Chapter 5

Experimental Results

In this chapter, we present experimental results on the performance of our methods, the heuristic method, ILP method, and region-wise ILP method. We implemented the three methods and call our heuristic, optimal (ILP) and region-wise ILP methods as ADVAN, ADVBIST and ADVBIST_h, respectively. We also compare the performance of our methods with two other BIST synthesis methods; RALLOC [17] proposed by Avra, BITS [20] proposed by Parullkar et al. We implemented the other two BIST synthesis methods (RALLOC, and BITS) in the C++ language. For the implementation of RALLOC and BITS, we followed the algorithms presented in [17] and [20]. A commercial ILP solver CPLEX [67] was used in ADVBIST and ADVBIST_h. We set the CPU time limit to 24 hours for the ILP solver. Hence, a BIST design may not be an optimal one if the design is obtained without searching all the possible solution space due to the time limit.

Another experiment that we conducted was to implement a large BIST circuit and to measure the performance (in area overhead and the clock speed) of the BIST circuit. We incorporated a BIST design into an 8×8 DCT (discrete cosine transform) circuit using the proposed region-wise ILP method. We synthesized the design to obtain a gate level circuit, and laid it out by following the ASIC design flow.

5.1 Background

We measured the performance of the five BIST synthesis systems for six data flow graphs, which are widely adopted for benchmarking high-level BIST synthesis. The data flow graphs include the ones studied by Tseng and Siewiorek, called tseng, and by Paulin and Knight, called paulin [32], [50]. The other four data flow graphs are a 6^{th} order FIR (finite impulse response) filter, a 3^{rd} order IIR (infinite impulse response) filter, a 4-point DCT (discrete cosine transformation) circuit, and a 6-tap wavelet filter. We adopted the scheduling and module assignment from [17] for tseng and paulin. The other four data flow graphs were synthesized using HYPER [68]. The width of the data path logic is eight for all the circuits. Details of the circuits are shown in Table 5.1. Column headings of the table are described below:

ckt: the name of circuit.

var: the number of variables in the DFG,

const: the number of constants in the DFG,

op: the number of operations in the DFG,

reg: the minimum number of necessary registers (equivalent to the maximal horizontal crossing), and

modules: the minimum number and types of necessary modules.

Table 5.1. Characteristics of the circuits

Ckt	Data Flow Graph			Data Path	
	Var	Const	Op	Reg	Modules
tseng	8	0	11	5	3 ALUs
paulin	10	2	12	5	2 multipliers, 1 adder, 1 subtractor
fir6	13	7	27	7	2 multipliers, 1 adder
iir3	14	8	26	6	1 multiplier, 1adder, 1 subtractor
dct4	15	4	23	6	2 multipliers, 1 adder, 1 subtractor
wavelet6	16	6	28	7	1 multiplier, 1adder, 1 subtractor

In this thesis, the area of a circuit is represented by the transistor count of registers and multiplexers in the circuit. *Data path logic of a circuit is not considered in the transistor count.* The number of transistors in test registers and multiplexers is based on the circuits of [57] and [58] and is given in Table 5.2. In the table, #Trs and #MuxIn denote the number of transistors and the number of multiplexer inputs, respectively. The heading "Avg" in Table 5.2(b) is the average number of transistors per multiplexer input. Note that the transistor counts of registers and the average number of transistors per multiplexer input is the weights[2] of our objective function in Section 3.5 for ADVBIST_h.

The reference circuits, which were used to measure the area overhead of BIST designs, were obtained through an ILP for data path synthesis. The objective function for the data path synthesis is the same as the one used for our optimal BIST synthesis, except that the number of test registers is set to 0. Hence, the reference circuits are optimal in area.

[2] To evaluate the performance, the number of transistors for each size of the multiplexers is used as the weight for our optimal BIST design ADVBIST.

Table 5.2 Number of transistors of 8-bit test registers and multiplexers

(a) Test registers

Type	Reg.	TPG	SR	BILBO	CBILBO
#Trs	208	256	304	388	596

(b) Multiplexers

#MuxIn	2	3	4	5	6	7	Avg
#Trs	80	176	208	300	320	350	-
#Trs/input	40	59	52	60	53	50	52

5.2 Experimental Results of the Three Proposed Methods

In this subsection, we present the performance of the three proposed methods. We compare the results of the three methods with other existing high-level BIST synthesis methods.

5.2.1 ADVAN: Heuristic Method

The first experiment was to measure the performance of the proposed heuristic method, ADVAN, and to examine the relationship between area and test time in high-level BIST synthesis. In allocating signature registers for our method, we considered *all possible optimal mergers* of the circuit for each test session. (Refer to Section 3.2 for optimal mergers.) Among all the mergers considered, we chose a merger that incurs the least area overhead.

Experimental results on area and test time (in terms of the number of test sessions) of synthesized BIST circuits are given in Table 5.3. The first row for each circuit entry is the reference circuit. Column headings for the table are explained below.

R : the total number of registers

T : the total number of test pattern generators
S : the total number of signature registers
B : the total number of BILBOs
C : the total number of CBILBOs
M : the total number of inputs of multiplexers
Area : the number of transistors of the registers and the multiplexers
OH : the area overhead of the BIST design (%)

From the table, the area overhead of ADVAN ranges from 8 percent to 40 percent. The area overhead of the last four circuits is less than 21 percent. ADVAN incurs relatively high area overhead for tseng and paulin due to the employment of CBILBOs and/or a large number of multiplexer inputs. However, since the area overhead of a circuit is computed without considering the area for the data path logic modules, the actual area overhead will be much lower than the ones presented in the table. For example, the area overhead of paulin is 40.8 percent for k=1. Circuit paulin contains two 8×8 multipliers, one 8-bit adder and one 8-bit subtractor. If the adder and the subtractor based on ripple carry propagation and array multipliers are used to implement the circuit, the actual area overhead would be reduced to about 18 percent. Therefore, it can be said that the area overhead of ADVAN is small to moderate for all the circuits tested.

The table shows that, in general, the area and test time are traded in high-level BIST synthesis. An illustrative example is fir6, in which the area overhead reduces monotonically with an increase in the number of test sessions. When the number of test sessions, k, increases from 1 to 2, there is a substantial reduction in the area overhead for most circuits. However, an increase of k beyond 2 is little help in reducing the area overhead. In fact, the area overhead increases for some circuits. Therefore, it is a good idea to avoid testing an entire circuit in one test session if the area overhead is a major concern.

Table 5.3. Performance of ADVAN

Ckt	*k*	R	T	S	B	C	M	Area	OH(%)
tseng		5					14	1600	
	1	5	2	2	0	1	21	2612	38.7
	2	5	3	2	0	0	21	2316	30.9
	3	5	2	1	0	0	23	2368	32.4
paulin		5					19	1856	
	1	5	1	2	0	2	22	3096	40.1
	2	5	2	1	0	1	22	2612	28.9
	3	5	2	1	0	1	21	2564	27.6
	4	5	3	1	0	0	26	2684	30.8
fir6		7					20	2576	
	1	7	0	1	0	2	24	3510	26.6
	2	7	2	1	1	0	26	3232	20.3
	3	7	2	1	0	0	28	3308	22.1
iir3		6					22	2224	
	1	6	2	2	0	1	28	3360	33.8
	2	6	1	2	0	0	31	3148	29.4
	3	6	3	1	0	0	32	3432	35.2
dct4		6					24	2320	
	1	6	2	3	0	0	36	3504	33.8
	2	6	2	2	0	0	38	3472	33.2
	3	6	3	1	0	0	36	3308	29.9
	4	6	3	1	0	0	35	3420	32.2
wave-let6		7					25	2880	
	1	7	2	2	0	0	48	3800	24.2
	2	7	1	1	1	0	46	3728	22.7
	3	7	2	1	0	0	46	4182	31.1

5.2.2 ADVBIST: ILP Method

We conducted experiments for the proposed ILP based method, ADVBIST. We set a limit on the execution time, which is 24 CPU hours, in running the ILP solver. Hence a solution may not be optimal if obtained without searching all the possible solution space due to the time limit. Experimental results on area overhead and processing time of ADVBIST are given in Table 5.4. The first row of each circuit entry is the reference circuit. Two new column headings, for the table are explained below. All the other headings are the same as those given in the previous section.

k: the number of test sessions, and
Time: the CPU time on a Sun Ultra 10 workstation.

As shown in the table, ADVBIST successfully synthesized a BIST data path for each test session for all six circuits. Among the 20 BIST designs obtained by ADVBIST, 16 circuits are optimal (i.e., minimal) in area, and the four circuits of dct4 may not be optimal. The area overhead of ADVBIST ranges from 11 percent to 46 percent. As described before, since the area overhead of a circuit is calculated without considering the area for the data path logic modules, the actual area overhead will be much lower than the ones presented in the table. Therefore, the area overhead of ADVBIST is small to moderate for all six circuits. Note that the area overhead of the 16 optimal BIST designs is minimal and cannot be reduced any further.

Table 5.4: Performance of the proposed method ADVBIST

Ckt	*k*	R	T	S	B	C	M	Area	OH(%)	Time (h,m,s)
tseng		5					14	1600		
	1	5	2	3	0	0	22	2416	33.8	58s
	2	5	2	2	1	0	18	2228	28.2	1m 22s
	3	5	2	1	2	0	14	2152	25.7	35s
paulin		5					19	1856		
	1	5	1	2	0	2	20	2968	37.5	4h 42m 0s
	2	5	2	2	1	0	24	2580	28.1	24m 55s
	3	5	2	2	1	0	23	2484	25.3	11m 40s
	4	5	2	2	1	0	23	2484	25.3	59m 34s
fir6		7					20	2576		
	1	7	3	2	0	1	29	3684	30.1	17m 34s
	2	7	3	1	0	1	23	3268	21.2	40m 16s
	3	7	4	1	0	0	26	3040	15.3	23h 56m 4s
iir3		6					22	2224		
	1	6	3	3	0	0	27	2912	23.6	3h 11m 8s
	2	6	4	2	0	0	24	2688	17.3	2h 6m 26s
	3	6	5	1	0	0	23	2656	16.3	2h 50m 8s
dct4		6					24	2320		
	1	6	2	4	0	0	27	3024	23.3*	24h
	2	6	3	2	0	0	34	3088	24.9*	24h
	3	6	2	2	0	0	59	4256	45.5*	24h
	4	6	3	1	1	0	32	3236	28.3*	24h
wavelet6		7					25	2880		
	1	7	2	3	0	0	31	3344	13.9	11m 9s
	2	7	2	2	0	0	31	3248	11.3	10h 5m 15s
	3	7	2	2	0	0	31	3248	11.3	14h 39m 24s

Note: The entries marked with “*” reached the time limit.

The experimental results confirm that the area and the test time are traded in high-level BIST synthesis. Excluding dct4 (whose BIST circuits may not be optimal), the area overhead of a BIST design reduces monotonically with the increase in the number of test sessions. The rate of reduction is the largest from k=1 to k=2 and is smaller for a larger k. For three circuits, paulin, iir3, and wavelet6, the area overhead is the same or nearly the same for k=2 and k=3. For such circuits, it is better to choose k=2 due to shorter testing time. It is a good idea to avoid testing the entire modules in a small number of (one or two) test sessions if the area overhead is a major concern. However, an excessively large number of test sessions wastes testing time without saving the area, which necessitates the exploration of various BIST designs with a different number of test sessions.

5.2.3 ADVBIST_h: Region-Wise ILP Method

Table 5.5 shows the processing time of the proposed region-wise method ADVBIST_h for five different region sizes of *s* ranging from 1 to 5 and that of our optimal BIST method ADVBIST. The column heading "speedup" is the ratio of the processing time of ADVBIST to that of ADVBIST_h for three different region sizes of *s*=1, 3, and 5. All processing times are in seconds and were measured on a SUN Ultra II workstation. We set a limit on the execution time, which is 2 CPU hours, in running the ILP solver.

Table 5.5: Processing time (in seconds) of the proposed region-wise ILP method

Circuit	Processing time for various region sizes					ADVBIST	Speedup		
	1	2	3	4	5		1	3	5
tseng	4	5	4	10	10	35	8.8	8.8	3.6
paulin	15	27	8	6	6	3574	238.3	446.8	567.3
fir6	8	9	36	336	2233	86164	10770.5	2393.4	38.6
iir3	12	67	207	218	198	10208	850.7	49.3	51.7
dct4	33	163	3773	1166	18831	86400	2618.2	22.9	4.6
wavelet6	13	9	12	145	147	52764	4058.8	4397.0	360.2
Average							**3090.9**	**1219.7**	**171.0**

The processing time for our region-wise method ADVBIST_h ranges from 4 seconds to 18831 seconds (=5.2 hours), while our optimal method ADVBIST takes as long as 86400 seconds (=24 hours) for the most complex circuit dct4. The average speedup of the region-wise method over the optimal BIST design is 3091 for *s*=1, 1220 for *s*=3, and 171 for *s*=5. As expected, the growth of the region size increases the processing time and improves the quality of the solution (to be shown later). Therefore, the heuristic used in region-wise ILP method enables us to trade the processing time (equivalently, the circuit size) and the quality of the solution. The

remaining key issue of the region-wise method is the quality (in terms of area overhead) of its BIST designs in comparison with that of optimal BIST designs.

Table 5.6 shows the area overhead of synthesized BIST circuits for the ADVBIST_h under the region size of *s*=1 to 5 and for ADVBIST. Although both methods generate BIST designs for each *k*-test session, we report only the results for the maximum of test session number (which is equal to the number of modules) to simplify the results. The number of test sessions for a circuit is given next to the circuit name in the table.

