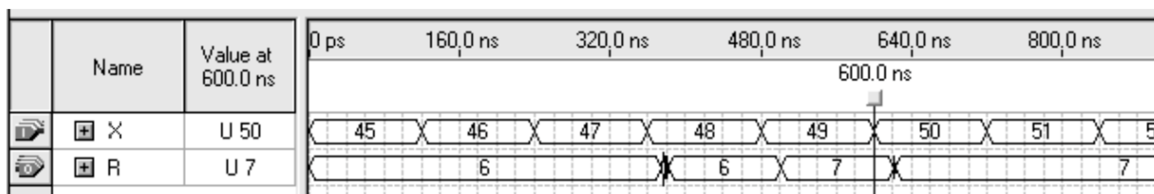

Figura 15 – Exemplo de simulação do cálculo da raiz quadrada paralela

Os resultados obtidos da síntese do projeto paralelo apresentaram um total de 23 células lógicas e 12 pinos utilizados, obtendo um atraso máximo de 18ns para o projeto de 8 bits. Já no projeto para 16 bits, foram utilizados 1032 células lógicas e 24 pinos, atingindo um atraso máximo de 104ns. Como o atraso indica o tempo para o processamento de todas as operações relativas a um cálculo completo, enquanto nas duas primeiras implementações seriais um cálculo completo é formado por um número variável de estados e, por conseqüência, por um número variável de ciclos de clock, na solução paralela é usado um tempo constante para qualquer cálculo, o que é uma importante vantagem.

Nestas implementações, os tempos para o maior ou para o menor cálculo são idênticos, sendo de 18ns para 8 bits e 104ns para 16 bits. Este tempo, além de ser fixo, é muito menor do que os obtidos para os cálculos seriais com ou sem *pipeline*, principalmente se forem comparados os tempos para os maiores cálculos.

A validação dos projetos desenvolvidos, tanto de 8 quanto de 16 bits, foram realizadas por meio simulações. A Fig. 15 apresenta um exemplo de simulação, onde “X” é a entrada (valor do qual se deseja extrair a raiz) e “R” é a saída (a qual possui o valor da raiz calculado), pode-se notar que, diferentemente das implementações seriais, a cada valor de entrada “X” a raiz “R” correspondente, é disponibilizada de forma completamente independente do ciclo de clock.

6. ANÁLISE DOS RESULTADOS

A Tab. 1 apresenta os resultados de síntese obtidos para as implementações serial sem *pipeline*, serial com *pipeline* e paralela, sempre considerando as versões com 8 e 16 bits. Nesta tabela estão apresentados os consumos de células lógicas (LCs) e de pinos do FPGA alvo, a máxima freqüência de operação, o mínimo período, o tempo gasto para o menor cálculo (TC <) e o tempo gasto para o maior cálculo (TC >).

A Tab. 1 apresenta a vantagem obtida com o uso de *pipeline* na versão serial. O cálculo para o maior valor na versão de 8 bits com *pipeline* é cerca de 3,4 vezes mais rápido do que na versão sem *pipeline.* Esta diferença é ainda maior se for considerada a versão de 16 bits, onde o cálculo do maior valor foi 4,8 vezes mais rápido.

O dado mais relevante apresentado na Tab.1 é o notável ganho de desempenho da solução paralela, em relação as soluções seriais, tanto pela velocidade mais elevada quanto pelo tempo constante usado para qualquer cálculo. Considerando a menor entrada possível, a solução paralela de 8 bits é, aproximadamente, 3 vezes mais rápida que as soluções seriais.

Na solução de 16 bits a solução paralela é até 1,3 vezes mais lenta do que as soluções seriais para a menor entrada possível. Por outro lado, as vantagens da implementação paralela em termos de desempenho fica ainda mais óbvia considerando dados de entrada muito grandes. Considerando a maior entrada de 8 bits, a solução paralela é 30 e 8,8 vezes mais rápida do que as versões seriais sem *pipeline* e com *pipeline*, respectivamente. Com a maior entrada de 16 bits, esta diferença cresce ainda mais, sendo a versão paralela 160,2 vezes mais rápida do que a versão serial sem *pipeline* e 33,6 vezes mais rápida do que a versão com *pipeline.*

