Pipelining of Register Transfer Netlists

by

Kenichi Kanehara
Daniel D. Gajski

Technical Report 91-12

Information and Computer Science Department
University of California, Irvine
Irvine, CA. 92717

Abstract

This paper describes a method for pipelining of register-to-register netlists. We define algorithms for inserting latches in a data path, both inside each unit and between the units as well as between control logic and the data path and for readjusting the state transition table. Experimental results on several benchmarks show 30%-40% improvement in performance.

TABLE OF CONTENTS

1. Introduction

The micro-architecture of a chip or a portion of it is usually described by the register transfer netlists that contain register transfer components such as counters, registers, alus, multiplexers, multipliers and other random logic consisting of gates and flip-flops. In order to satisfy performance constraints such a netlist is usually optimized by reducing delays on all register-to-register critical paths.

Two techniques are usually employed: retiming and pipelining. Retiming moves latches and storage elements in order to shorten long paths. These may result in more storage elements but improved performance [MaSi90] [MaSe90]. Pipelining method inserts new latches or registers on critical paths, thus shortening the clock period [McCa90] [NoCa90].

There are three possible places for insertion of pipeline latches.

(1) large functional units such as multipliers, or floating point adders can be pipelined in several stages

(2) latches can be inserted between any two combinational operators or logic partitions, and

(3) latches can be inserted not only in the data path but also in the control portion of the design.

All of the above methods require some restructuring of the design to make sure that original computation is preserved, that is, the arrival time of corresponding operations for each functional unit must stay the same. Thus if different number of registers are inserted on the convergent paths, control sequence must be denied, by

rescheduling operations into different control steps.

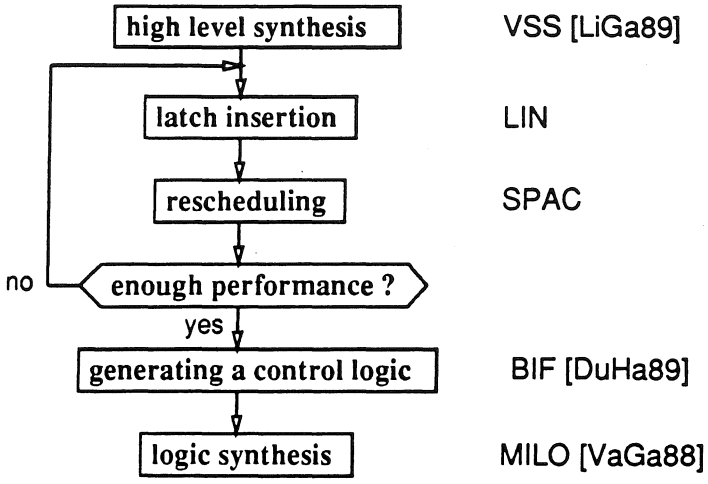


Figure 1-1. Design flow

Two resynthesis tools for pipelining register-transfer netlist are described in this paper.

(i) The first tool LIN inserts latches in a data path, both inside each unit and between the units as well as between control logic and the data path

(ii) The other tool SPAC readjusts the state transition table which contains conditional operations and jumps

The design methodology using these new tools are shown in Figure 1-1. New pipelining tools, LIN (Latch Insertion on Netlist) and SPAC (SPlit And Compaction) are incorporated with other UCI tools.

The rest of the paper is organized as follows: Latch insertion is discussed in Section 2, while rescheduling is discussed in Section 3. Experiments and results are discussed in Section 4. Section 5 concludes this report.

2. Latch insertion

2.1. Problem description

Pipelining is a well known engineering method for performance enhancement.

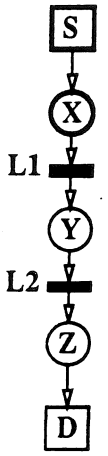


Figure 2-1. Pipeline example

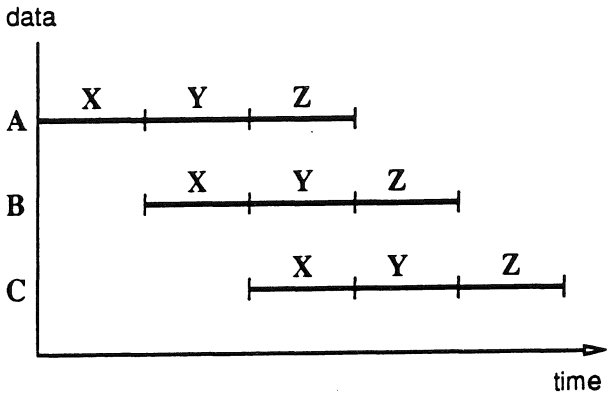


Figure 2-2. Parallel execution of pipeline

Figure 2-1 shows a pipelined design that obtained by inserting latches L1 and L2 into an original register-transfer netlist that performs operations X, Y and Z on data A, B and C and deposits the result in register D. After inserting latches, operators X, Y and Z can be used simultaneously on different sets of data A, B, C, stored in S. Figure 2-2 shows how computation X, Y and Z on data A, B, C is accomplished over time.

The delay from S to D determines the clock period CP:

$$CP = Tx + Ty + Tz + Tst$$

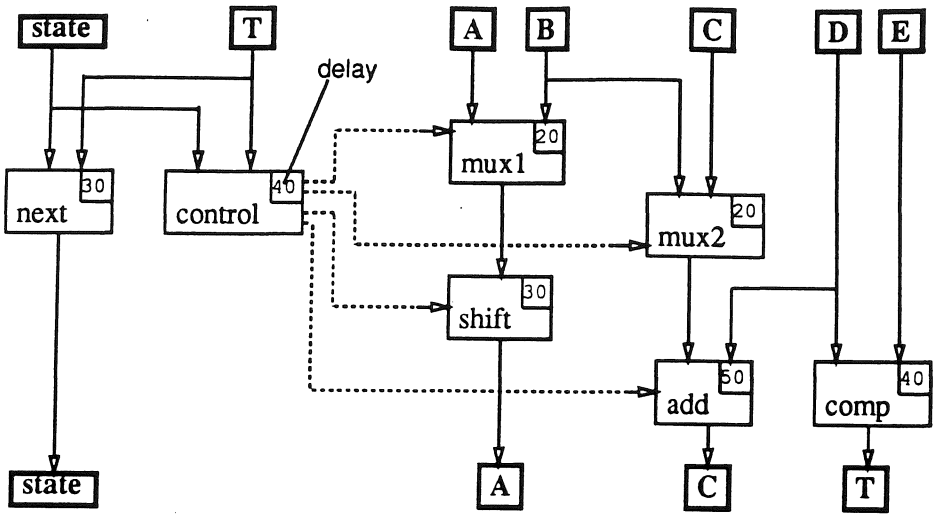


Figure 2-3. Original circuit (clock period = 110)

where Tx, Ty and Tz are delays of operations X, Y and Z, and Tst is the setup time of register D. By inserting latches L1 and L2 the pipelined clock period becomes CPP:

$$CPP = \max\{Tx,Ty,Tz\} + Tst$$

Parallel operations and a reduced clock period increase a system throughput.

Performance of a complete synchronous pipelined system is given by C * (1 / clock_period) * (pipeline_efficiency) where C is a constant, and pipeline_efficiency defines average concurrent use of available operators. The pipeline efficiency will be discussed in section 3 in detail.

In an arbitrary register transfer netlist a clock period is equal to the largest propagation delay on all paths from any register to any register. Figure 2-3 shows a

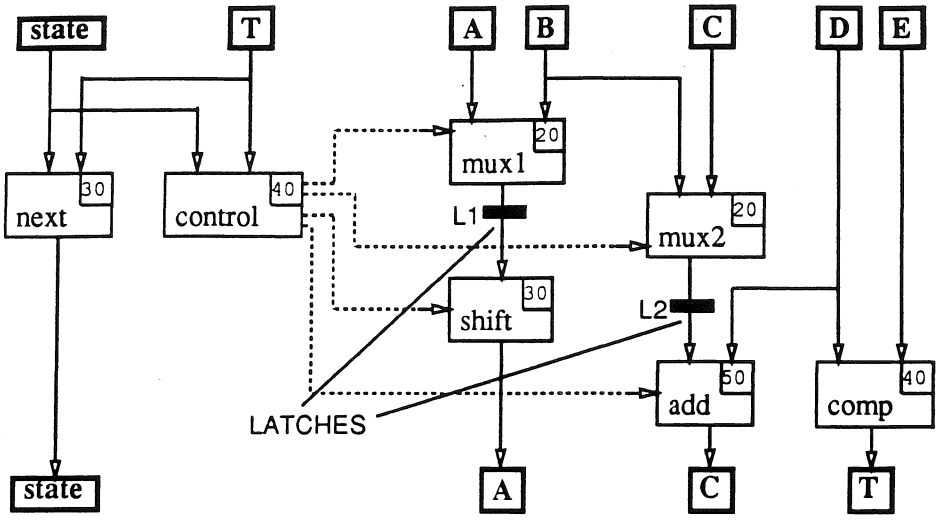


Figure 2-4. Design example with latches in the data path (clock period = 90)

micro-architecture example. The longest path is from registers state or T to register C through units control, mux2 and add. Therefore the clock period is equal to

$$CP = 40 + 20 + 50 + \text{setup} = 110 + \text{setup}$$

Latches can be inserted in three different places which require different design adjustments. First, the latches can be inserted between two units. Figure 2-4 shows the original design with 2 latches inserted between mux1 and shift units and between mux2 and add units. The longest path is now from state/T to C through control and add. The clock period is reduced from 110 to 90.

Second place for inserting latches is inside a functional unit. Using the first method the clock period can be only reduced to the delay of the slowest unit. Thus replacing units with multi stage pipes can reduce clock period further, to the delay of

the slowest stage of the pipelined units. Figure 2-5 shows the new design which obtained by replacing one adder with a 2-stage adder. The longest path is now from state/T to A through control and shift. The clock period has been reduced from 90 to 70.

The last place for inserting latches is between control logic and data path, since usually the longest path contains control logic. Figure 2-6 shows the design obtained by inserting latches L4, L5, L6 and L7 between control logic and a data path in Figure 2-5. The longest is now from state/T to L4/L5/L6/L7 or from D/E to T. The clock period is reduced from 70 to 40.

Thus the main problem in pipelining arbitrary register-transfer netlists consists of inserting minimal number of latches to satisfy a desired clock period. The cost of latches is computed as the sum of all bits in every latch. Therefore, inserting latches into buses with fewer bits is preferred. Thus we can define the pipelining problems as follows:

given a micro-architectural design that includes control logic and a state register and desired clock period, insert minimal number of latch bits in such a way that any delay from a storage element to another storage element is smaller than the desired clock period and that number of latches is minimal.

