Máquinas de Fluxo de Dados

Ricardo S Ferreira

October 11, 2004

1 Introdução

As aplicações que envolvem o processamento de sequências (streams) vem crescendo em diversas áreas como vídeo, áudio, busca de dados na internet, redes de computadores, bioinformática, entre outras. Os processadores superscalares, vem sendo a solução mais utilizada como plataforma de hardware, seja na versão desktop, em máquinas multiprocessadas, em clusters ou nas rede locais de computadores. O aumento exponencial de desempenho e a redução dos custos permitiu a popularização desta solução nos últimos 40 anos.

Na década de 60, apenas uma dezena de transistores era sintetizada em um único chip. Na década de 70, milhares de componentes já eram integrados, passando para centenas de milhares na década de 80 e chegando a milhões na década de 90. Atualmente, a densidade de transistores está passando a barreira de um bilhão de componentes em um único chip. Apesar das arquiteturas baseadas em processadores superscalares terem sido a solução mais explorada nos últimos quinze anos, a alta complexidade dessa abordagem, devido a problemas intrínsecos dessa solução, vem limitando a exploração do potencial oferecido pela tecnologia de sílicio.

Os maiores problemas das arquiteturas superescalares são: alto consumo, falta de escalabilidade, alta complexidade de conexões, tempo de verificação e projeto. O consumo de energia é um fator muito importante nas últimas décadas sob dois pontos de vista. Primeiro, para equipamentos portáteis, como celulares e notebooks, a autonomia é fortemente dependente do consumo do processador, pois a tecnologia de baterias avança muito lentamente. Portanto, as novas gerações de processadores não podem consumir muito mais energia que as gerações precedentes. O segundo ponto, é o aquecimento, devido a alta densidade e atividade nas arquiteturas síncronas. Para dissipar o calor gerado, se faz necessário o uso de encapsulamentos mais caros e sistemas eficientes de refrigeração. O que aumenta o custo final do equipamento, e reduz sua vida útil por se tratar, em grande maioria, de sistemas de refrigeração mecânicos.

A escalabilidade é um fator chave, já que a tecnologia oferece a cada ano, uma capacidade de componentes 50% maior. Porém, dobrar a taxa de busca de instruções ou o número e o controle das unidades de ponto flutuante em uma arquitetura superscalar não é direto, pode envolver custos não lineares e muitas vezes, não traz um ganho proporcional ao custo. Cada vez mais, a área dos transistores vem sendo bem menor que a área gasta pelas conexões, ou seja, se gasta mais espaço com as conexões que com os elementos de processamento. Atualmente, apenas 10% da área do chip é ocupada por unidades de cálculo, mais da metade é ocupada por memória, e o restante por complexas estruturas de controle.

A alta complexidade dos projetos atuais, acaba aumentando o tempo de verificação. Além disso, o tamanho cada vez maior dos chips aumenta a probabilidade de falhas. Porque não explorar a localidade de fluxo de dados (dataflow locality) ? Já que os resultados produzidos por uma instrução serão consumidos pelas duas próximas instruções com probabilidade maior que 90%. Nas arquiteturas tradicionais, baseadas no modelo de Von Neumann (paradigma

processador-memória), a busca centralizada das instruções acaba sendo o gargalho do sistema, mesmo que várias unidades de cálculo estejam disponíveis.

As arquiteturas de fluxo de dados foram apontadas como uma solução na década de 70 e posteriomente no início dos anos 80[Vee86]. Porém, naquela época, devido a tecnologia de implementação e o alto custo e tempo para o desenvolvimento de chip específicos, esta abordagem foi deixada de lado. Atualmente, com o rápido tempo de desenvolvimento oferecido pelas arquiteturas reconfiguráveis baseadas em FPGA, as arquiteturas de fluxo de dados aparecem com uma solução escalável, de alto desempenho, baixo consumo e fácil conectividade, tornando-se uma alternativa viável e de grande potencial frente as arquiteturas superescalares[BG02, SNL+03, BGP03, BGD+03, SMO03].

Os FPGA oferecem uma alta flexibilidade, podendo ser reconfiguráveis dinamicamente a nível de bit. Porém, devido a alta densidade dos componentes atuais, da ordem de milhões de elementos, o processo de síntese é muito caro computacionalmente para explorar o potencial oferecido. Uma solução é adotar arquiteturas regulares e modulares como os arranjos de processadores com um grão mais grosso, com 16,32 ou 64 bits. A vantagem é a redução do tempo de síntese, que passa a ser imediato. Porém se faz necessário investigar quais devem ser as operações básicas implementadas nos elementos de processamento, bem como a topologia empregada e o protocolo de comunicação.

O ganho no desempenho pode ser alcançado graças a exploração eficiente do paralelismo disponível nas aplicações com sequências (multimedia, telecomunicações, internet, bioinformática) pelo arranjos de processadores baseados em fluxo de dados.

Nas próximas seções iremos ver alguns exemplos de modelos de elementos de processamento ilustrados através de exemplos de mapeamento de algoritmos em grafos de fluxo de dados.

2 Wavescalar

O artigo [SMO03] apresenta uma arquitetura chamada Wavescalar. Nosso objetivo aqui é apenas mostrar como um trecho de código é mapeado em um grafo do tipo dataflow, descrever os operadores e mostrar a solução mapeada na arquitetura.

O código fonte e o grafo correspondente são apresentados na figura 1

O grafo é composto por operadores. O operador WR é usado para gravar uma variável na memória (no caso o array a, os inteiros sum e i). O operador ADD executa uma soma, no caso o cálculo do endereço de A[i] para busca pelo operador load (LD). O operador A&a (triangular) executa um comando condicional. A condição (linha pontilhada) determina se o sinal de entrada será propagado para o ramo *true* ou *false*. O operador WC faz a geração do wavenumber para sincronização de memória, para maiores detalhes consulte o artigo [SMO03].

2.1 Exercício

Faça o grafo para o código:

```
for (i=0; i < 100; i+=2) {
   x = 2 + b[i];
   if ( a[i] > 0 ) {
     sum += a[i] + 2;
   }
   sum += a[i] + x;
}
```

