Building Conflict-Free FFT Schedules

Stephen Richardson, *Member, IEEE*, Dejan Marković, *Member, IEEE*, Andrew Danowitz, *Member, IEEE*, John Brunhaver, *Member, IEEE*, and Mark Horowitz, *Fellow, IEEE*

***Abstract*—A *conflict-free schedule* lets an FFT run to completion without ever having to pause for memory-conflict resolution. We show how to build such schedules for FFTs having any number of butterfly units B operating at any radix R, transforming any number of datapoints D. Our algorithm works for FFT datapaths with or without pipeline overlap, and for memory banks having any number of access ports. Specifically, it enables construction of conflict-free schedules using single-ported memory banks, which require less area than more traditional multi-ported designs.**

***Index Terms*—Conflict-free scheduling, digital signal processor, fast Fourier transform, FFT, single-ported memory.**

I. Introduction

DEMAND FOR low-power and low-cost solutions underscores the proliferation of custom FFTs embedded in battery-driven devices like smart-phones and tablets, where they drive OFDM-based WiFi/WLAN and 4G cellular communication applications like LTE [1]. A myriad of architectural choices underlies the design of these FFTs, nearly all of which involve one or more radix-2, radix-2^n, and/or prime-radix FFT stages. We develop a radix-2 schedule that is both power- and performance-efficient, then demonstrate how it naturally extends to *any* radix, thus pointing the way to its use in existing and future FFT designs.

Our schedule improves cost efficiency by reducing the die area required by custom FFT hardware. It targets smaller single-ported rather than traditional multi-ported memory, which has been shown to reduce the physical size of required on-chip memory by 30%–53% [2].

FFT implementations tend to fall into one of two main architecture classes, serial-pipeline and memory-based [3]. Serial-pipeline architectures generally require more hardware resources. Therefore we target memory-based architectures, operating with as few as one butterfly unit, because our prime concern is to minimize die area. In particular, we target *in-place* algorithms, generally chosen for their lower resource requirement versus pipeline architectures, and which can lead to lower power implementations [4].

Each stage in an FFT must read and then write back its entire data set, and each time in a different order, so there are ample opportunities for memory-access conflict. **This paper describes an optimal placement and access strategy for FFTs such that data can be fetched with *zero conflicts* so as to maximize performance while using minimal area for data storage**.

Note that our minimal-area goal goes beyond the traditional concern of minimizing memory-locations-per-datapoint. Our *in-place* implementation indeed requires a minimum of only D memory locations (data-words) to transform D datapoints. But the area savings go a step further. Like recent work by Luo [2], our algorithm enables the use of single-ported memory structures, meaning *less die area per data-word*.

To build an efficient FFT using only single-ported memory, we need to use multiple SRAMs and place the data such that butterfly unit(s) can fetch operands without conflict. We will show how to derive an algorithm to compute this placement, one that covers FFT designs operating on any given number of datapoints D, using any number of butterflies B operating in parallel, and where each butterfly operates at any radix R using any pipeline depth P.

Initially, we restrict B, R, and D to fixed powers of two, but later we will indicate how to relax this restriction. Also, we will show how to further extend the algorithm to cover overlapped/pipelined execution, where results get written to memory at the same time that new operands are being read, all without conflict or collision.[1]

Our previous paper [5] presented an algorithm that worked empirically for any number of datapoints D from $D = 8$ up to $D = 8192$ and for $B = 1, 2, 4$, or 8 butterfly units of radix $R = 2$, operating with an overlapping pipeline of depth $P = 2$. The paper postulated that the algorithm should work for any values of D, B, R, and P, but could not show *why* the algorithm would work. **To address that shortcoming, this paper presents a refined and simplified algorithm, and explains why it works for all D, B, R, and P.**

We chose a basic radix-2 Cooley-Tukey (CT) algorithm (Fig. 1), rather than e.g., a constant-geometry (CG) design, to develop our method. While CG alternatives can simplify the addressing requirements for an FFT [6], the problem of parallel access to stride-separated data still remains [7]. Thus a conflict-free *map* such as the one we will propose should work for CG designs, especially considering that the map is heavily based on earlier work that specifically targeted CG [7]. The map should work as well for many other DFT forms, because it simply shows how to distribute data among memory banks to prevent collision regardless of stride. In particular, by using the radix-2 CT algorithm to develop a conflict-free map, we can simultaneously build a conflict-free CT *schedule* to go along with the map, using the same set of principles and the same simple hardware.

S. Richardson and M. Horowitz are with Stanford University, Stanford, CA 94305 USA (e-mail: steveri@stanford.edu; horowitz@stanford.edu).

D. Marković is with Electrical Engineering Department, University of California, Los Angeles, Los Angeles, CA 90095 USA (e-mail: dejan@ee.ucla.edu).

A. Danowitz is with Computer Engineering Department, California Polytechnic State University, San Luis Obispo, CA 93407 USA (e-mail: adanowitz@gmail.com).

J. Brunhaver is with Arizona State University, Tempe, AZ 85281 USA (e-mail: jbrunhaver@gmail.com).

[1] Note pipelining (Section IV) is different than time-multiplexing.

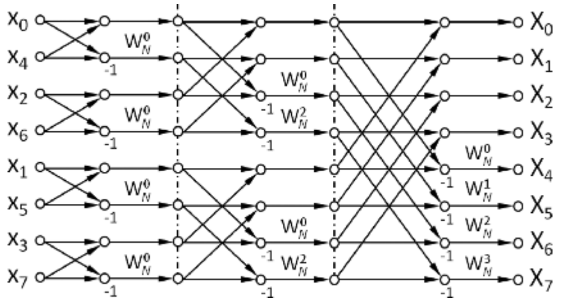


Fig. 1. Signal flow graph, radix-2 decimation in time (DIT).

Our method should thus extend to many other common stride-based DFT algorithms. E.g., instead of CT, a designer might consider prime-factor algorithms such as Good-Thomas, Winograd, or a combination thereof [8], with the possible benefit of reducing/eliminating twiddle factors and complex multiplies, or providing a better fit for applications with a non-power-of-two number of datapoints [2]. Such designs could still be time-multiplexed and/or pipelined such that complex address generation would be required for conflict-free access, at which point our algorithm may be considered for precise data placement and access patterns.

Note that we target only the *address generation* portion of FFT design and do not concern ourselves with unrelated design issues like precise implementation of the trig table. Our sample design used a single ROM to hold pre-computed twiddle factors [5]; alternately, one might use *distributed* ROM tables [9] or even calculate twiddle factors on-the-fly [10].

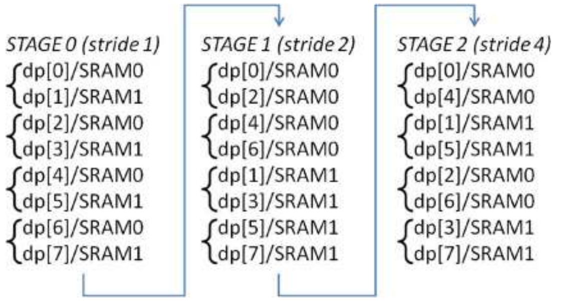
Section II introduces *schedules* and *groups*, key concepts to understanding the derivation of our algorithm. Section III discusses how to map datapoints into memory banks such that they can be fetched and written without conflict, at least for simple FFTs without overlapped pipelines; and Section IV extends the algorithm to *pipelined* FFTs, as well as FFTs with radix $R > 2$. Section V discusses background work, and then we conclude.

II. Schedules and Groups

A **schedule** tells us the order in which an FFT will process its data set. Our schedules are designed with a specific targeted group size G. Each sequential **group** of G datapoints within the schedule represents operands that can be *processed* all at once. Thus for maximum performance the group should be *accessed* all at once. When using single-ported memory, this means that the G operands must live in G separate SRAMs.

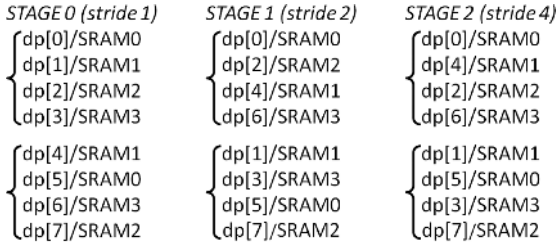
Fig. 2 shows the complete schedule for a radix-2 FFT designed to transform eight datapoints, a toy example we can use to demonstrate the important features of our algorithm. In our nomenclature, this is a schedule for $D = 8$ and $G = BR = 2$. This schedule is based on the original Cooley-Tukey algorithm [11]. (The *map* part of the schedule will be discussed later.) [2]

Unlike a signal flow graph, which can only show how data flows from stage to stage, this *group-2 schedule* shows the order in which a single radix-2 butterfly unit will read and process datapoints within each stage. In the schedule, we have bracketed each data pair to show that, conceptually, both are accessed at the same time.



STAGE 0 (stride 1)	STAGE 1 (stride 2)	STAGE 2 (stride 4)
dp[0]/SRAM0	dp[0]/SRAM0	dp[0]/SRAM0
dp[1]/SRAM1	dp[2]/SRAM0	dp[4]/SRAM0
dp[2]/SRAM0	dp[4]/SRAM0	dp[1]/SRAM1
dp[3]/SRAM1	dp[6]/SRAM0	dp[5]/SRAM1
dp[4]/SRAM0	dp[1]/SRAM1	dp[2]/SRAM0
dp[5]/SRAM1	dp[3]/SRAM1	dp[6]/SRAM0
dp[6]/SRAM0	dp[5]/SRAM1	dp[3]/SRAM1
dp[7]/SRAM1	dp[7]/SRAM1	dp[7]/SRAM1

Fig. 2. Group-2 schedule and SRAM map for 8-point FFT with one radix-2 butterfly, i.e., $D = 8$ and $G = RB = 2$. If the operands in each bracketed pair live in separate memory banks, we call it a *conflict-free* schedule. This particular schedule is not conflict free.