2.2. Algorithm

We use a heuristic method to solve latch insertion problem. Linear programming method can be used for this problem also [NoCa90]. It only inserts latches between units. Our heuristic approach inserts latches between unit in the control and data path

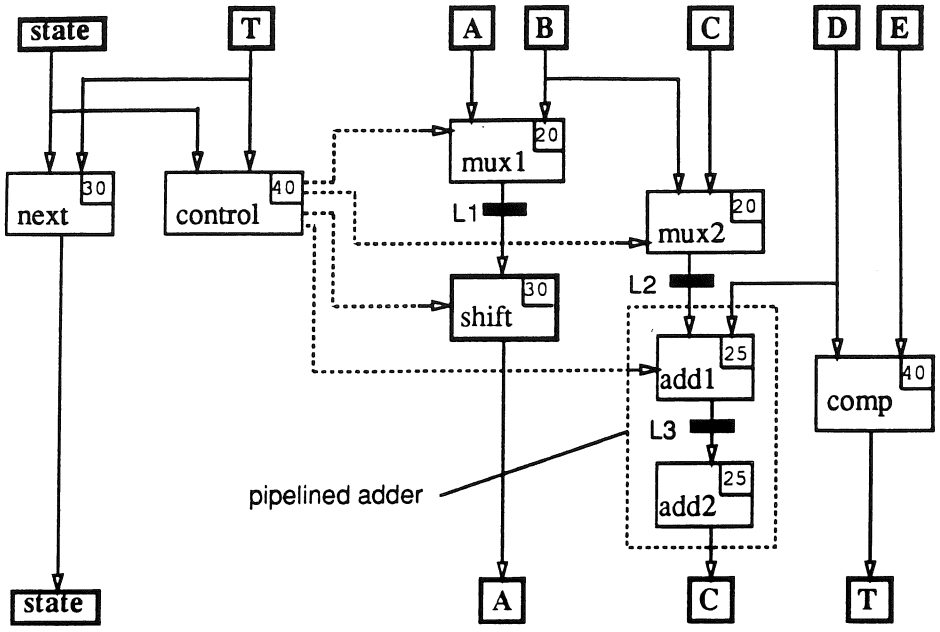


Figure 2-5. Circuit example with 2 stage pipeline adder (clock period = 70)

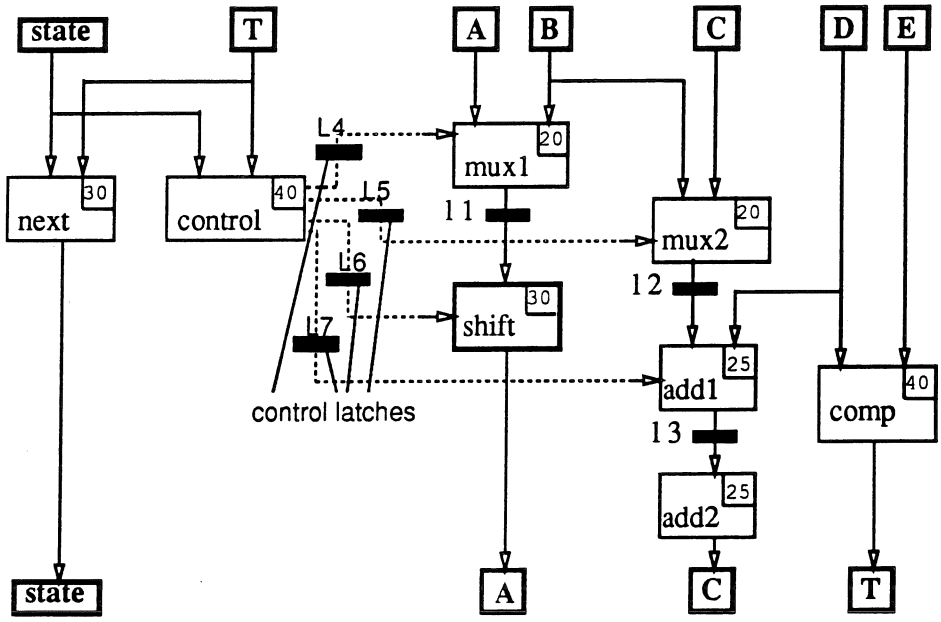


Figure 2-6. Circuit example with latches in control path (clock period = 40)

as well as inserts latches inside the functional units. Figure 2-7 shows a flow chart of latch insertion algorithm.

(i) If delay of a functional unit is greater than a given clock period, it is replaced with multi-stage units in which each stage delay is less or equal to a given clock period. We select the unit with minimal number of stages among all the units performing above condition. For example, a functional unit G in Figure 2-8 is replaced with B for given clock period of 20 since unit B has less stages than C although, both B and C have stage delay, less than 20.

(ii) Minimum number of latches that satisfies given clock period is computed for each register-to-register path.

(iii) Pathes are sorted in descending order according to the number of latches. Procedure (iv) will be applied on path by path because minimizing latches on the longest path has priority over minimizing latches on other shorter paths. Operations of each original cycle are divided into stages the number of which is equal to the number of the longest path's stages. So, the procedure is applied on the longest path ar first.

(iv) Latch insertion is considered for every path. Candidate edges for latch insertion are determined as follows:

(a) insert latches from start to end as far apart as possible still satisfying given clock period. Call those ALAP latches

(b) insert latches from end to start as far apart as possible still satisfying given clock period. Call those ASAP latches

(c) candidate edges are between ASAP and ALAP latches

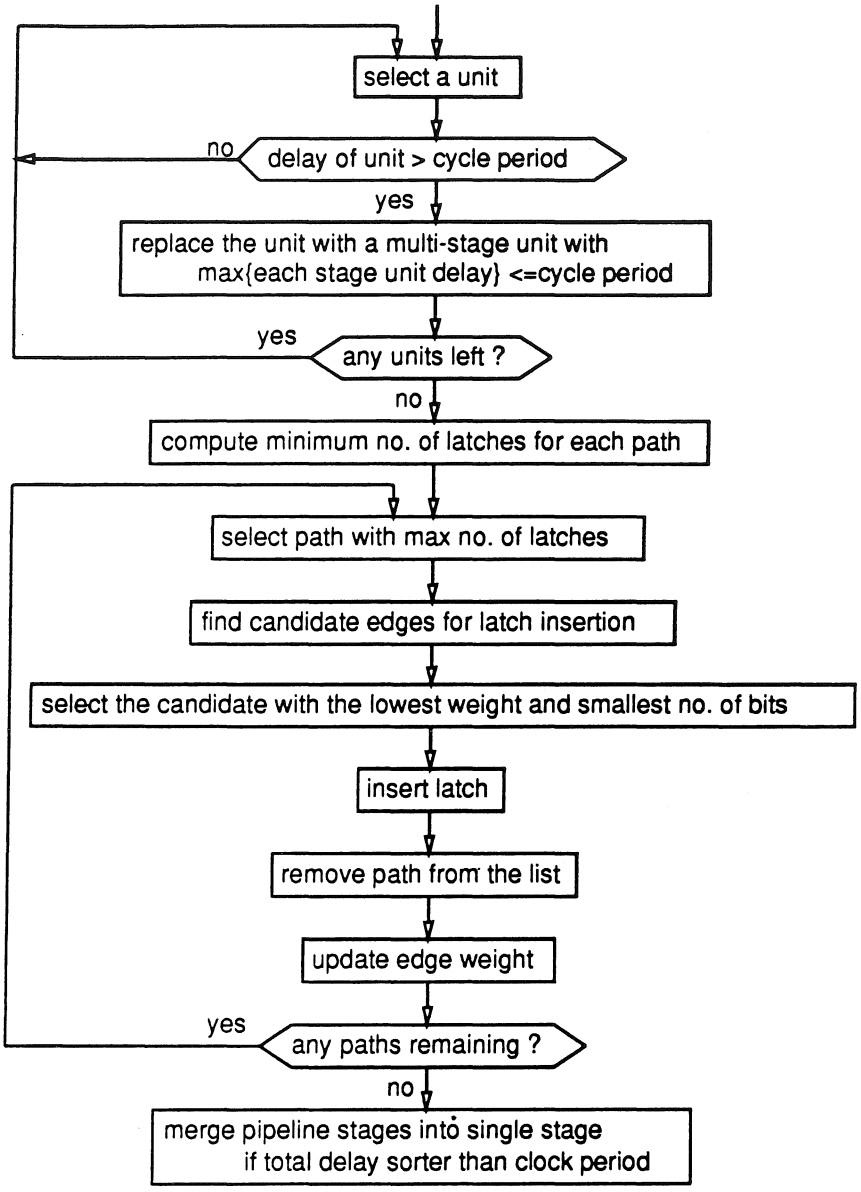


Figure 2-7. Latch insertion argorithm

	no. of stages	delay of each stage	pipeline delay
G	1	30	30
A	2	5, 25	25
B	2	15, 15	15
C	3	10, 10, 10	10

Figure 2-8. Multi stage unit example

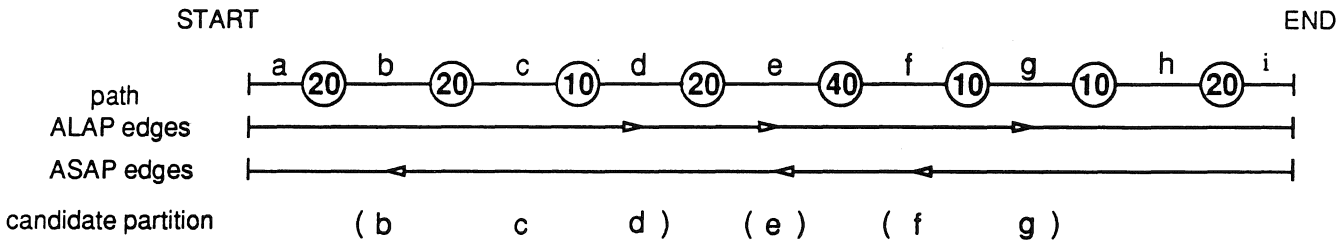


Figure 2-9. Latch insertion candidate edges for cycle period 50

Figure 2-9 shows a path with candidate edges for latch insertion for given clock period of 50. ALAP edges are d, e and g. ASAP edges are f, e and b. So, set of candidate edges for the first latch is { b, c, d }, for the second latch is { e } and { f, g } for the third one. When there are multiple candidates for latch insertion, the one with the smallest edge weight and area cost is selected.

Edge weight is computed as the max number of latches of any path passing through the edge. For example, edges of all paths passing through edge (E,H) in Figure 2-10-a have weight of 1. After insertion of L2 all edges on all paths that include L1 and L2 have weight of 2.

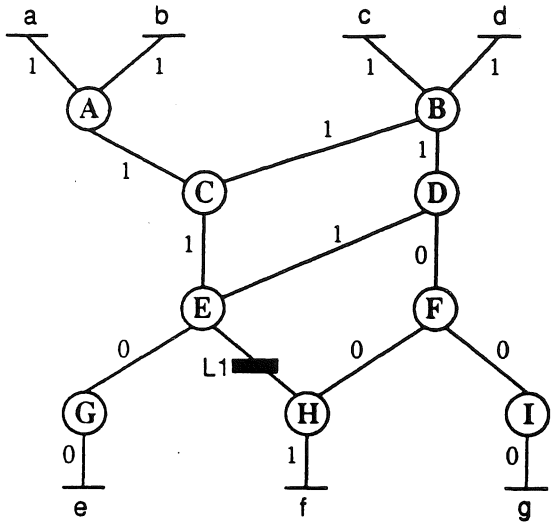


Figure 2-10-a. Edge weights after insertion of L1

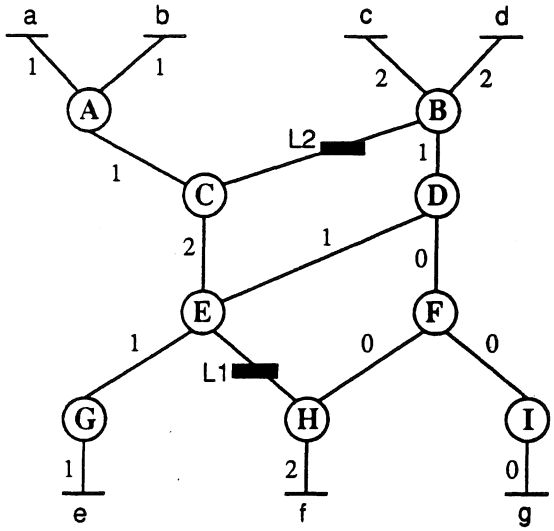


Figure 2-10-b. Edge weights after insertion of L2

3. Rescheduling

3.1. Introduction

When pipeline latches are inserted, the control sequence is changed due to different number of latches are inserted on different register-to-register paths. Thus a state transition table should be rewritten for this new micro architecture.

In order to explain the control sequence adjustment, we will use a simple example of a straight line code. Figure 3-1 shows a design with two latches inserted while Figure 3-2 shows its corresponding state transition table. The longest path in the original design was from the state register through control, mux and adder/subtracter to registers a and b. The clock period is computed to be 20 + 30 + 50 + setup time = 100 + setup time. When clock period is reduced to 50 ns, the L1 and L2 must be inserted.