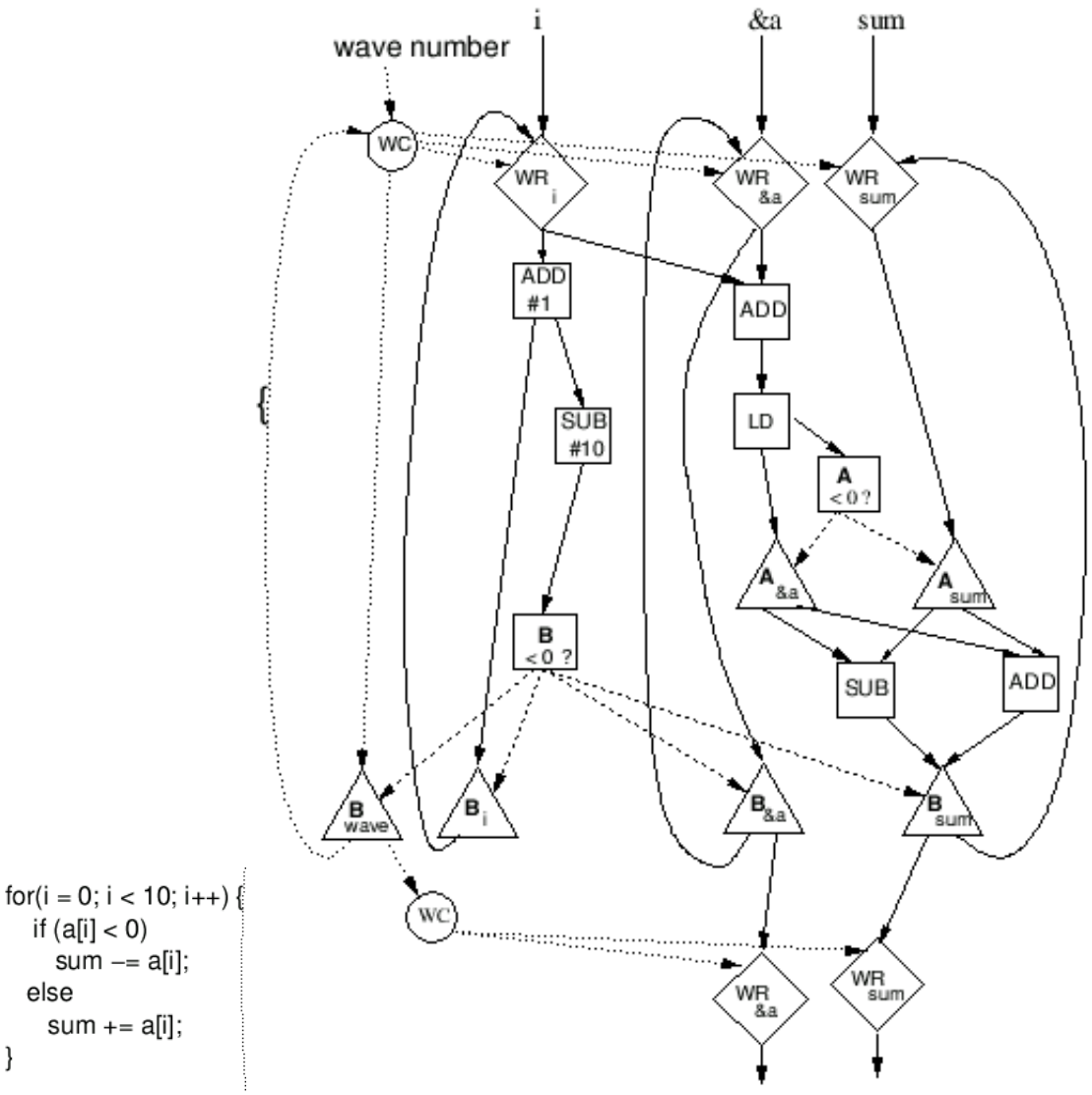

Figure 1: Wavescalar (extraído [SMO03]) : (a) Fonte. (b) Grafo

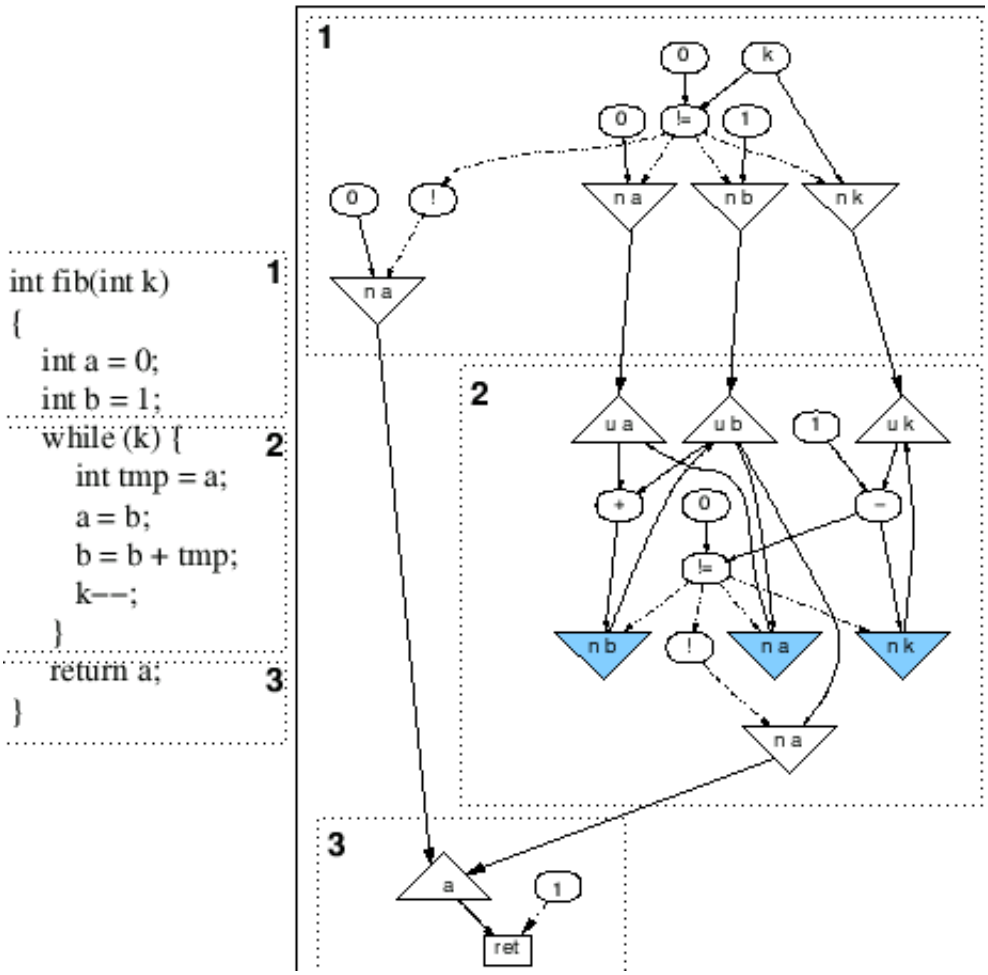

Figure 2: Modelo ASH: código e grafo[BG02]

3 ASH

Recente, um modelo dataflow de arquitetura foi apresentado em [BG02], denominado Application Specific Hardware (ASH). A figura 2 mostra um exemplo de código para a sequência de Fibonnacci e o grafo correspondente. Os nodos na forma de triângulo (apontados para face norte) são do tipo Merge. Um nodo merge recebe duas entradas e as propaga para a saída. Por exemplo, o nodo *ua* recebe o valor da variável *a* ora de fora do loop ora do final do loop. Os nodos na forma de triângulo (apontados para face sul), são denominados nodos *eta* no artigo, porém iremos usar a denominação de nodos condicionais, já que o nodo recebe uma condição booleana de entrada e propaga o valor presente na entrada de dados para saída quando a condição é verdadeira. O nome mais usual para este nodo é Tgate (True-gate). O nodo *nb* é um exemplo que repassa o valor de b para a próxima iteração do loop.

3.1 Exercícios

Faça o grafo para o exemplo abaixo.

```
int a = 5;
int b = 3;
while (k < 25) {
```

```
input (w, x)
    y := x;  t := 0;
    repeat
        if y > 1 then y := y ÷ 2
                 else y := y × 3;
        t := t + 1;
    until t = w;
output y
```

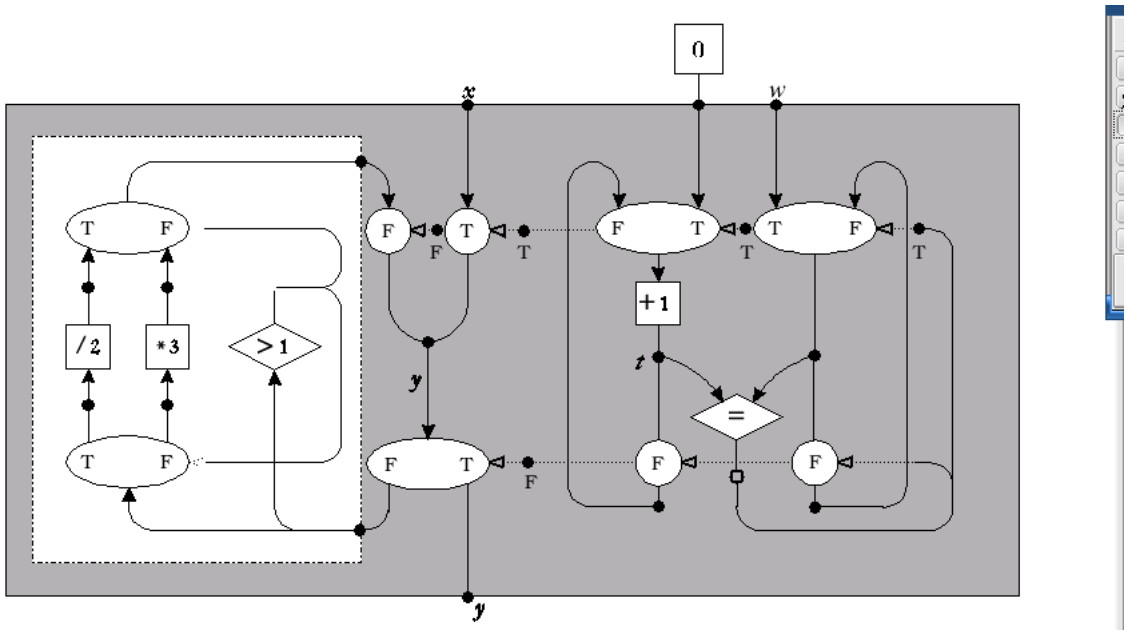

Figure 3: Modelo com fluxos separados de controle e dados (extraído [VV94]) : (a) Fonte. (b) Grafo

```
  int tmp = a - b;
  a = b * 2;
  if ( a ) b = tmp + b;
  k ++;
}
```

4 Atores Condicionais

O trabalho apresentado por [VV94], mostra um modelo de dataflow que não faz a distinção entre sinais de controle e de dados. Nas seções 2 e 3, o fluxo de controle (as saídas dos comparadores) seguia um caminho distinto do de dados.

Para ilustrar seus conceitos, considere o exemplo de código e o grafo (figura 3) correspondente no modelo de dataflow com tokens distintos para controle e dados.

Pode-se observar que no modelo com fluxos de tokens separados, é necessário ter tokens de condição para inicialização (pontos pretos com os marcadores F e T). Os operadores estão definidos na Figura 4.