STAGE 0 (stride 1)	STAGE 1 (stride 2)	STAGE 2 (stride 4)
dp[0]/SRAM0	dp[0]/SRAM0	dp[0]/SRAM0
dp[1]/SRAM1	dp[2]/SRAM2	dp[4]/SRAM1
dp[2]/SRAM2	dp[4]/SRAM1	dp[2]/SRAM2
dp[3]/SRAM3	dp[6]/SRAM3	dp[6]/SRAM3
dp[4]/SRAM1	dp[1]/SRAM1	dp[1]/SRAM1
dp[5]/SRAM0	dp[3]/SRAM3	dp[5]/SRAM0
dp[6]/SRAM3	dp[5]/SRAM0	dp[3]/SRAM3
dp[7]/SRAM2	dp[7]/SRAM2	dp[7]/SRAM2

Fig. 3. Conflict-free schedule and SRAM map for 8-point data transform operating with group size 4 ($G = 4$, $D = 8$), designed to work with $M = G = 4$ memory banks. (Note slight but necessary variation in Stage-2 datapoint order vs. Fig. 2, explained later in detail.).

Reading Fig. 2 from top to bottom and left to right, the first bracket says that Stage 0 begins by reading operands dp[0] and dp[1]. After processing the operands, two results get written back to the same locations dp[0] and dp[1], overwriting previous contents. The butterfly then reads and subsequently writes back locations dp[2] and dp[3], and so on. A conflict-free layout will need only eight memory cycles per stage: four read pairs and four write pairs. The operand pairs in each stage s must be separated by a distance of 2^s, thus Stage 0 is a stride-1 stage, Stage 1 is stride-2, and so on.

Because $G = BR$, an FFT with two radix-2 butterfly units operating in parallel, or an FFT with a single radix-4 butterfly, would process operands in groups of four instead of groups of two. To illustrate this *group-4 schedule* we place brackets next to groups of four datapoints instead of two, like in Fig. 3.

III. Mapping Datapoints to Memory Banks

Our goal is to map data into single-ported memory in a way that avoids conflicts. When we annotate a schedule to explicitly show these data-point-to-SRAM assignments, we call it a **map**. Our "schedule" of Fig. 2 is really a map, because it explicitly states which SRAM contains what datapoint. It maps out a simple but naïve scheme for two banks of SRAM such that *even* datapoints go to bank 0 and *odd* to bank 1.

Throughout Stage 0, each bracketed group of two operands lives one apiece in the two memory banks; we call these groups *nonredundant*. Starting in Stage 1, however, each bracketed pair of operands lives in the same memory bank; dp[0] and dp[2] both live in bank SRAM0, dp[1] and dp[3] both live in bank SRAM1, etc. Because both operands live in the same bank, and because the bank only has a single port for reading, we cannot access both operands in the same memory cycle. We call this group *redundant*, because at least one memory bank is represented more than once within the group.

Redundant groups cause *conflicts*, where more than one operand needs to be accessed from the same bank at the same time. A schedule like that of Fig. 3, with *no redundant*

[2] Stage-to-stage arrows, added here for clarity, are left off future diagrams.

groups (and consequently no conflicts) is called a *conflict-free schedule*, or CFS. Note Fig. 2 is *not* conflict free.

A. *Conflict-Free Schedule (CFS) with Optimal Memory Area*

Our challenge, then, is to come up with an algorithm that can build a conflict-free schedule for FFT designs using any number of data points D, any number of butterfly units B and any radix of butterfly unit R. Or more simply, because group size $G = BR$, we want an algorithm that works for FFT designs using any number of data points D and any group size G. Clearly a conflict-free schedule for group size G will require at least G single-ported memory banks. Our goal is to use *exactly* G banks so as to produce a *minimum-area design* [5] also known as *matched interleaved memory* [12].

Fig. 3 shows a conflict-free schedule and map for an eight-point data transform operating with a group size of four. We will show how to produce this map, and how the same procedure can generate a map for any number of datapoints, butterflies, groups, etc.

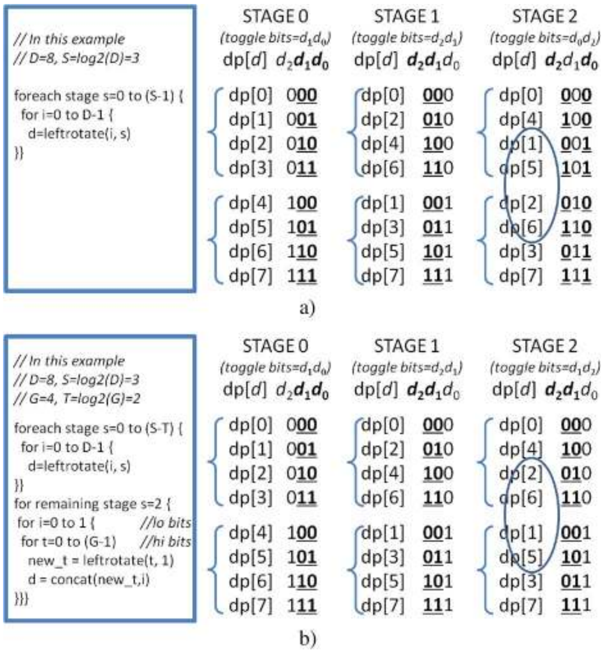
B. *Toggle Bits*

The group-4 schedule of Fig. 4(a) was built using a traditional Cooley-Tukey algorithm [11]. Here, instead of dp[0], dp[1], dp[2], etc., we only list the binary form of the index for each datapoint—000,001,010, etc., still in groups of four.[3] Notice that, within each group of four in every stage, there are two bits that always count 00,01,10,11 (these would be bits d_1d_0 in Stage 0, bits d_2d_1 in Stage 1 and bits d_0d_2 in Stage 2). We call these the *toggle bits*, because within any given group of four, these are the only bits that change, while the other bits remain constant. This is true regardless of the number of data points in the schedule; a group-4 schedule for 4M datapoints would have 1M groups per stage, but each group would still have two toggle bits counting 00,01,10,11 while *the non-toggle bits are constant*.

Why is this important? Because 1) for conflict-free schedules we need non-redundant groups; 2) for non-redundancy, we need to map the four datapoints in each group to the four memory banks 00,01,10,11 in some order; 3) the toggle bits in each group count 00,01,10,11; so 4) *we might achieve our CFS goal by calculating the memory bank for each datapoint as a function of its toggle bits*.

In particular, we know there are $T = (s_2G) = 2$ toggle bits (call them t_1 and t_0) when group size $G = 4$; and we know that there are $\log_2 M$ or two memory-bank bits m_1 and m_0 because we have $M = G = 4$ memory banks. So one thing we could try is simply setting $m_1 = t_1$ and $m_0 = t_0$ to guarantee that each of the four memory banks were represented in each group; i.e., when t_1t_0 counts 00,01,10,11, m_1m_0 would count 00,01,10,11. In Stage 0 of Fig. 4(a) the toggle bits are $t_1t_0 = d_1d_0$. Stage-1 toggle bits are $t_1t_0 = d_2d_1$ and Stage-2 toggle bits are $t_1t_0 = d_0d_2$ (**not** d_2d_0).

If we could simply set the memory bank m_1m_0 equal to the toggle-bit number t_1t_0, we would easily accomplish our goal of nonredundant groups. Unfortunately in Fig. 4(a) that would mean dp[1] maps to SRAM1 in Stage 0 ($m_1m_0 = t_1t_0 = d_1d_0 = 01$), but in Stage 1 it maps to SRAM0 ($m_1m_0 = t_1t_0 = d_2d_1 = 00$). For the FFT to work correctly as an in-place algorithm, each datapoint must live in the same memory location *throughout all stages*.



```
// In this example
// D=8, S=log2(D)=3

foreach stage s=0 to (S-1) {
  for i=0 to D-1 {
    d=leftrotate(i, s)
}}
```

STAGE 0 (toggle bits=d_1d_0)		STAGE 1 (toggle bits=d_2d_1)		STAGE 2 (toggle bits=d_0d_2)	
dp[d]	$d_2d_1d_0$	dp[d]	$d_2d_1d_0$	dp[d]	$d_2d_1d_0$
dp[0]	000	dp[0]	000	dp[0]	000
dp[1]	001	dp[2]	010	dp[4]	100
dp[2]	010	dp[4]	100	dp[1]	001
dp[3]	011	dp[6]	110	dp[5]	101
dp[4]	100	dp[1]	001	dp[2]	010
dp[5]	101	dp[3]	011	dp[6]	110
dp[6]	110	dp[5]	101	dp[3]	011
dp[7]	111	dp[7]	111	dp[7]	111

a)

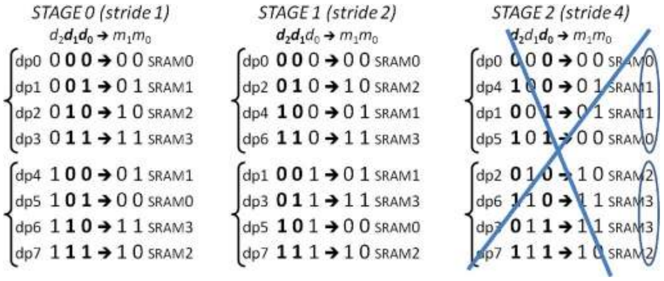
```
// In this example
// D=8, S=log2(D)=3
// G=4, T=log2(G)=2

foreach stage s=0 to (S-T) {
  for i=0 to D-1 {
    d=leftrotate(i, s)
}}
for remaining stage s=2 {
 for i=0 to 1 {          //lo bits
  for t=0 to (G-1)       //hi bits
    new_t = leftrotate(t, 1)
    d = concat(new_t,i)
}}}
```

STAGE 0 (toggle bits=d_1d_0)		STAGE 1 (toggle bits=d_2d_1)		STAGE 2 (toggle bits=d_1d_2)	
dp[d]	$d_2d_1d_0$	dp[d]	$d_2d_1d_0$	dp[d]	$d_2d_1d_0$
dp[0]	000	dp[0]	000	dp[0]	000
dp[1]	001	dp[2]	010	dp[4]	100
dp[2]	010	dp[4]	100	dp[2]	010
dp[3]	011	dp[6]	110	dp[6]	110
dp[4]	100	dp[1]	001	dp[1]	001
dp[5]	101	dp[3]	011	dp[5]	101
dp[6]	110	dp[5]	101	dp[3]	011
dp[7]	111	dp[7]	111	dp[7]	111

b)

Fig. 4. Cooley-Tukey algorithm (a), modified to keep toggle bits adjacent (b). $D = 8$ and $G = RB = 4$. a) Schedule based on original Cooley-Tukey: toggle bits are non-adjacent in final stage. This can be remedied by e.g., rotating the lower bits d_1d_0 in Stage 2. b) Modified schedule: toggle bits always adjacent. Note Stage-2 rotation (swap) of d_1d_0 vs. original algorithm.