Table 5.6: Area overhead of the proposed heuristic

Circuit	Area overhead of various region sizes (%)					ADVBIST (%)
	1	2	3	4	5	
tseng (3)	28.7	28.7	28.7	28.7	28.7	25.7
paulin (4)	39.2	29.3	29.3	29.3	29.3	25.3
fir6 (3)	17.9	17.9	17.9	15.3	15.3	15.3
iir3 (3)	24.5	24.0	23.6	21.5	21.0	16.3
dct4 (4)	32.0	32.3	28.0	29.0	28.0	28.3
wavelet6 (3)	15.5	13.0	13.0	15.5	13.0	11.3
Average	**26.3**	**24.2**	**23.4**	**23.2**	**22.6**	**20.4**

As can be seen from the Table 5.6, the difference in area overhead for ADVBIST and ADVBIST_h is small to moderate for most cases except circuit paulin under the step size of *s*=1. Excluding the exceptional case, the average difference in area overhead is 6% for *s*=1 and is reduced to 2% under *s*=5. The maximum difference is, at most, 5% for *s*=5. Hence, it can be said that the quality of BIST circuits synthesized by the region-wise method is close to that of the optimal circuits.

Again, it is important to note that since the area overhead of a circuit is computed without considering the area for the data path logic modules, the actual area overhead will be much lower than the ones presented in the table. Therefore, the quality of BIST circuits of the heuristic

method would be even closer to the optimal BIST designs when the actual area overhead is considered. It is interesting to note that the area overhead of the heuristic method for dct4 under s=5 is slightly less than that of the optimal method, since the optimal method could not generate an optimal BIST design due to the CPU time limit.

5.2.4 Comparison of Performance

We compare the performance of our methods (heuristic, optimal, and region-wise optimal) with two other BIST synthesis systems, RALLOC [17] and BITS [20]. In order to make the comparison meaningful, the six data flow graphs used in the experiment employed the same scheduling and the same module assignment for all five systems. The performance of the five high-level BIST synthesis systems is presented in Table 5.7. The region size s for our heuristic method ADVBIST_h is 5 in the table, and the number of test sessions is maximal for a given circuit. Column headings for the table are the same as in the previous section.

From the table, our heuristic method (ADVAN) is slightly better than other methods. However, our ILP based methods (ADVBIST and ADVBIST_h) perform better than the other methods in area for all circuits. For some complex circuits such as iir3, dct4, and wavelet6, the area overhead of the ILP methods is substantially less than that for other methods. The better performance of the proposed ILP based methods is largely due to a smaller number of transistors for multiplexers. This fact is encouraging since our ILP methods will take less area for routing to make our methods even more attractive.

Table 5.7. Performance of various high-level BIST synthesis systems

Ckt	Method	R	T	S	B	C	M	Area	OH(%)
tseng (3)	Reference	5					14	1600	
	ADVBIST_h	5	2	1	1	0	18	2244	28.7
	ADVBIST	5	2	1	2	0	14	2152	25.7
	ADVAN	5	2	1	0	0	23	2368	32.4
	RALLOC	5	1	0	3	0	14	2300	30.4
	BITS	5	2	1	1	0	20	2436	34.3
paulin (4)	Reference	5					19	1856	
	ADVBIST_h	5	3	2	0	0	27	2624	29.3
	ADVBIST	5	2	2	1	0	23	2484	25.3
	ADVAN	5	3	1	0	0	26	2684	30.8
	RALLOC	5	1	0	3	0	25	2892	35.8
	BITS	5	2	0	0	1	27	3024	38.6
fir6 (3)	Reference	7					20	2576	
	ADVBIST_h	7	4	1	0	0	26	3040	15.3
	ADVBIST	7	4	1	0	0	26	3040	15.3
	ADVAN	7	2	1	0	0	28	3308	22.1
	RALLOC	8	1	1	2	0	36	4212	38.8
	BITS	7	1	0	0	1	24	3280	21.5
iir3 (3)	Reference	6					22	2224	
	ADVBIST_h	6	5	1	0	0	24	2816	21.0
	ADVBIST	6	5	1	0	0	23	2656	16.3
	ADVAN	6	3	1	0	0	32	3432	35.2
	RALLOC	7	1	0	2	0	38	4212	47.2
	BITS	6	2	0	2	0	29	3176	30.0

Table 5.7. Performance of various high-level BIST synthesis systems (Continued)

Ckt	Method	R	T	S	B	C	M	Area	OH(%)
dct4 (4)	Reference	6					24	2320	
	ADVBIST_h	6	3	1	1	0	32	3220	28.0
	ADVBIST	6	3	1	1	0	32	3236	28.3
	ADVAN	6	3	1	0	0	35	3420	32.2
	RALLOC	6	1	1	2	0	37	3812	39.1
	BITS	7	1	1	0	1	38	4180	44.5
wavelet6 (3)	Reference	7					25	2880	
	ADVBIST_h	7	2	2	0	0	31	3312	13.0
	ADVBIST	7	2	2	0	0	31	3248	11.3
	ADVAN	7	2	1	0	0	46	4182	31.1
	RALLOC	8	1	0	3	0	50	5186	44.5
	BITS	7	1	0	2	0	40	3946	27.0

5.3 Experimental Results of the DCT Circuits

We designed two DCT circuits, one with BIST design using the proposed region-wise ILP method and the other one without BIST. We laid them out in using our ASIC design flow environment and measured the performance. Our objective of the experiment is two fold:

i) to verify the practicality of the proposed heuristic to large industry circuits, and

ii) to measure the overhead (in area and clock speed) of the BIST circuit.

ADVBIST_h (as well as ADVAN and ADVBIST) employs a cost model based on the number of transistors. Although the cost model is a reasonable choice in high-level BIST synthesis (in which layout information is unavailable), it does not account for the area for modules. Therefore, as noted earlier, the actual area overhead of a BIST circuit would deviate from the one reported by ADVBIST_h. The second objective given above is intended to measure

the deviation. In addition, we are also interested in finding out the performance degradation due to the additional delay of BIST circuitry.

We decided to implement the 2-dimensional (2-D) 8×8 DCT (discrete cosine transformation) algorithm, whose specific implementation is described in Chapter 4. After scheduling and module assignment using HYPER [68], 15 control steps were assigned to the 1-D DCT. Six modules (two adders, two subtractors, two multiplier), and 15 registers necessary for the implementation. More detailed characteristic of the data flow graph is shown in Table 5.8 (a).

Then ADVBIST_h with the region size from 1 to 5 was applied to the scheduled and module assigned data flow graph. The number of test sessions is set to 4. The performance of the ADVBIST_h with the various region sizes is shown in Table 5.8 (b). (Refer to the previous section for column headings.) As shown in the table the area overhead decreases as the region size increases and is minimal when region size becomes 3. As the region size further increases to 4 and 5, the area overhead increases. This is because the size of regions becomes too large for ILP solver to handle and as a result it generates only feasible result. Thus, the data path generated from the program with the region *size of 2* was selected. Table 5.8(c) shows the performance of ADVBIST_h compared with other various high-level BIST synthesis systems for circuit DCT8. The overhead of ADVBIST_h is about 11 % while those of others are about 30 %. Hence, it can be said that our region-wise ILP heuristic method is efficient for large circuits.

The resultant BIST data path and a control unit (designed manually) were described in VHDL and were synthesized into a gate level circuit. The circuit was placed and routed using our HP 0.5 μm CMOS ASIC library with triple metal layers. It should be noted that BIST is incorporated into the data path, and the control unit. Scan design is employed for the rest of modules as explained in Section 4.7. We also implemented a circuit without BIST in a similar manner. The characteristics of the two circuits are summarized in Table 5.8.

Table 5.8: Characteristics of the DCT circuit and performance of its BIST implementation

(a) The data flow graph and the data path

Data Flow Graph			Data Path	
# Csteps	# Var's	# Op's	# Reg's	Modules
15	62	47	15	two adders, two subtractors, and two multipliers

(b) Performance of the ADVBIST_h with the various region sizes

Circuit	*s*	R	T	S	M	Area	OH (%)	Processing time of ADVBIST_h (sec)
w/o BIST		15			81	7074		
w/ BIST	1	15	4	2	93	8152	13.2	11089
	2	15	4	2	92	8128	13.0	17440
	3	15	4	2	89	7994	11.5	14800
	4	15	4	3	88	8060	12.2	10953
	5	15	4	2	94	8242	14.2	21691

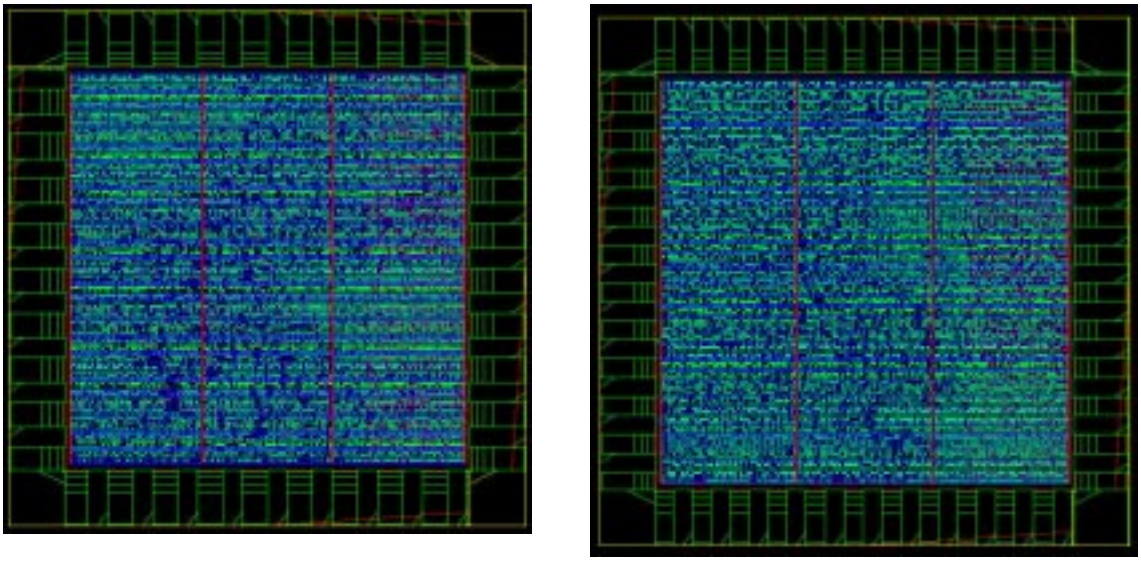
(c) Performance of various high-level BIST synthesis systems for circuit DCT8

Method	R	T	S	B	C	M	Area	OH(%)
Reference	15					81	7074	
ADVBIST_h	15	4	2	0	0	89	7994	11.5
ADVBIST	-	-	-	-	-	-	-	N/A
ADVAN	15	2	2	0	0	128	9996	29.2*
RALLOC	15	3	4	0	0	119	10096	29.9
BITS	16	3	2	0	0	115	101786	30.5

Note: The entries marked with "*" reached the time limit.

Table 5.9 Characteristics of two chips

	w/o BIST	w/ BIST
Area of the core logic	10.84 mm^2	11.92 mm^2
Maximum operable clock rate	32.4 MHz	30.8 MHz
Area overhead	-	9.95 %
Clock speed degradation	-	5.2 %
Equiv. NAND2 gate counts		
of the data path	9926	10956
of the control unit	317	553
of the memory unit	10,030	12451
Total	22,612	27,160



(a) Without BIST (b) With BIST

Figure 5.1 Layouts of 8×8 DCT

As shown in Table 5.9, the two experimental circuits are quite large. The area of the core logic for the original circuit (without BIST) is about 10.8 mm^2, and its equivalent NAND2 gate count is 22612. ADVBIST_h successfully generated a BIST design for the 2-D DCT algorithm. The processing time of ADVBIST_h is four hours. The processing time is long but tolerable, since the BIST design is a one-time effort. It should be noted that ADVBIST, our optimal method, failed to generate even a sub-optimal solution for the circuit.

ADVBIST_h predicts the overhead of the BIST design would be 11.5 % as shown in Table 5.8(c), but the actual area overhead is 9.95 % as in Table 5.9. As noted earlier, the discrepancy is due to the fact that the overhead estimated by ADVBIST_h does not account for the modules in the data path. Note that the actual area overhead of the BIST circuit is moderate and acceptable in industry.

The test circuitry of the BIST circuit slows down the maximal operating clock frequency to 30.8 MHz from 32.4 MHz or by 5.2 %. The performance degradation is small. The performance degradation can be reduced easily to a certain level for our ILP method possibly at the cost of higher area overhead. Increased the cost of multiplexers (which are a major factor contributing the performance degradation) in the objective function would reduce the number of multiplexers and/or would decrease the size of multiplexers in our BIST synthesis process. Hence, the performance of a BIST circuit is likely to increase.

The data path with six modules is tested in four sub-test sessions. The test schedule for each test session is shown in Table 5.10. In the test session 0 and 1, subractors (S24 and S22) are tested. In the session 2 and 3, multipliers (M23 and M24) and adders (A26 and A27) are tested. Fig. 5.2 shows BIST configuration for testing subtractor S25 module in session 0. R1 and R14 are used as TPGs and R11 is used as an SR. Necessary control signals for multiplexers are generated by the controller.