O modelo com fluxo conjuntos é ilustrado na Figura 5, que corresponde ao mesmo programa fonte da Figura 3. O operador SEL funciona com um operador MERGE. Por exemplo, o operador SEL recebe x e y como entradas. Primeiro apenas o valor de x que estará presente e será passado para a saída. Depois, o valor de y que irá sendo gerado durante a execução

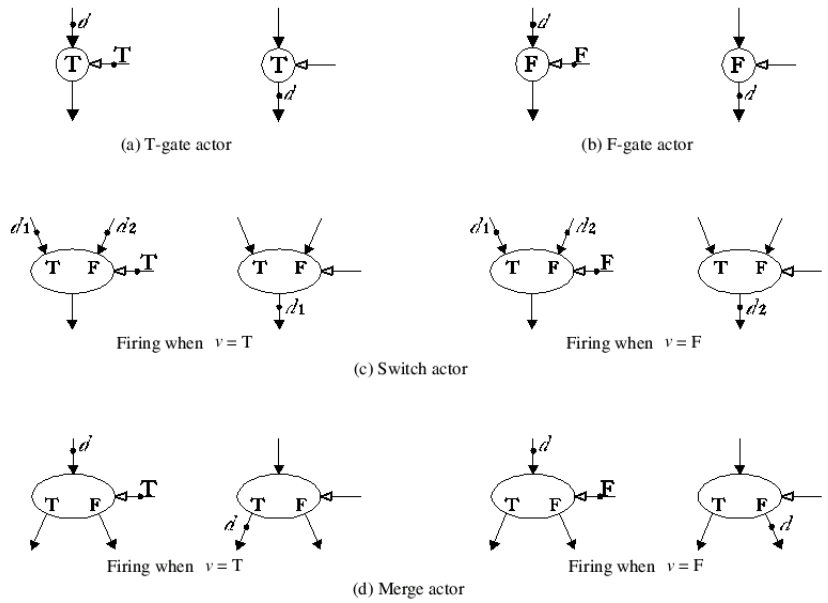

Figure 4: Operadores de dataflow

do laço. O operador + faz o papel de um operador condicional. Ele irá receber um valor a ser propagado em uma entrada e um valor inteiro ou vazio na outra entrada que servirá para implementar o controle. Se o valor inteiro for 0, ele faz a soma e funciona como um Tgate. Se o valor gerado for vazio, o valor presente na outra entrada será absorvido e nenhum sinal de saída será gerado. Note que neste exemplo, o operador + recebe sempre um valor de dados e outro valor de controle (saídas dos comparadores). Porém, o operador + poderá receber dois valores e executar a soma, se comportando como um operador aritmético, como é o caso do $*, /$ no mesmo exemplo.

Finalmente, a Figura 6 mostra um exemplo com dois laços aninhados (nested loop). Pode-se notar que é uma extensão direta do exemplo anterior.

4.1 Exercícios

Faça o grafo para o exemplo abaixo.

```
y = 0; t = 3;
while ( t < w ) {
   t = t+ 1;
   if  ( y > 0 ) y = y + t
   else y = 2 + w - t;
}
```

5 Exemplo de Filtro FIR

Nesta seção iremos mostrar como a técnica de expansão de laços (loop unrolling) pode ser usada para obter dataflow para implementações de algoritmos de Filtros FIR. Os filtros FIR (Finite Impulse Response) tem muitas aplicações na área de processamento de sinais e imagens, sendo amplamente estudados e utilizados.

Considere o código na linguagem C mostrado abaixo:

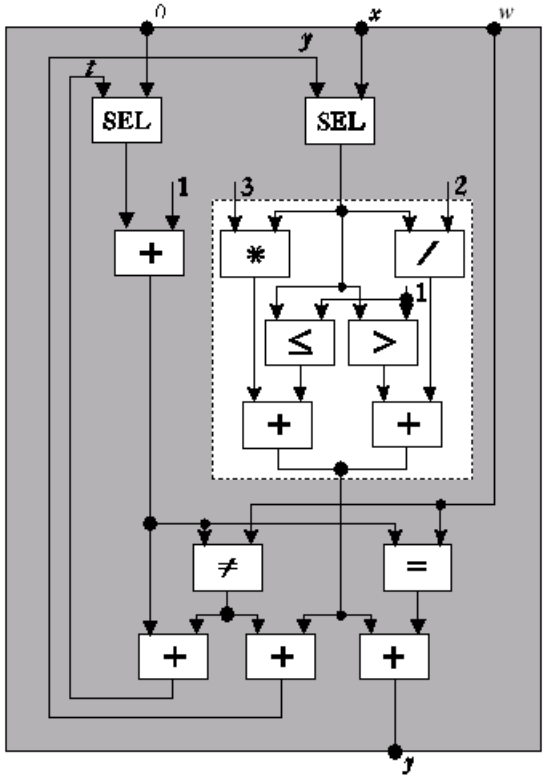

Figure 5: Mesmo código com fluxo conjuntos

```
/* TEXAS INSTRUMENTS, INC.
* USAGE This routine is C Callable and can be called as:
*
* void fir8(short *x, short *h, short *y, int N, int M)
*
* x = input array
* h = coefficient array
* y = output array
* N = number of coefficients (MULTIPLE of 8 >= 8)
* M = number of output samples (EVEN >= 2)
*/
void fir8(short x[], short h[], short y[], int N, int M)
{
   int i, j, sum;

   for (j = 0; j < M; j++) {
      sum = 0;
      for (i = 0; i < N; i++)
          sum += x[i + j] * h[i];
          y[j] = sum >> 15;
}
}
```

O laço interno implementa um somatório onde h são os coeficientes aplicados ao stream x. Temos $y_j = \sum_i x_{i+j} * h_i$. Vamos supor um caso mais simples onde o laço interno trata apenas

```
input  (w, x)
    y := x;
    repeat
        t := 0;
        repeat
            if y > 1    then y := y ÷ 2
                        else y := y × 3;
            t := t + 1;
        until t = w;
        w := w + t/2;
    until y ≤ w
output  y
```

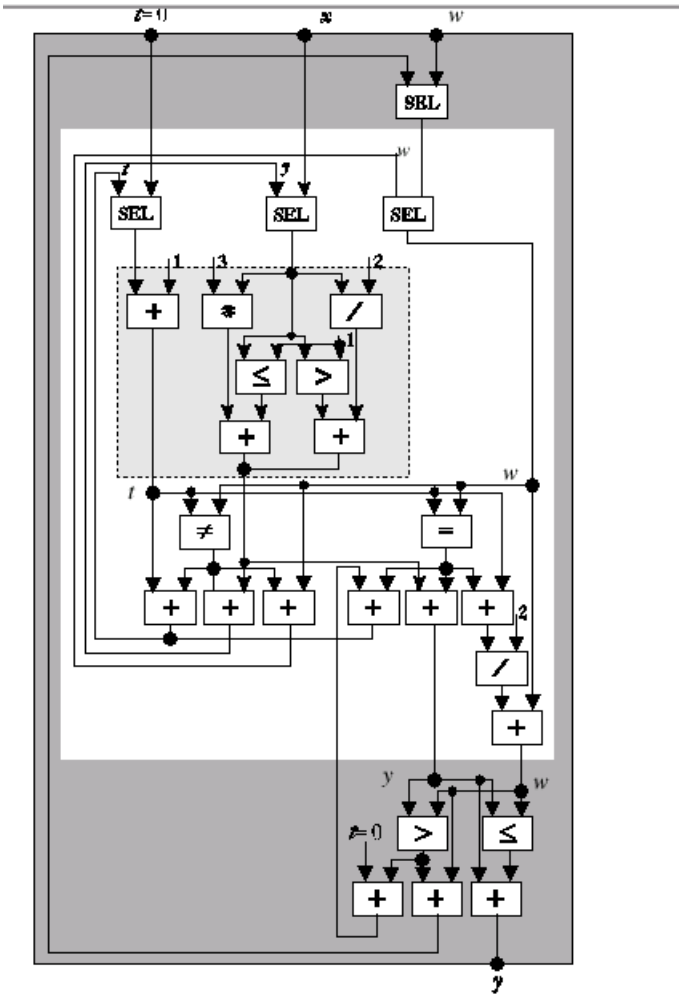

Figure 6: Modelo com fluxos separados de controle e dados (extraído [VV94]) : (a) Fonte. (b) Grafo

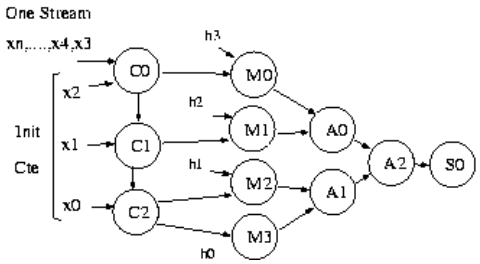

Figure 7: Dataflow expandido para loop FIR4

4 coeficientes. Se usarmos a técnica de desenrolar o loop interno teremos

```
void fir8(short x[], short h[], short y[], int N, int M)
{
   int i, j, sum;

   for (j = 0; j < M; j++) {
      sum = 0;
          sum = x[j] * h[0]+ x[j+1] * h[1] + x[j+2] * h[2]+ x[j+3] * h[3];
          y[j] = sum >> 15;
}
}
```

ou na forma de equações:

$$\begin{aligned} y_0 &= X_0 * h_0 + X_1 * h_1 + X_2 * h_2 + x_3 * h_3 \\ y_1 &= X_1 * h_0 + X_2 * h_1 + x_3 * h_2 + x_4 * h_3 \\ y_2 &= X_2 * h_0 + x_3 * h_1 + x_4 * h_2 + x_5 * h_3 \\ y_3 &= x_3 * h_0 + x_4 * h_1 + x_5 * h_2 + x_6 * h_3 \\ &.... \end{aligned}$$