STAGE 0 (stride 1)				STAGE 1 (stride 2)				STAGE 2 (stride 4)			
	$d_2d_1d_0$	$\to m_1m_0$			$d_2d_1d_0$	$\to m_1m_0$			$d_2d_1d_0$	$\to m_1m_0$	
dp0	0 0 0	→ 0 0	SRAM0	dp0	0 0 0	→ 0 0	SRAM0	dp0	0 0 0	→ 0 0	SRAM0
dp1	0 0 1	→ 0 1	SRAM1	dp2	0 1 0	→ 1 0	SRAM2	dp4	1 0 0	→ 0 1	SRAM1
dp2	0 1 0	→ 1 0	SRAM2	dp4	1 0 0	→ 0 1	SRAM1	dp1	0 0 1	→ 0 1	SRAM1
dp3	0 1 1	→ 1 1	SRAM3	dp6	1 1 0	→ 1 1	SRAM3	dp5	1 0 1	→ 0 0	SRAM0
dp4	1 0 0	→ 0 1	SRAM1	dp1	0 0 1	→ 0 1	SRAM1	dp2	0 1 0	→ 1 0	SRAM2
dp5	1 0 1	→ 0 0	SRAM0	dp3	0 1 1	→ 1 1	SRAM3	dp6	1 1 0	→ 1 1	SRAM3
dp6	1 1 0	→ 1 1	SRAM3	dp5	1 0 1	→ 0 0	SRAM0	dp3	0 1 1	→ 1 1	SRAM3
dp7	1 1 1	→ 1 0	SRAM2	dp7	1 1 1	→ 1 0	SRAM2	dp7	1 1 1	→ 1 0	SRAM2

Fig. 5. Toggle-based map successfully provides non-redundant groups for Stages 0 and 1. Stage 2 still has conflicts; its non-adjacent toggle bits d_2 and d_0 mean only two SRAMs per group of four.

The problem is that the toggle bits change for each stage. Since the datapoint-to-memory-bank mapping must remain the same regardless of what stage we're in, we need to build a mapping such that 1) the memory bank number is always a function of the two toggle bits and 2) the datapoint-to-memory-bank mapping stays the same regardless of stage. When $G = 4$, *and for each stage where the toggle bits are adjacent*, the following equation maps exactly four different memory banks $m = m_1m_0$ to each group of four datapoints:

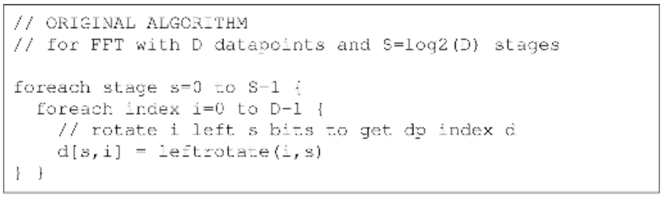
$$m_1 = \text{XOR}(d_1, d_3, d_5, \ldots) \text{ and } m_0 = \text{XOR}(d_0, d_2, d_4, \ldots)$$

or, for our example when $D = 8$ datapoints and each datapoint index d only has three bits $d = d_2d_1d_0$,

$$m_1 = \text{XOR}(d_1) = d_1 \text{ and } m_0 = \text{XOR}(d_0, d_2)$$

This gives us the schedule and map of Fig. 5. We see that this is now a conflict-free schedule throughout Stages 0 and 1, while Stage 2 still has conflicts.

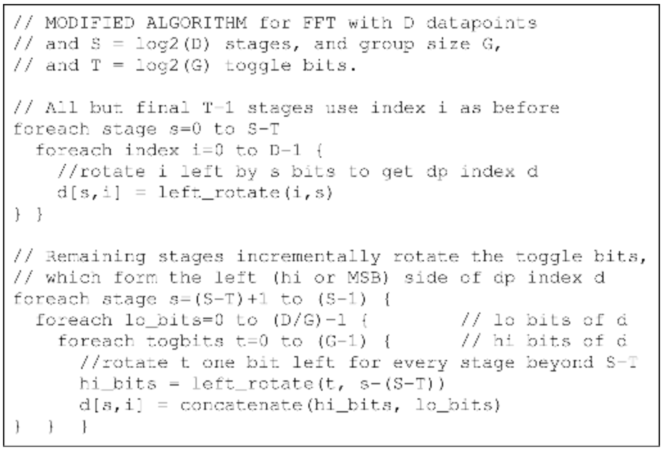
[3] We denote the individual bits of an integer using subscripts, e.g., a three-bit integer i is composed of three bits $i_2i_1i_0$. We use the convention that the least-significant bit (LSB) is the rightmost bit, and it has subscript 0.



```
// ORIGINAL ALGORITHM
// for FFT with D datapoints and S=log2(D) stages

foreach stage s=0 to S-1 {
  foreach index i=0 to D-1 {
    // rotate i left s bits to get dp index d
    d[s,i] = leftrotate(i,s)
} }
```

Fig. 6. Original Cooley-Tukey-based algorithm. It produces non-adjacent toggle bits in the later stages.

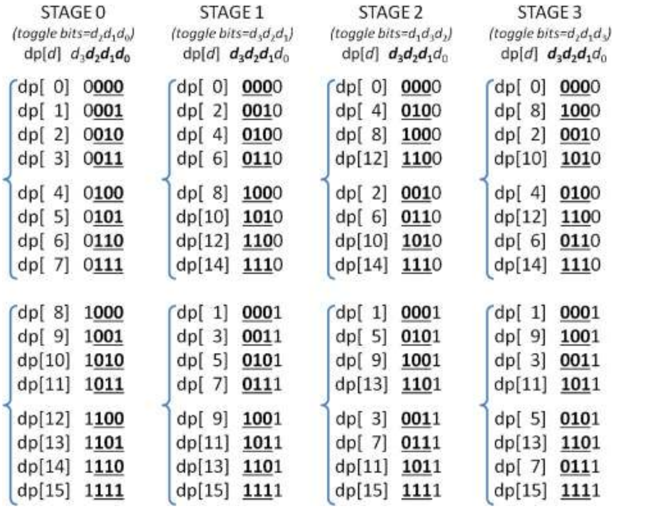


```
// MODIFIED ALGORITHM for FFT with D datapoints
// and S = log2(D) stages, and group size G,
// and T = log2(G) toggle bits.

// All but final T-1 stages use index i as before
foreach stage s=0 to S-T
  foreach index i=0 to D-1 {
    //rotate i left by s bits to get dp index d
    d[s,i] = left_rotate(i,s)
} }

// Remaining stages incrementally rotate the toggle bits,
// which form the left (hi or MSB) side of dp index d
foreach stage s=(S-T)+1 to (S-1) {
  foreach lo_bits=0 to (D/G)-1 {          // lo bits of d
    foreach togbits t=0 to (G-1) {        // hi bits of d
      //rotate t one bit left for every stage beyond S-T
      hi_bits = left_rotate(t, s-(S-T))
      d[s,i] = concatenate(hi_bits, lo_bits)
}   }   }
```

Fig. 7. Our new algorithm keeps the toggle bits adjacent in all stages. This is the algorithm from Fig. 4(b), generalized to any group size G.



STAGE 0		STAGE 1		STAGE 2		STAGE 3	
(toggle bits=$d_2d_1d_0$)		*(toggle bits=$d_3d_2d_1$)*		*(toggle bits=$d_1d_3d_2$)*		*(toggle bits=$d_2d_1d_3$)*	
dp[d]	$d_3\boldsymbol{d_2d_1d_0}$	dp[d]	$\boldsymbol{d_3d_2d_1}d_0$	dp[d]	$\boldsymbol{d_3d_2d_1}d_0$	dp[d]	$\boldsymbol{d_3d_2d_1}d_0$
dp[0]	0**000**	dp[0]	**000**0	dp[0]	**000**0	dp[0]	**000**0
dp[1]	0**001**	dp[2]	**001**0	dp[4]	**010**0	dp[8]	**100**0
dp[2]	0**010**	dp[4]	**010**0	dp[8]	**100**0	dp[2]	**001**0
dp[3]	0**011**	dp[6]	**011**0	dp[12]	**110**0	dp[10]	**101**0
dp[4]	0**100**	dp[8]	**100**0	dp[2]	**001**0	dp[4]	**010**0
dp[5]	0**101**	dp[10]	**101**0	dp[6]	**011**0	dp[12]	**110**0
dp[6]	0**110**	dp[12]	**110**0	dp[10]	**101**0	dp[6]	**011**0
dp[7]	0**111**	dp[14]	**111**0	dp[14]	**111**0	dp[14]	**111**0
dp[8]	1**000**	dp[1]	**000**1	dp[1]	**000**1	dp[1]	**000**1
dp[9]	1**001**	dp[3]	**001**1	dp[5]	**010**1	dp[9]	**100**1
dp[10]	1**010**	dp[5]	**010**1	dp[9]	**100**1	dp[3]	**001**1
dp[11]	1**011**	dp[7]	**011**1	dp[13]	**110**1	dp[11]	**101**1
dp[12]	1**100**	dp[9]	**100**1	dp[3]	**001**1	dp[5]	**010**1
dp[13]	1**101**	dp[11]	**101**1	dp[7]	**011**1	dp[13]	**110**1
dp[14]	1**110**	dp[13]	**110**1	dp[11]	**101**1	dp[7]	**011**1
dp[15]	1**111**	dp[15]	**111**1	dp[15]	**111**1	dp[15]	**111**1

Fig. 8. Schedule produced by modified algorithm (Fig. 7) when $D = 16$ and $G = 8$. In every stage s: 1) toggle bits are adjacent, so the resulting SRAM map will be non-redundant and consequently conflict free; and 2) the fastest-toggling toggle bit (LSB) is bit position d_s so Stage 0 is stride 1, Stage 1 is stride 2, and so on. LSB rotation goes: $t_2t_1\boldsymbol{t_0} = d_2d_1\boldsymbol{d_0} \rightarrow d_3d_2\boldsymbol{d_1} \rightarrow d_1d_3\boldsymbol{d_2} \rightarrow d_2d_1\boldsymbol{d_3}$.