Table 5.10 Test schedule

Session	Module	TPG	TPG	SR
0	S25	R1	R14	R11
1	S22	R0	R1	R2
2	M23	R0	R3	R2
	M24	R3	R0	R11
3	A26	R3	R1	R11
	A27	R1	R0	R2

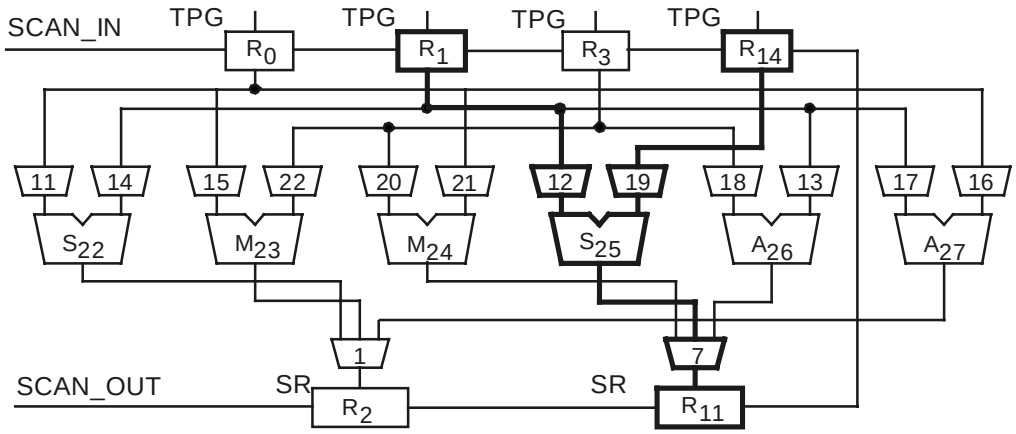


Figure 5.2 BIST Configuration at session 0

Fig. 5.3 through Fig. 5.6 show fault coverage of the adder, the subtractor, the multiplier, and the controller, respectively. The fault coverage of the adder and the subtractor reaches 100 % with 40 and 50 test patterns, respectively. The fault coverage of the multiplier reaches 99.0 % for 50 test patterns. The fault coverage reaches 99.65 % for 200 test patterns. Thus, we used 150 test patterns for the multiplier, which achieves 99.6 % fault coverage. In the case of controller, the fault coverage reaches 97.0 % for 50 test patterns and remains the same over 50 test patterns. Thus, 50 patterns are used for the testing of controller.

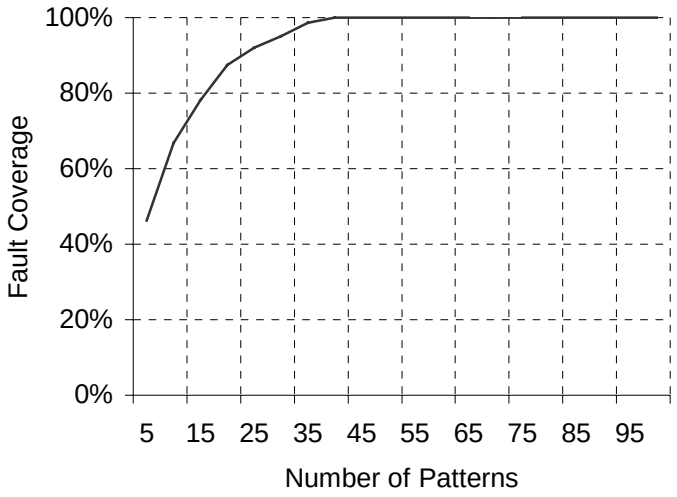


Figure 5.3 Fault coverage of the adder

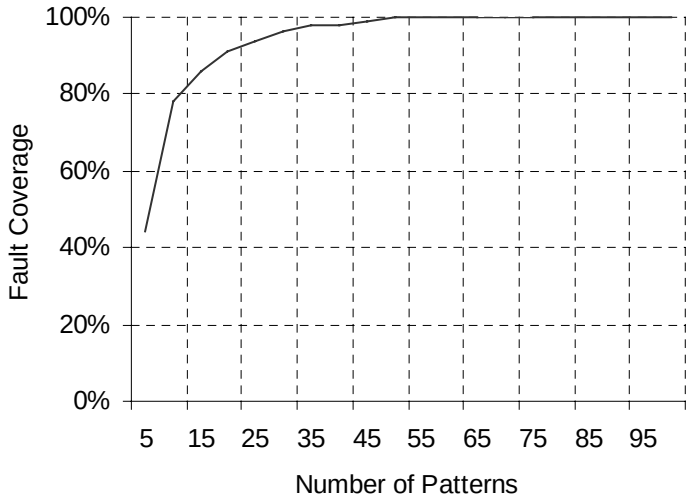


Figure 5.4 Fault coverage of the subtractor

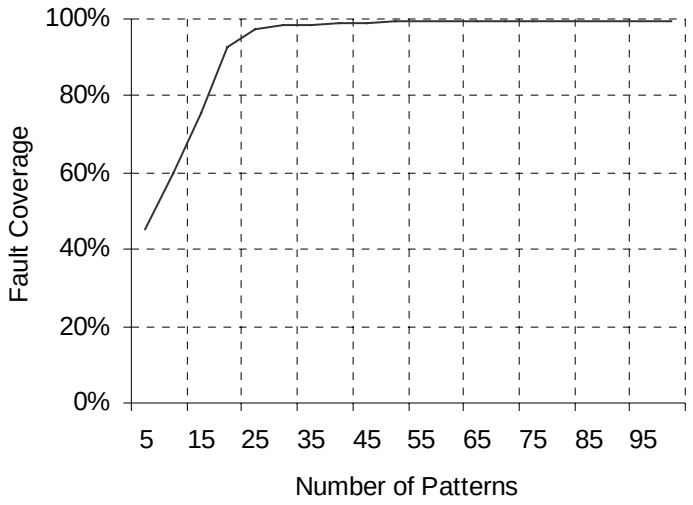


Figure 5.5 Fault coverage of the multiplier

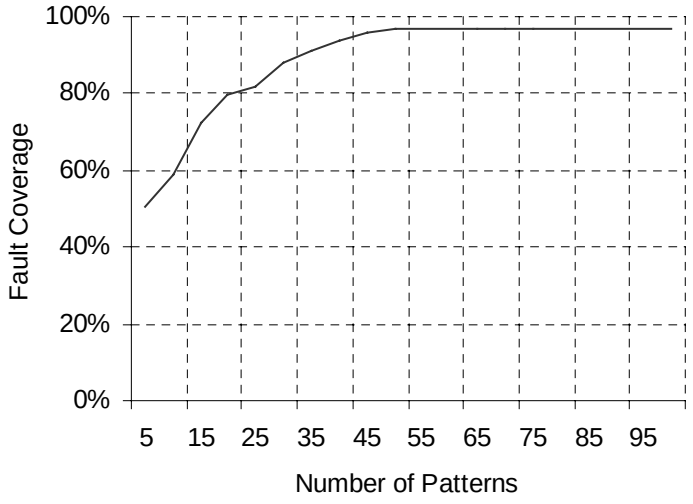


Figure 5.6 Fault coverage of the controller

Conclusion

A high-level built-in self-test (BIST) synthesis is a process of transforming a behavioral description into a register-transfer level structural description while minimizing BIST overhead. Existing high-level BIST synthesis methods focus on one objective, minimizing either area overhead or test time. Hence, those methods do not render exploration of large design space, which may result in a local optimum.

In this thesis, we present three methods that aim to address the problem. The first method tries to find a register assignment for each *k*-test session in a heuristic manner. Therefore, it offers a range of designs to the designer with different figures of merit in area and test time. The second method is based on integer linear programming (ILP). In order to reduce the complexities involved in the register assignment process, existing high-level BIST synthesis methods decouple the three tasks, assignments of registers, interconnections, and BIST registers, and perform the tasks sequentially at the cost of global optimality. The proposed ILP based method performs the three tasks concurrently to yield optimal or near-optimal designs. However, the ILP based method takes a long processing time. The third method, a region-wise heuristic ILP method, intends to address the problem. It partitions a given data flow graph into smaller regions based on control steps and applies the ILP for each region successively to reduce the processing time.

Experimental results show that all of our high-level BIST synthesis methods perform better than or comparable to existing BIST synthesis systems. Our ILP based method successfully generates optimal solution for small to midrange circuits. It enables designers to s a trade between area overhead and test time. The experimental results show that it is a good idea to avoid testing the entire modules in a small number of (one or two) test sessions if the area overhead is a major concern. However, an excessively large number of test sessions wastes testing time with little saving area, which necessitates the exploration of various BIST designs with a different figure of merits. Our region-based method reduces the processing time by several orders of magnitude, while the quality of the solution is slightly compromised compared with our ILP-based optimal method. The BIST area overhead of the region-wise approach is around 5 % to 15 % less than those of other existing methods for the six circuit experimented.

To evaluate our high-level BIST synthesis method, two DCT circuits, one with BIST and the other without BIST, were implemented using an ASIC design flow developed at Virginia

Tech. Experimental results show that our method is applicable to large industry circuits. The BIST area overhead of the circuit is about 9.9 %, which is acceptable to industry. We achieved fault coverage over 97.0 % for the modules tested by BIST.

Finally, our high-level BIST synthesis methods concern only register and interconnection assignments. They can be extended to include scheduling and transformation in the synthesis process, which would result in higher performance. Traditionally performance and area have been the major metrics in VLSI design. Recently, power consumption has been added as the third major metric. Our ILP based methods can incorporate power consumption in the design process by introducing power consumption as a cost factor in the objective function. The two areas are open for future research.

Appendix A: User's Guide for High-Level BIST Synthesis Tools

A.1 Introduction

This manual explains the steps necessary to create an RT-level implementation of a circuit from a behavioral VHDL description. Most of the tools used in the process described were developed by a third party. However, the tools "hls", "advbist", and "advbist_h" were developed at Virginia Polytechnic Institute and State University. All the related files are located underneath /home/cadtool/bist_synthesis. Source programs, executables, and input circuits are located underneath src/, bin/, and circuits/, respectively.

A.2 Scheduling and Module Assignment

The first steps to synthesizing a circuit from a behavioral model are scheduling and module assignment. The scheduling step assigns a particular clock cycle (relative to the start of the computation) to each operation that needs to be performed. Module assignment assigns functional units (such as adders) to particular steps in the schedule. The goals of these functions are often at odds. We would like to schedule the operations so that the computation finishes in as few cycles as possible. We would also like to use as few functional units as possible. Scheduling operations to occur in the same clock cycle requires more logic units, but spreading out the work over more clock cycles takes more time. For example, adding two pairs of numbers at the same time would require two adders. If the additions took place in different clock cycles, the same adder could be used, but the operation would take two clock cycles.

A.2.1 6^{th} Order FIR Filter

Throughout this report, we use a 6^{th} order finite impulse response (FIR) filter as a demonstrative circuit. This circuit implements the function:

$$y = h_0 x_0 + h_1 x_1 + h_2 x_2 + h_3 x_3 + h_4 x_4 + h_5 x_5 + h_6 x_6$$

where h_n is a filter coefficient, and x_n is the delayed value of x_{n-1}. This equation requires seven multiplications and one summation. Figure A.0.1 shows a block diagram of the filter.

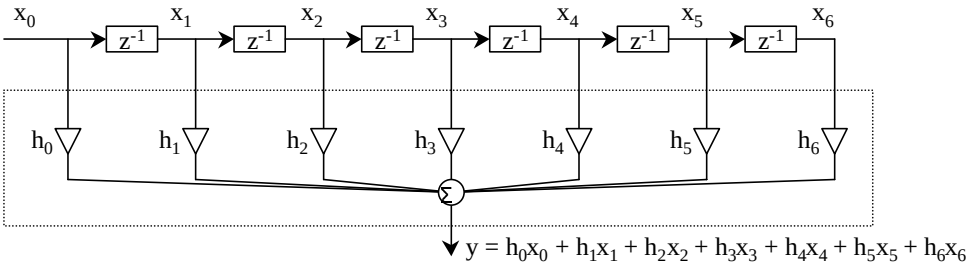


Figure A.0.1 A 6th order FIR filter

Figure A.0.2 is a Silage program which describes the behavior of the 6th order FIR filter. We assume that the delay elements (the "z^{-1}" blocks in Figure A.0.1) are implemented using a shift register.

```
#define num8    num<8,7>

#define h0      num8(0.9)
#define h1      num8(0.8)
#define h2      num8(0.7)
#define h3      num8(0.6)
#define h4      num8(0.5)
#define h5      num8(0.4)
#define h6      num8(0.3)

func main(x0, x1, x2, x3, x4, x5, x6 : num8) y : num8 =
begin
        t0  = num8(h0  * x0);
        t1  = num8(h1  * x1);
        t2  = num8(h2  * x2);
        t3  = num8(h3  * x3);
        t4  = num8(h4  * x4);
        t5  = num8(h5  * x5);
        t6  = num8(h6  * x6);
        y = (((((t0 + t1) + t2) + t3) + t4) + t5) + t6;
end;
```

Figure A.0.2 Silage program for the 6th order FIR filter.

A.2.2 Hyper

Hyper is a high-level synthesizer that creates a register transfer (RT) level circuit description from a behavioral description of the circuit. The behavioral description must be written in the Silage language. Hyper was developed at U.C. Berkeley.

When run, Hyper searches the current directory for files with ".sil" extensions. Before running the program, create a directory for the 6th order FIR filter, and place the silage file in the new directory (the silage file name should end with ".sil"). Change directory to the new directory. To run Hyper, type:

```
% ahyper
```

This brings up the Hyper command prompt. The command prompt will show the name of the input file read (without the ".sil" extension). Once the command prompt comes up, the following steps should be performed:

```
fir6 -> parse
...
fir6 -> select
...
fir6#0 -> estimate
...
fir6#1 -> set TSAMPLE 8
...
fir6#1 -> estimate
...
fir6#2 -> schedule
...
```

The commands above can generally be abbreviated with their first three letters. "parse" (par) reads the input file and transforms the silage code into a flow graph. "select" (sel) selects primitive hardware modules via the nodes. "estimate" (est) estimates design parameters and performs initial allocation. "schedule" (sch) performs the actual scheduling and module assignment. This step produces a ".sched" file containing a summary of the schedule produced. Included in the summary is the number of modules and control steps required. If the results of a run are unsatisfactory, the "TSAMPLE" value can be changed and the "estimate" and "schedule" steps can be performed again. The value of "TSAMPLE" is used to determine the number of control steps and the number of modules used; as "TSAMPLE" is increased, the number of

control steps tends to increase (and thus the number of modules tends to decrease). Once a satisfactory solution has been achieved, the scheduled and module assigned data flow graph can be found in a numbered ".afl" file created in the working directory; the number of the current ".afl" file is displayed as part of the Hyper command prompt. Figure A.0.3 shows a data flow graph for the 6^{th} order FIR filter with the scheduling and module assignment completed.