Podemos notar que apenas x_0, x_1 e x_2 são os termos iniciais a circular, posteriormente, os termos x_3 em diante irão passar por todos os coeficientes (seguindo a diagonal). Tirando proveito desta propriedade regular, podemos propor o grafo de dataflow da Figura 7. O stream X entra em uma sequência de nodos do tipo Copy. Um nodo Copy serve para duplicar o fluxo. Recebe um fluxo de entrada e gera dois fluxos de saída. Para implementar a propagação de constantes de inicialização como x_0 a x_2, o nodo pode ter a funcionalidade de possuir uma entrada constante com uma função predefinida. Por exemplo, suponha que o nodo Copy tem as entradas A, B e as saídas Y, Z. Algumas funções para processamento de streams seriam:

1. Copy (default) com entrada stream A.

$$\begin{aligned} y &= a_0, a_1, \ldots \\ z &= a_0, a_1, \ldots \end{aligned}$$

2. Copy com entrada stream A, entrada B constante

$$\begin{aligned} y &= b, a_0, a_1, \ldots \\ z &= b, a_0, a_1, \ldots \end{aligned}$$

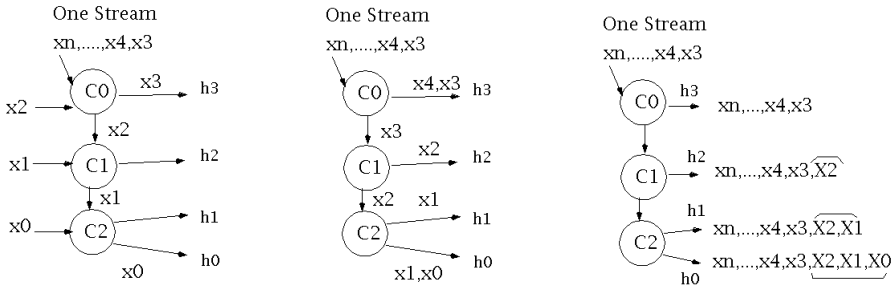

Figure 8: Sequência de propagação do stream com inicialização pelos Copy's

3. Copy com entrada stream A, entrada B constante para saída y

$$\begin{aligned} y &= b, a_0, a_1, \ldots \\ z &= a_0, a_1, \ldots \end{aligned}$$

Temos três casos exemplos, onde o copy pode ser usado em aplicações diferentes simplificando o controle. O caso três está sendo utilizado no nosso exemplo. Faz passar o valor de x_0 como constante para uma saída e depois o stream proviniente do copy anterior é propagado. Como mostra a sequência da figura **??**. Graças ao copy, o desenrolamento do loop pode ser feito sem controle adicional para fazer a distinção entre as variáveis x_0 que só interage com h_0, x_1 que só interage com h_0, h_1 e x_2 que só interage com h_0, h_1, h_2. O restante dos termos de x, de x_3 em diante, formam um único stream.

Entretanto falta controlar o final do loop, pois a constante h_0 só irá multiplicar o termo x_n, os termos adicionais não irão propagar na diagonal da expansão mostrada a seguir.

$$\begin{aligned} &\ldots\ldots\ldots \\ y_{n-3} &= x_{n-3} * h_0 + x_{n-2} * h_1 + x_{n-1} * h_2 + x_n * h_3 \\ y_{n-2} &= x_{n-2} * h_0 + x_{n-1} * h_1 + x_n * h_2 + x_{n+1} * h_3 \\ y_{n-1} &= x_{n-1} * h_0 + x_n * h_1 + x_{n+1} * h_2 + x_{n+2} * h_3 \\ y_n &= x_n * h_0 + x_{n+1} * h_1 + x_{n+2} * h_2 + x_{n+3} * h_3 \end{aligned}$$

Uma solução é usar a seguinte configuração para o copy com a entrada A com um stream de n termos, a entrada B constante para saída y, teremos

$$\begin{aligned} y &= b, a_0, a_1, \ldots, a_{n-1} \\ z &= a_0, a_1, a_2 \ldots, a_n \end{aligned}$$

Assim, os termos x_{n+1} a x_{n+3} não serão propagados para todos os copy's.

Os nodos do tipo M e A fazem a multiplicação e adição, respectivamente. Repare que a multiplicação tem uma entrada constante (coeficiente h). Finalmente, temos um nodo que faz o deslocamento e gera a saída do stream para y.

Entretanto, se o loop interno tiver $N = 64$, ou seja as constantes $h_0, \ldots, h_63$, e existir uma limitação no número de multiplicadores, por exemplo apenas 8 multiplicadores por ciclo, como poderemos compartilhar os termos, manter um único stream sem gerar estruturas de controle centralizados com várias interconexões ?

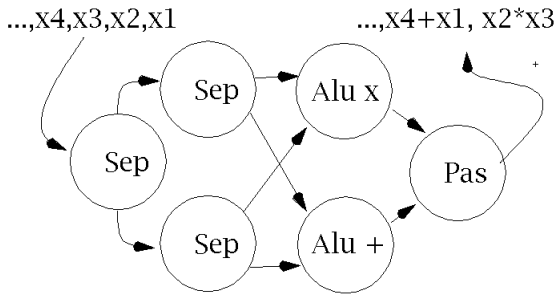

Figure 9: Operador Série-Paralelo

Uma solução distribuída é utilizar os operadores série-paralelo e paralelo-série propostos por [**?**, cardoso04] mostrado na Figura 9. O operador SEP recebe um fluxo da entrada a e gera um fluxo alternado na saída, ou seja, $y = a_0, z = a_1, y = a_2, z = a_3, \ldots$. O operador PAS recebe fluxos em ambas as entradas (a e b) e gera um fluxo serial na saída y, ou seja, $y = a_0, b_0, a_1, b_1, \ldots$.

Suponha um caso mais simples do FIR com $N = 8$ e quatro multiplicadores. O loop interno teria que executar duas interações com 4 termos em cada. Fazendo a expansão temos

$$\begin{aligned} y_0 &= x_0 * h_0 + \ldots + x_3 * h_3 + x_4 * h_4 + \ldots + x_7 * h_7 \\ y_1 &= x_1 * h_0 + \ldots + x_4 * h_3 + x_5 * h_4 + \ldots + x_8 * h_7 \\ &\ldots. \end{aligned}$$

A figura 10 mostra o FIR com operadores de copy e série-paralelo. Por exemplo, o multiplicador mais a direita na figura, executa as operações $x_0 \cdot h_0$ e $x_4 \cdot h_4$, cuja a alimentação é controlada por um operador sére-paralelo. Na primeira parte da propagação, as multiplicações $x_0 * h_0 + \ldots + x_3 * h_3$ ser"ao realizadas, depois o série paralelo irá fornecer a segunda parte $x_4 * h_4 + \ldots + x_7 * h_7$. Vale ressaltar que para agrupar as duas partes, um operador paralelo-série é adicionado ao final, na saída da árvore de somadores.

Apesar da regularidade, o código do FIR CLEX envolve mais operações e compartilhamento de termos.

```
for (i = 0; i < 2*nr; i += 2)
 {
     imag = 0;
     real = 0;
     for (j = 0; j < 2*nh; j += 2)
     {
         real += h[j+0] * x[i-j+0] - h[j+1] * x[i-j+1];
         imag += h[j+1] * x[i-j+0] + h[j+0] * x[i-j+1];
     }
     r[i  ] = (real >> 15);
     r[i+1] = (imag >> 15);
 }
```

Fazendo a expansão do loop interno para $nh = 2$, teremos as equações

$$r_0 = x_0 * h_0 - x_1 * h_1 + x_{-2} * h_2 - x_{-1} * h_3$$

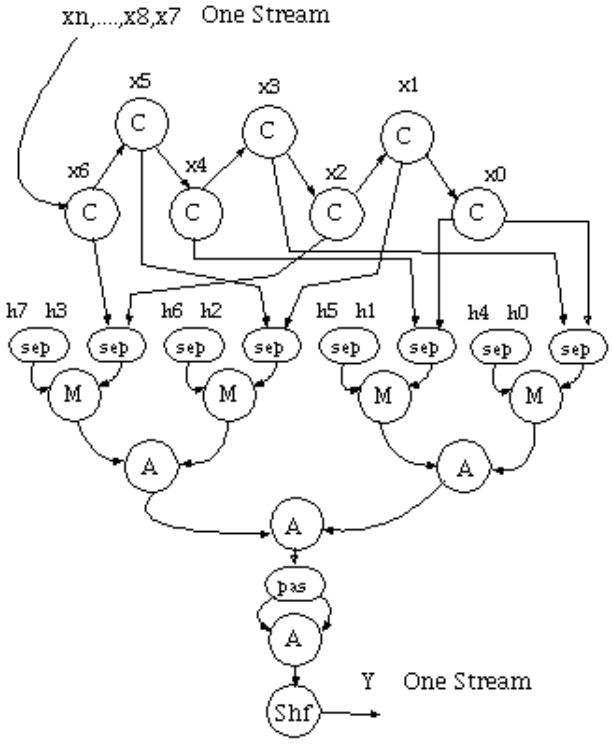

Figure 10: Fir com termos compartilhados e Operador Série-Paralelo

$$
\begin{aligned}
r_1 &= x_1 * h_0 + x_0 * h_1 + x_{-1} * h_2 + x_{-2} * h_3 \\
r_2 &= x_2 * h_0 - x_3 * h_1 + x_0 * h_2 - x_1 * h_3 \\
r_3 &= x_3 * h_0 + x_2 * h_1 + x_1 * h_2 + x_0 * h_3 \\
&....
\end{aligned}
$$

Os operadores copy's com entradas constantes e série-paralelo podem ser usados para manter um único stream. Neste exemplo, iremos considerar apenas dois multiplicadores compartillhados. Vale ressaltar que nas linhas pares, isto é $r_0, r_2, \ldots$ existem subtrações. Apesar do loop andar de dois em dois passos ($j+=2$), cada constante h multiplica todos os termos do stream que passam por ela. Uma solução é ilustrada na figura 11.