Unfortunately, the Stage-2 toggle bits d_0 and d_2 are *non-adjacent*. And while $m_0 = \text{XOR}(d_0, d_2)$ is a function of *both* toggle bits, $m_1 = d_1$ only depends on the non-toggle-bit d_1, and we need each memory bank bit to be a function of the toggle bits. We can fix this by altering the schedule such that toggle bits are always adjacent, regardless of stage.

Consider the alternate schedule for an eight-point FFT shown in Fig. 4(b). Here we've swapped the least-significant two bits d_1d_0 of each datapoint number in Stage 2, so that the circled portion of Stage 2 now counts dp[2,6,1,5] instead of the original dp[1,5,2,6]. *This is still a valid Stage-2 schedule* because each pair of datapoints is separated by stride 4. But now, instead of Stage 2 having non-adjacent toggle bits $t_1t_0 = d_0d_2$, we have *adjacent* toggle bits $t_1t_0 = d_1d_2$.

As it happens, our new Stage-2 ordering is the same as Stage 1 except the toggle bits (the first two bits) have been rotated. This rotation yields a reordering that, combined with the previously shown XOR-mappings for m_1 and m_0, gives the desired conflict-free result originally shown in Fig. 3.

We can use this toggle-rotation trick to generate a schedule for *any number of datapoints D* and *any group size G* such that the toggle bits within each group are always adjacent. We begin with the Cooley-Tukey algorithm for producing a standard FFT schedule [11], shown in Fig. 6. We modify this to account for group size (Fig. 7), producing the toggle-normal schedule as before, up until the stage at which the toggle bits would have wrapped to become non-adjacent. At that point, the algorithm switches to the rotated-toggle form.

Fig. 8 shows the schedule our algorithm produces for a 16-point transform with group size of eight. Instead of counting 000,001,010,011 with LSB as the rightmost bit, rotated Stage-2 toggle bits count 000,010,100,110 with the LSB as the middle bit, which doubles the stride vs. the previous Stage 1, while still preserving toggle-bit adjacency. Further-rotated Stage-3 toggle bits count 000,100,001,101 with the LSB as the leftmost bit, again doubling the stride versus the previous stage. Thus for every stage s, the LSB of the toggles $t_0 = d_s$ so Stage 0 is stride 1, Stage 1 is stride 2, etc.

C. General Algorithm for Producing Nonredundant Groups

So, to produce a conflict-free group-G mapping for D datapoints: First, use the algorithm of Fig. 7 to create a valid group-G schedule with adjacent toggle bits. Then, for each S-bit datapoint $d = d_{(S-1)}d_{(S-2)}\ldots d_1d_0$ calculate a T-bit memory bank number m such that

$$
\begin{aligned}
m &= m_{(T-1)}m_{(T-2)}\ldots m_2m_1m_0 \text{ where} \\
&m_0 = \text{XOR}(d_0, d_{(T)}, d_{(2T)}, d_{(3T)}, \ldots) \\
&m_1 = \text{XOR}(d_1, d_{(T+1)}, d_{(2T+1)}, d_{(3T+1)}, \ldots) \\
&m_2 = \text{XOR}(d_2, d_{(T+2)}, d_{(2T+2)}, d_{(3T+2)}, \ldots) \\
&\ldots \\
&m_{(T-2)} = \text{XOR}(d_{(T-2)}, d_{(2T-2)}, d_{(3T-2)}, d_{(4T-2)}, \ldots) \\
&m_{(T-1)} = \text{XOR}(d_{(T-1)}, d_{(2T-1)}, d_{(3T-1)}, d_{(4T-1)}, \ldots)
\end{aligned}
\tag{1}
$$

and where

T the number of toggle bits, is equal to $\log_2 G$ and

d is an S-bit datapoint $d_{(S-1)}d_{(S-2)}\ldots d_3d_2d_1d_0$ such that

S the number of stages, is equal to $\log_2 D$ and

D is the number of datapoints to be transformed.

Like Takala's algorithm that inspired this work [7], a hardware implementation for mapping address d to bank m requires only T XOR gates each with a fan-in of S/T. Thus the same logic can accommodate FFTs of any variable length D simply by designing for the maximum size D_{MAX} and using 0's for the high bits $d_Sd_{S+1}d_{S+2}\ldots d_{S_{\text{MAX}}}$ when $D < D_{\text{MAX}}$. Moreover, the reordering step (discussed later in more detail) entirely eliminates the earlier work's need for a "rotation unit" in the mapping hardware.

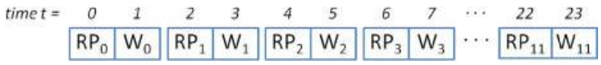


Fig. 9. Two-stage pipeline with no overlap.

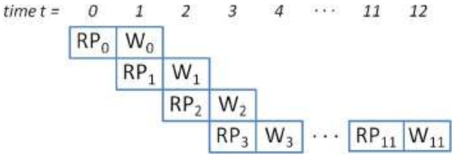


Fig. 10. Two-stage pipeline with overlap of 1. This *overlapped* pipeline can do two reads and two writes in a single cycle. Its two-stage RP/W pipe only takes 13 cycles to complete the $D = 8$ point transform. Because it accesses four datapoints at a time, it wants a schedule with group size $G = 4$.

IV. FFT With Pipeline Overlap

The algorithm as developed so far works only for FFTs without *pipeline overlap*, a term we shall soon explain. For greater performance and efficiency, designers typically prefer *overlapped* designs [13]. To extend our algorithm to overlapped designs, we need to understand their access patterns and how they differ from non-overlapped designs.

Our original example of Fig. 2 represents a schedule for a simple FFT with a single radix-2 butterfly. Using a simple two-stage *non-overlapped* read-process/write (or RP/W) pipeline, the FFT 1) **R**eads its first two operands dp[0] and dp[1] from memory and **P**rocesses them; 2) **W**rites the two results back to memory locations dp[0] and dp[1]; and then starts over again by reading the next two operands. Each complete RP/W butterfly operation takes two cycles, times twelve butterfly operations means that a complete 8-point transform requires 24 cycles. We call this a *2-stage 0-overlap pipeline*, shown graphically as the diagram of Fig. 9.

Fig. 10 shows the same FFT, this time reconfigured as an *overlapping* RP/W pipelined design. In the first three cycles of operation this FFT 1) reads its first two operands from memory and processes them; 2) writes the two results back to memory while *at the same time* fetching the next two operands and processing them; then 3) repeats the previous write/read-process cycle with the next two operands and so on until done. After the read in the first cycle, the pipeline is full, and each write/read-process combination thereafter takes only *one* cycle to complete. The entire 8-point transform thus now takes only 13 total cycles instead of the previous 24: one RP cycle to load the pipe, and then one cycle for each of twelve successive W/RP butterfly operations.

A group-2 schedule will not suffice for this *2-stage* 1-overlap pipeline, which now accesses four locations at once, for instance writing dp[0,1] while reading dp[2,3]. It will need a group-4 schedule.

There are many other ways to construct an FFT pipeline. Our original design described in Section II had a three-stage non-overlapping pipeline aka a *3-stage 0-overlap pipeline*. Meanwhile, we could just as easily construct an overlapped design that takes multiple cycles to complete the butterfly operation, like the five-stage pipeline shown in Fig. 11.

A. Ordered Groups Create CFS for Overlapped Pipelines

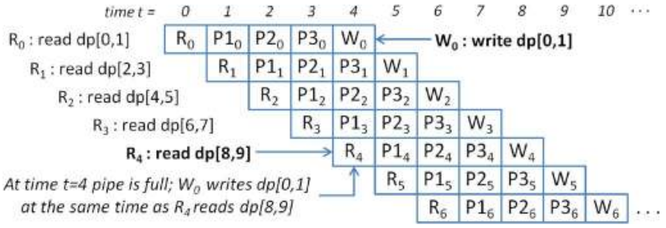


Fig. 11. Five-stage overlapping pipeline.

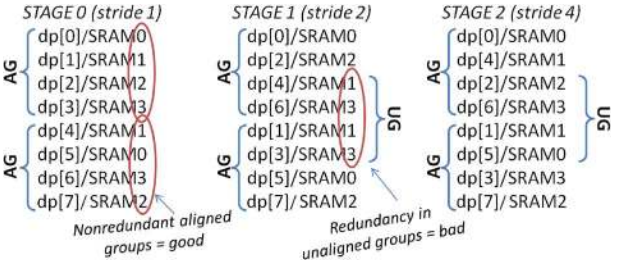


Fig. 12. The four operands in each aligned group AG map one apiece to the four memory banks. Unfortunately, unaligned groups UG in Stages 1 and 2 use only two memory banks each, that is, these groups are *redundant*. For the schedule to work with overlapping pipelines, all groups of G operands must be nonredundant, whether aligned or unaligned. In this example, $D = 8$ and $G = M = 4$.

FFT's with overlapping pipelines pose a special challenge for conflict-free scheduling. As mentioned earlier, a non-overlapping 2-stage pipeline can be satisfied with a group-2 schedule, but an overlapping 2-stage pipe needs a group-4 schedule. Not a group-4 schedule like that of Fig. 3, however; it needs an *overlapping* group-4 schedule.

While the original group-4 schedule accesses non-overlapping groups of four operands dp[0,1,2,3] then dp[4,5,6,7] then dp[8,9,10,11] and so on, the new *overlapping* group-4 schedule must accommodate overlapping groups of operands dp[0,1,2,3] then dp[2,3,4,5] then dp[4,5,6,7] and so on, where the last two datapoints of one group overlap the first two of the next group. The original non-overlapping groups beginning with dp[0] in each stage—dp[0,1,2,3], dp[4,5,6,7], and so on—are called *aligned* groups, while the new overlap groups dp[2,3,4,5], dp[6,7,8,9] are *unaligned* groups.

We show unaligned groups in the schedule with overlapping brackets to the right of each column. Once we do this for one of our schedules with maps, as in Fig. 12, we immediately see a problem. While the original non-overlapping (or *aligned*) groups (bracketed on the left side of each column) all map to non-redundant groups of four memory banks, the new overlapping (*unaligned*) groups (bracketed on the right) do not. And for a conflict-free schedule, we need for *all* the groups to be non-redundant, including the overlap groups.