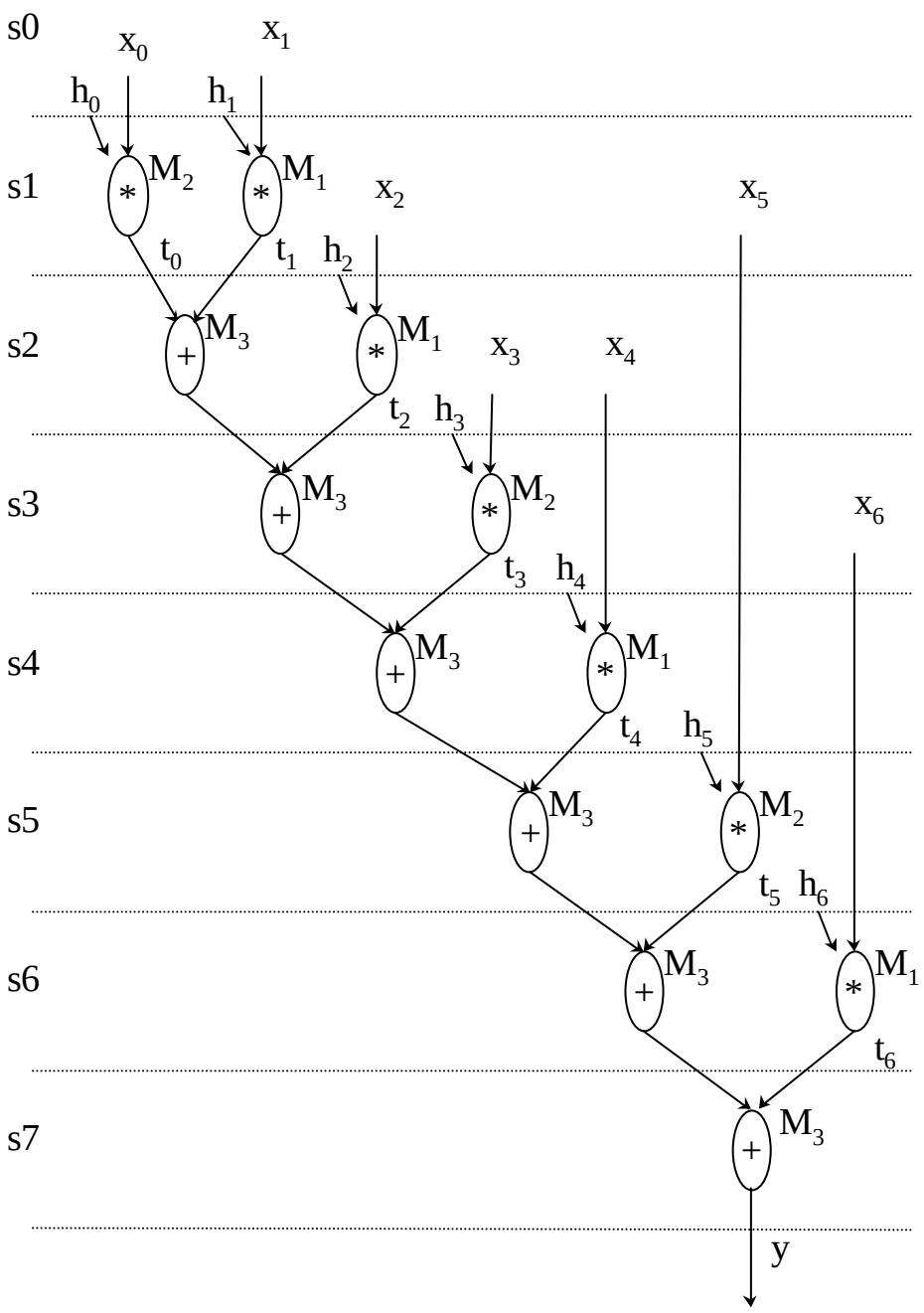


Figure A.0.3 Data flow graph for the 6th order FIR filter

A.3 Output File Conversion

The ".afl" file generated by Hyper cannot be used directly as an input file for the integer linear programming (ILP) based tools described in the upcoming sections. Converting the flow graph file to an acceptable format is a two-step process; first, the ".afl" is run through the "sfggen" program, then the output of this program is run through the "ilpnum" program.

A.3.1 sfggen

The "sfggen" (SFG GENerator) program converts the data flow graph file that was created by Hyper into an intermediate format that can be fed to ilpnum. To run sfggen, type:

```
% sfggen <filename>
```

This will print out the intermediate file to standard output. Typical usage will redirect the output to a file:

```
% sfggen fir6.afl > fir6.sfg
```

Figure A.0.1 contains a portion of the sfggen output file for the 6th order FIR filter.

```
Node(n24) Master(multiply) Mod(mult#160) C(y) Step(6)  In(e24,x6) Out(t6)
Node(n0) Master(multiply) Mod(mult#161) C(y) Step(1)  In(e0,x0) Out(t0)
Node(n33) Master(add) Mod(add#80) C(y) Step(2)  In(t0,t1) Out(e28)
Node(n32) Master(add) Mod(add#80) C(y) Step(3)  In(e28,t2) Out(e29)
Node(input_0) Master(input) Step(0)  Out(x0)
Node(output_0) Master(output) Step(8)  In(y)
Edge(e0) Con(y) Out(n0)
Edge(x0) In(input_0) Out(n0)
Edge(t3) In(n12) Out(n31)
```

Figure A.0.1 Partial listing of "fir6.sig"

A.3.2 ilpnum

The "ilpnum" (Integer Linear Program NUMbering) program takes the intermediate output of sfggen and converts it into a format usable by the ILP-based tools described in the following sections. To run this program, type:

```
% ilpnum [-d] [-s <latency>] <filename>
```

"-d" instructs ilpnum to number based on separate delayed variables, and "-s" instructs ilpnum to number for scheduling with the given latency. ilpnum will print the output file to standard output. Typical usage will redirect the output to a file. To continue with the 6th order FIR filter example, the command would be:

```
% ilpnum fir6.sfg > fir6.in
```

Figure A.0.2 contains a portion of the ilpnum output file for the 6th order FIR filter.

```
( NTA  40 )
( NAG  17 )
( SOP  27 )
( SMO  14 )
( NLTA 2 )
( NK   1 )
( In
(  1 4 7 10 13 16 19 )
...
)
( constant
( 0 3 6 9 12 15 18 )
)
( TaskTime
( 0 1 3 4 7 10 13 16 19 )
...
)
( x
( 27 14 )
( 28 15 )
( 29 14 )
( 30 15 )
  ...
)
( IntConComm
  ( 18 27 0 )
  ( 19 27 1 )
  ...
)
( IntCon
  ( 27 20 0 )
  ( 35 22 0 )
  ( 38 25 0 )
  ( 39 26 0 )
  ...
)

( InpPair
  ( 18 19 27 )
  ( 15 16 28 )
  ( 12 13 29 )
  ...
)
```

Figure A.0.2 Partial listing of "fir6.in"

A.4 Register Assignment and Interconnection Assignment

Once the scheduling and module assignment is completed, registers must be allocated to store the data computed during a control step, and interconnections must be determined. Several different approaches for this are presented here. These methods generally trade execution time and quality of solution.

A.4.1 CPLEX

CPLEX is a program that solves integer linear programming (ILP) problems. This program is used by the register and interconnection assignment programs discussed in this section; the tools either generate input files for CPLEX, or they use CPLEX during their execution.

To run CPLEX, type:

```
% cplex
```

CPLEX accepts interactive commands once it has been started. A typical session involves the following sequence of commands:

```
CPLEX> read <lp file>
...
CPLEX> set timelimit <seconds>
...
CPLEX> optimize
...
CPLEX> display sol var <variable name>
...
```

The first command reads in an ILP file containing the problem's data. The second command tells CPLEX how much time to spend trying to optimize the solution; an alternative command is "`set mip lim node 10000`". Complicated problems can take over a day to optimize completely, so the ability to limit the run time of CPLEX is useful. The third command

initiates the optimization process. Finally, the display command prints out the solution result for a variable. A sequence of display commands is often used to print out all the variables. For sample CPLEX output, see Figure A.0.1.

A sequence of commands is often redirected into CPLEX in a command file; the output can be redirected to a file in the same step, with the following syntax:

```
% cplex < <script file> > <output file>
```

```
Welcome to CPLEX Linear Optimizer 6.0
  with Mixed Integer & Barrier Solvers
Copyright (c) ILOG 1997-1998
CPLEX is a registered trademark of ILOG

Type 'help' for a list of available commands.
Type 'help' followed by a command name for more
information on commands.

CPLEX> set timelimit 86400
New value for time limit in seconds: 86400
CPLEX> read fir6.lp
Problem 'fir6.lp' read.
Read time =    0.07 sec.
CPLEX> optimize
Tried aggregator 2 times.
MIP Presolve eliminated 1437 rows and 662 columns.
Aggregator did 54 substitutions.
Reduced MIP has 269 rows, 268 columns, and 946 nonzeros.
Presolve time =    0.03 sec.
Maximum infeasibility variable selection.
Objective is integral.

        Nodes                                         Cuts/
   Node  Left     Objective  IInf  Best Integer     Best Node    ItCnt

      0     0      792.5000    56                    792.5000      181
*    24     9     1285.0000     0     1285.0000      792.5000      392
    100    45      861.1429    15     1285.0000      856.5000      879
    200    89    infeasible           1285.0000      861.1429     1429
...
Elapsed b&b time =   2.18 sec. (tree size =  0.12 MB)
...

Integer optimal solution:  Objective =    1.1270000000e+03
Solution time =  149.86 sec.  Iterations = 324892  Nodes = 60963

CPLEX> display sol var x0_0
The variable 'x0_0' is 0.
...
CPLEX> display sol var x0_7
Variable Name           Solution Value
x0_7                          1.000000
...
```

Figure A.0.1 Sample CPLEX output

A.4.2 Optimal Assignment

This section presents two register and interconnection assignment tools that attempt to find a globally optimal solution. The first performs register and interconnection assignment only, while the second performs BIST register assignment as well as register and interconnection assignment. These tools were developed at Virginia Tech.

These tools produce linear programming files and command files to be used as input to the CPLEX program. See Figure A.0.2 for a sample command file, and Figure A.0.3 for a sample linear programming file.

```
set timelimit 86400
read fir6.lp
optimize
d sol var x0_0
d sol var x0_1
d sol var x0_2
d sol var x0_3
...
```

Figure A.0.2 Sample generated CPLEX command file

```
Minimize
Obj:    1 Nm1 + 80 Nm2 + 176 Nm3 + 208 Nm4 + 300 Nm5
    + 320 Nm6 + 350 Nm7 + 400 Nm8 + 450 Nm9
Subject To
  c0: x27_14 = 1
  c1: x27_15 = 0
  c2: x27_16 = 0
  c3: x28_14 = 0
...
  c601:  x0_0 + x1_0 + x3_0 + x4_0 + x7_0 + x10_0 + x13_0 + x16_0 + x19_0 <= 1
  c602:  x2_0 + x5_0 + x6_0 + x7_0 + x10_0 + x13_0 + x16_0 + x19_0 <= 1
  c603:  x8_0 + x9_0 + x10_0 + x13_0 + x16_0 + x19_0 + x21_0 <= 1
...
Bounds
  0 <= x0_0 <= 1
  0 <= x0_1 <= 1
  0 <= x0_2 <= 1
  0 <= x0_3 <= 1
...
Integers
  x0_0
  x0_1
  x0_2
  x0_3
...
End
```

Figure A.0.3 Sample generated linear programming file

A.4.2.1 High Level Synthesizer

The first program performs optimal register and interconnection assignment. The syntax for running this tool is:

```
% hls <input file> <command file>
```

The “input file” argument is the name of the “.in” file containing the ILP problem (see Section 0), and the “command file” argument is the name of the CPLEX command file to be created (a “.cmd” extension will be added to this name). The linear programming file generated is printed to standard output.

For our 6th order FIR filter, the command line would read:

```
% hls fir6.in fir6 > fir6.lp
```

The above command generates 2 files: “fir6.lp”, containing the integer linear program representing the problem, and “fir6.cmd”, a CPLEX command file to be used to solve the linear program. To find the solution, CPLEX must be run using these two files as input. See Section 00 for instructions on running CPLEX.

A.4.2.2 High Level BIST Synthesizer

The second program performs register, interconnection, and BIST register assignments concurrently. The syntax for running this tool is:

```
% advbist <input file> <command file> <nk>
```

The “input file” argument is the name of the “.in” file containing the ILP problem (see Section 0), and the “command file” argument is the name of the CPLEX command file to be created (a “.cmd” extension will be added to this name). “nk” is the number of the test session. The linear programming file generated is printed to standard output.

For our 6th order FIR filter, the command line would read:

```
% advbist fir6.in fir6 3 > fir6.lp
```

The above command generates 2 files: "fir6.lp", containing the integer linear program representing the problem, and "fir6.cmd", a CPLEX command file to be used to solve the linear program. To find the solution, CPLEX must be run using these two files as input. See Section 00 for instructions on running CPLEX.

A.4.3 Region by Region Assignment

When execution speed is important, region by region register and interconnection assignment can be used to find a good solution in less time than it would take to find an optimal solution. The program presented in this section performs register, interconnection, and BIST register assignment concurrently, but processes the control steps in a region by region fashion. The user can specify the region size to trade execution time for solution quality. Larger region size generally yields a better solution at the cost of more processing time.

The command name of this tool is "advbist_h". The syntax for running it is:

```
% advbist_h <input file> <region size> <nk> [-m]
```

"input file" is the name of the ".in" file containing the ILP problem. "region size" is the region size to consider when assigning registers and interconnection. "nk" is the number of the test session. The "–m" option causes advbist_h to consider multiplexer cost in the objective function. This produces slightly improved quality solutions, but takes much more time to execute.

advbist_h does not generate input for CPLEX, but rather uses CPLEX directly. The script file "runcplex" should be accessible in the path when advbist_h is called. Two output files are produced by advbist_h. The first file has the same name as the input file, but with a ".out" extension; this file contains the output from CPLEX. The second file created is named "cputime", and contains the amount of processor time that was consumed in finding a solution.