Os operadores Copy tem funções diferentes. O copy C_0 apenas duplica o stream de entrada para $y = z = x_0, \ldots, x_n$. O copy C_1 envia para a saída y conectada a C_2 o stream com uma constante de controle inicial $2, x_0, \ldots, x_n$. A constante 2 indica para o copy c_2 que duas variáveis constantes serão aplicadas a entrada constante de C_2 e para manter o stream, o mesmo número de variáveis, 2 no caso, devem ser descartadas por C_2, no caso as duas últimas x_{n-1} e x_{n-2}. A operação de descarte pode ser realizada por um buffer interno ao copy, já que não se sabe a priori quem são as duas últimas. O operador paralelo-série é usado para injetar as constantes no copy C_2. O problema dessa variação do copy é o tamanho do buffer, para loops com padrões de 2 em 2, como o exemplo acima, ou até de 4 em 4, seria viável. Para generalizar, se faz necessário o uso de outros operadores. A inicialização é simples, o buffer deve existir é para garantir a última iteração do loop, descartando as variáveis extras. Uma solução é incluir contadores nos copy ou acoplar contadores, mas mantendo os contadores distribuídos. Outra solução é usar um operador de stream, que iremos denotar por operador I que insere uma constante e retira o último termo do stream. A vantagem desse operador é a simplicidade, tem um buffer de apenas 1 elemento e resolve casos típicos de inicialização e finalização de loops. A figura 12 ilustra a

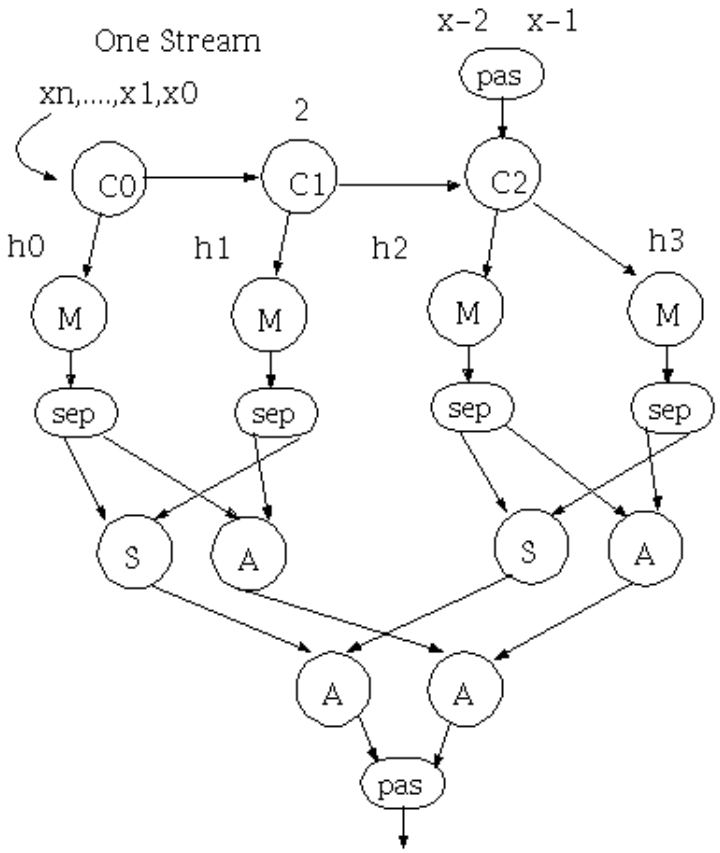

Figure 11: CPLX com termos compartilhados e Operador Série-Paralelo

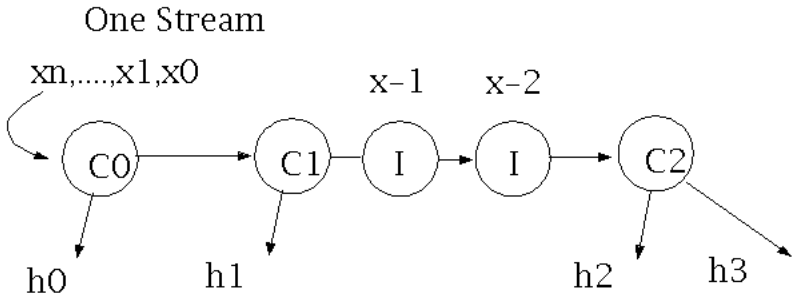

Figure 12: Ligação do operador I para finalização correta do Loop

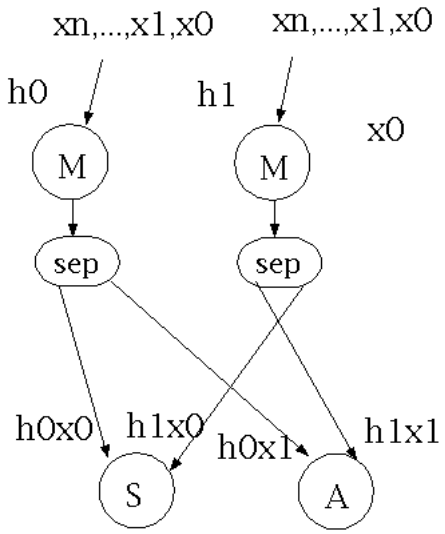

Figure 13: Ligação do operador SEP aos somadores e subtratores

configuração de copys para o CPLX usando dois operadores I.

Os nodos S indicam uma subtração. A conexão dos nodos sep´s nos somadores e subtratores é alternada, como está destacado na figura 13.

Os loops analizados acima são bem comportados. Apesar da existência de vários exemplos similares, um aspecto importante é garantir um modo formal que possa ser usado por um compilador para transformar diretamente loops bem comportados, mantendo o código fonte em alto nível. Várias técnicas foram e continuam a ser usadas e propostas para o tratamento de loops. Recentemente, uma técnica para processamento vetorial [CEL+03] propos o uso de operadores strike, span e skip. O strike indica o passo com o qual o stream é percorrido, de 1 em 1, de 2 em 2. O span indica o número de repetições do loop interno. O skip indica, a partir do último item da iteração precedente, qual será o termo inicial da próxima iteração. Por exemplo, suponha a sequência abaixo:

$$s_0 \qquad 0, 3, 6, 9, 12, 15$$

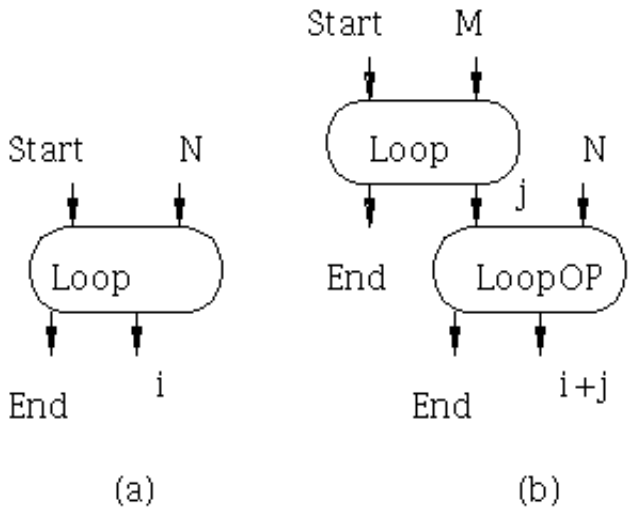

Figure 14: Operador Loop

$$\begin{array}{ll} s_1 & 1, 4, 7, 10, 13, 16 \\ s_2 & 2, 5, 8, 11, 14, 17 \end{array}$$

As sequências acima podem ser geradas com $stride = 3$, $span = 6$ e $skip = -14$.

6 Operadores para Loop

Como já foi visto nas seções anteriores, os loops são parte integrante de todos os benchmark mostrados. Técnicas com operadores condicionais para implementação do loop foram apresentadas nas primeiras seções. Outras técnicas com a expansão de loop foram apresentadas para o caso específico dos FIR's. Existe a possibilitade da criação de novos operadores para simplificar o fluxo de controle e permite a gerência do loop. Recentemente, vários trabalhos colocam em evidência importância dessa abordagem. Para maiores referências, o leitor pode connsultar [Car04].

Por exemplo, o operador Loop da figura 14(a) recebe um token na entrada start para disparar uma sequência na saída i, onde a sequência irá variar de 0 a N-1. Quando a sequência termina um token é gerado na saída End.

Considere o caso de dois laços aninhados com o código do FIR da seção . Existem muitos laços que geram sequência do tipo $i+j$. Este tipo de laço pode ser tratado com dois operadores do tipo Loop. A figura 14(b) mostra uma solução onde o operador LoopOP recebe o valor de j na entrada START e gera a sequência $j+0, j+1, \ldots, j+N-1$ na saída i, ou seja, gera $i+j$.