To be clear: sequential groups of G operands beginning at cycle 0 in each stage of a schedule are **aligned** groups. Any other group of G or fewer operands in a schedule is an **unaligned** group. A schedule with nonredundant *aligned* groups is conflict-free for *non-overlapping* pipelines only. But if we could build a schedule with nonredundant *unaligned* groups, it will be conflict-free for all pipelines, overlapped and non-overlapped.

We are going to take a very simple approach toward achieving this goal. Focusing only on the aligned groups, we have solved the problem of redundancy, such that each group is nonredundant. Now, we add a further constraint: each aligned group within a stage must not only be *nonredundant*, but it must also be *strictly ordered*. That is, each of the four datapoints in an aligned group must map one-for-one to the four memory banks, *and* they must map to those memory banks always in the same order.

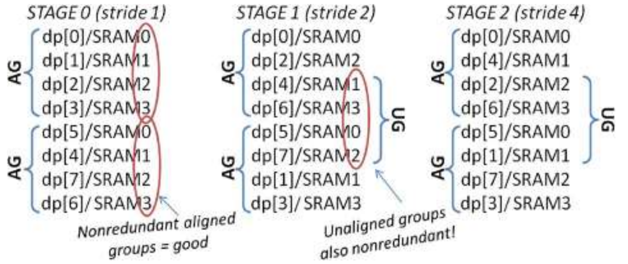


Fig. 13. We sorted the four operands in each aligned group AG so as to make them *strictly ordered* per stage. As a result, unaligned groups UG are now guaranteed to be nonredundant. $D = 8$ and $G = M = 4$.

In the schedule of Fig. 12, produced by our algorithm as developed so far, the SRAM mapping for each group is nonredundant, but the SRAM sequence within each group is *unordered* with respect to neighboring groups. Again: to work for overlapping pipelines, the schedule needs groups that are both nonredundant and *strictly ordered* within each stage.

Fig. 13 shows the result of just such a map, where, e.g., each aligned group in Stage 0 has been *ordered* such that the datapoints map to SRAM0, 1, 2, 3 *in that order*. The special consequence of making ordered aligned groups is that now *all unaligned groups are also ordered* and therefore nonredundant and therefore conflict-free.

When group size $G = 4$ as in Fig. 13, this mapping means that aligned groups in even-numbered stages (Stage 0, 2, 4, ...) keep bank order SRAM0,1,2,3 while odd-numbered stages are ordered SRAM0,2,1,3. And now, because the aligned groups are strictly ordered, this means that *the unaligned groups are also strictly ordered*: unaligned groups in even stages are ordered SRAM2,3,0,1 and unaligned groups in odd stages are ordered SRAM1,3,0,2.[4]

This reordering is possible because datapoint pairs can be processed *in any order* so long as the stride relationship within each pair is preserved, i.e., stride 1 for Stage 0, stride 2 for Stage 1 and so on, i.e., the operand pairs in each stage s must be separated by 2^s. (Remember, we already changed the order of the datapoints once before, when our algorithm moved from producing the schedule of Fig. 4(a) to the schedule of Fig. 4(b). The datapoint order in Stage 2 changed slightly, but the result of the transform is the same for either schedule.)

The extension of this principle to arbitrarily deep pipelines is given in Section IV-C.

B. Producing Ordered Groups

To produce a schedule with strictly ordered, aligned groups of size G, then, we use the following *generate-map-reorder* sequence:

1) **Generate** a base schedule with adjacent toggle groups using the modified Cooley-Tukey algorithm of Fig. 7.
2) **Map** datapoints to memory banks according to the simple parity mapping (1) at the end of Section III.
3) **Reorder** the datapoints such that the memory banks within each group follow the same strict order.

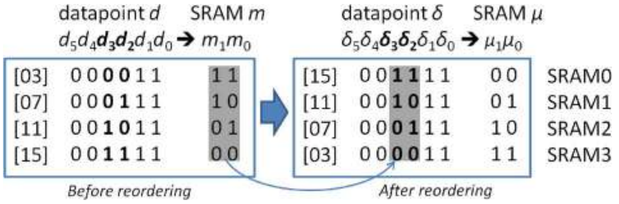


Fig. 14. Middle of Stage 2 (stride-4 stage) of a 64-point transform. **Before reordering**: As d counts 3,7,11,15, toggle bits d_3d_2 count 00,01,10,11 while non-toggle bits d_5d_4, d_1d_0 remain constant. The four datapoints map to four separate memory banks, but the memory banks are not in sequential order. **After reordering**: δ counts 15,11,7,3, and reordered toggle bits $\delta_3\delta_2$ count 11,10,01,00 while non-toggle still constant (00)(11). In the end, the four datapoints δ map to four separate memory banks μ *in sequential order*. In this example $D = 64$ and $G = M = 4$.

The previously-discussed *generate* and *map* steps produce a sequence of datapoints d similar to that shown in Fig. 3, whose map has groups with *unordered* memory banks. The goal of the new *reorder* step is to rearrange each group's sequence d to produce a new sequence δ such that the memory banks in each group are strictly *ordered*.

We begin the *reorder* process by observing that the original memory bank order in any given group is strictly determined by the *toggle bits* of the datapoints in that group. This should be obvious considering the fact that, as we noted earlier, only the toggle bits differ from datapoint to datapoint within any given aligned group of G operands. The toggle bits count from 0 to (G-1) in strict numeric order, while the corresponding memory banks m count from 0 to (G-1) in scrambled order determined by the XOR functions in (1) at the end of Section III. For example, Fig. 14 shows a group of four datapoints in the middle of Stage 2 of a 64-point transform. Here the toggle bits are two bits d_3d_2 in the middle of the datapoint index d. As d counts 3,7,11,15, the toggle bits count 00,01,10,11.

If we take the toggle bits d_3d_2=(00,01,10,11) of Fig. 14 and change their order to match the calculated memory bank bits $m_1m_0 = (11, 10, 01, 00)$, we get the new datapoint order $\delta = \text{dp}[15, 11, 7, 3]$. If we now perform our d-to-m mapping (1) on this new order we get the desired canonical memory-bank order $\mu_1\mu_0 = (00, 01, 10, 11) = \text{SRAM}(0, 1, 2, 3)$. In other words, to get the new sequence δ we replaced each toggle bit t_i in each datapoint index d with memory-bank bit m_i.

The reordering works because in Stage 2 m_1 is a function of t_1 and m_0 is a function of t_0. Remember, $t_1 = d_3$ and $t_0 = d_2$, and so from (1) we get $m_1 = \text{XOR}(d_1, d_3, d_5) = f(t_1)$ and $m_0 = \text{XOR}(d_0, d_2, d_4) = f(t_0)$.

Unfortunately, this does not hold true for every stage. In fact, when group size $G = 4$ as in our example, m_1 is a function of t_1 only for even-numbered stages. Remember $t_0 = d_s$ so in even-numbered stages $t_0 \in \{d_0, d_2, d_4 \ldots\}$ and thus $t_1 \in \{d_1, d_3, d_5 \ldots\}$. In (even-numbered) Stage 2, $t_1 = d_3$ and $m_1 = f(d_1, d_3, d_5) = f(t_1)$, and we achieve the desired final order SRAM0,1,2,3. In, e.g., (odd-numbered) Stage 1, however, we would have $t_1 = d_2$ and $m_1 = f(d_1, d_3, d_5) \neq f(t_1)$, and reordering would fail to achieve the desired result.

So instead of replacing each toggle bit t_i with memory bank m_i, let us replace t_i with $m_{(s+i) \bmod T}$. We will do this because we know that, depending on stage number s, it may or may not be true that m_i is a function of toggle bit t_i. However, because t_i for each stage s is datapoint bit $d_{(s+i)}$, and because $m_{(s+i) \bmod T}$ is always a function of $d_{(s+i)}$, we therefore know

[4] *Note* conflicts still exist when an unaligned group crosses an interstage boundary. E.g., in Fig. 13 the last two accesses dp[7,6] of Stage 0 use SRAMs 2,3 while the first two accesses dp[0,2] of Stage 1 access SRAMs 0,2. SRAM2 gets used twice in this same unaligned group of 4, thus the conflict. Our earlier paper [5] explains why such conflicts are rare, and describes a simple way to prevent them from impacting performance.

that $m_{(s+i)\bmod T}$ is always a function of t_i regardless of stage number s.

To be even more precise, we must take into account the fact that, in the final stages $(S-T+1)$ through $(S-1)$ of a schedule produced by our *generate* algorithm (Fig. 7), toggle bits are aligned at the top (MSB end) of the data word. For these stages, instead of $t_i = d_{(s+i)}$, it is the case that the toggle bits are the top $T-1$ bits $(d_{S-1}, d_{S-2}, \ldots d_{S-T+1})$ i.e., $t_i \in \{d_{S-(i+1)}\}$. Thus the final reordering procedure is

Reordering

$$\begin{array}{ll} \forall s \in (0..S-T) & \{\forall\, i \in (0..D-1)\delta_{s,i} = \mathrm{LREP}(d_{s,i})\} \\ \forall s \in (S-T+1..S-1) & \{\forall i \in (0..D-1)\delta_{s,i} = \mathrm{RREP}(d_{s,i})\} \end{array} \quad (2)$$

where $d_{s,i}$ is a datapoint calculated using the algorithm of Fig. 7, and $S = \log_2 D$=no. of transformation stages, and

$\mathrm{LREP}(d)$

$\Rightarrow \forall i \in (0..T-1)$ replace $d_{(s+i)}$ with $m_{((s+i)\bmod T)}$,

$\mathrm{RREP}(d)$

$\Rightarrow \forall i \in (1\ldots T)$ replace $d_{(S-i)}$ with $m_{((S-i)\bmod T)}$,

where s is stage number and T is number of toggle bits.

This is the *reorder* part of the *generate-map-reorder* sequence we introduced at the beginning of this subsection. The complete procedure appears more formally in Appendix A where A1, A2 give the *generate* and *reorder* formulas for non-overlapping and overlapping pipelines respectively, while Sec. A3 recaps the *map* portion common to both.