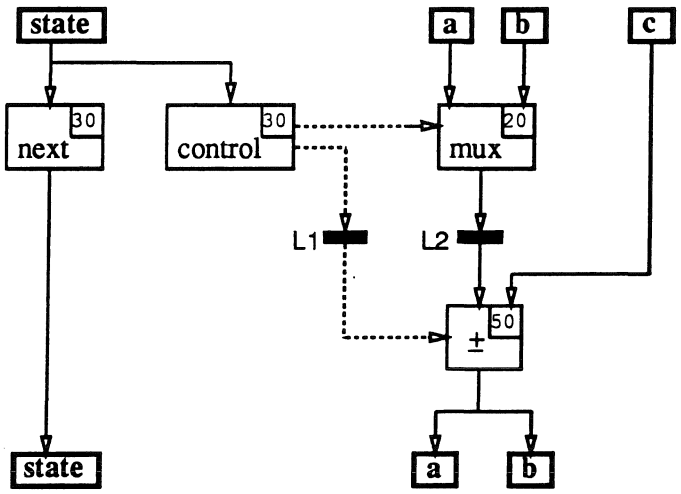


Figure 3-1. Example design for straight line code

To adjust the table for the pipelined design the state transition table in Figure 3-2 is transferred into a graph shown Figure 3-3. This graph, called APG(Active Path Graph), is a sequence of graphs for each cycle. It contains all active paths in a particular control cycle. Active paths include all control signals except register/latch load signals.

The control step graph starts and ends with registers. When latches are inserted, such a graph is split into smaller graphs at latch points. Figure 3-4 shows result of splitting of Figure 3-3 at latch points.

If new table is generated from split graph, it will generate proper results. It can be compacted however. Pipeline latches make possible executing independent operations in parallel. New transition table can be generated by compaction. Figure 3-5 shows the result of such compaction.

Originally, the clock period was 100 ns and transition table in Figure 3-2 was executed in 3 cycles. Thus the total execution time was 300 ns before insertion of pipelined latches. After insertion, the clock period was reduced to 50 ns and execution takes 4 cycles after compaction. Thus, total execution time is 200 ns, or 33% reduction of the original execution time.

3.2. Representation by APG

When the state transition table is transferred into APG, the design model shown in Figure 3-6 is used. This model can handle both a conditional actions and conditional branches to next state. When there is a conditional action, the path from outputs generating conditions in a data path to control logic is activated on APG. When there

state	operation	next
0	a = a + c	1
1	b = b - c	2
2	b = a + c	–

Figure 3-2. State transition table for design in Figure 3-1

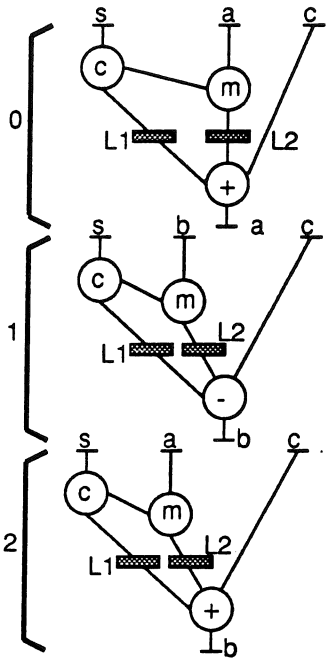


Figure 3-3. APG representation of Figure 3-1 and Figure 3-2

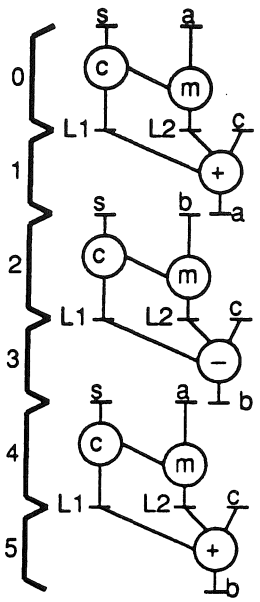


Figure 3-4. APG of Figure 3-3 after splitting

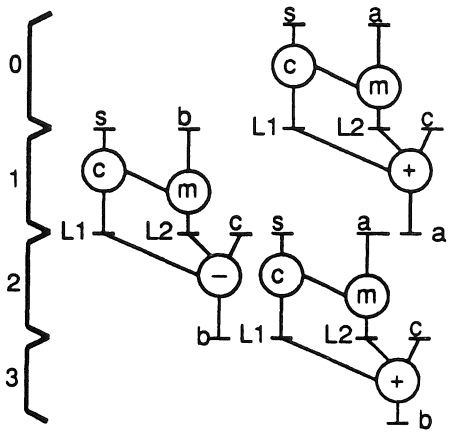


Figure 3-5. APG from Figure 3-4 after compacting

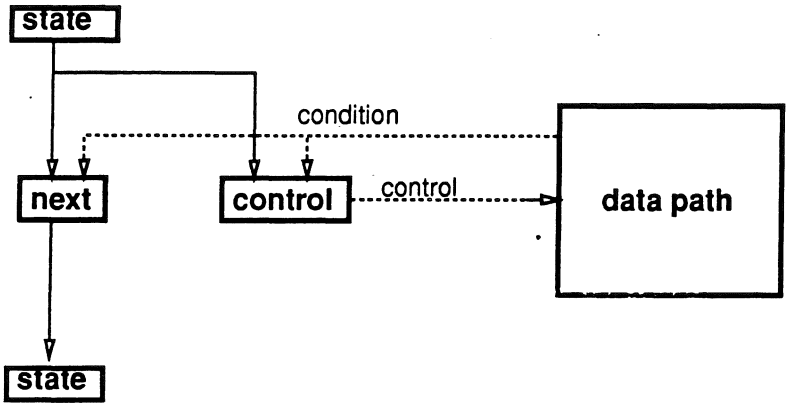


Figure 3-6. Design model for APG creation

is a conditional next state, the path from outputs of conditions to next logic is also activated. Active paths representing load signals of registers/latches are omitted because control signals directly connected with registers/latches are never latched. Control logic for load signals will be generated from the result of splitting and compaction.

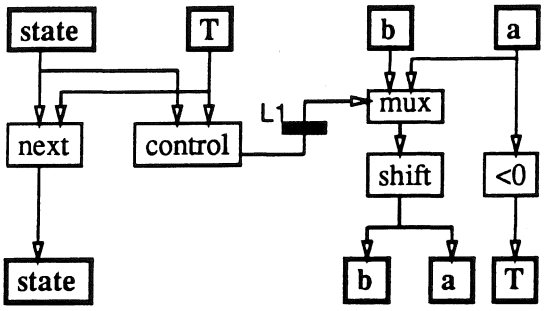


Figure 3-7. Loop example

state	condition	T/F	action	next
0			a <- shift(b)	1
1			T <- (a<0)	2
2	T=1	T	b <- shift(a)	3
		F	b <- shift(b)	4
3			a <- shift(b)	2
4			a <- shift(a)	—

Figure 3-8. State transition table for the loop example

Figure 3-7 shows a circuit and Figure 3-8 shows a state transition table for an example including loop operations. APG for the same example is shown in Figure 3-9.

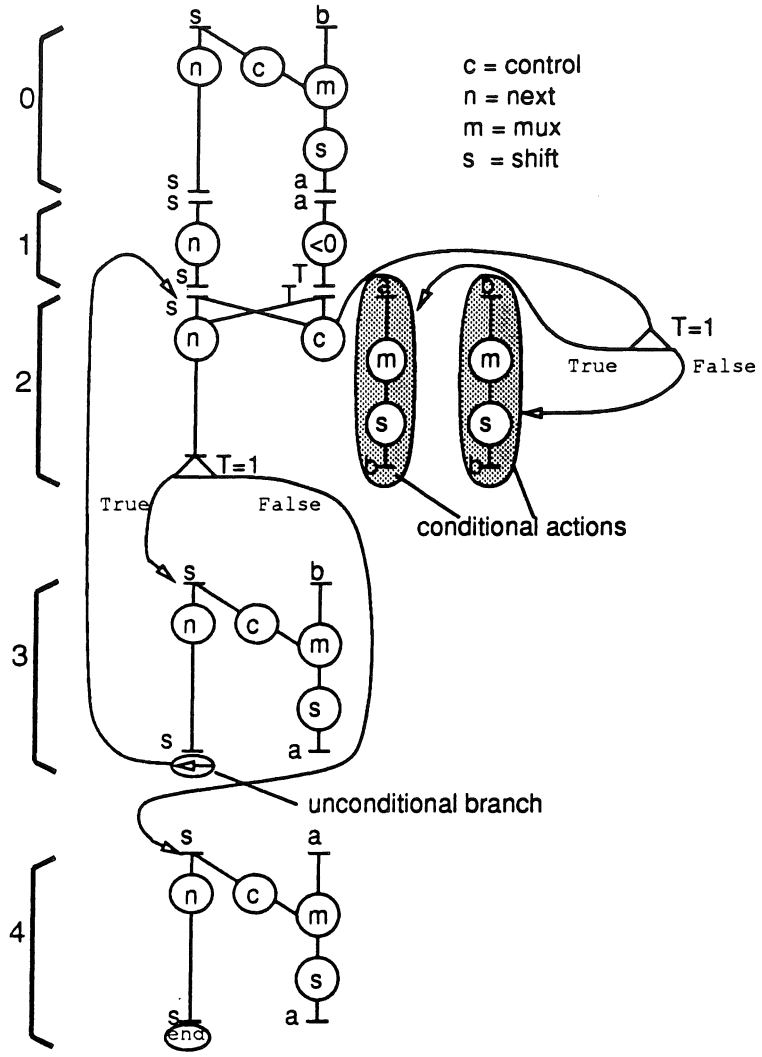


Figure 3-9. Original scheduling APG for loop example

3.3. Splitting and Compaction

3.3.1. Rescheduling flow

Figure 3-10 shows a flow chart of rescheduling algorithm.

(i) Conditional actions are converted into conditional next states and unconditional actions. A cycle including conditional actions are represented as mixed active paths of true paths, false paths and unconditional paths. Inserting latches into the active paths generates multi stages including conditional next states and unconditional actions. Sooner or later, conditional actions will be converted. Removing conditional actions before inserting latches is simpler than during inserting latches.

(ii) Latches are inserted as functional units into positions selected by a latch insertion tool.

(iii) Each cycle is split into stages at latch points. After splitting, latches become equivalent registers. Details of the splitting will be explained in the following section.

(iv) New latches make more operations executable concurrently. Thus, each independent operation can be moved over control step boundaries. If a null step appears after moving all the operations out of step, it is removed. This process of moving operations upward is called compaction. Details will be explained later.

(v) Conditional next states are reconverted into conditional actions to reduce the number of control steps if possible. For example, state transition table with no conditional actions, shown in Figure 3-11, could be converted into state transition table with conditional actions, shown in Figure 3-12.

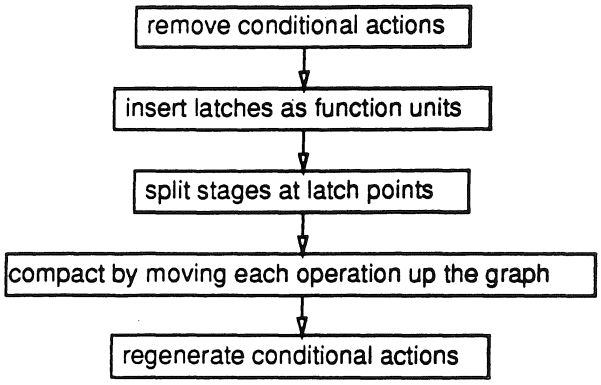


Figure 3-10. Rescheduling algorithm

	action			next		
state	condition	T/F	action	condition	T/F	next
j			a = a + b	T=1	T	k
					F	m
k			c = d - 1			k+1
m			e = f - 1			m+1

Figure 3-11. State transition table with no conditional actions

	action			next		
state	condition	T/F	action	condition	T/F	next
j	T=1	T	a = a + b c = d - 1	T=1	T	k+1
		F	a = a + b e = f - 1		F	m+1

Figure 3-12. State transition table with conditional actions

The APG of the loop example in Figure 3-9 is shown after procedure (**i**), (**iii**), (**iv**), (**v**) in Figure 3-13,Figure 3-14, Figure 3-15, Figure 3-16. State 3 in Figure 3-16 is an empty cycle which includes no data path operations but sets a control latch.

3.3.2. Splitting

A flow chart of splitting algorithm is shown in Figure 3-17. All graph tree routed in a destination register/latch in the same active path graph of a control step are independent. For example APG shown in Figure 3-18 has six independent operation trees as shown in Figure 3-19.