Assim como os operadores de Loop, um operador tipo MAC também é uma solução interessante. O operador MAC (multiply-accumulator) realiza a Multiplicação e o somatório, ou seja recebe dois termos a e b, e executa o somatório $a_0 * b_0 + a_1 * b_1 + \ldots a_n * b_n$.

6.1 Exercício

Modifique o operador LOOP se necessário, criando operadores derivados.

1. Monte um grafo usando o operador LOOP para o código da figura 1.
2. Monte um grafo usando o operador LOOP para o código da figura 2.
3. Monte um grafo usando o operador LOOP para o código da figura 3.
4. Monte um grafo usando o operador LOOP para o código da seção 6.
5. Monte um grafo usando o operador LOOP para o código da seção 9.

7 Exemplo ADCP

Nesta seção, iremos ilustrar o algoritmo do adpcm (Adaptive delta compression for audio samples). Neste exemplo, nosso objetivo é estudar alternativas de mapeamento para os comandos condicionais. Existem diferentes possibilidades. Suponha o sub-trecho (extraído do código ilustrado na seção 11 abaixo:

```
if ( diff >= step ) {
    delta = 4;
    diff -= step;
    vpdiff += step;
}
```

A figura 15 mostra algumas implementações possíveis, para linha *diff -= step;*. Suponha que C seria a condição do if, proviniente de um comparador. A implementação (a) usa o nodo condicional que irá propagar diff na saída T se a condição for verdadeira, caso contrário será propagado pela saída F. No caso de verdadeiro, para o exemplo, o novo valor de diff será obitdo da operação $diff - step$. A implementação (b) usa o operador MUX. A implementação (c) usa o operador T-gate e F-Gate. Note que é necessário o nodo copy para duplicar os tokens da condição. Note também que o copy (que não está presente) deve aparecer para duplicar o valor de diff que se propaga em mais de um ponto do grafo. A implementação (d) usa um operador aritmético condicional, *if (c) Y=A op B else Y=A* que permite implementar as atribuições comuns em todo o código.

7.1 Exercícios

Considere o código abaixo. Para cada uma das implementações da figura 15, construa o grafo usando os operadores. Não se esqueça de implementar os operadores copy quando for necessário duplicar o fluxo.

```
step >>= 1;
if ( diff >= step  ) {
    delta |= 2;
    diff -= step;
    vpdiff += step;
}
step >>= 1;
```

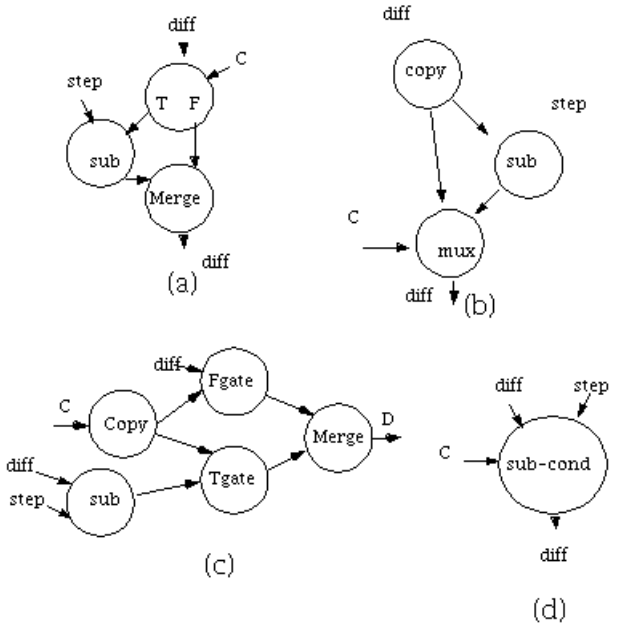

Figure 15: Operadores Condicionais

8 Exemplo DCT

Considere um subconjunto do código DCT. A seção 12 mostra o trecho completo para o laço abaixo.

```
   for (j = 0; j < N; j++) {
      /* ------------------------------------------------ /
/  Load the spatial-domain samples.                 /
/ ------------------------------------------------ */
      f0 = dct_io_ptr[ 0+i_1];
      f1 = dct_io_ptr[ 8+i_1];
      f2 = dct_io_ptr[16+i_1];
      f3 = dct_io_ptr[24+i_1];
      f4 = dct_io_ptr[32+i_1];
      f5 = dct_io_ptr[40+i_1];
      f6 = dct_io_ptr[48+i_1];
      f7 = dct_io_ptr[56+i_1];

      /* ------------------------------------------------ /
/  Stage 1:  Separate into even and odd halves.     /
/ ------------------------------------------------ */
      g0 = f0 + f7;               h2 = f0 - f7;
      g1 = f1 + f6;               h3 = f1 - f6;
      h1 = f2 + f5;               g3 = f2 - f5;
      h0 = f3 + f4;               g2 = f3 - f4;
  ....
     F1 = c7 * Q1 + c1 * S1;     F7 = c7 * S1 - c1 * Q1;
      F5 = c3 * Q0 + c5 * S0;     F3 = c3 * S0 - c5 * Q0;
```

```
    /* ---------------------------------------------- /
/  Store the frequency domain results.             /
/ ---------------------------------------------- */
    dct_io_tmp[ 0+i_1] = F0;
    dct_io_tmp[ 8+i_1] = F1 >> 13;
    dct_io_tmp[16+i_1] = F2 >> 13;
    dct_io_tmp[24+i_1] = F3 >> 13;
    dct_io_tmp[32+i_1] = F4;
    dct_io_tmp[40+i_1] = F5 >> 13;
    dct_io_tmp[48+i_1] = F6 >> 13;
    dct_io_tmp[56+i_1] = F7 >> 13;

    //dct_io_ptr++;
    i_1++;
  }
```

Neste exemplo, oito elementos do vetor dct_io_ptr são injetados em um dataflow e depois armazenados no vetor dct_io_tmp. Pode-se notar que existem muitas operações que podem ser realizadas em paralelo, como o primeiro estágio de somas e subtração. Entretanto o fluxo pode ficar limitado a um módulo sequencial de memória, o controle pode gerar muitos nodos condicionais ou nodos de load. Alguns operadores podem ser usados para facilitar esta tarefa. Suponha que um gerador de endereço seja aplicado ao vetor resultando em um stream dos elementos. No caso específico do DCT, seria gerado um fluxo (ou stream) dos elementos do vetor dct_io_ptr. Uma solução seria fazer uso dos operadores definidos como série paralelo (SEP) e paralelo série (PAS). O fluxo proviniente do vetor dct_io_ptr pode ser aplicado a uma estrutura em arvore de operadores SEP que irá gerar os sinais f_0 a f_7 nas suas respectivas folhas. O processo complementar, ou seja, uma arvore invertida, usando operadores PAS pode coletar os sinais F_0 a F_7 e armazenar no vetor dct_io_tmp.

8.1 Exercícios

Considere o código do DCT, versão resumida. Mostre como devem ser conectados os operadores SEP e PAS para gerar os fi e para coletar os Fi. Não é necessário mostrar a parte intermediária do código que está disponível no final desta apostila.

9 Exemplo FIR CLEX

Neste seção iremos considerar outro código de Filtro, semelhante ao da seção **??**. Apesar da regularidade, o código do FIR CLEX envolve mais operações e compartilhamento de termos.

```
for (i = 0; i < 2*nr; i += 2)
 {
     imag = 0;
     real = 0;
     for (j = 0; j < 2*nh; j += 2)
     {
         real += h[j+0] * x[i-j+0] - h[j+1] * x[i-j+1];
         imag += h[j+1] * x[i-j+0] + h[j+0] * x[i-j+1];
     }
     r[i  ] = (real >> 15);
```

```
        r[i+1] = (imag >> 15);
    }
```

9.1 Exercícios

Faça a expansão do loop considerando que temos 8 termos no laço interno, isto é, $real_0 = h_0x_0 - h_1x_1 + \ldots - h_7x_{-5}$, etc. Implemente um arvore para cálculo em paralelo do corpo do laço com operadores copy, sep, pas, alu, etc. Suponha que voce tenha um número ilimitado de alu. Faça outra versão, onde só tenha disponível 8 multiplicadores. Pense como compartilhar o stream x e as ALU com multiplicadores. O número de ALU com somadores pode ser ilimitado.

10 Mapeamento

Nesta seção iremos mostrar que para implementar um dataflow em uma arquitetura do tipo array se faz necessário aplicar algumas restrições, já que a topologia, capacidade de comunicação com os vizinhos e o posicionamento dos nodos no array poderá modificar alguns propriedades (como a distância entre os nodos) do dataflow original.

10.1 Hexagonal

A maioria da arquitetura proposta são baseadas em estrutura de 2 dimens oes com uma estrutura matricial linha-coluna. A topologia hexagonal pode ser mapeada intercalando as linhas e colunas, como ilustra a Figura **??**, semelhante a fafos de mel. Os custos de implementação aumentam muito pouco e geram um ganho em capacidade de roteamento e tolerancia a falhas.

10.2 Multiplicador 32x32

Nesta seção, um exemplo de multiplicador C=AxB de 32x32 usando 3 multiplicadores de 16x16 e somadores, seguindo a estrutura do Dataflow da Figura 16, onde m1,m2 e m3 são multiplicadores de 16x16, add1 e add2 são somadores, a0 e a1 são a parte menos e mais significativa de A, respectivamente.