Fig. 15 shows how the process works for a more complicated reordering. This is a schedule and map for part of Stage 3 in a 256-point transform with group size $G = 8$. Highlighted columns and arrows explain the two-step process, whereby

1) Every n^{th} bit of datapoint index d gets XOR'ed together to form one of the memory bank bits m_i. The top-left quadrant of Fig. 15 illustrates how m_1 is formed by XORing datapoint bits d_7, d_4 and d_1 (i.e., very third bit starting with d_1).
2) Memory bits replace toggle bits to produce a new ordering for the datapoints. The bottom half of Fig. 15 shows how bits $m_0m_2m_1$ replace toggle bits $d_6d_5d_4$ to form bits $\delta_6\delta_5\delta_4$.

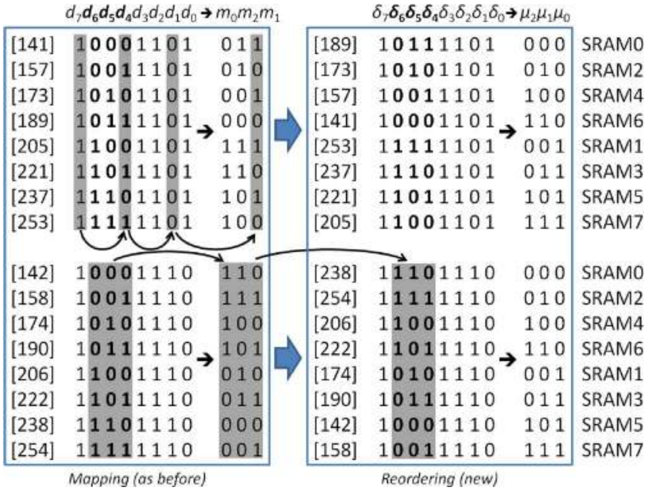
The net effect is that the original stride-16 order $d = (141, 157, \ldots, 254)$ is replaced by a new, scrambled order $\delta = (189, 173, \ldots, 158)$ such that the final memory bank order is a repeating series (SRAM0,2,4,6,1,3,5,7).

C. Extension to Arbitrarily Deep Pipelines

Earlier, we showed how to create a conflict-free schedule for a 2-stage pipeline with overlap of 1. Here, we show how the same principle can be used to create schedules for arbitrarily deep pipelines.

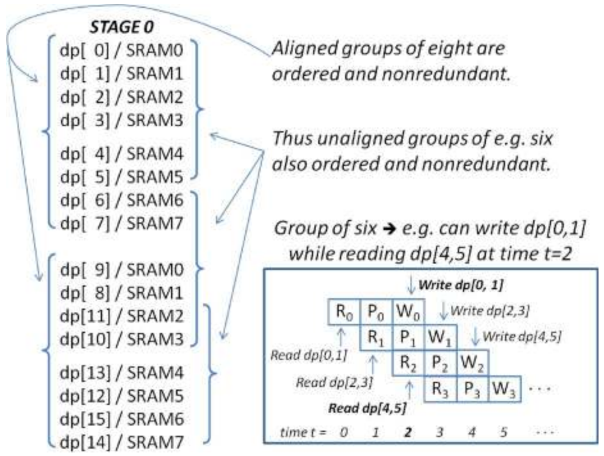
For instance, say we have a three-stage pipeline with overlap of two (Fig. 16). Such a pipeline would write operands dp[0,1] while reading operands dp[4,5] and would thus need nonredundant groups of six operands. Because our algorithm only produces schedules for power-of-two group sizes, we create a group-8 schedule whose aligned groups are *strictly ordered* as shown in Fig. 16. Because the aligned groups are ordered, this means that all groups of G or smaller, whether aligned or not, are nonredundant. In particular, all unaligned groups of six operands are nonredundant, meaning, e.g., that the first and last pairs in each group of six map to four separate memory banks and thus all four operands can be accessed simultaneously, which is what we need for this particular pipeline to work conflict-free.

The algorithm thus extends to any pipeline depth P with overlap $P-1$ simply by building a group-G schedule with $M = G$ memory banks where, as in this example, G is the largest power-of-2 equal to or greater than $2P$ or (as we shall see) BRP where $B = 1$ and $R = 2$.



	$d_7d_6d_5d_4d_3d_2d_1d_0$ → $m_0m_2m_1$			$\delta_7\delta_6\delta_5\delta_4\delta_3\delta_2\delta_1\delta_0$ → $\mu_2\mu_1\mu_0$		
[141]	10001101	011	[189]	10111101	000	SRAM0
[157]	10011101	010	[173]	10101101	010	SRAM2
[173]	10101101	001	[157]	10011101	100	SRAM4
[189]	10111101	000	[141]	10001101	110	SRAM6
[205]	11001101	111	[253]	11111101	001	SRAM1
[221]	11011101	110	[237]	11101101	011	SRAM3
[237]	11101101	101	[221]	11011101	101	SRAM5
[253]	11111101	100	[205]	11001101	111	SRAM7
[142]	10001110	110	[238]	11101110	000	SRAM0
[158]	10011110	111	[254]	11111110	010	SRAM2
[174]	10101110	100	[206]	11001110	100	SRAM4
[190]	10111110	101	[222]	11011110	110	SRAM6
[206]	11001110	010	[174]	10101110	001	SRAM1
[222]	11011110	011	[190]	10111110	011	SRAM3
[238]	11101110	000	[142]	10001110	101	SRAM5
[254]	11111110	001	[158]	10011110	111	SRAM7

Fig. 15. Generating the schedule for a couple of groups in the middle of Stage 3 (stride 16) in a 256-point transform where $G = 8$. Four highlighted columns in the top group show how $m_1 = \mathrm{XOR}(d_7, d_4, d_1)$. Highlighted columns in the bottom group show how toggle bits $d_6d_5d_4$ get replaced by $m_0m_2m_1$ to form the new datapoint sequence. The final SRAM order is consistent (0,2,4,6,1,3,5,7) for each group. Note transformations are the same for both groups, we simply highlighted them differently in top and bottom to show the two steps of the transformation sequence.



Aligned groups of eight are ordered and nonredundant.

Thus unaligned groups of e.g. six also ordered and nonredundant.

Group of six → e.g. can write dp[0,1] while reading dp[4,5] at time t=2

Fig. 16. Ordered groups allow arbitrary pipeline depth and overlap. This three-deep pipeline requires a group $G \geq 6$ schedule because it accesses, e.g., dp[0,1] at the same time it accesses dp[4,5]. We use $G = 8$ because the algorithm only works for power-of-two group sizes. So in this example $G = 8$ and $D = 16$.

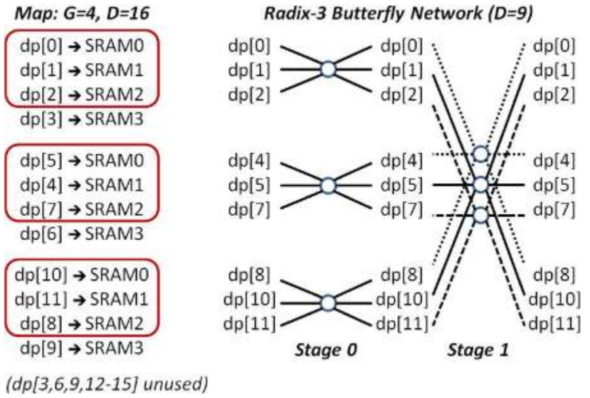


Fig. 17. To produce a CFS for e.g., $D = 9$ and $R = 3$, use a group-4 schedule ($D = 16$ and $G = 4$) and ignore extraneous data locations.

D. Extension to Other Radices

The extension of our algorithm to transforms with mixed radix and/or non-power-of-two radices is fairly straightforward. Simply use a group-G schedule, where G is the next power-of-two greater than or equal to the largest radix R being used, and distribute the operands such that each butterfly uses one group, with $(G - R)$ unused data locations per group when/if R is less than G.

For example, when using radix-3 butterfly units to transform nine datapoints, we could use a group-4 schedule for sixteen datapoint locations and map the nine datapoints to dp[0,1,2], dp[4,5,6], and dp[8,9,10], respectively. That is, we put three datapoints in each group of $G = 4$ datapoint locations, while one datapoint location in each group goes unused. Note that while the resulting schedule will indeed be conflict-free, the packing is suboptimal, using four SRAMs with four data-words each, or sixteen datapoint locations for just nine datapoints. But we can do better than that.

A better approach is to use only data locations that map to SRAMs 0, 1, and 2. Fig. 17 shows a group-4 map for sixteen datapoints, and the subsequent radix-3 butterfly network using it to produce a CFS for transforming nine datapoints. The nine datapoints thus use locations dp[0,1,2], dp[5,4,7], and dp[10,11,8], which map to only three SRAMs instead of four.

Thus the algorithm further extends to any radix R, simply by using $M = R$ memory banks along with a group-G schedule where G is the largest power-of-2 equal to or greater than R.

V. Background and Related Work

Pease [14] presented a clever way of assigning operands to memory banks such that operands can be accessed without conflict within a given FFT stage. To keep running smoothly from one stage to the next, however, requires double buffering: a redundant second block of memory that can be "unloaded and reloaded" while operating on the first block and vice versa.

Johnson [15] used an in-place algorithm to produce conflict-free mappings, and one that works for any given radix of butterfly, not just radix-2. However (according to Takala [7]) the algorithm does not extend to multiple butterflies operating in parallel. Also, Johnson does not discuss how to avoid intercycle conflicts that result from pipeline overlap, i.e., where the write-back of results from a previous cycle interferes with the reading of a new set of operands.

Ma [16] improves on these earlier schemes, with a way to map operands to memory banks using less computation and thus speeding the address generation. Ma reduces address-generation time down to 7 gate delays from Johnson's original 12. Ma's scheme requires a minimum of two banks of two-port memory plus a bypass buffer to hold an extra value between successive computations and, like Johnson, does not eliminate intercycle conflicts.

Chang [17] avoids conflicts by using three 2-port RAMs that can read-and-then-write a given address in one clock cycle. Each RAM holds $D/2$ data points, thus over-provisioning by $D/2$. The extra memory is used to buffer intermediate results and prevent collisions, by writing the data back in a slightly different order than the original read.

Hidalgo [6] goes a step further to include parallel butterfly computations, requiring a complex "perfect unshuffle" interconnection network for reordering, as well as a serial-in parallel-out (SIPO) delay line to further prevent stalls.