To use this tool on our 6th order FIR filter, type:

```
% advbist_h fir6.in 2 3
```

This command generates the files “fir6.out” and “cputime”. Oftentimes, the files are combined:

```
% cat cputime >> fir6.out
```

A.4.4 Heuristic Assignment

Register and interconnection assignment can be done more quickly (at the cost of solution quality) by using a heuristic. The ADVAN program presented here implements such an algorithm. ADVAN is not ILP-based, although one of the programs that it uses is, so it operates directly on a “.afl” file generated by Hyper (see Section A.0). ADVAN can be configured to use one of three different algorithms to perform its tasks; the following subsections describe how to run each version.

ADVAN is actually three programs that work together. The main program (“advan”) first performs register assignment, then calls on “Iassign” to perform interconnection assignment, and “BassignILP” to perform BIST register assignment. When running ADVAN, the files “advan”, “Iassign”, “BassignILP”, and “bistilp” should all reside in a directory appearing in the path.

A.4.4.1 Default Algorithm

The syntax for running ADVAN is:

```
% advan [-h] [-e] [-s <n>] [-p <priority file>]
     <flow graph file>
```

The “–h” option displays help, “–e” displays detailed results, “–s” lets the user specify the number of the test session, and “–p” lets the user specify a priority file. The priority file can be used to specify priorities for goal measures, such as number of common neighbors, number of non-common edges, and number of self-adjacent edges. The “flow graph file” argument contains the name of the file with the scheduled and module assigned circuit generated by Hyper

(see Section 0). Execution results are printed to standard output (see Figure A.0.4), and an execution summary is printed to the file “bist.out”.

Typical usage of ADVAN is as follows:

```
% advan –s 2 fir6#3.afl > fir6.out
```

```
        ****************************
        *                          *
        *    ADVAN (version 1.0)   *
        *                          *
        ****************************

**** DETAILED RESULTS ****
k R N T S B C M  Cost
... Results written to rtl.net!
... Results written to ilp.in!
2 7 3 2 2 0 0 26 3088  (t6_D7 t5_D6 )(e28_D3 ) [ 2 2 ] [ 3 3 ] [ 4 2 ] [ 5 1 ]
... Results written to rtl.net!
... Results written to ilp.in!
2 7 4 1 2 0 0 26 2772  (t6_D7 t5_D6 )(e32_D7 ) [ 2 2 ] [ 3 3 ] [ 4 2 ] [ 5 1 ]
...

**** SUMMARY OF REGISTER ASSIGNMENT RESULTS ****
1. Flow Graph Information
    Input AFL file        : fir6#3.afl
    Number of modules     : 3
    Priority              :

2. Costs for Each Session Constraint
    session=2 min=2456 max=3660 ave=2919  193/216

3. CPU time               : 1666.43
```

Figure A.0.4 Sample execution results from ADVAN

A.4.4.2 Avra’s Algorithm

ADVAN can be configured to run using Avra’s algorithm (RALLOC) by using the following syntax:

```
% advan –m a <flow graph file>
```

Typical usage redirects the assignment results to a file:

```
% advan –m a fir6#3.afl > fir6_a.out
```

A.4.4.3 Parullkar’s Algorithm

ADVAN can be configured to run using Parullkar's algorithm (BITS) by using the following syntax:

```
% advan –m u <flow graph file>
```

Typical usage redirects the assignment results to a file:

```
% advan –m u fir6#3.afl > fir6_u.out
```

A.5 Design Analysis

Once a design has been generated, it is useful to gather some statistics about the design in order to evaluate it. This section presents several data collection programs for this purpose.

A.5.1 ilpformlist

The "ilpformlist" tool organizes values collected from a CPLEX run. It presents a summary of the number of registers, TPGs, SRs, BILBOs, CBILBOs, and n-input multiplexers used by a design. The syntax for running this program is:

```
% ilpformlist <list file> [<list file> ...] [-w]
```

Each "list file" is the name of a file containing the names of CPLEX output files to be processed, one file name per line. Each CPLEX output file should contain variable printouts from an optimization run. The "–w" option is used when the number of n-input multiplexers is derived from wire rather than from ILP equations. Output from this command is printed to standard output (see Figure A.0.1).

```
k R N T S B C M
Circuit: fir
Method: ILP
6 7 7 0 0 0 0 20 [ 3 4 ] [ 4 2 ]
```

Figure A.0.1 "ilpformlist" output for FIR6 circuit from "hls"

A.5.2 gentabilp

This program tabulates values calculated by the “ilpformlist” program. It also reports the area overhead percentages compared with the optimal result. This program requires that the file “refckt” be in the current directory. The syntax for running “gentabilp” is:

```
% gentabilp <ilpformlist output file>
```

“ilpformlist output file” is the name of a file containing the output from the “ilpformlist” program. Output for “gentabilp” is written to standard output (see Figure A.0.2).

```
Ckt    #regs #muxs Area
tseng    5  14  1600
paulin    5  19  1856
fir    7  20  2576
iir    6  22  2224
dct    6  24  2320
wavelet    7  25  2880
...
fir
k    Method  R  T  S  B  C  M  Area
    ILP        7 0 0 0 0 20 2576   0.0
    ILP        0 0 0 0 0 0  2583   0.0
    ILP        0 0 0 0 0 0  2576   0.0
    ILP        0 0 0 0 0 0  2576   0.0
    ILP        0 0 0 0 0 0  2576   0.0
    ILP        0 0 0 0 0 0  2576   0.0
...
```

Figure A.0.2 “gentabilp” output for FIR6 circuit from “hls”

A.5.3 form

`form` performs a similar function to “ilpformlist”, but organizes values collected from ADVAN rather than from CPLEX. This tool requires that “rparse” and “bparse” be accessible in the path. The syntax for running “form” is:

```
% form <ref file>
```

The “ref file” is the name of an ADVAN output file containing assignment results. Output from this command is printed to standard output (see Figure A.5.3).

```
k R N T S B C M
Circuit: fir4
Method: Advan
1 5 1 1 2 0 1 24 [ 2 9 ] [ 3 2 ]
1 5 1 1 1 0 2 19 [ 2 8 ] [ 3 1 ]
1 6 1 2 2 0 1 24 [ 2 6 ] [ 3 4 ]
1 5 1 1 1 0 2 21 [ 2 7 ] [ 3 1 ] [ 4 1 ]
1 5 1 1 2 0 1 25 [ 2 8 ] [ 3 3 ]
1 6 1 2 2 0 1 24 [ 2 3 ] [ 3 6 ]
1 5 0 2 3 0 0 27 [ 2 7 ] [ 3 3 ] [ 4 1 ]
1 5 0 2 2 0 1 27 [ 2 7 ] [ 3 3 ] [ 4 1 ]
1 5 1 1 1 0 2 19 [ 2 5 ] [ 3 3 ]
1 5 2 0 1 0 2 25 [ 2 3 ] [ 3 5 ] [ 4 1 ]
```

Figure A.0.3 “form” output for FIR circuit

A.5.4 gentab

This program performs a similar function to “gentabilp”, but compares the results of different register and interconnection assignment runs. Like “gentabilp”, “gentab” reports area overhead percentages compared with the optimal result. Also like “gentabilp”, this program requires that the file “refckt” be in the current directory. The syntax for running this program is:

```
% gentab <form output file>
```

“form output file” is the name of a file containing the output from the “form” program. Output for “gentab” is written to standard output (see Figure A.5.4).

```
Ckt      #regs #muxs Area
tseng       5  14  1600
paulin      5  19  1856
fir6a_m3    7  20  2576
iir3c_m3    6  22  2224
dct4        6  24  2320
wavelet6_m3 7  25  2880
tseng
k    Method     R  T  S  B  C  M  Area
    ILP-n        5 2 1 2 0 14 2152   25.7
    ILP-1        5 2 2 0 0 21 2160   25.9
    ADVAN        5 2 1 0 0 23 2368   32.4
3   RALLOC       5 1 0 3 0 14 2300   30.4
    BITS         5 2 1 1 0 20 2436   34.3
```

Figure A.0.4 **“**gentab” output for TSENG circuit

A.6 Register Transfer Level VHDL Generation

Once circuit generation is complete, it is desirable to create a register transfer (RT) level VHDL model of the circuit. Such a model is useful to verify the correctness and timing of the circuit. The program "genvhd" generates information about a synthesized circuit that can be used to create a register transfer level VHDL model of the circuit. This section discusses the mapping of "genvhd" output to VHDL code.

To illustrate the operation of "genvhd", we will use an example circuit consisting of a controller module and a data path module. All circuits generated by the procedures in this document will have a structure similar to this example. The data path consists of registers, modules, and multiplexers. The registers store the necessary data values at each control step. Modules (such as adders and multipliers) operate on the data. The multiplexers control which values the registers store at each control step, as well as on which data the modules operate. The control signals for the multiplexers are generated by the controller module on the "sel" signal line, while the control signals for the registers are generated on the "ren" signal line.

A.6.1 Data Path VHDL Description

This section describes the creation of a VHDL model for the data path portion of the circuit. The data path module contains multiplexers, registers, and modules. To model the data path, we also need constants and output ports.

A.6.1.1 Multiplexer Description

In the genvhd output, multiplexers are describe in the following way:

```
mux2 ( In1 sub#80out mult#160out add#81out mux2out)
```

The genvhd description can be converted to VHDL in the following way:

```
mux2: MUXDP4
     port map (Inp(0) => IN81, Inp(1) => sub80out,
```

```
Inp(2) => mult160out,Inp(3) => add81out,
Sel => sel(3 to 4), Outp => mux2out);
```

For an n-input multiplexer, the component MUXDP*n* is used in VHDL. In this example, a portion of the "sel" signal from the control module is used as the control signal for the multiplexer.

A.6.1.2 Register Description

In the genvhd output, registers are described in the following way:

```
reg1 (mux1out reg1out)
```

The genvhd description can be converted to VHDL in the following way:

```
reg1: REG
      port map (D => mux1out, RB => RB, CLK => CLK,
      REN => ren(1), Q => reg1out);
```

The "REN" signal enables the register.

A.6.1.3 Module Description

In the genvhd output, modules are described in the following way:

```
add#80 (mux20out mux18out add#80out)
mult#160 (mux21out mux11out mult#160out)
```

The genvhd description can be converted to VHDL in the following way:

```
add80out <= mux20out + mux18out;
mult160outw <= mux21out * mux11out;
mult160out <= mult160outw(31 downto 16);
```

We see above that when multiplication is performed, the result is twice the width of the operands. For this example, the operands “mux21out” and “mux11out” are 16 bits wide, and the result of their multiplication is 32 bits wide. After the multiplication is performed, though, the result is truncated back to the width of the operands (16 bits wide here).

A.6.1.4 Constant Desription

In the genvhd output, constants are described in the following way:

```
Cst Name:e13 Value:0.35355339 Id:21 mId:15
```

The genvhd description can be converted to VHDL in the following way:

```
constant ce13: SIGNEDV := conv_signed(23170, 16);
e13 <= ce13;
```

Note that, in general, signal processing algorithms involve coefficient constants which are less than 1 but greater than –1. The conversion from an integer type to the “SIGNEDV” type can be done by the “CONV_SIGNED” function, as was done in the example. To make the coefficient number and integer, the coefficient is multiplied by 2^{16}; to convert the coefficient back to its original value at a later time, the value is divided by 2^{16}.

A.6.1.5 Output Port Description

In the genvhd output, output ports are described in the following way:

```
Out0 : reg9
Out1 : reg7
```

The genvhd description can be converted to VHDL in the following way:

```
OUT80 <= reg9out;
OUT81 <= reg7out;
```

A.6.2 Controller VHDL Description

The controller generates the “SEL” and “REN” control signals for the multiplexers and registers, respectively. The values of “SEL” and “REN” are determined by the current control step number from the input data flow graph. In the genvhd output, the values of the SEL signal for each control step are printed in table format. Each row represents the value of the signal at a particular control step; the first row represents the first control step, the second row represents the second control step, and so on. The “SEL” signal is a bit vector, and it is printed as such in the genvhd output. The following is the genvhd output for the “SEL” signal for our sample circuit:

```
0000000000-0----------------------------
----------------------------------------
----------------------------------------
-01---------------00000---0000-----------
1---------0-----0101001-------00--0---001
--------------00-------------------010---
------------0-0000-----------------001011
1--10-1------1010010010---1010-----000000
---101----111-0100111000010101000001011010
-1010---01----0110-----000------10-010001
---01-11-----0101001001011111101111000011
---1111---1---0010100110100001000100000010
-10--1----1101001010100011111110110001110
-------110--0-0011-----000----01101011100
---11-1-01-10-1110-----100001100011001O1
----------------------------------------
```

The values of the “REN” signal at each control step are also printed in the genvhd output. The format is similar to that of the “SEL” signal, as can be seen from the genvhd output for our sample circuit:

```
 0: 111111110000000
 1: 000000000000000
 2: 000000000000000
 3: 010000000010000
 4: 100000001000010
 5: 000000000001000
 6: 000000000100100
 7: 101100000000001
 8: 001000110100110
 9: 011001000000000
10: 001110000011001
11: 001100101001010
12: 010000110100011
13: 000011001100000
14: 001101010100000
15: 000000000000000
```