We define ready time of an operation in a graph to be the time when all source operands are ready. Time is an integer number beginning with 0. It is increased by 1 when a latch is reached by tracing from source registers to a destination register. Ready time basically shows the new stage number that an operation is to be rescheduled into. Figure 3-20 shows ready time of a single cycle APG example. Ready time is represented by integers 0,1 and 2 at inputs of each unit destinations.

We also define read time of a source register to keep a correct time sequence of reading and writing to a register/latch. A register in a single cycle APG might be both a source and a destination. Such register should not be rewritten until its contents is last used. Read time is the time when data is used at last. Read time of a source register/latch is obtained by backward tracing from a destination to a source on a tree. Figure 3-21 shows read times of the same example in Figure 3-2. Read time is represented by integer 0, 1 and 2 within squares at outputs of each source. If certain register is both a source and a destination in a cycle, its ready time should be rewritten

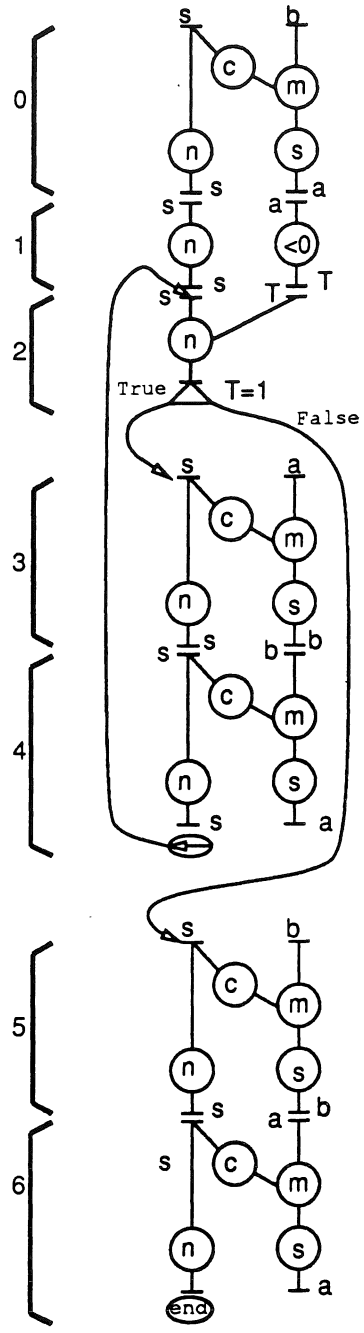


Figure 3-13. APG after removing conditional actions

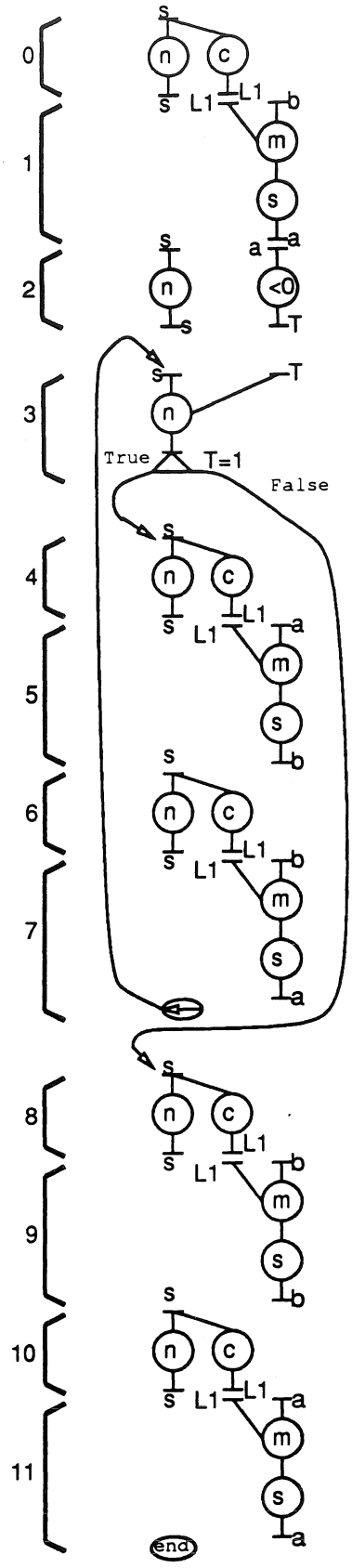


Figure 3-14. APG after splitting

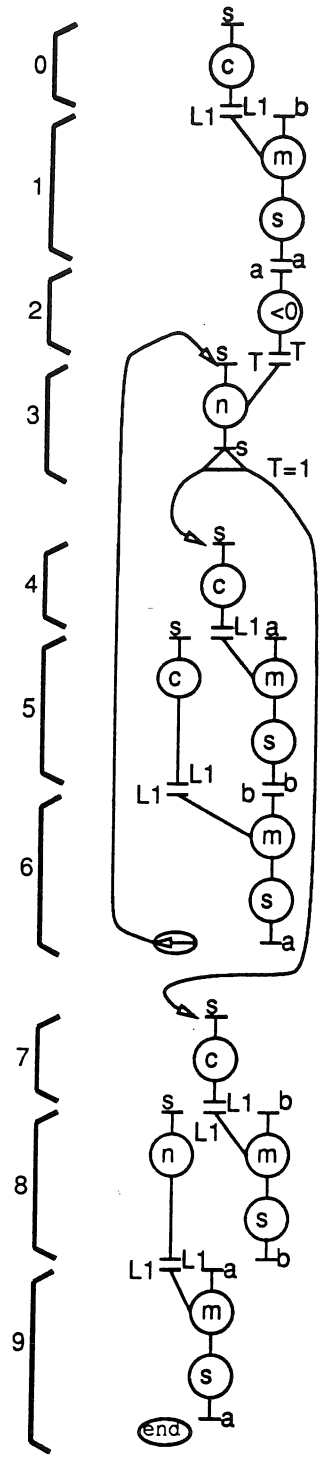


Figure 3-15. APG after compacting

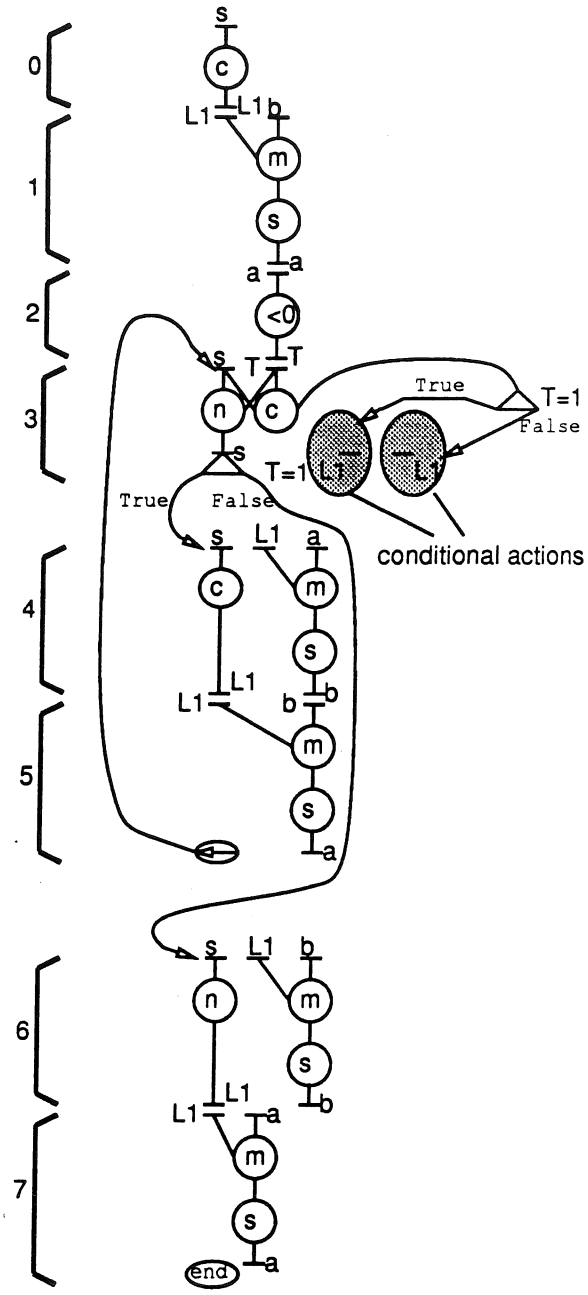


Figure 3-16. APG after regeneration of conditional actions

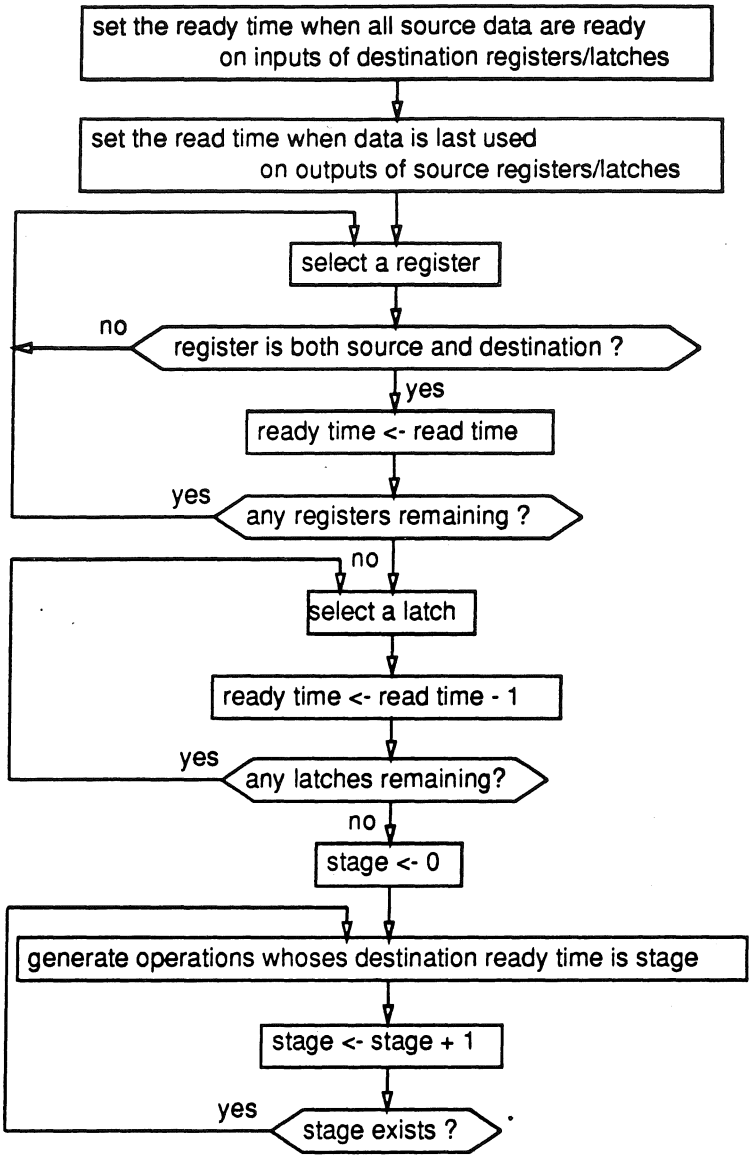


Figure 3-17. Splitting algorithm

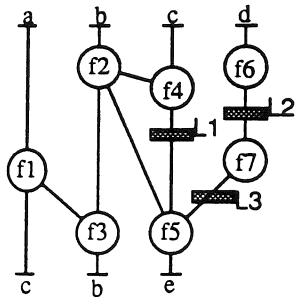
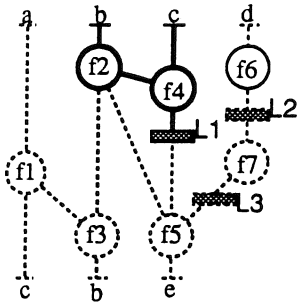
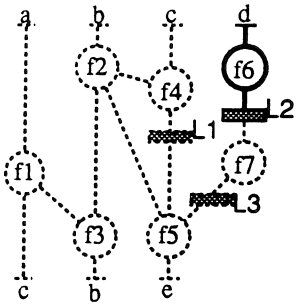