Takala [7] extends Johnson's work, providing conflict-free mappings for an arbitrarily large number of operand pairs being either read or written in a given cycle, for any number of butterfly units operating in parallel, and for any given radix of butterfly unit, without requiring the additional registers of Hidalgo. However, there is no discussion about how to avoid intercycle conflicts.

Inspired by Takala's algorithm, the scheme presented herein uses near-minimal memory to implement FFTs with or without pipeline overlap. The resulting schedule lets the FFT operate in near-minimal time with no intracycle, intercycle, or inter-stage memory conflicts. At constant throughput, FFTs based on this work are strictly smaller than implementations based on earlier algorithms; at constant area, they match or exceed the throughput [5]. So the resulting design space is pareto-optimal in area and throughput relative to prior work.

We say "near-minimal" rather than "minimal" memory and time because of extremely small (relative to data memory) per-butterfly bypass buffers required for maximum performance in pipelined implementations only. The buffers make the practical pipelined version of the algorithm *truly* conflict-free, rather than *almost* conflict free, at least in our implementation [5]. Without the buffers, a pipelined 1024-point FFT with one butterfly unit might have 10 conflicts (depending on pipeline depth), resulting in 5130 cycles of operation instead of 5120, a difference of 0.2%. With the buffer, the algorithm completes in minimal time, but the buffer increases total memory size by one word above the 1024 already needed to store data, for a difference of 0.1%. More information can be found in the previous paper [5], which emphasized experimental results but did not sufficiently develop the underlying algorithm—hence the need for this more scholarly follow-up article.

To verify that it indeed produces conflict-free operation, the scheme has been tested at multiple levels, from functional Perl scripts down to Verilog RTL, for various D-point FFT lengths where D varies from as little to 8 to as high as 8192, and at multiple levels of parallelism from $B = 1$ butterfly up to $B = 8$ by powers of two. Moreover, the Verilog implementation has been used as part of an online FFT generator [18].

VI. Conclusions

We have shown how to produce conflict-free schedules for in-place FFTs having any number of butterflies B operating at any radix R, with or without pipeline overlap, operating on any number of datapoints D. The resulting *group-G* schedule uses G banks of single-ported memory, where G is the next power-of-2 greater than or equal to *BRP*, and where $P = 1$ for pipelines with no overlap and $P =$ pipedepth for overlapping pipes. E.g., an FFT having one radix-two butterfly unit with no pipeline overlap will need a group-2 schedule and two banks of single-ported SRAM. An FFT having two radix-four butterflies and overlapping 3-deep pipelines will need a group-32 schedule with 32 memory banks, because $G = 2^T$ **where** $T = \lceil \log_2(2 \times 4 \times 3) \rceil$ **thus** $G = 32$.

Because single-port RAM is faster and requires less area to implement versus multiported RAM, this ability to construct conflict-free schedules for FFTs based on single-ported memory lets us approach minimum area at maximum throughput.

In fact, experimental results show that 1) at constant area, a conflict-free schedule, such as that produced by this type of algorithm, exceeds the throughput performance of implementations based on earlier algorithms; and 2) at constant throughput, FFTs based on this algorithm are strictly smaller [5].

Such conflict-free schedules can be constructed by the *generate-map-reorder* sequence developed in Section IV-B above and summarized by the equations in Appendix A. The resulting schedule and map works for FFTs that use B radix-R butterfly units to transform D datapoints, for any given values of B, R and D, operating with *overlapping or non-overlapping pipelines of any depth*.

APPENDIX

Schedule and Map Formulas: In each formula below

i	is an S-bit number and t is a T-bit number, and
$S = \log_2 D$	number of stages for the transformation
$S = \log_2 D$	also the number of bits in each datapoint d
$T = \log_2 G$	number of toggle bits
G	group size for the transformation
D	number of datapoints to be transformed
M	$G =$ req'd # of mem. banks for conflict-free operation

A1. Scheduling FFTs With Non-Overlapping Pipelines: Given D datapoints to transform and desired group size G. To produce datapoint sequence $\mathrm{dp}[d_{s,i}]$ for nonoverlapping pipelines, where $d_{s,i}$ is the i^{th} datapoint of stage s:

$$\forall s \in (0..S-T) \qquad \{\forall i \in (0..D-1)\{d_{s,i} = \mathbf{LROT}_s(i)\}\}$$

$$\forall s \in (S - \mathrm{T} + 1..S-1) \quad \{\forall i \in (0..D/G-1)\{$$

$$\forall t \in (0..G-1) \;\; d_{s,i} = [\mathbf{LROT}_{s-(S-T)}(t)] \times (D/G) + i\}\}$$

where

$$\mathbf{LROT}_s(i) \Rightarrow \text{left} - \text{rotate } i \text{ by } s \text{ bits, and}$$

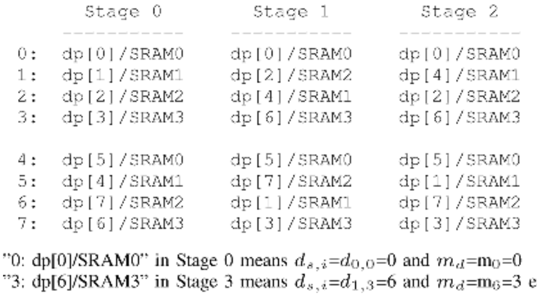
$$\mathbf{LROT}_{s-(S-T)}(t) \Rightarrow \text{ left} - \text{rotate } t \text{ by } s-(S-T) \text{ bits.}$$

(Note the final mul-add $[X \times (D/G) + i]$ is just a shift/concat.)

A2. Scheduling FFTs With Overlapping Pipelines: Given D datapoints to transform and desired group size G. To produce datapoint sequence $\mathrm{dp}[\delta_{s,i}]$ for *overlapping* pipelines, where $\delta_{s,i}$ is the i^{th} datapoint of stage s:

$$\forall s \in (0..S-T) \qquad \{\forall i \in (0..D-1) \qquad \delta_{s,i} = \mathbf{LREP}(d_{s,i})\}$$

$$\forall s \in (S-T+1..S-1) \;\; \{\forall i \in (0..D-1) \;\; \delta_{s,i} = \mathbf{RREP}(d_{s,i})\}$$

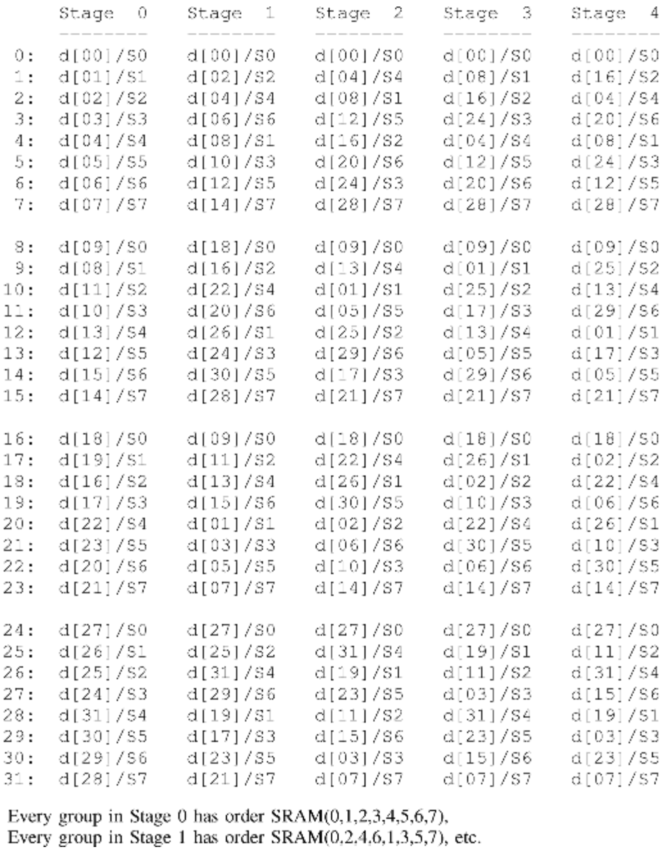


```
        Stage 0        Stage 1        Stage 2
       -----------    -----------    -----------
   0:  dp[0]/SRAM0    dp[0]/SRAM0    dp[0]/SRAM0
   1:  dp[1]/SRAM1    dp[2]/SRAM2    dp[4]/SRAM1
   2:  dp[2]/SRAM2    dp[4]/SRAM1    dp[2]/SRAM2
   3:  dp[3]/SRAM3    dp[6]/SRAM3    dp[6]/SRAM3

   4:  dp[5]/SRAM0    dp[5]/SRAM0    dp[5]/SRAM0
   5:  dp[4]/SRAM1    dp[7]/SRAM2    dp[1]/SRAM1
   6:  dp[7]/SRAM2    dp[1]/SRAM1    dp[7]/SRAM2
   7:  dp[6]/SRAM3    dp[3]/SRAM3    dp[3]/SRAM3
```

"0: dp[0]/SRAM0" in Stage 0 means $d_{s,i}$=$d_{0,0}$=0 and m_d=m$_0$=0
"3: dp[6]/SRAM3" in Stage 3 means $d_{s,i}$=$d_{1,3}$=6 and m_d=m$_6$=3 etc.