Tabela 1 – Resultados de síntese para as versões do cálculo da raiz quadrada

		LCs	Pinos	Freqüência (MHz)	Período (ns)	TC < (ns)	TC > (ns)
8 bits	**Serial**	51	15	133	7,5	52,5	540
	Pipeline	47	15	120,5	8,3	49,8	157,7
	Paralela	23	12	55,6	18	18	18
16 bits	**Serial**	99	27	76,3	13,1	91,7	16.663,2
	Pipeline	91	27	77,5	12,9	77,4	3.496,5
	Paralela	1.032	24	9,6	104	104	104

A Fig. 16 apresenta um gráfico onde são comparados os tempos para o cálculo da raiz para a maior entrada entre as três arquiteturas desenvolvidas, considerado as larguras de dados de entrada de 8 e 16 bits. Na Fig. 16 fica clara a diferença de desempenho entre as soluções e a vantagem explícita da versão paralela, principalmente para dados de 16 bits, pois a curva da solução paralela cresce muito mais lentamente do que as demais.

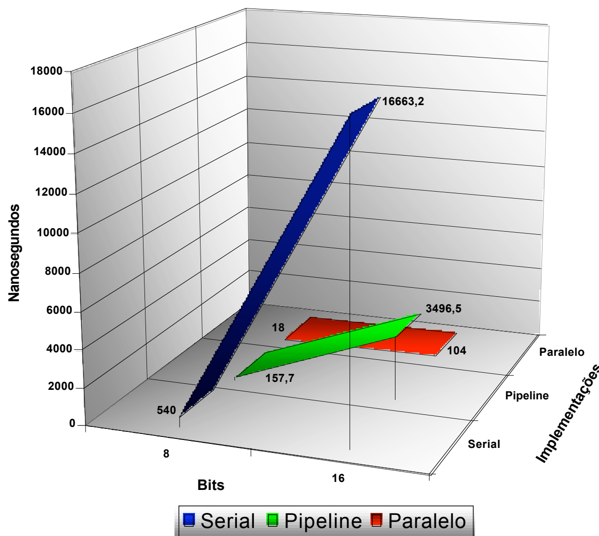

Figura 16 – Comparação de tempos para maior cálculo entre as soluções

Ao considerar a comparação em termos de utilização de recursos, é possível observar a partir da Tab. 1, que as implementações seriais com *pipeline* utilizaram um número menor de células lógicas do que a duas versões com *pipeline*. As versões com *pipeline* utilizaram 1,1 vezes menos células lógicas do que a versão sem *pipeline* e a causa principal desta redução encontra-se na construção da parte de controle, que ficou mais simples na versão com *pipeline*.

A implementação paralela de 8 bits, surpreendentemente, utilizou cerca de 2 vezes menos LCs do que as versões seriais. A ausência da parte de controle e a largura reduzida na palavra de entrada são as prováveis causas deste decréscimo na utilização de células lógicas.

Por outro lado, se considerada a implementação com 16 bits, a implementação paralela apresenta os impactos inicialmente previstos em termos de uso de recursos. A versão paralela de 16 bits utilizou 10,4 vezes mais células lógicas do que a versão serial sem *pipeline* e 11,3 vezes mais LCs do que a versão serial com *pipeline*.

A Fig. 17 apresenta um gráfico com a comparação de consumo de células lógicas. Salienta-se, neste gráfico, a tendência de crescimento muito acelerado no uso de LCs por parte das soluções paralelas a medida que a palavra de entrada cresce. Este crescimento era esperado, uma vez que os cálculos são todos feitos em paralelo e, mesmo sem necessitar de um controle, a versão paralela necessita de muitos operadores para poder realizar todos os cálculos ao mesmo tempo. O que é surpreendente é que, para 8 bits, a implementação paralela utilizou uma quantidade de LCs menor do que as implementações seriais.