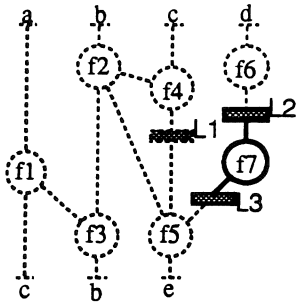
Figure 3-18. Single cycle APG



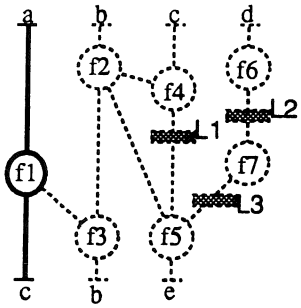
(1) Tree for L1



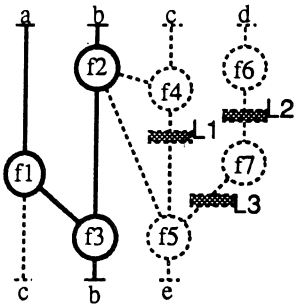
(2) Tree for L2



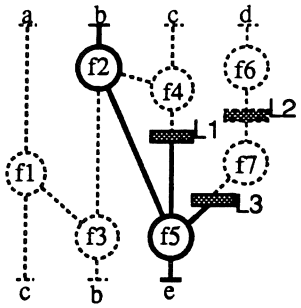
(3) Tree for L3



(4) Tree for c



(5) Tree for b



(6) Tree for e

Figure 3-19. Six graph trees

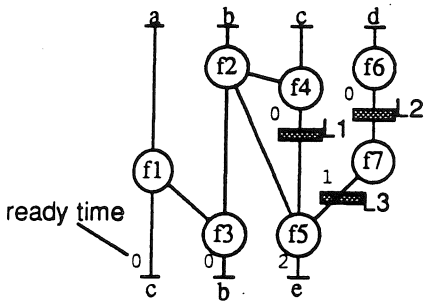


Figure 3-20. Ready time of a single cycle APG

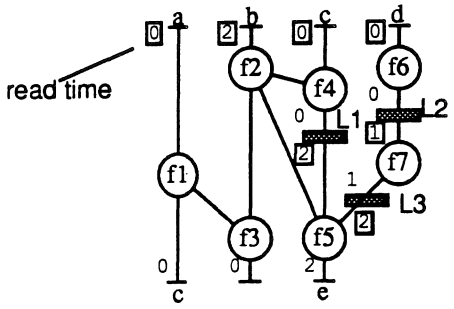


Figure 3-21. Read time in a single cycle APG

with the same time as its read time. For example source b of Figure 3-21 is used at time 2, i.e. read time is 2, and current ready time of destination b is 0. Therefore, ready time of b is to be adjusted to be 2.

In our approach latch data is to be read in a next of the cycle when data is set. So ready time of a latch is to be rewritten with read time - 1.

Figure 3-22 shows rewriting ready time of b and L1. Figure 3-23 shows a rewriting ready time of c caused by rewriting of Figure 3-22. Finally operations are sorted by ready time and rescheduled into each ready time stage. Figure 3-24 shows APG after splitting Figure 3-18.

3.3.3. Compaction

Figure 3-25 shows a flow chart of compaction algorithm. Compaction is performed within for each straight line code block. A cycle is selected one after another from second cycle to last one in each block. Each operation represented by a tree in a selected cycle is moved to the earlier cycle until data dependency is accountered. After an operation is moved to a cycle with dependency, it is moved to the next cycle until resource dependency does not exist any more.

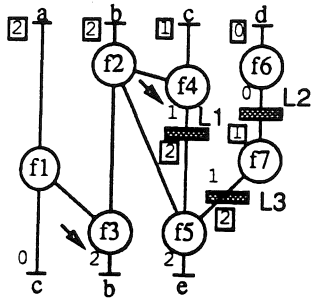


Figure 3-22. Ready and read time after adjusting b and L1

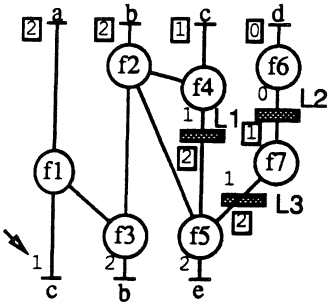


Figure 3-23. Ready and read time after adjusting c

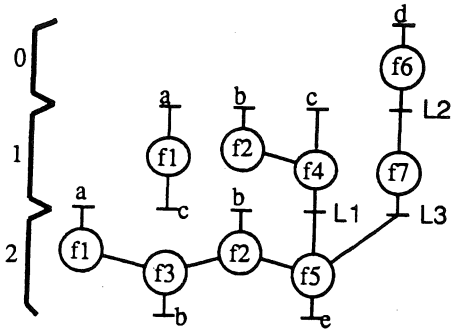


Figure 3-24. PAG after splitting

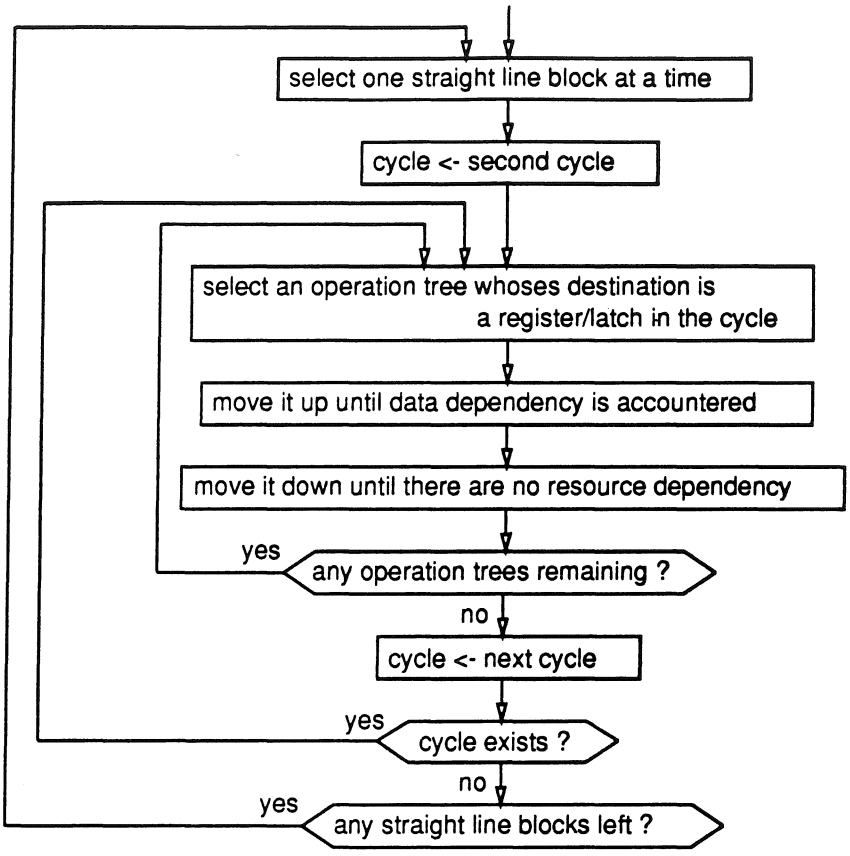


Figure 3-25. Compaction algorithm

4. Experiments and results

LIN and SPAC tools have been implemented on SUN4 running unix operating system 4.01. VHDL is used to describe micro-architecture in latch insertion tool LIN. BIF [DuHa89] is used to describe state transition table for the rescheduling program SPAC. A control logic is generated from a rescheduled state transition table. Control latches for pipeline operations are included not in data path but in control logic.

Several examples were used to test our tools.

Figure 4-1, Figure 4-2 and Figure 4-3 show shift-and-add multiplier [BrGa86] before, after pipelining without multi-stage units and after pipelining with multi-stage units. 2-stage adder was added in the third case as shown in Figure 4-3. Figure 4-4, Figure 4-5 and Figure 4-6 shows an original state transition table, state transition tables of pipelined design with no multi-stage units and with 2-stage adder respectively. The results is summarized in Figure 4-7. The shift-and-add example shows that pipelining without multi-stage units will improve total execution time by 70.7% while adding a 2-stage adder will reduce execution time to 59.1% of original time needed when shift-and-add multiplexer was not pipelined.

The "HAL" example [PaKG86] is shown in Figures 4-8 through 4-15. In case of HAL example pipelining with two 3-stage multiplexers decreased total execution time to 63.4% of original. The pipelining of HAL design without multi-stage multiplier increased total execution time since clock period was not drastically reduced and the number of states increased from 6 to 9. The clock period was not reduced because of multiplier delay. The number of states increased since not much overlap of states was possible with latches inserted in front of multiplexers.