Uma multiplicação de 32x32 resulta em um número de 64 bits, mas como se trata de uma multiplicação MOD32, só é necessário gerar os 32 bits menos significativos.

O mapeamento no array hexagonal não necessita de células especiais para roteamento, apenas uma das células exerce uma função dupla, faz o ADD1 e o roteamento de B1 para M3. A implementação foi simplificada considerando que existe uma unidade funcional que decompõe um número de 32 bits e gera duas saídas, ou seja, com apenas 1 nó se implementa os nós a,a0 e a1. Na Figura 17, as células em vermelho mostram que existe tres possibilidades diretas para receber A, tres possibilidades para B (em azul) e quatro possibilidades para gerar o resultado C (em verde). Se for possível usar as células como roteamento, A,B e C podem ser recebidos ou transmitidos ao mais células.

10.3 FIR4

Um exemplo de mapeamento do grafo do FIR4 é ilustrado na Figura 18. Pode-se notar que não foram necessários nodos de roteamento (que não implementam funções, apenas servem de passagem para as conexões). A maioria das conexões são entre vizinhos e apenas 3 conexões tem comprimento 2 (passam por um nodo intermediário).

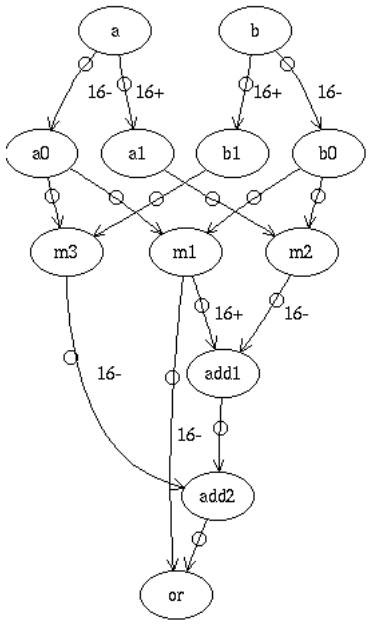

Figure 16: Dataflow do Multiplicador

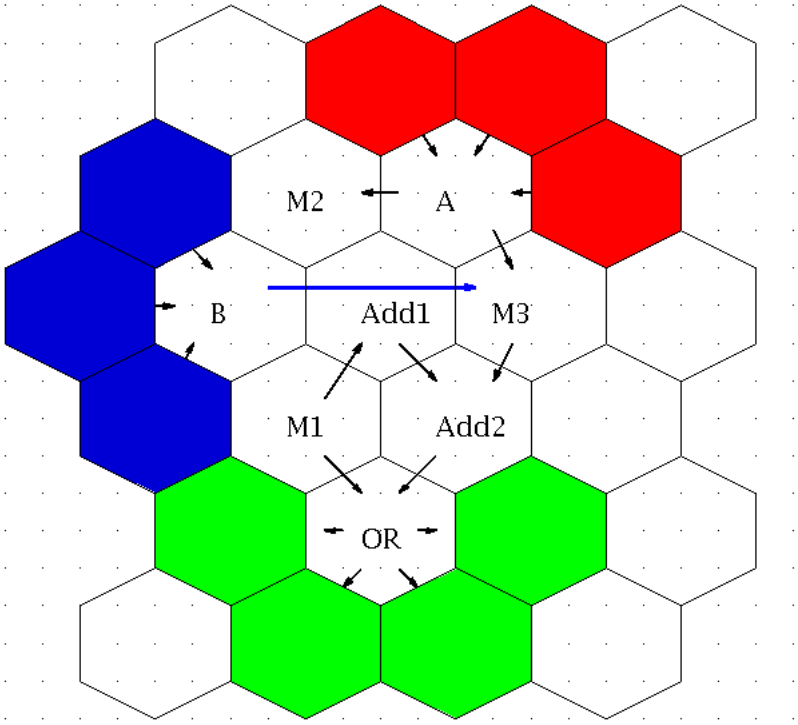

Figure 17: Multiplicador no Array Hexagonal

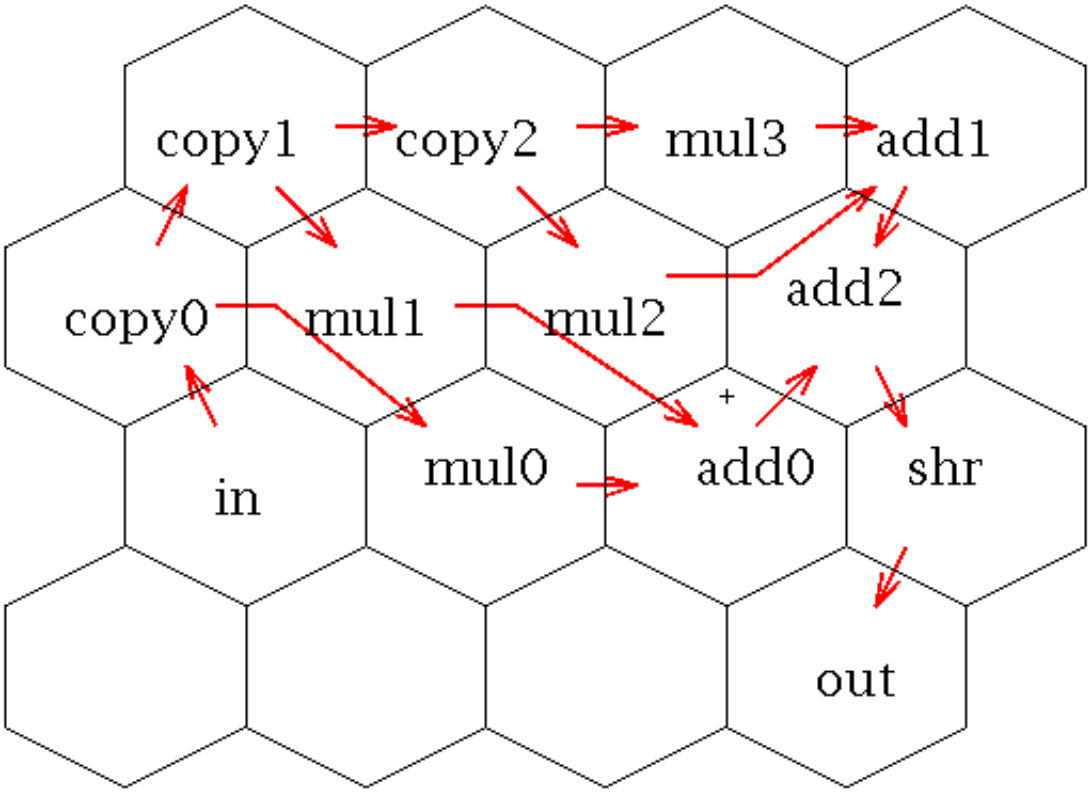

Figure 18: FIR4 em uma array hexagonal

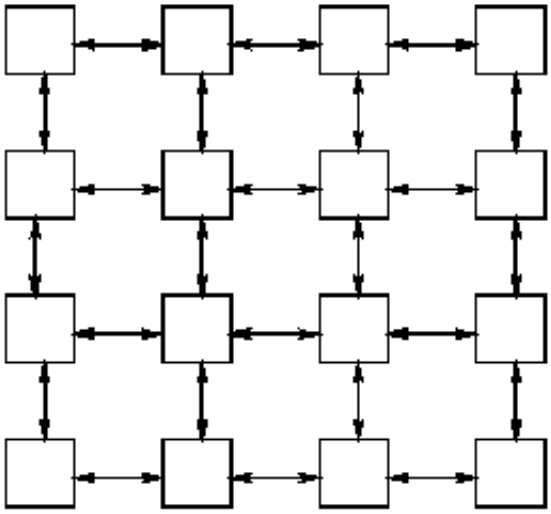

Figure 19: Estrutura de Grid com vizinhos diretos

10.4 Exercícios

Considere os grafos dataflow das seções anteriores e dos exercícios.

1. Faça o mapeamento em uma estrutura hexagonal.
2. Faça o mapeamento em uma estrutura em Grid da figura 19.
3. Faça o mapeamento em uma estrutura em Grid da figura 20.
4. Faça o mapeamento em uma estrutura em Grid da figura 22.
5. Faça o mapeamento em uma estrutura em Grid da figura 19 usando as três formas de posicioamento da Figura **??**.

References

[BG02] Mihai Budiu and Seth Copen Goldstein. Compiling application-specific hardware. *Lecture Notes in Computer Science*, 2438:853–??, 2002.

[BGD+03] N. Bansal, S. Gupta, N. Dutt, A. N, and R. Gupta. Network topology exploration of mesh-based coarse-grain reconfigurable architectures. Technical report, CECS Technical Report 03-27, 2003.

[BGP03] L. Bossuet, G. Gogniat, and J.L. Philippe. Fast design space exploration method for reconfigurable architectures. In *The International Conference on Engineering of Reconfigurable Systems and Algorithm, ERSA'03*, Las Vegas, Nevada, USA, June 23-26 2003.

[Car04] João Cardoso. Self loop pipelining and reconfigurabel dataflow arrays. In *SAMOS*, 2004.