Once we have the values of the “SEL” and “REN” signals, creating a VHDL model of the controller is simple. A “STATES” type is declared with the number of control steps necessary for operation, and a signal of that type is declared to hold the current control step. One process is written to move through the control steps, starting at zero, and ending at the last step. Once the controller reaches the last control step, it remains in that control step for an indefinite number of clock cycles, until the reset signal is activated. The last control step is actually a ‘do nothing’ step, in which none of the registers are enabled. This allows the result of the calculation to be held. To generate the “SEL” and “REN” output signals, case statements are used to select the proper values for the current control step. The values are pasted in directly from the tables generated by genvhd. See Figure A.0.1 for the VHDL description.

```
architecture RTL of CONTROLLER is
  type STATES is (S0, S1, S2, ...., S14, S15);

begin
  STATE: process(R, CLK, FSM_STATE) begin
    if R = '0' then -- Reset
      FSM_STATE <= S0;
    elsif CLK'EVENT and CLK = '1' then -- Clock event
      case FSM_STATE is
        when S0 => FSM_STATE <= S1;
        ....
        ....
        when S14 => FSM_STATE <= S15;
        when S15 => FSM_STATE <= S15;
      end case;
    end if;
  end process STATE;

  SELGEN: process (FSM_STATE) begin
    case FSM_STATE is
      when S0  => SEL <= "0000000000-0----------------------------";
      ....
      ....
      when S14 => SEL <= "---11-1-01-10-1110-----100000110001100101";
      when S15 => SEL <= "----------------------------------------";
    end case;
  end process SELGEN;
  RENGEN: process (FSM_STATE) begin
    case FSM_STATE is
      when S0  => REN <= "111111110000000";
      ....
      ....
      when S14 => REN <= "001101010100000";
      when S15 => REN <= "000000000000000";
    end case;
   end process RENGEN;
end RTL;
```

Figure A.0.1 VHDL description of the controller module

Appendix B: DSP Benchmark Circuits

B.1 Introduction

This appendix contains six data flow graphs which were used as benchmark circuits in [40]-[42]. The data flow graphs include the following ones:

- a 6^{th} order FIR (finite impulse response) filter
- a 3^{rd} order IIR (infinite impulse response) filter
- a 6-tap wavelet filter
- a 4-point DCT (discrete cosine transformation) circuit
- tseng studied by Tseng and Siewiorek [32]
- paulin by Paulin and Knight [50]

We adopted the scheduling and module assignment from [17] for tseng and paulin. The other four data flow graphs were synthesized using HYPER [68]. The width of the data path logic is eight for all the circuits.

B.2 6th Order FIR Filter

A 6th order FIR filter [71] is represented as

$$y = h_0 x_0 + h_1 x_1 + h_2 x_2 + h_3 x_3 + h_4 x_4 + h_5 x_5 + h_6 x_6$$

where h_n is a filter coefficient, and x_n is a delayed value of x_{n-1}. The equation consists of seven multiplications and a summation. Fig. B.1 shows a block diagram of a 6th order FIR filter.

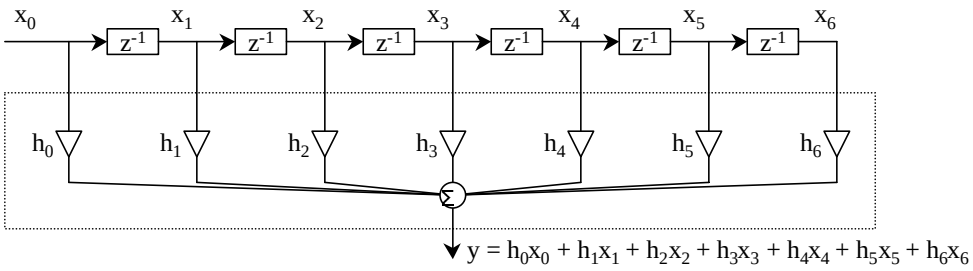


Fig. B.1 A 6th order FIR filter

Fig. B.2 is a Silage program which describes the behavior of the 6th order FIR filter. We assume that delay elements (outside the dashed line in Fig. A.1) are implemented as a shift register.

```
#define num8    num<8,7>
#define h0 0.717
:

func main(x0, x1, x2, x3, x4, x5, x6 : num8) y : num8 =
begin
      t0  = num8(h0  * x0);
      t1  = num8(h1  * x1);
      t2  = num8(h2  * x2);
      t3  = num8(h3  * x3);
      t4  = num8(h4  * x4);
      t5  = num8(h5  * x5);
      t6  = num8(h6  * x6);
      y = (((((t0 + t1) + t2) + t3) + t4) + t5) + t6;
end;
```

Fig. B.2 Silage program of a 6th order FIR filter

Fig. B.3 shows a data flow graph in which the scheduling and the module assignment are completed.

Characteristic	Value
Critical path delay	7 units
Scheduling constraint	8 units
Modules	2 multipliers, 1 adder

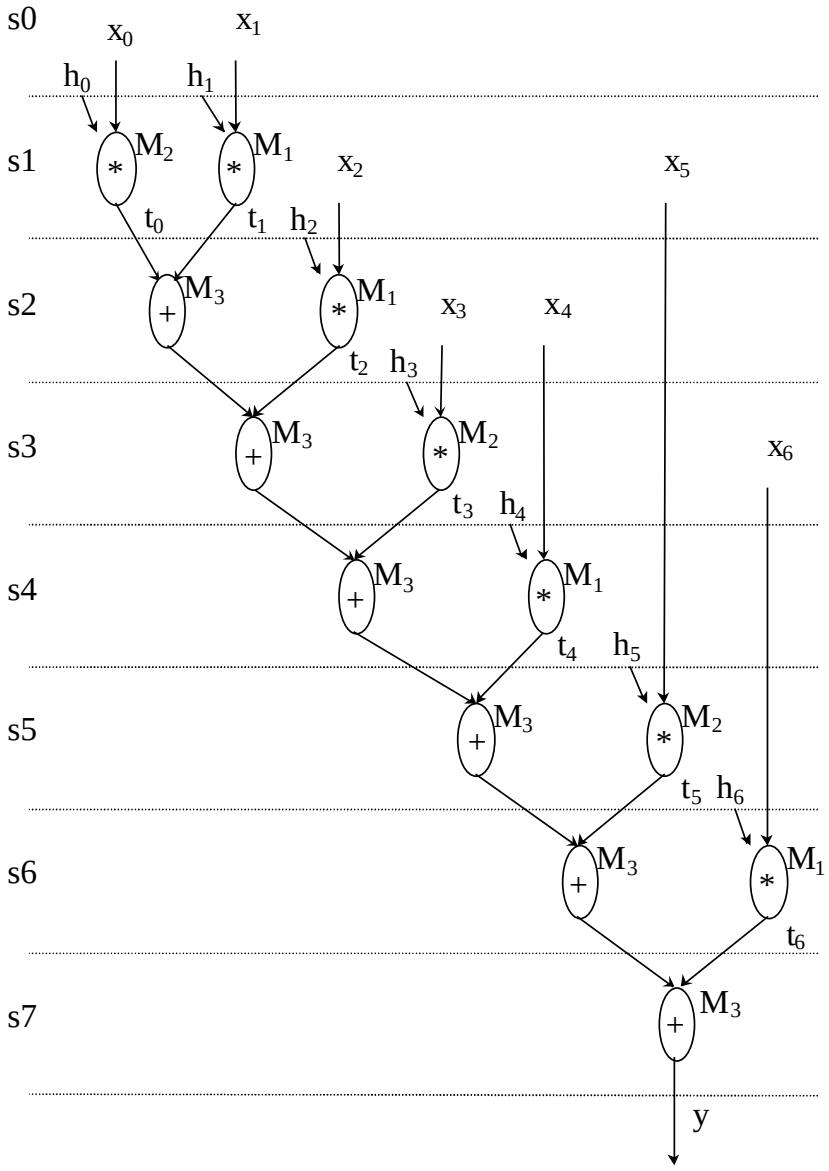


Fig. B.3 Data flow graph of a 6th order FIR filter

B.3 3rd Order IIR Filter Cascade Connection

Fig. A.4 shows a 3rd order IIR filter [71] which is implemented as a cascade of second- and first-order building blocks. Fig. B.5 is the Silage program.

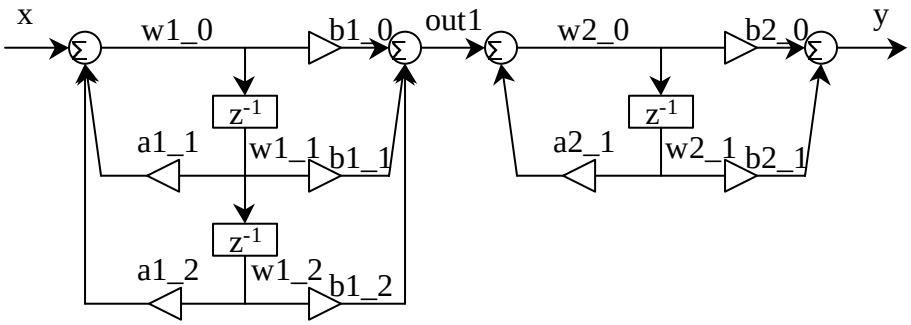


Fig. B.4 A 3rd order IIR filter (cascade connection)

```
#define num8    num<8,7>

#define a1_1    num8(0.010)
:

func main(x, w1_1, w1_2, w2_1 : num8) w1_0, w2_0, y : num8 =
begin
      t1_1 = num8(a1_1 * w1_1);
      t1_2 = num8(a1_2 * w1_2);
      w1_0 = x + (t1_1 + t1_2);

      u1_0 = num8(b1_0 * w1_0);
      u1_1 = num8(b1_1 * w1_1);
      u1_2 = num8(b1_2 * w1_2);
      out1 = u1_0 + (u1_1 + u1_2);


      t2_1 = num8(a2_1 * w2_1);
      w2_0 = out1 + t2_1;

      u2_0 = num8(b2_0 * w2_0);
      u2_1 = num8(b2_1 * w2_1);


      y    = u2_0 + u2_1;
end;
```

Fig. B.5 Silage program of a 3rd order IIR filter (cascade connection)

Fig. B.6 shows a data flow graph in which the scheduling and the module assignment are completed.

Characteristic	Value
Critical path delay	9 units
Scheduling constraint	9 units
Modules	2 multipliers, 1 adder

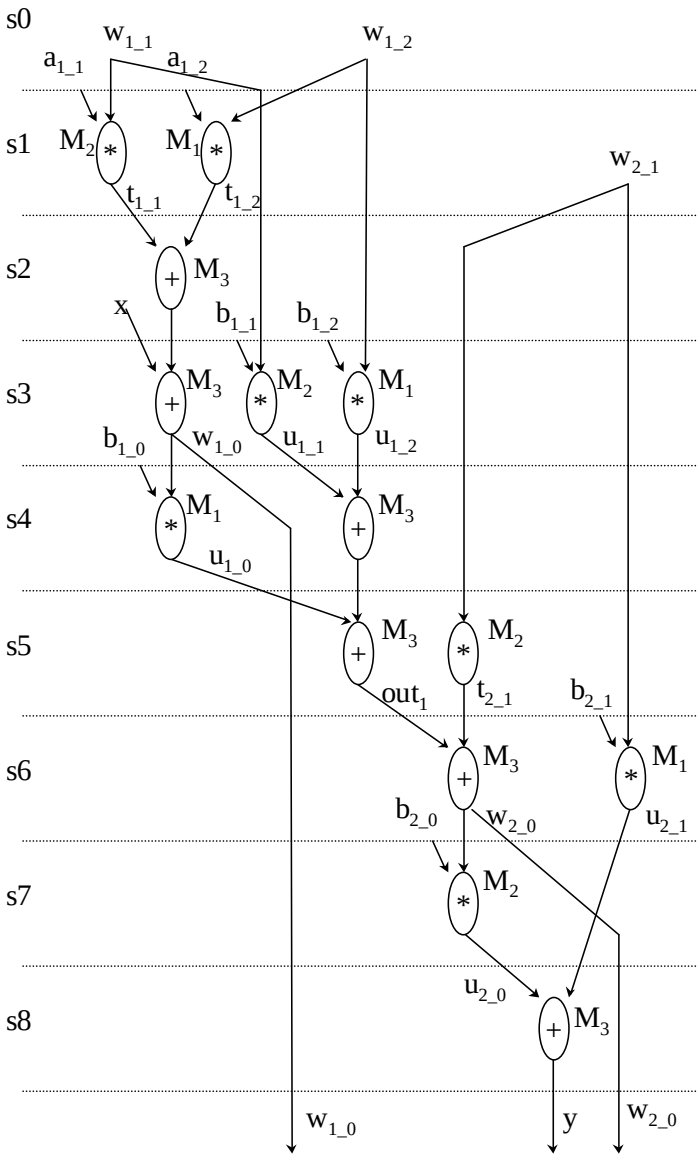


Fig. B.6 Data flow graph of a 3rd order IIR filter (cascade connection)

B.4 6-Tap Wavelet Filter

The low pass filter A_n and the high pass filter D_n of a 6-tap (Daubcheis 6) wavelet filter [72],[73] are given as

$$A_n = a_0 x_n + a_1 x_{n-1} + a_2 x_{n-2} + a_3 x_{n-3} + a_4 x_{n-4} + a_5 x_{n-5}$$
$$D_n = a_0 x_n - a_1 x_{n-1} + a_2 x_{n-2} - a_3 x_{n-3} + a_4 x_{n-4} - a_5 x_{n-5}$$

where a_0, a_1, …, a_5, are the Daubechies 6 coefficients. Fig. B.7 shows the block diagram. Fig. B.8 shows the Silage program.