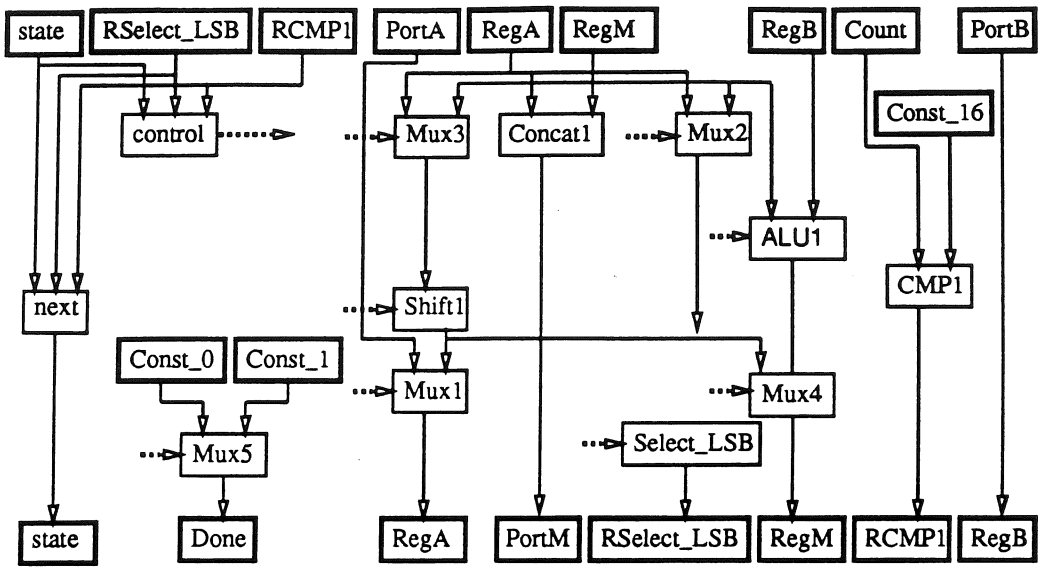


Figure 4-1. Shift-and-add design

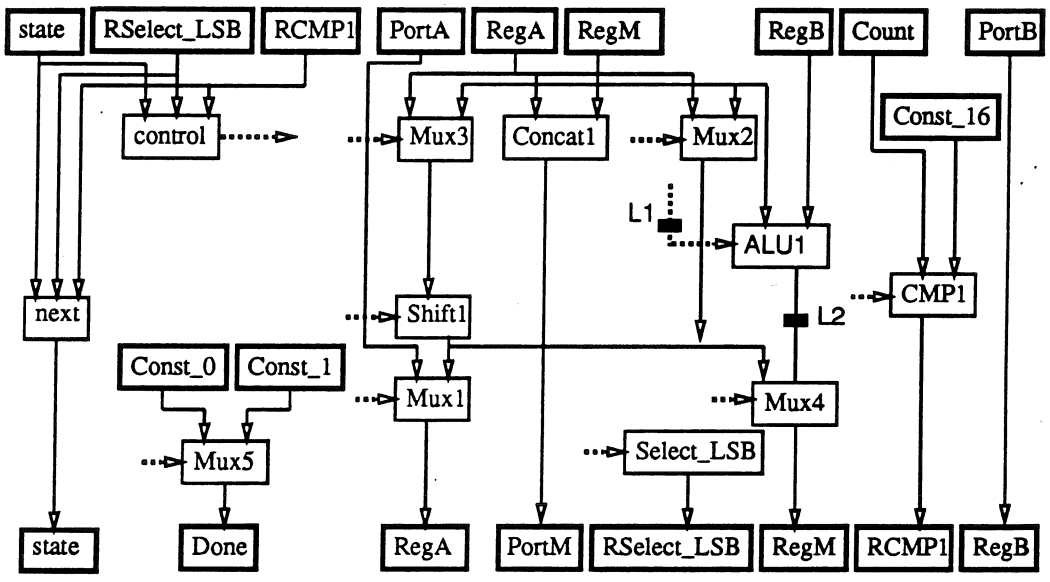


Figure 4-2. Shift-and-add design without pipelined adder

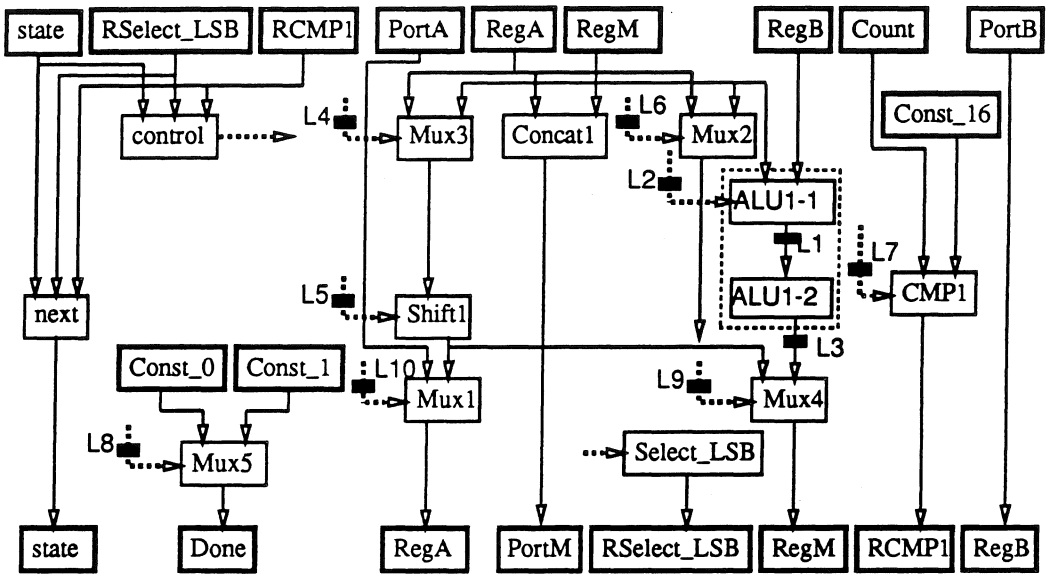


Figure 4-3. Shift-and-adder design with 2-stage adder

STATE	CONDITION T/F	ACTION	NEXT
0		Count(COUNTER; OPS: clear) Const_0(REG; OPS: clear)	1
1		Mux1(MUX2; OPS: C0; INPS: PortA.OO) RegA(REG; OPS: write; INPS: Mux1.OO) RegB(REG; OPS: write; INPS: PortB.OO) RegM(REG; OPS: clear) Const_1(REG; OPS: set) Mux5(MUX2; OPS: C0; INPS: Const_0.OQ) Done(OUTPORT; OPS: write; INPS: Mux5.OO) CMP1(COMPAR; OPS: lt; INPS: Count.OO, Const_16.OQ) RCMP1(REG; OPS: load; INPS: CMP1.OLT)	2
2	RCMP1.OQ == '1'		
	True	Mux2(MUX2; OPS: C0; INPS: RegA.OQ) Select_LSB(SELECT1; OPS: select; INPS: Mux2.OO) RSelect_LSB(REG; OPS: load; INPS: Select_LSB.OO)	3
	False	Concat1(CONCAT2; OPS: concat; INPS:RegM.OQ,RegA.OQ) PortM(OUTPORT; OPS: write; INPS: Concat1.OO) Mux5(MUX2; OPS: C1; INPS: Const_1.OQ) Done(OUTPORT; OPS: write; INPS: Mux5.OO)	8
3	RSelect_LSB.OQ == '1'		
	True	ALU1(ALU; OPS: add; INPS: RegM.OQ, RegB.OQ) Mux4(MUX2; OPS: C0; INPS: ALU1.OO) RegM(REG; OPS: write; INPS: Mux4.OO) Mux2(MUX2; OPS: C1; INPS: RegM.OQ) Select_LSB(SELECT1; OPS: select; INPS: Mux2.OO) RSelect_LSB(REG; OPS: load; INPS: Select_LSB.OO)	4
	False	Mux2(MUX2; OPS: C1; INPS: RegM.OQ) Select_LSB(SELECT1; OPS: select; INPS: Mux2.OO) RSelect_LSB(REG; OPS: load; INPS: Select_LSB.OO)	4
4	RSelect_LSB.OQ == '0'		
	True	Mux3(MUX2; OPS: C0; INPS: RegA.OQ) Shift1(SHIFTER; OPS: shr; INPS: Mux3.OO) Mux1(MUX2; OPS: C1; INPS: Shift1.OO) RegA(REG; OPS: write; INPS: Mux1.OO)	5
	False	Mux3(MUX2; OPS: C0; INPS: RegA.OQ) Shift1(SHIFTER; OPS: shl; INPS: Mux3.OO) Mux1(MUX2; OPS: C1; INPS: Shift1.OO) RegA(REG; OPS: write; INPS: Mux1.OO)	5
5		Mux3(MUX2; OPS: C1; INPS: RegM.OQ) Shift1(SHIFTER; OPS: shr; INPS: Mux3.OO) Mux4(MUX2; OPS: C1; INPS: Shift1.OO) RegM(REG; OPS: write; INPS: Mux4.OO)	6
6		Count(COUNTER; OPS: inc)	7
7		CMP1(COMPAR; OPS: lt; INPS: Count.OO, Const_16.OQ) RCMP1(REG; OPS: load; INPS: CMP1.OLT)	2
8		empty	8

Figure 4-4. Original state transition tablefor shift-and-add design

STATE	CONDITION T/F	ACTION	NEXT
0		Mux1(MUX2; OPS: C0; INPS: PortA.O0) Count(COUNTER; OPS: clear) Const_0(REG; OPS: clear) RegB(REG; OPS: write; INPS: PortB.O0) RegM(REG; OPS: clear) Const_1(REG; OPS: set) RegA(REG; OPS: write; INPS: Mux1.O0)	1
1		Mux5(MUX2; OPS: C0; INPS: Const_0.OQ) CMP1(COMPAR; OPS: lt; INPS: Count.O0, Const_16.OQ) Done(OUTPORT; OPS: write; INPS: Mux5.O0) RCMP1(REG; OPS: load; INPS: CMP1.OLT)	2
2	RCMP1.OQ == '1'		
	True	Select_LSB(SELECT1; OPS: select; INPS: Mux2.O0) Mux2(MUX2; OPS: C0; INPS: RegA.OQ) RSelect_LSB(REG; OPS: load; INPS: Select_LSB.O0)	3
	False	Concat1(CONCAT2; OPS: concat; INPS: RegM.OQ,RegA.OQ) Mux5(MUX2; OPS: C1; INPS: Const_1.OQ) PortM(OUTPORT; OPS: write; INPS: Concat1.O0) Done(OUTPORT; OPS: write; INPS: Mux5.O0)	9
3	RSelect_LSB.OQ == '1'		
	True	Select_LSB(SELECT1; OPS: select; INPS: Mux2.O0) Mux2(MUX2; OPS: C1; INPS: RegM.OQ) RSelect_LSB(REG; OPS: load; INPS: Select_LSB.O0) ALU1(ALU; OPS: add [CL]; INPS: RegM.OQ, RegB.OQ)	4
	False	Select_LSB(SELECT1; OPS: select; INPS: Mux2.O0) Mux2(MUX2; OPS: C1; INPS: RegM.OQ) RSelect_LSB(REG; OPS: load; INPS: Select_LSB.O0)	6
4		L2(REG; OPS: load; INPS: ALU1.O0)	5
5		Mux4(MUX2; OPS: C0; INPS: L2.Q) RegM(REG; OPS: write; INPS: Mux4.O0)	6
6	RSelect_LSB.OQ == '0'		
	True	Mux1(MUX2; OPS: C1; INPS: Shift1.O0) Shift1(SHIFTER; OPS: shr; INPS: Mux3.O0) Mux3(MUX2; OPS: C0; INPS: RegA.OQ) RegA(REG; OPS: write; INPS: Mux1.O0)	7
	False	Mux1(MUX2; OPS: C1; INPS: Shift1.O0) Shift1(SHIFTER; OPS: shl; INPS: Mux3.O0) Mux3(MUX2; OPS: C0; INPS: RegA.OQ) RegA(REG; OPS: write; INPS: Mux1.O0)	7
7		Mux3(MUX2; OPS: C1; INPS: RegM.OQ) Mux4(MUX2; OPS: C1; INPS: Shift1.O0) Shift1(SHIFTER; OPS: shr; INPS: Mux3.O0) Count(COUNTER; OPS: inc) RegM(REG; OPS: write; INPS: Mux4.O0)	8
8		CMP1(COMPAR; OPS: lt; INPS: Count.O0, Const_16.OQ) RCMP1(REG; OPS: load; INPS: CMP1.OLT)	2
9		empty	9

Figure 4-5. State transition table for shift-and-adder without multi-stage units

STATE	CONDITION T/F	ACTION	NEXT
0		Count(COUNTER; OPS: clear) Const_0(REG; OPS: clear) RegB(REG; OPS: write; INPS: PortB.OO) RegM(REG; OPS: clear) Const_1(REG; OPS: set) CMP1(COMPAR; OPS: lt [CL];INPS:Count.OO,Const_16.OQ) Mux5(MUX2; OPS: C0 [CL]; INPS: Const_0.OQ) Mux1(MUX2; OPS: C0 [CL]; INPS: PortA.OO)	1
1		RegA(REG; OPS: write; INPS: Mux1.OO) Done(OUTPORT; OPS: write; INPS: Mux5.OO) RCMP1(REG; OPS: load; INPS: CMP1.OLT)	2
2	RCMP1.OQ == '1'		
	True	Mux2(MUX2; OPS: C0 [CL]; INPS: RegA.OQ)	3
	False	Concat1(CONCAT2; OPS: concat; INPS: RegM.OQ,RegA.OQ) PortM(OUTPORT; OPS: write; INPS: Concat1.OO) Mux5(MUX2; OPS: C1 [CL]; INPS: Const_1.OQ)	14
3		Select_LSB(SELECT1; OPS: select; INPS: Mux2.OO) RSelect_LSB(REG; OPS: load; INPS: Select_LSB.OO)	4
4	RSelect_LSB.OQ == '1'		
	True	ALU1(ALU; OPS: add [CL, CP 2]; INPS:RegM.OQ,RegB.OQ) Mux2(MUX2; OPS: C1 [CL]; INPS: RegM.OQ)	5
	False	Mux2(MUX2; OPS: C1 [CL]; INPS: RegM.OQ)	8
5		Select_LSB(SELECT1; OPS: select; INPS: Mux2.OO) RSelect_LSB(REG; OPS: load; INPS: Select_LSB.OO)	6
6		L3(REG; OPS: load; INPS: ALU1.OO) Mux4(MUX2; OPS: C0 [CL]; INPS: L3.Q)	7
7		RegM(REG; OPS: write; INPS: Mux4.OO)	8
8		Select_LSB(SELECT1; OPS: select; INPS: Mux2.OO) RSelect_LSB(REG; OPS: load; INPS: Select_LSB.OO)	9
9	RSelect_LSB.OQ == '0'		
	True	Mux3(MUX2; OPS: C0 [CL]; INPS: RegA.OQ) Shift1(SHIFTER; OPS: shr [CL]; INPS: Mux3.OO) Mux1(MUX2; OPS: C1 [CL]; INPS: Shift1.OO)	10
	False	Mux3(MUX2; OPS: C0 [CL]; INPS: RegA.OQ) Shift1(SHIFTER; OPS: shl [CL]; INPS: Mux3.OO) Mux1(MUX2; OPS: C1 [CL]; INPS: Shift1.OO)	11
10		RegA(REG; OPS: write; INPS: Mux1.OO)	12
11		RegA(REG; OPS: write; INPS: Mux1.OO)	12
12		Count(COUNTER; OPS: inc) Mux3(MUX2; OPS: C1 [CL]; INPS: RegM.OQ) Shift1(SHIFTER; OPS: shr [CL]; INPS: Mux3.OO) Mux4(MUX2; OPS: C1 [CL]; INPS: Shift1.OO) CMP1(COMPAR; OPS: lt [CL];INPS:Count.OO,Const_16.OQ)	13
13		RegM(REG; OPS: write; INPS: Mux4.OO), RCMP1(REG; OPS: load; INPS: CMP1.OLT)	2
14		Done(OUTPORT; OPS: write; INPS: Mux5.OO)	15
15		empty	15