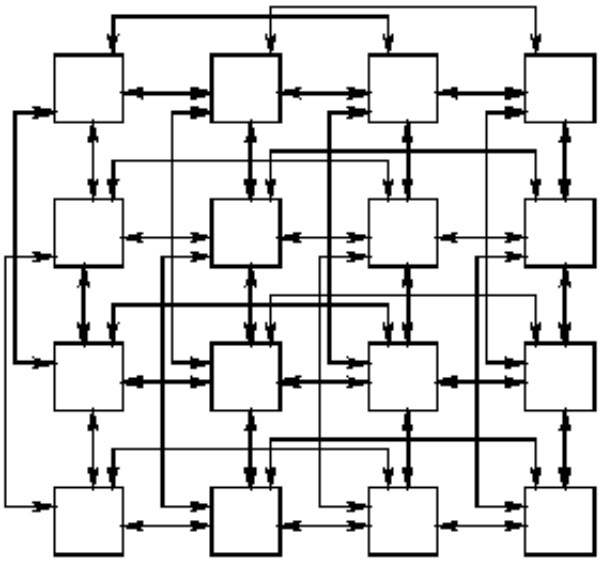

Figure 20: Estrutura de Grid com vizinhos de distância 2

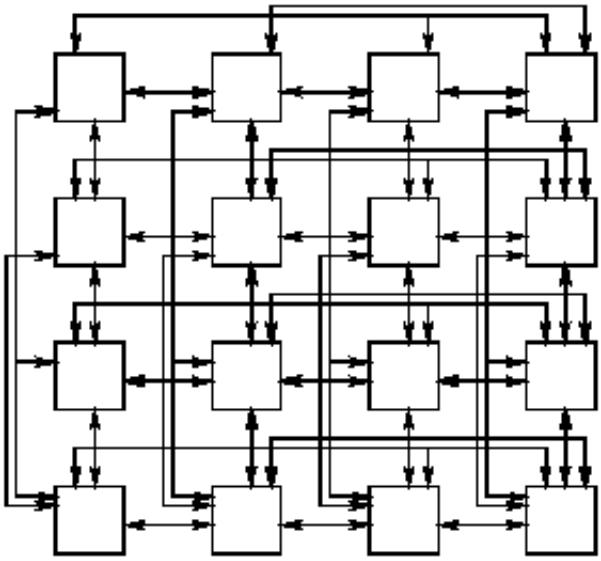

Figure 21: Estrutura de Grid com vizinhos linha/coluna

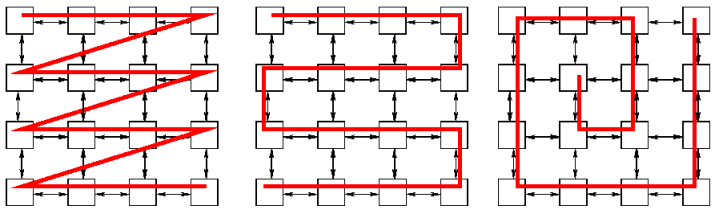

Figure 22: Mapeamento varredura, zig-zag e espiral

[CEL+03] Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi. The reconfigurable streaming vector processor (rsvptm). In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 141. IEEE Computer Society, 2003.

[SMO03] Steven Swanson, Ken Michelson, and Mark Oskin. Wavescalar. Technical Report UW-CSE-03-01-01, University of Washington, 2003.

[SNL+03] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In Doug DeGroot, editor, *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA-03)*, volume 31, 2 of *Computer Architecture News*, pages 422–433, New York, June 9–11 2003. ACM Press.

[Vee86] Arthur Veen. Dataflow machine archirecture. *ACM Computing Surveys*, 18:365–396, 1986.

[VV94] Lorenzo Verdsoscia and Roberto Vaccaro. Conditional and iterative structures using a homogeneous static dataflow graph model. Technical report, Istituto per la Ricerca sui Sistemi Informatici Paralleli - CNR, Naples, Italy, 1994.

11 ADPCM

O trecho acima é apenas um subconjunto do algoritmo codificado em C.

```
Copyright 1992 by Stichting Mathematisch Centrum, Amsterdam, The
Netherlands.
 ** Intel/DVI ADPCM coder/decoder.
   for ( ; len > 0 ; len-- ) {
val = *inp++;

/* Step 1 - compute difference with previous value */
diff = val - valpred;
sign = (diff < 0) ? 8 : 0;
if ( sign ) diff = (-diff);

/* Step 2 - Divide and clamp */
/* Note:
** This code *approximately* computes:
**    delta = diff*4/step;
**    vpdiff = (delta+0.5)*step/4;
** but in shift step bits are dropped. The net result of this is
** that even if you have fast mul/div hardware you cannot put it to
** good use since the fixup would be too expensive.
*/
delta = 0;
vpdiff = (step >> 3);

if ( diff >= step ) {
    delta = 4;
```

```
    diff -= step;
    vpdiff += step;
}
step >>= 1;
if ( diff >= step  ) {
    delta |= 2;
    diff -= step;
    vpdiff += step;
}
step >>= 1;
if ( diff >= step ) {
    delta |= 1;
    vpdiff += step;
}

/* Step 3 - Update previous value */
if ( sign )
  valpred -= vpdiff;
else
  valpred += vpdiff;

/* Step 4 - Clamp previous value to 16 bits */
if ( valpred > 32767 )
  valpred = 32767;
else if ( valpred < -32768 )
  valpred = -32768;

/* Step 5 - Assemble value, update index and step values */
delta |= sign;

index += indexTable[delta];
if ( index < 0 ) index = 0;
if ( index > 88 ) index = 88;
step = stepsizeTable[index];

/* Step 6 - Output value */
if ( bufferstep ) {
    outputbuffer = (delta << 4) & 0xf0;
} else {
    *outp++ = (delta & 0x0f) | outputbuffer;
}
bufferstep = !bufferstep;
    }

    /* Output last step, if needed */
    if ( !bufferstep )
      *outp++ = outputbuffer;

    state->valprev = valpred;
    state->index = index;
```

```
}
```

12 DCT

```
   /* ------------------------------------------------- /
      /  Perform Vert 1-D FDCT on columns within each block. /
      / ------------------------------------------------- */
   for (j = 0; j < N; j++) {
     /* ------------------------------------------------ /
/  Load the spatial-domain samples.                 /
/ ------------------------------------------------ */
     f0 = dct_io_ptr[ 0+i_1];
     f1 = dct_io_ptr[ 8+i_1];
     f2 = dct_io_ptr[16+i_1];
     f3 = dct_io_ptr[24+i_1];
     f4 = dct_io_ptr[32+i_1];
     f5 = dct_io_ptr[40+i_1];
     f6 = dct_io_ptr[48+i_1];
     f7 = dct_io_ptr[56+i_1];

     /* ------------------------------------------------ /
/  Stage 1:  Separate into even and odd halves.     /
/ ------------------------------------------------ */
     g0 = f0 + f7;               h2 = f0 - f7;
     g1 = f1 + f6;               h3 = f1 - f6;
     h1 = f2 + f5;               g3 = f2 - f5;
     h0 = f3 + f4;               g2 = f3 - f4;

     /* ------------------------------------------------ /
/  Stage 2                                          /
/ ------------------------------------------------ */
     p0 = g0 + h0;               r0 = g0 - h0;
     p1 = g1 + h1;               r1 = g1 - h1;
     q1 = g2;                    s1 = h2;

     s0a= h3 + g3;               q0a= h3 - g3;
     s0 = (s0a * c0 + 0x7FFF) >> 16;
     q0 = (q0a * c0 + 0x7FFF) >> 16;

     /* ------------------------------------------------ /
/  Stage 3                                          /
/ ------------------------------------------------ */
     P0 = p0 + p1;             P1 = p0 - p1;
     R1 = c6 * r1 + c2 * r0;     R0 = c6 * r0 - c2 * r1;

     Q1 = q1 + q0;               Q0 = q1 - q0;
     S1 = s1 + s0;               S0 = s1 - s0;
```

```
    /* ---------------------------------------------- /
/  Stage 4                                          /
/ ---------------------------------------------- */
    F0 = P0;                  F4 = P1;
    F2 = R1;                  F6 = R0;

    F1 = c7 * Q1 + c1 * S1;   F7 = c7 * S1 - c1 * Q1;
    F5 = c3 * Q0 + c5 * S0;   F3 = c3 * S0 - c5 * Q0;

    /* ---------------------------------------------- /
/  Store the frequency domain results.              /
/ ---------------------------------------------- */
    dct_io_tmp[ 0+i_1] = F0;
    dct_io_tmp[ 8+i_1] = F1 >> 13;
    dct_io_tmp[16+i_1] = F2 >> 13;
    dct_io_tmp[24+i_1] = F3 >> 13;
    dct_io_tmp[32+i_1] = F4;
    dct_io_tmp[40+i_1] = F5 >> 13;
    dct_io_tmp[48+i_1] = F6 >> 13;
    dct_io_tmp[56+i_1] = F7 >> 13;

    //dct_io_ptr++;
    i_1++;
  }
  /* ---------------------------------------------------- /
     /  Update pointer to next 8x8 FDCT block.             /
     / ---------------------------------------------------- */
  //dct_io_ptr += 56;
  i_1 += 56;
 }
```