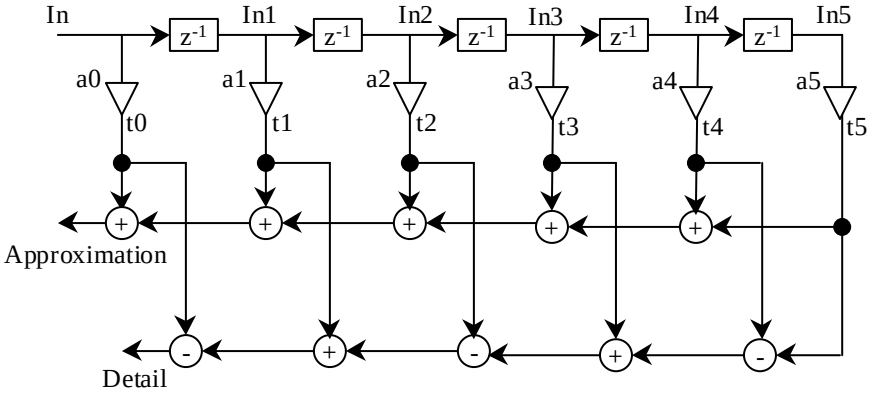


Fig. B.7 A 6-tap wavelet filter

```
#define num8    num<8,7>

#define a0      num8(0.3327)
#define a1      num8(0.8069)
#define a2      num8(0.4598)
#define a3      num8(-0.1350)
#define a4      num8(-0.0854)
#define a5      num8(0.0352)

func main(In, In1, In2, In3, In4, In5 : num8) Approximation, Detail : num8 =
begin
      t0 = num8(a0 * In);
      t1 = num8(a1 * In1);
      t2 = num8(a2 * In2);
      t3 = num8(a3 * In3);
      t4 = num8(a4 * In4);
      t5 = num8(a5 * In5);
      Approximation = t0 + (t1 + (t2 + (t3 + (t4 + t5))));
      Detail        = t0 - (t1 + (t2 - (t3 + (t4 - t5))));
end;
```

Fig. B.8 Silage program of a 6-tap wavelet filter

Fig. B.9 shows a data flow graph in which the scheduling and the module assignment are completed.

Characteristic	Value
Critical path delay	7 units
Scheduling constraint	12 units
Modules	1 multiplier, 1 adder, 1 subtractor

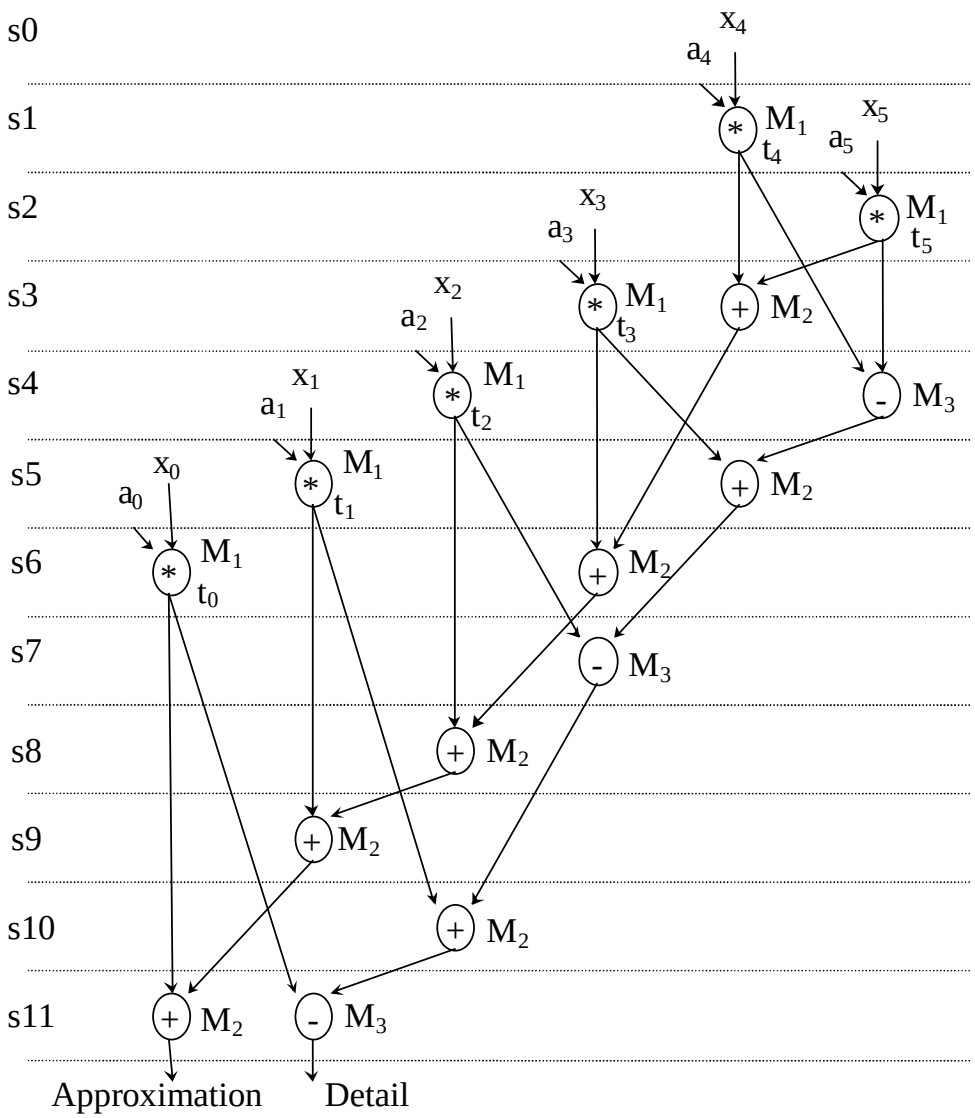


Fig. B.9 Data flow graph of a 6-tap wavelet filter

B.5 4-point Discrete Cosine Transformation (DCT)

Fig. B.10 is a Silage program which describes the behavior of 4-point discrete cosine transformation.

```
#define num16 fix<8,3>

#define beta num16(0.707107)
#define cosa num16(0.923880)
#define sina num16(0.382683)

func main (In0, In1, In2, In3 : num16)
   Out0, Out1, Out2, Out3 : num16 =
begin
   /* Stage 1 */
   (a0, a3) = butterfly(In0,In3);
   (a1, a2) = butterfly(In1,In2);

   /* Stage 2 */
   (b0, b1) = butterfly(a0,a1);
   (Out2, Out3) = complex_mult(a2, a3);

   /* Stage 3 */
   Out0 = num16 (b0 * beta);
   Out1 = num16 (b1 * beta);
end;

func complex_mult(In1, In2 : num16) Out1, Out2 : num16 =
begin
   Out1 =   num16 (In1 * sina) + num16 (In2 * cosa);
   Out2 = - num16 (In1 * cosa) - num16 (In2 * sina);
end;

func butterfly(In1, In2: num16) Out1, Out2 : num16 =
begin
   Out1 = In1 + In2;
   Out2 = In1 - In2;
end;
```

Fig. B.10 Silage program of a 4-point discrete cosine transformation

Fig. B.11 shows a data flow graph in which the scheduling and the module assignment are completed.

Characteristic	Value
Critical path delay	5 units
Scheduling constraint	7 units
Modules	1 multiplier, 1 adder, 1 subtractor

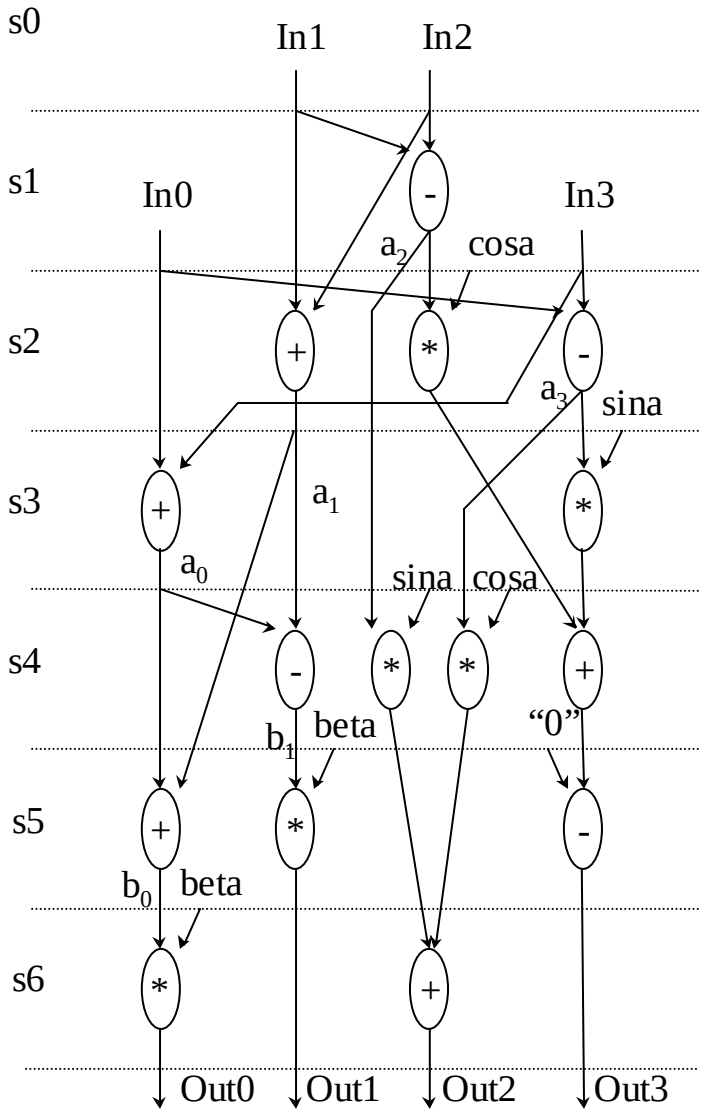


Fig. B.11 Data flow graph of a 4-point discrete cosine transformation

B.6 Tseng

Fig. B.12 shows a data flow graph which was studied by Tseng and Siewiorek in [32].

Characteristic	Value
Critical path delay	5 units
Modules	a1, a2, a3 (ALUs)

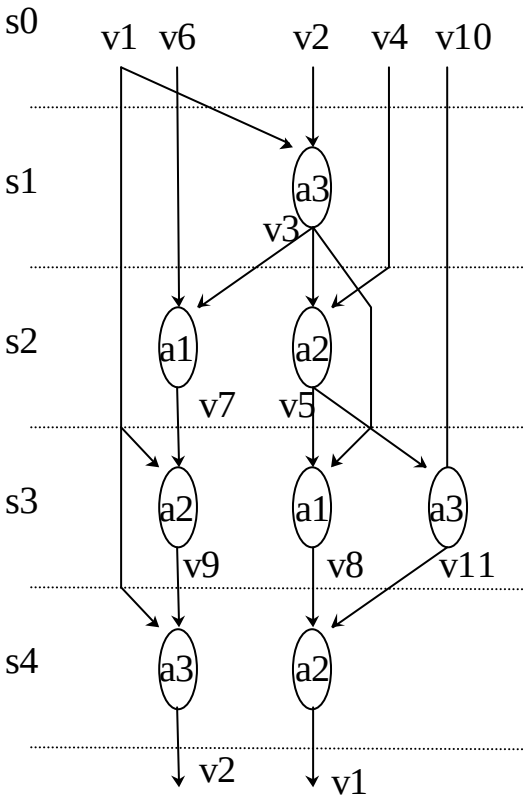


Fig. B.12 Data flow graph of Tseng

B.7 Paulin

Fig. B.13 shows a data flow graph which was studied by Paulin and Knight in [50].

Characteristic	Value
Critical path delay	5 units
Modules	2 multipliers, 1 adder, 1 subtractor

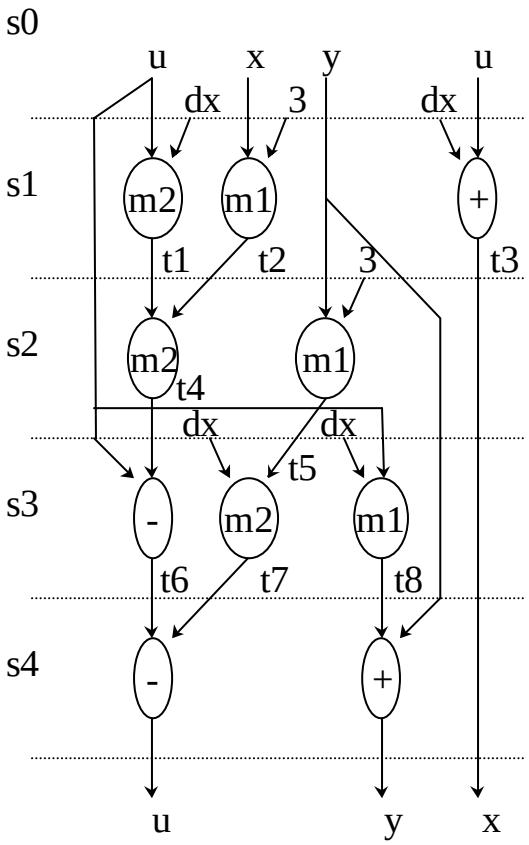


Fig. B.13 Data flow graph of Paulin

Bibliography

[1] L.J. Avra and E.J. McCluskey, “High-Level Synthesis of Testable Designs: an Overview of University Systems,” *Proc. Int’l. Test Conf.*, TS Paper 1.1, pp. 1-8, Oct. 1994.

[2] S. Chiu and C.A. Papachristou, "A Design for Testability Scheme with Applications to Data Path Synthesis," *Proc. 28th Design Automation Conf.*, pp. 271-277, June 1991.

[3] C.-H. Chen, T. Karnik and D.G. Saab, "Structure and Behavioral Synthesis for Testability Techniques," *IEEE Trans. on Computer-Aided Design*, Vol. 13, No. 6, pp 777-785, June 1994.

[4] A. Majumdar, R. Jain, and K. Saluja, “Incorporating Testability Considerations in High-Level Synthesis,” *J. Electronic Testing: Theory & Applications*, pp. 43-55, Feb. 1994.

[5] P. Vishakantaiah, J.A. Abraham, and M. Abadir, "Automatic Test Knowledge Extraction from VHDL (ATKET)," *Proc. 29th Design Automation Conf.*, pp. 273-278, June 1992.

[6] T. Kim, K.-S. Chung, and C.L. Liu, "A Stepwise Refinement Data Path Synthesis Procedure for Easy Testability," *European Test Conf.*, pp.586-590, Mar. 1994.