Figure 4-6. State transition table for shift-and-adder with 2 stage adder

shift-and-adder design	no. of latches	clock period(nsec)	no. of states	clock * states	%
without pipelining	0	316	9	2844	100.0
without multi-stage units	2	201	10	2010	70.7
with multi-stage units	10	105	16	1680	59.1

Figure 4-7. Comparison of pipelining shift-and-add design

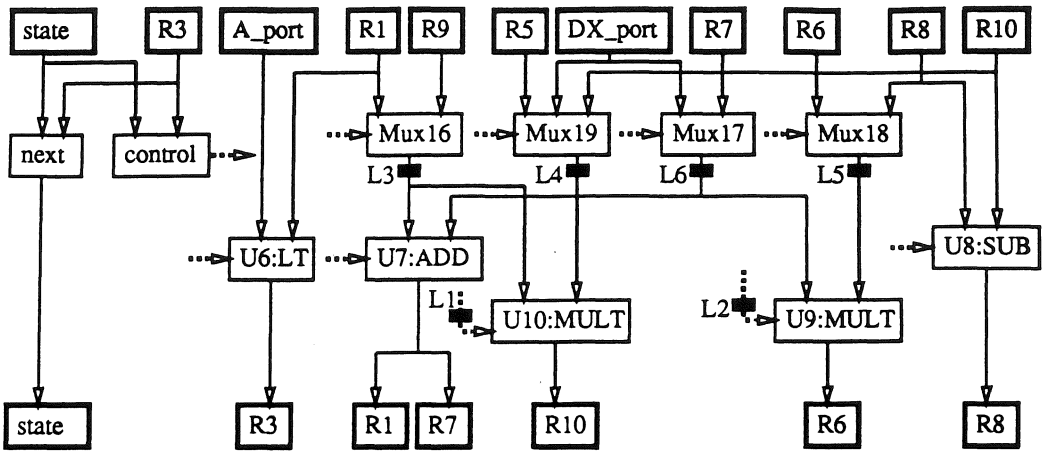


Figure 4-8. Pipelined Hal example without multi-stage units

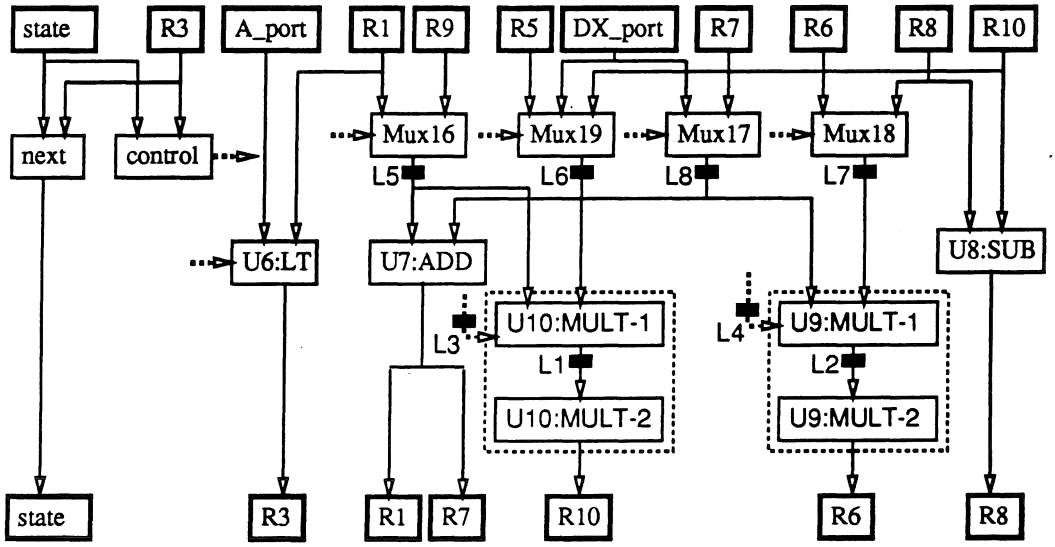


Figure 4-9. Pipelined Hal example with a 2-stage multiplier

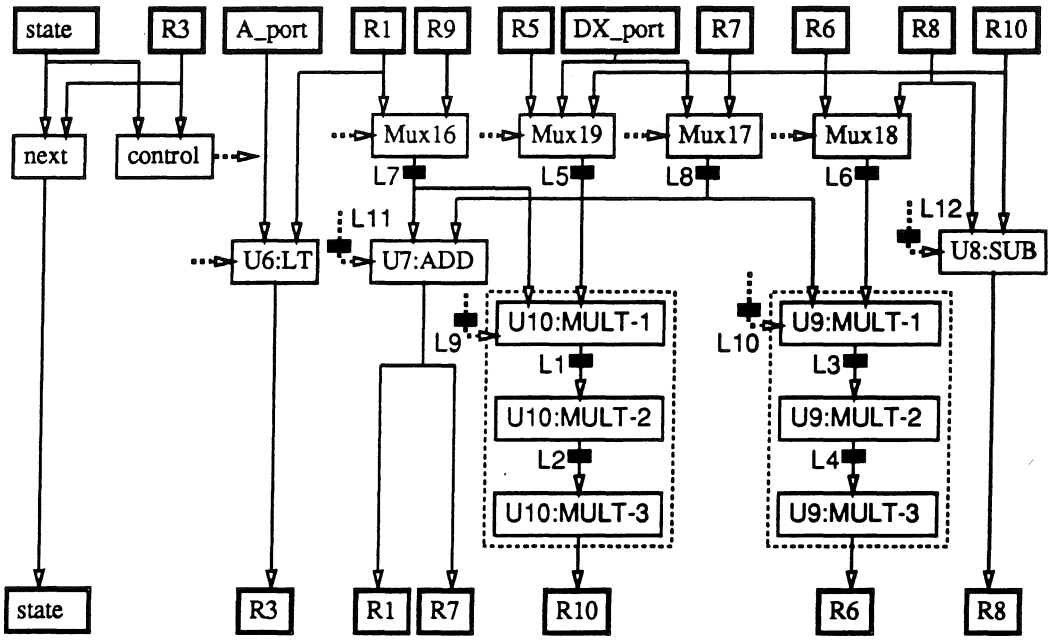


Figure 4-10. Pipelined Hal example with a 3-stage multiplier

STATE	CONDITION T/F	ACTION	NEXT
0		R3(REGI; OPS: WRITE; INPS: U6.O0) U6(ALU; OPS: LT; INPS: R1.O0, A_PORT.O0)	1
1	R3.O0 == '1'		
	True	R9(REGI; OPS: WRITE; INPS: U9.O0) U9(ALU; OPS: MULT; INPS: MUX18.O0, MUX17.O0) MUX18(MUX2;OPS:C1; INPS: R8.O0) MUX17(MUX2;OPS:C0; INPS: DX_PORT.O0) U10 (ALU; OPS: MULT; INPS: MUX16.O0, MUX19.O0) MUX19(MUX4;OPS:C1; INPS: R5.O0) MUX16(MUX2;OPS:C0; INPS: R1.O0) R1(REGI; OPS: WRITE; INPS: U7.O0) U7(ALU; OPS: ADD; INPS: MUX17.O0, MUX16.O0) R10(REGI; OPS: WRITE; INPS: U10.O0)	2
	False	empty	5
2		R10(REGI; OPS: WRITE; INPS: U10.O0) U10(ALU; OPS: MULT; INPS: MUX16.O0, MUX19.O0) MUX16(MUX2;OPS:C1; INPS: R9.O0) MUX19(MUX4;OPS:C0; INPS: R10.O0) R9(REGI; OPS: WRITE; INPS: U9.O0) U9(ALU; OPS: MULT; INPS: MUX18.O0, MUX17.O0) MUX18(MUX2;OPS:C0; INPS: R6.O0) MUX17(MUX2;OPS:C1; INPS: R7.O0) U6(ALU; OPS: LT; INPS: R1.O0, A_PORT.O0) R3(REGI; OPS: WRITE; INPS: U6.O0)	3
3		R8(REGI; OPS: WRITE; INPS: U8.O0) U10(ALU; OPS: MULT; INPS: MUX16.O0, MUX19.O0) MUX19(MUX4;OPS:C2; INPS: DX_PORT.O0) MUX16(MUX2;OPS:C1; INPS: R9.O0) U8(ALU; OPS: SUB; INPS: R8.O0, R10.O0) R9(REGI; OPS: WRITE; INPS: U9.O0) U9 (ALU; OPS: MULT; INPS: MUX18.O0, MUX17.O0) MUX18(MUX2;OPS:C1; INPS: R8.O0) MUX17(MUX2;OPS:C0; INPS: DX_PORT.O0) R10(REGI; OPS: WRITE; INPS: U10.O0)	4
4		R7(REGI; OPS: WRITE; INPS: U7.O0) U7(ALU; OPS: ADD; INPS: MUX17.O0, MUX16.O0) MUX17(MUX2;OPS:C1; INPS: R7.O0) MUX16(MUX2;OPS:C1; INPS: R9.O0) R8(REGI; OPS: WRITE; INPS: U8.O0) U8(ALU; OPS: SUB; INPS: R8.O0, R10.O0)	1
5		empty	5

Figure 4-11. Original state transition table for HAL example

STATE	CONDITION T/F	ACTION	NEXT
0		U6(ALU; OPS: LT; INPS: R1.O0, A_PORT.O0) R3(REGI; OPS: WRITE; INPS: U6.O0)	1
1	R3.O0 == '1'		
	True	MUX16(MUX2; OPS: C0; INPS: R1.O0) MUX19(MUX4; OPS: C1; INPS: R5.O0) MUX18(MUX2; OPS: C1; INPS: R8.O0) MUX17(MUX2; OPS: C0; INPS: DX_PORT.O0) U10(ALU; OPS: MULT [CL]; INPS: L3.Q, L4.Q) U9(ALU; OPS: MULT [CL]; INPS: L5.Q, L6.Q) L3(REG; OPS: load; INPS: MUX16.O0) L4(REG; OPS: load; INPS: MUX19.O0) L5(REG; OPS: load; INPS: MUX18.O0) L6(REG; OPS: load; INPS: MUX17.O0)	2
	False	empty	8
2		U7(ALU; OPS: ADD; INPS: L6.Q, L3.Q) MUX18(MUX2; OPS: C0; INPS: R6.O0) MUX17(MUX2; OPS: C1; INPS: R7.O0) R9(REGI; OPS: WRITE; INPS: U9.O0) R1(REGI; OPS: WRITE; INPS: U7.O0) U9(ALU; OPS: MULT [CL]; INPS: L5.Q, L6.Q) R10(REGI; OPS: WRITE; INPS: U10.O0) L5(REG; OPS: load; INPS: MUX18.O0) L6(REG; OPS: load; INPS: MUX17.O0)	3
3		U6(ALU; OPS: LT; INPS: R1.O0, A_PORT.O0) MUX16(MUX2; OPS: C1; INPS: R9.O0) MUX19(MUX4; OPS: C0; INPS: R10.O0) R3(REGI; OPS: WRITE; INPS: U6.O0) U10(ALU; OPS: MULT [CL]; INPS: L3.Q, L4.Q) L3(REG; OPS: load; INPS: MUX16.O0) L4(REG; OPS: load; INPS: MUX19.O0) R9(REGI; OPS: WRITE; INPS: U9.O0)	4
4		MUX16(MUX2; OPS: C1; INPS: R9.O0) MUX19(MUX4; OPS: C2; INPS: DX_PORT.O0) MUX17(MUX2; OPS: C0; INPS: DX_PORT.O0) MUX18(MUX2; OPS: C1; INPS: R8.O0) R10(REGI; OPS: WRITE; INPS: U10.O0) U10(ALU; OPS: MULT [CL]; INPS: L3.Q, L4.Q) U9(ALU; OPS: MULT [CL]; INPS: L5.Q, L6.Q) L3(REG; OPS: load; INPS: MUX16.O0) L4(REG; OPS: load; INPS: MUX19.O0) L5(REG; OPS: load; INPS: MUX18.O0) L6(REG; OPS: load; INPS: MUX17.O0)	5
5		U8(ALU; OPS: SUB; INPS: R8.O0, R10.O0) R8(REGI; OPS: WRITE; INPS: U8.O0) R10(REGI; OPS: WRITE; INPS: U10.O0) R9(REGI; OPS: WRITE; INPS: U9.O0)	6
6		U8(ALU; OPS: SUB; INPS: R8.O0, R10.O0) MUX16(MUX2; OPS: C1; INPS: R9.O0) MUX17(MUX2; OPS: C1; INPS: R7.O0) R8(REGI; OPS: WRITE; INPS: U8.O0) L3(REG; OPS: load; INPS: MUX16.O0)	7
7		U7(ALU; OPS: ADD; INPS: L6.Q, L3.Q) R7(REGI; OPS: WRITE; INPS: U7.O0)	1
8		empty	8