Fig. 18. EXAMPLE: D = 8, G = 4.



```
      Stage  0    Stage  1    Stage  2    Stage  3    Stage  4
      --------    --------    --------    --------    --------
  0:  d[00]/S0    d[00]/S0    d[00]/S0    d[00]/S0    d[00]/S0
  1:  d[01]/S1    d[02]/S2    d[04]/S4    d[08]/S1    d[16]/S2
  2:  d[02]/S2    d[04]/S4    d[08]/S1    d[16]/S2    d[04]/S4
  3:  d[03]/S3    d[06]/S6    d[12]/S5    d[24]/S3    d[20]/S6
  4:  d[04]/S4    d[08]/S1    d[16]/S2    d[04]/S4    d[08]/S1
  5:  d[05]/S5    d[10]/S3    d[20]/S6    d[12]/S5    d[24]/S3
  6:  d[06]/S6    d[12]/S5    d[24]/S3    d[20]/S6    d[12]/S5
  7:  d[07]/S7    d[14]/S7    d[28]/S7    d[28]/S7    d[28]/S7

  8:  d[09]/S0    d[18]/S0    d[09]/S0    d[09]/S0    d[09]/S0
  9:  d[08]/S1    d[16]/S2    d[13]/S4    d[01]/S1    d[25]/S2
 10:  d[11]/S2    d[22]/S4    d[01]/S1    d[25]/S2    d[13]/S4
 11:  d[10]/S3    d[20]/S6    d[05]/S5    d[17]/S3    d[29]/S6
 12:  d[13]/S4    d[26]/S1    d[25]/S2    d[13]/S4    d[01]/S1
 13:  d[12]/S5    d[24]/S3    d[29]/S6    d[05]/S5    d[17]/S3
 14:  d[15]/S6    d[30]/S5    d[17]/S3    d[29]/S6    d[05]/S5
 15:  d[14]/S7    d[28]/S7    d[21]/S7    d[21]/S7    d[21]/S7

 16:  d[18]/S0    d[09]/S0    d[18]/S0    d[18]/S0    d[18]/S0
 17:  d[19]/S1    d[11]/S2    d[22]/S4    d[26]/S1    d[02]/S2
 18:  d[16]/S2    d[13]/S4    d[26]/S1    d[02]/S2    d[22]/S4
 19:  d[17]/S3    d[15]/S6    d[30]/S5    d[10]/S3    d[06]/S6
 20:  d[22]/S4    d[01]/S1    d[02]/S2    d[22]/S4    d[26]/S1
 21:  d[23]/S5    d[03]/S3    d[06]/S6    d[30]/S5    d[10]/S3
 22:  d[20]/S6    d[05]/S5    d[10]/S3    d[06]/S6    d[30]/S5
 23:  d[21]/S7    d[07]/S7    d[14]/S7    d[14]/S7    d[14]/S7

 24:  d[27]/S0    d[27]/S0    d[27]/S0    d[27]/S0    d[27]/S0
 25:  d[26]/S1    d[25]/S2    d[31]/S4    d[19]/S1    d[11]/S2
 26:  d[25]/S2    d[31]/S4    d[19]/S1    d[11]/S2    d[31]/S4
 27:  d[24]/S3    d[29]/S6    d[23]/S5    d[03]/S3    d[15]/S6
 28:  d[31]/S4    d[19]/S1    d[11]/S2    d[31]/S4    d[19]/S1
 29:  d[30]/S5    d[17]/S3    d[15]/S6    d[23]/S5    d[03]/S3
 30:  d[29]/S6    d[23]/S5    d[03]/S3    d[15]/S6    d[23]/S5
 31:  d[28]/S7    d[21]/S7    d[07]/S7    d[07]/S7    d[07]/S7
```

Every group in Stage 0 has order SRAM(0,1,2,3,4,5,6,7),
Every group in Stage 1 has order SRAM(0,2,4,6,1,3,5,7), etc.

Fig. 19. EXAMPLE: D = 32, G = 8.

where $d_{s,i}$ is a datapoint calculated using the formula for non-overlapping pipelines, above, and

$$\mathbf{LREP}(i) : \text{replace bits } (i_{s+(T-1)} i_{s+(T-2)} \dots i_s)$$

$$\text{w/ } (\mathbf{P}_{s+(T-1)} \mathbf{P}_{s+(T-2)} \dots \mathbf{P}_s), \text{ and}$$

$$\mathbf{RREP}(i) : \text{replace bits } (i_{S-1} i_{S-2} \dots i_{S-T})$$

$$\text{w/ } (\mathbf{P}_{S-1} \mathbf{P}_{S-2} \dots \mathbf{P}_{S-T}), \text{ and}$$

$$\text{where } \mathbf{P}_b = \mathbf{XOR}(i_b, i_{b+T}, i_{b+2T}, i_{b+3T}, \dots)$$

A3. Map: Each datapoint d gets stored in memory bank m_d which is calculated as follows:

$$\forall (d = d_{S-1} d_{S-2} \dots d_0) \in (0..D-1)\{$$

$$m_d = m_{T-1} m_{T-2} \dots m_0 = \mathbf{P}_{T-1} \mathbf{P}_{T-2} \dots \mathbf{P}_0\}$$

$$\text{where } \mathbf{P}_b = \mathbf{XOR}(d_b, d_{b+T}, d_{b+2T}, d_{b+3T}, \dots)$$

A4. Examples: Fig. 18 shows the result of using these formulae to produce a schedule and map for an 8-point FFT with group size 4. Fig. 19 shows the schedule and map for a 32-point FFT with group size 8.

Acknowledgment

This material is based upon work supported by the Defense Advanced Research Projects Agency under Contract No. HR0011-11-C-0007. Any opinions, findings and conclusion or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

References

[1] T. Patyk, D. Guevorkian, T. Pitkanen, P. Jaaskelainen, and J. Takala, "Low-power application-specific FFT processor for LTE applications," in *Proc. Int. Conf. Embedded Comput. Syst.: Archit., Model., Simul. (SAMOS XIII)*, Jul. 2013, pp. 28–32.

[2] H.-F. Luo, Y.-J. Liu, and M.-D. Shieh, "Efficient memory-addressing algorithms for FFT processor design," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 2014 [Online]. Available: http://ieeexplore.ieee.org

[3] J. Wu, K. Liu, B. Shen, and H. Min, "A hardware efficient VLSI architecture for FFT processor in OFDM systems," in *Proc. 6th Int. Conf. ASIC (ASICON 2005)*, Oct. 2005, vol. 1, pp. 232–235.

[4] Z. Qian and M. Margala, "A novel low-power and in-place split-radix FFT processor," in *Proc. 24th Ed. Great Lakes Symp. VLSI*, 2014, ser. GLSVLSI '14, pp. 81–82 [Online]. Available: http://doi.acm.org/10.1145/2591513.2591563

[5] S. Richardson, O. Shacham, D. Markovié, and M. Horowitz, "An area-efficient minimum-time FFT schedule using single-ported memory," in *Proc. IFIP/IEEE 21st Int. Conf. Very Large Scale Integr. (VLSI-SoC)*, Istanbul, Turkey, 2013, pp. 39–44.

[6] J. A. Hidalgo, J. López, F. Arguello, and E. L. Zapata, "Area-efficient architecture for fast Fourier transform," *IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process.*, vol. 46, no. 2, pp. 187–193, 1999.

[7] J. H. Takala, T. Jarvinen, and H. Sorokin, "Conflict-free parallel memory access scheme for FFT processors," in *Proc. Int. Symp. Circuits Syst. (ISCAS'03)*, Bangkok, Thailand, vol. 4, pp. IV.524–IV.527.

[8] Z.-X. Yang, Y.-P. Hu, C.-Y. Pan, and L. Yang, "Design of a 3780-point ifft processor for tds-ofdm," *IEEE Trans. Broadcast.*, vol. 48, no. 1, pp. 57–61, Mar. 2002.

[9] J. Baek and K. Choi, "New address generation scheme for memory-based FFT processor using multiple radix-2 butterflies," in *Proc. SoC Int. Design Conf. (ISOCC '08)*, Nov. 2008, vol. 01, pp. I-273–I-276.

[10] S. Johnson and M. Frigo, "A modified split-radix FFT with fewer arithmetic operations," *IEEE Trans. Signal Process.*, vol. 55, no. 1, pp. 111–119, Jan. 2007.

[11] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.*, vol. 19, no. 90, p. 297, 1965.

[12] D. T. Harper, III, "Block, multistride vector, FFT accesses in parallel memory systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 1, pp. 43–51, 1991.

[13] D. A. Patterson and J. L. Hennessy, "Appendix A: Pipelining: Basic and intermediate concepts," in *Computer Architecture: A Quantitative Approach*, 3rd ed. San Mateo, CA, USA: Morgan Kauffman, 2003.

[14] M. C. Pease, "Organization of large scale Fourier processors," *J. ACM (JACM)*, vol. 16, no. 3, pp. 474–482, 1969.

[15] L. Johnson, "Conflict free memory addressing for dedicated FFT hardware," *IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process.*, vol. 39, no. 5, pp. 312–316, 1992.

[16] Y. Ma, "An effective memory addressing scheme for FFT processors," *IEEE Trans. Signal Process.*, vol. 47, no. 3, pp. 907–911, 1999.

[17] C.-H. Chang, C.-L. Wang, and Y.-T. Chang, "A novel memory-based FFT processor for DMT/OFDM applications," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process.*, Phoenix, AZ, USA, 1999, vol. 4, pp. 1921–1924.

[18] Interactive chip generator, powered by Genesis [Online]. Available: http://www-vlsi.stanford.edu/genesis

Stephen Richardson received the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, USA, and has worked in industry at Weitek and MIPS, as well as at Sun Microsystems and Hewlett-Packard research labs. He is currently a Research Associate in the Stanford University EE Department.

Dejan Marković is a Professor of Electrical Engineering at the University of California, Los Angeles, CA, USA. He is also affiliated with UCLA Bioengineering Department as a co-chair of the Neuroengineering field. His current research is focused on emerging radio and healthcare systems, programmable ICs, design with post-CMOS devices, optimization methods and CAD flows. Dr. Markovié received the 2007 David J. Sakrison Memorial Prize at UC Berkeley for his PhD research. He received an NSF CAREER Award in 2009. In 2010, he was a co-recipient of the ISSCC Jack Raper Award for Outstanding Technology Directions

Andrew Danowitz received his M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, USA, in 2010 and 2014 respectively. He is currently an Assistant Professor of Computer Engineering at California Polytechnic State University, San Luis Obispo, CA, USA, where he conducts research in the fields of digital design methodologies, and energy efficient computing, and hardware/software co-design.

John Brunhaver is an Assistant Professor at Arizona State University, Tempe, AZ, USA, as of 2015. His current research focuses on energy efficient, programmable, domain specific computer architectures, and the design automation techniques for implementing them. His Stanford University Ph.D. thesis, *The Design and Optimization of A Stencil Engine*, examines the virtual machine model for an image processing and image understanding domain specific processor.

Mark Horowitz (F'00) is the Yahoo! Founders Professor at Stanford University, Stanford, CA, USA, and was chair of the Electrical Engineering Department from 2008 to 2012. He co-founded Rambus, Inc. in 1990 and is a fellow of the ACM and a member of the National Academy of Engineering and the American Academy of Arts and Science. Dr. Horowitz's research interests are quite broad and span using EE and CS analysis methods to problems in molecular biology to creating new design methodologies for analog and digital VLSI circuits.