[7] F.F. Hsu, E.M. Rudnick and J.H. Patel, "Enhancing High-Level Control-Flow for Improved Testability," *Intl. Conf. on Computer-Aided Design*, Nov. 1996.

[8] P. Vishakantaiah, T. Thomas, J.A. Abraham, and M. Abadir, "AMBIANT-Automatic Generation of Behavioral Modifications for Testability," *Intl. Conf. on Computer Design*, pp. 63-66, Oct. 1993.

[9] C.-H. Chen and D.G. Saab, "A Novel Behavioral Testability Measure," *IEEE Trans. on Computer-Aided Design*, Vol. 12, No. 12, pp. 1960-1970, Dec. 1993.

[10] T.-C. Lee, N.K. Jha, and W.H. Wolf, "Behavioral Synthesis for Highly Testable Data Paths under the Non-Scan and Partial Scan Environments," , *Proc. 30th Design Automation Conf.*, pp.292-297, June 1993.

[11] T.-C. Lee, N.K. Jha, and W.H. Wolf, "A Conditional Resource Sharing Method for Behavioral Synthesis of Highly Testable Data Paths," *Proc. Int’l. Test Conf.*, pp.744-753,

Oct. 1993.

[12] S. Bhatia and N.K. Jha, “Genesis: A Behavioral Synthesis System for Hierarchical Testability,” *European Test Conference*, pp.272-276, Mar. 1994.

[13] A. Majumdar, R. Jain, and K. Saluja, "Behavioral Synthesis of Testable Designs," *24th Fault-Tolerant Computing Symposium,* pp.436-445, June 1994.

[14] S. Bhatia and N.K. Jha, “Behavioral Synthesis for Hierarchical Testability of Controller/Data Path Circuits with Conditional Branches,” *Intl. Conf. on Computer Design*, Oct. 1994.

[15] T.-C. Lee, W.H. WoIf, N.K. Jha and J.M. Acken, "Behavioral Synthesis for Easy Testability in Data Path Allocation,” *Intl. Conf. on Computer Design***,** pp.29-32, Oct. 1992.

[16] T.-C. Lee, W.H. Wolf, and N.K. Jha, "Behavioral Synthesis for Easy Testability in Data Path Scheduling," *Intl. Conf. on Computer-Aided Design*, pp.616-619, Nov. 1992.

[17] L.J. Avra, "Allocation and Assignment in High-Level Synthesis for Self-Testable Data Paths," *Proc. Int. Test Conf.*, pp. 463-472, Oct. 1991.

[18] C.A. Papachristou, S. Chiu and H. Harmanani, "A Data Path Synthesis Method for Self-Testable Designs, " *Proc. 28rd Design Automation Conf*., pp. 378-384, June 1991.

[19] C.A. Papachristou and J. Carletta, "Test Synthesis in the Behavioral Domain*," Proc. Int’l. Test Conf*., pp. 693-702, Oct. 1995.

[20] I. Parulkar, S. Gupta, and M.A. Breuer, “Data Path Allocation for Synthesizing RTL Designs with Low BIST Area Overhead,” *Proc. 32nd Design Automation Conf*., pp. 395-401, June 1995.

[21] I. Parulkar, S. Gupta, and M.A. Breuer, “Introducing Redundant Computations in a Behavior for Reducing BIST Resources,” *Proc. 35th Design Automation Conf*., pp. 548-553, June 1998.

[22] I. Parulkar, S. Gupta, and M.A. Breuer, “Scheduling and Module Assignment for Reducing BIST Resources,” *Proc. DATE*, pp. 66-73, Feb. 1998.

[23] I. Parulkar, S.K. Gupta and M.A. Breuer, "Lower Bounds on Test Resources for Scheduled Data Flow Graphs," *Proc. 33rd Design Automation Conf*., pp. 143-148, June 1996.

[24] H. Harmanani and C.A. Papachristou, "An Improved Method for RTL Synthesis with Testability Tradeoff," *Intl. Conf. on Computer-Aided Design*, pp. 30-35, Nov. 1993.

[25] I.G. Harris and A. Orailoglu, "Microarchitectural Synthesis of VLSI Designs with High Test Concurrency," *Proc. 31st Design Automation Conf.*, pp. 206-211, June 1994.

[26] A. Orailoglu and I.G. Harris, "Microarchitectural Synthesis for Rapid BIST Testing," *IEEE Trans. Computer-Aided Design*, Vol.16, No. 6, pp. 573-586, June 1997.

[27] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.

[28] M. Abramovici, M.A. Breuer and A.D. Friedman, *Digital Systems Testing and Testable Design*, IEEE, Jan. 1998.

[29] S. Pilarski, A. Krasniewski, and T. Kameda, "Estimating Testing Effectiveness of the Circular Self-Test Path Technique," *IEEE Trans. on Computer-Aided Design*, Vol. 11, No. 10, 1301-1316, Oct. 1992.

[30] G. DeMichelli, *Synthesis and Optimization of Digital Circuits*, McGraw Hill, 1994.

[31] A. Hashimoto and J. Stevens, "Wire Routing by Optimizing Channel Assignment within Large Apertures," *Proc. 8th Design Automation Workshop*, pp. 155-163, 1971.

[32] C. Tseng and D.P. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems," *IEEE Trans. on Computer-Aided Design*, pp. 379-395, July 1986.

[33] M. Gomunbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, San Diego, CA, 1980.

[34] F.J. Kurdahi and A.C. Parker, "REAL: A Program for Register Allocation," *Proc. 24th Design Automation Conference*, pp. 210-215, 1987.

[35] L. Hafer and A. Parker, "A formal method for the specification, analysis, and design of register-transfer level digital logic," *IEEE Trans. on Computer-Aided Design*, Vol. 2, Jan. 1983.

[36] C.T. Hwang, J.H. Lee, and Y.C. Hsu, "A formal approach to the scheduling problem in high level synthesis," *IEEE Trans. on Computer-Aided Design*, Vol. 10, Apr. 1991.

[37] C.T. Hwang and Y.C. Hsu, “Zone scheduling,” *IEEE Trans. on Computer-Aided Design*, Vol. 12, No. 7, pp. 926-pp.934, July 1993.

[38] C.H. Gebotys and M.I. Elmasry, “Global optimization approach for architecture synthesis,” *IEEE Trans. Computer-Aided Design*, vol. CAD-12, pp.1266-1278, Sept. 1993.

[39] M. Rim, A. Mujumdar, R. Jain, and R. DeLeone, “Optimal and heuristic algorithms for solving the binding problem, ” *IEEE Trans. on VLSI Systems*, vol. 2, No. 2, pp. 211-225, June 1994.

[40] H.B. Kim, T. Takahashi, and D.S. Ha, "Test session oriented built-in self-testable data path synthesis,” *Proc. Int. Test Conf.*, pp. 154-163, Oct. 1998.

[41] H.B. Kim, D.S. Ha, and T. Takahashi, “On ILP Formulations for Built-In Self-Testable Data Path Synthesis,” *Proc. IEEE 36th Design Automation Conference*, pp. 742-747, June 1999.

[42] H.B. Kim and D.S. Ha, “A High-Level BIST Synthesis Method Based on a Region-wise Heuristic for an Integer Linear Programming,” *IEEE International Test Conference*, pp. 903-912, Sept. 1999.

[43] K. P. Rao, *Discrete Cosine Transform: Algorithms, Advantages, applications*, Academic Press, 1990.

[44] K.P. Rao and J.J. Hwang, *Techniques and Standards for Image, Video and Audio Coding*, Prentice Hall PTR, 1996.

[45] P. Hilfinger, "A High-level Language and Silicon Compiler for Digital Signal Processing", *Proc. IEEE Custom Integrated Circuits Conference*, pp. 213-216, May 1985

[46] W.H. Chen, C.H. Smith, and S.C. Fralick, “A Fast Computational Algorithm for the Discrete Cosine Transform,” *IEEE Trans. Commun.*, Vol. COM-25, pp. 1004-1009, Sept. 1977.

[47] B.G. Lee, “A New Algorithm to Compute the Discrete Cosine Transform,” *IEEE Trans. Acoust., Speech, and Signal Processing*, Vol. ASSP-32, pp.1243-1245, Dec. 1984.

[48] N.I. Cho and S.U. Lee, “Fast Algorithm and Implementation of 2-D Discrete Cosine Transform”, *IEEE Trans. Circuits and Systems*, Vol. 38, No. 3, pp. 297-305, Mar. 1991

[49] ftp://mcnc.mcnc.org/pub/benchmark/HLSynth89.

[50] P.G. Paulin and J.P. Knight, “Force-directed Scheduling for the Behavioral Synthesis of ASICs,” *IEEE Trans. on Computer-Aided Design*, Vol. 8, No. 6, pp. 661-679, June 1989.

[51] J. Nestor and D. Thomas, “Behavioral Synthesis with Interfaces,” *Intl. Conf. on Computer-Aided Design*, pp.112-115, Nov. 1986.

[52] D. Kim and K. Choi, “Power-conscious High Level Synthesis Using Loop Folding,” *Proc. 34th Design Automation Conf.*, pp.441-445, June 1997.

[53] A. Chandrakasan, S. Sheng and R. Brodersen, “Low-power CMOS Digital Design,” *IEEE Journal of Solid-State Circuits*, vol, 27, no. 4, pp. 473-484, Apr. 1992.

[54] C.-Y. Huang, Y.-S. Chen, Y.-L. Lin, and Y.-C. Hsu, “Data Path Allocation Based on Bipartite Weighted Matching,” *Proc 27th Design Automation Conference*, pp. 499-504, 1990.

[55] N. West and K. Eshraghian, *Principles of CMOS VLSI Design: A systems Perspective*, Second Edition, Addison Wesley, 1993.

[56] B.Pangrle, “On the Complexity of Connectivity Binding, ” *IEEE Trans. on Computer-Aided Design*, Vol. 10, No 11, pp. 1460-1465, Nov. 1991.

[57] Konemann, B.J. Mucha, and G. Zwiehoff, “Built-In Logic Block Observation Techniques,” *Proc. Int’l Test Conf.*, pp. 37-41, Oct. 1979.

[58] L.-T. Wang and E.J. McCluskey, “Concurrent Built-In Logic Block Observer (CBILBO),” *Int. Symp. On Circuits and Systems*, pp. 1054-1057, May 1986.

[59] K. Ogawa et al., "A Single Chip Compression/Decompression LSI Based On JPEG," *IEEE Trans. on Consumer Electronics*, vol. 38, pp.703-710, Aug. 1992.

[60] M. Nakagawa et al., "DCT-Based Still Image Compression ICs with Bit-Rate Control," *IEEE Trans. Consumer Electronics*, vol. 38, pp. 711-717, Aug. 1992.

[61] P.A. Ruetz et al., "A Video-Rate JPEG Chip Set," *VLSI video/image signal processing*, Edited by T. Nishitani, P.H. Ang, and F. Catthoor, Kluwer Academic, 1993.

[62] E. Scopa et al., "A 2D-DCT Low-Power Architecture For H.261 Coders," *IEEE Int'l Conf. on Acoust., Speech, and Signal Processing*, pp. 3271-3274, May 1995.

[63] D. Slawecki and Weiping Li, "DCT/IDCT Processor Design For High Data Rate Image Coding," *IEEE Trans. Circuits and Systems for Video Technology*, Vol. 2, No. 2, pp. 135-146, June 1992.

[64] A. Madisetti and A.N. Wilson, Jr., "A 100 MHz 2-D 8x8 DCT/IDCT Processor For HDTV Applications, " *IEEE Trans. Circuits and Systems for Video Technology*, vol. 5, pp 158-165, Apr. 1995.

[65] T. Xanthopoulos and A.P. Chandrakasan, "A Low-Power IDCT Macrocell for MPEG-2 MP@ML Exploiting Data Distribution Properties for Minimal Activity," *IEEE Journal of Solid-State Circuits*, Vol. 34-, No. 5, pp. 693-703, May 1999.

[66] V. Srinivasan and K.J.R. Liu, "VLSI Design of High-Speed Time-Recursive 2-D DCT/IDCT Processor for Video Application", *IEEE Trans. on Circuits and Systems for Video Technology*, Vol 6, no 1, pp.87-96, Feb. 1996.

[67] *CPLEX* 6.0 *Reference Manual*, ILOG, 1998.

[68] M. Potkonjak and J. Rabaey, "A Scheduling and Resource Allocation Algorithm for Hierarchical Signal Flow Graphs," *Proc. 26th Design Automation Conf.*, pp. 7-12, June 1989.

[69] T. Takahashi, H.B. Kim, and D.S. Ha, "BIST Synthesis - I," Technical Report, VISC-DSH-1-98, Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, July 1998.

[70] N. Park and A. Parker, "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications," *IEEE Trans. on Computer-Aided Design*, Vol. 7, No. 3, pp. 356-370, March 1988.

[71] E. C. Ifeachor and B. W. Jervis, *Digital Signal Processing*, Addison-Wesley, 1993.

[72] G. Strang and T. Nguyen, *Wavelets and Filter Banks*, Wellesley Cambridge, 1996.

[73] M. Vetterli and J. Kovacevic, *Wavelets and Subband Coding*, Prentice Hall, 1995.

Vita

Han Bin Kim was born in Seoul, Korea on July 20, 1965. He received a Bachelor of Science and a Master of Science in Electronic Engineering from Sogang University, Seoul, Korea in 1988 and 1990, respectively. From 1990 to 1996, he was with Samsung Electronics Company, Kihung, Korea, where he was involved in the development and application of CAD tools such as logic synthesis, static timing verification. He then became a Ph.D. student in Electrical and Computer Engineering at Virginia Polytechnic Institute and State University, Blacksburg VA. In January 2000, he began employment at Sun Microsystems, Sunnyvale CA as a hardware engineer involved with timing verification of Ultrasparc CPU. His research interests are in IC design and verification with emphasis on high performance, low power consumption, verification and test.