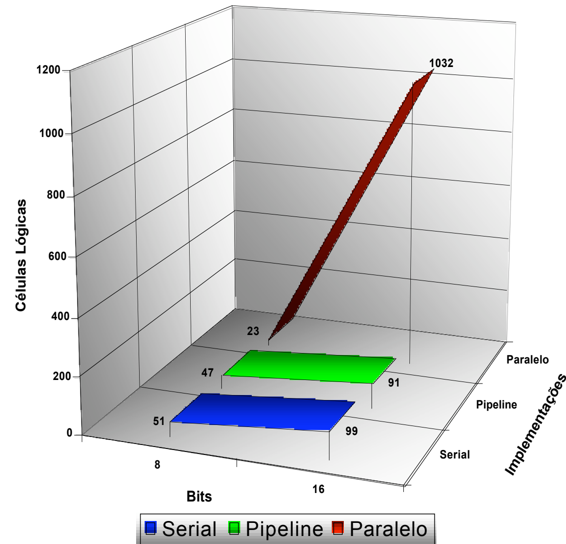

Figura 17 – Comparação em termos de consumo de células lógicas entre as soluções

7. CONCLUSÕES

Os resultados de síntese das soluções seriais com e sem *pipeline* indicaram que a solução com *pipeline* foi superior à solução serial sem *pipeline* em todos os aspectos considerados nas comparações. A versão com *pipeline* utilizou 1,1 vezes menos células lógicas do que a versão sem *pipeline* e teve tempos de cálculo menores para a menor e a maior entrada. Considerando as implementações com 8 e 16 bits, a versão com *pipeline* foi, respectivamente, 3,4 e 4,8 vezes mais rápida para o cálculo do maior valor de entrada se comparada com a versão sem *pipeline*.

A implementação paralela destacou-se em termos de desempenho tanto na implementação com 8 bits quanto na implementação com 16 bits. Considerando-se os cálculos sobre entradas de maior valor, na implementação com 8 bits a implementação paralela foi 30 vezes mais rápida que a implementação serial sem *pipeline* e 8,8 vezes mais rápida que a implementação com *pipeline*. Na implementação com 16 bits esta diferença cresceu ainda mais, sendo que a implementação paralela foi 33,6 e 160,2 vezes mais rápida que as implementações seriais com e sem *pipeline* respectivamente.

Em termos de consumo de LCs, a solução paralela foi, surpreendentemente, a melhor opção dentre as implementações de 8 bits pois, mesmo utilizando um número maior de operadores, em virtude da simplificação no algoritmo, utilizou cerca de duas vezes menos células lógicas que as implementações seriais. Porém, na implementação com 16 bits, a solução paralela utilizou

uma cerca de 11 vezes mais LCs do que as soluções seriais.

Com base na análise dos resultados foi possível concluir que a implementação paralela é a mais vantajosa em termos de desempenho em ambas implementações (8 e 16 bits), possuindo a importante característica de realizar cálculos em tempos constantes e independentes do valor de entrada. Deste modo, salvo para aplicações onde a largura de dados de entrada é maior que oito bits e onde o consumo de LCs é fator crítico, a implementação paralela é a mais apropriada para ser inserida em projetos onde o cálculo da raiz quadrada inteira se faz necessária.

Por fim, mesmo sem uma avaliação mais precisa, pode-se afirmar que a potência consumida pela solução paralela será muito menor do que a potência consumida pela demais soluções, pois a solução paralela não utiliza registradores nem sinal de clock, os quais são, sabidamente, grandes responsáveis pela potência consumida em circuitos digitais CMOS [1].

Como trabalho futuro está planejada a avaliação da potência consumida por todas as implementações, para construir uma base comparativa mais qualificada entre as implementações desenvolvidas neste artigo. Assim será possível desenvolver uma comparação completa entre a potência, a utilização de recursos e o desempenho das várias soluções arquiteturais para o cálculo da raiz quadrada inteira.

8. REFERÊNCIAS

[1] N. Weste, K. Eshraghian. *Principles of CMOS VLSI Design*. Addison-Wesley, 2nd edition, 1993.

[2] P. Ashenden. *The Designer's Guide to VHDL second edition*. San Francisco: Morgan Kaufmann Publishers, 2002.

[3] Altera Corporation, "Altera: The Programmable Solutions Company", <http://www.altera.com>, 2004.

[4] L. Carro, *Projeto e Prototipação de Sistemas Digitais*, 1ª ed. Ed. da Universidade/UFRGS, Porto Alegre, 2001.

[5] "FLEX 10KE – Embedded Programmable Logic Devices Data Sheet – version 2.3". San Jose, Altera Corporation, 2001.

[6] S. Brown, Z. Vranesic. *Fundamentals of Digital Logic With VHDL Design*. New York: Mc Graw Hill, 2000.