Figure 4-12. State transition table for pipelined HAL example with no multi stage unit

STATE	CONDITION T/F	ACTION	NEXT
0		U6(ALU; OPS: LT; INPS: R1.O0, A_PORT.O0) R3(REGI; OPS: WRITE; INPS: U6.O0)	1
1	R3.O0 == '1'		
	True	MUX16(MUX2; OPS: C0; INPS: R1.O0) MUX19(MUX4; OPS: C1; INPS: R5.O0) MUX17(MUX2; OPS: C0; INPS: DX_PORT.O0) U10(ALU; OPS: MULT [CL, CP 2]; INPS: L5.Q, L6.Q) U9(ALU; OPS: MULT [CL, CP 2]; INPS: L7.Q, L7.Q) L5(REG; OPS: load; INPS: MUX16.O0) L6(REG; OPS: load; INPS: MUX19.O0) L7(REG; OPS: load; INPS: MUX17.O0) L8(REG; OPS: load; INPS: MUX17.O0)	2
	False	empty	10
2		U7(ALU; OPS: ADD; INPS: L8.Q, L5.Q) MUX17(MUX2; OPS: C1; INPS: R7.O0) R1(REGI; OPS: WRITE; INPS: U7.O0) U9(ALU; OPS: MULT [CL, CP 2]; INPS: L7.Q, L7.Q) L7(REG; OPS: load; INPS: MUX17.O0)	3
3		U6(ALU; OPS: LT; INPS: R1.O0, A_PORT.O0) R9(REGI; OPS: WRITE; INPS: U9.O0) R10(REGI; OPS: WRITE; INPS: U10.O0) R3(REGI; OPS: WRITE; INPS: U6.O0)	4
4		MUX16(MUX2; OPS: C1; INPS: R9.O0) MUX19(MUX4; OPS: C0; INPS: R10.O0) U10(ALU; OPS: MULT [CL, CP 2]; INPS: L5.Q, L6.Q) L5(REG; OPS: load; INPS: MUX16.O0) L6(REG; OPS: load; INPS: MUX19.O0) R9(REGI; OPS: WRITE; INPS: U9.O0)	5
5		MUX16(MUX2; OPS: C1; INPS: R9.O0) MUX19(MUX4; OPS: C2; INPS: DX_PORT.O0) MUX17(MUX2; OPS: C0; INPS: DX_PORT.O0) U10(ALU; OPS: MULT [CL, CP 2]; INPS: L5.Q, L6.Q) U9(ALU; OPS: MULT [CL, CP 2]; INPS: L7.Q, L7.Q) L5(REG; OPS: load; INPS: MUX16.O0) L6(REG; OPS: load; INPS: MUX19.O0) L7(REG; OPS: load; INPS: MUX17.O0)	6
6		R10(REGI; OPS: WRITE; INPS: U10.O0)	7
7		U8(ALU; OPS: SUB; INPS: R8.O0, R10.O0) R8(REGI; OPS: WRITE; INPS: U8.O0) R10(REGI; OPS: WRITE; INPS: U10.O0) R9(REGI; OPS: WRITE; INPS: U9.O0)	8
8		U8(ALU; OPS: SUB; INPS: R8.O0, R10.O0) MUX16(MUX2; OPS: C1; INPS: R9.O0) MUX17(MUX2; OPS: C1; INPS: R7.O0) R8(REGI; OPS: WRITE; INPS: U8.O0) L5(REG; OPS: load; INPS: MUX16.O0) L8(REG; OPS: load; INPS: MUX17.O0)	9
9		U7(ALU; OPS: ADD; INPS: L8.Q, L5.Q) R7(REGI; OPS: WRITE; INPS: U7.O0)	1
10		empty	10

Figure 4-13. State transition table for pipelined HAL example with 2-stage multipliers

STATE	CONDITION T/F	ACTION	NEXT
0		U6(ALU; OPS: LT; INPS: R1.00, A_PORT.00) R3(REGI; OPS: WRITE; INPS: U6.00)	1
1	R3.00 = '1'		
	True	MUX19(MUX4; OPS: C1; INPS: R5.00) MUX18(MUX2; OPS: C1; INPS: R8.00) MUX16(MUX2; OPS: C0; INPS: R1.00) MUX17(MUX2; OPS: C0; INPS: DX_PORT.00) 15(REG; OPS: load; INPS: MUX19.00) 16(REG; OPS: load; INPS: MUX18.00) 17(REG; OPS: load; INPS: MUX16.00) 18(REG; OPS: load; INPS: MUX17.00) U10(ALU; OPS: MULT [CL, CP 3]; INPS: 17.Q, 15.Q) U9(ALU; OPS: MULT [CL, CP 3]; INPS: 18.Q, 16.Q) U7(ALU; OPS: ADD [CL]; INPS: 18.Q, 17.Q)	2
	False	empty	12
2		MUX18(MUX2; OPS: C0; INPS: R6.00) MUX17(MUX2; OPS: C1; INPS: R7.00) R1(REGI; OPS: WRITE; INPS: U7.00) 16(REG; OPS: load; INPS: MUX18.00) U9(ALU; OPS: MULT [CL, CP 3]; INPS: 16.Q, 18.Q) 18(REG; OPS: load; INPS: MUX17.00)	3
3		U6(ALU; OPS: LT; INPS: R1.00, A_PORT.00) R3(REGI; OPS: WRITE; INPS: U6.00)	4
4		R9(REGI; OPS: WRITE; INPS: U9.00) R10(REGI; OPS: WRITE; INPS: U10.00)	5
5		MUX19(MUX4; OPS: C0; INPS: R10.00) MUX16(MUX2; OPS: C1; INPS: R9.00) U10(ALU; OPS: MULT [CL, CP 3]; INPS: 17.Q, 15.Q) 15(REG; OPS: load; INPS: MUX19.00) 17(REG; OPS: load; INPS: MUX16.00) R9(REGI; OPS: WRITE; INPS: U9.00)	6
6		MUX18(MUX2; OPS: C1; INPS: R8.00) MUX17(MUX2; OPS: C0; INPS: DX_PORT.00) 16(REG; OPS: load; INPS: MUX18.00) U9(ALU; OPS: MULT [CL, CP 3]; INPS: 16.Q, 18.Q) 18(REG; OPS: load; INPS: MUX17.00)	7
7		MUX19(MUX4; OPS: C2; INPS: DX_PORT.00) MUX16(MUX2; OPS: C1; INPS: R9.00) 15(REG; OPS: load; INPS: MUX19.00) U10(ALU; OPS: MULT [CL, CP 3]; INPS: 17.Q, 15.Q) 17(REG; OPS: load; INPS: MUX16.00)	8
8		R10(REGI; OPS: WRITE; INPS: U10.00) U8(ALU; OPS: SUB [CL]; INPS: R8.00, R10.00)	9
9		R8(REGI; OPS: WRITE; INPS: U8.00) R9(REGI; OPS: WRITE; INPS: U9.00)	10
10		MUX16(MUX2; OPS: C1; INPS: R9.00) MUX17(MUX2; OPS: C1; INPS: R7.00) R10(REGI; OPS: WRITE; INPS: U10.00) 17(REG; OPS: load; INPS: MUX16.00) U7(ALU; OPS: ADD [CL]; INPS: 18.Q, 17.Q) U8(ALU; OPS: SUB [CL]; INPS: R8.00, R10.00) 18(REG; OPS: load; INPS: MUX17.00)	11
11		R7(REGI; OPS: WRITE; INPS: U7.00) R8(REGI; OPS: WRITE; INPS: U8.00)	1
12		empty	12

Figure 4-14. State transition table for pipelined HAL example with 3-stage multipliers

HAL example	no. of latches	clock period (nsec)	no. of states	clock * states	%
without pipelining	0	685	6	4110	100.0
pipeining without multi-stage units	6	600	9	5400	131.4
pipelining with 2-stage multiplier	8	300	11	3300	80.3
pipelining with 3 stage multiplier	12	200	13	2600	63.4

Figure 4-15. Performance comparison for HDL example

The "Elliptic" example [KuWK85] is shown in Figures 4-16 through 4-21. This experiment shows that insertion of pipelined operation units improves performance from 28% to 32%. Inserting latches between multiplexers and operation units may result in decrease of performance. This was the case for design with no pipelined adder and 2-stage multiplier.

There are special operations for pipeline represented by "CL" and "CP n" in an operation field of a state transition table. "CL" means the operation is control latched. "CL" operation in state N causes setting of a control latch in state N and execution with a function unit in state (N + 1). "CP n" means the operation is done with n stage pipeline unit. "CP n" operation in state N completes at the end of state (N + n - 1).

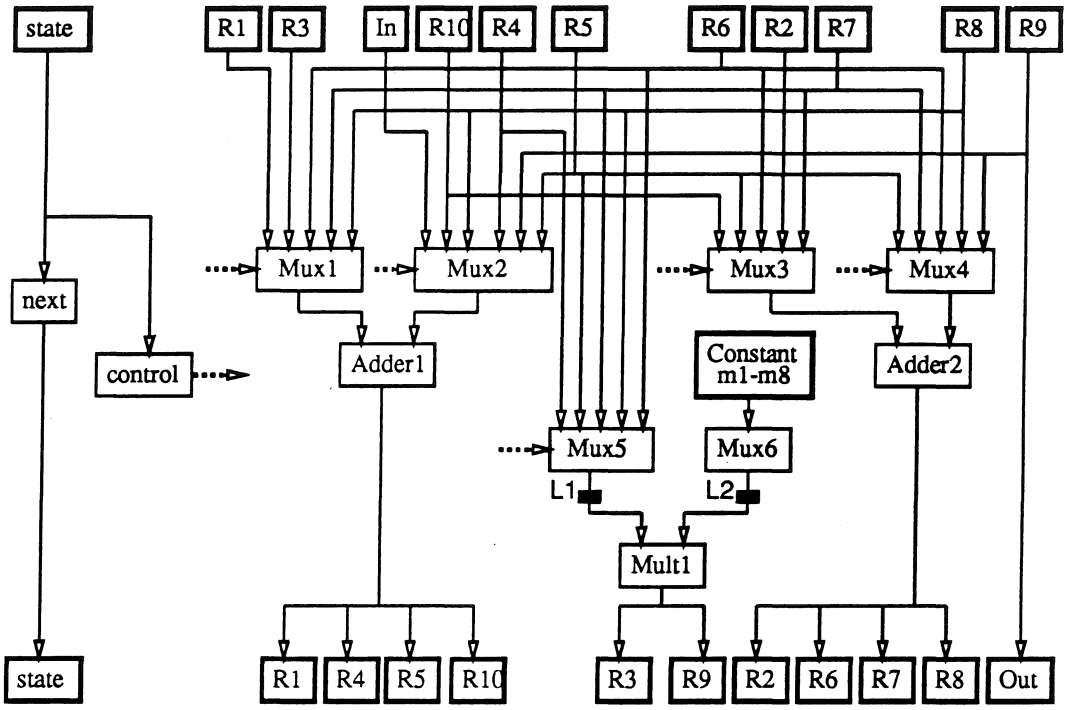


Figure 4-16. Pipelined Elliptic example without multi-stage units

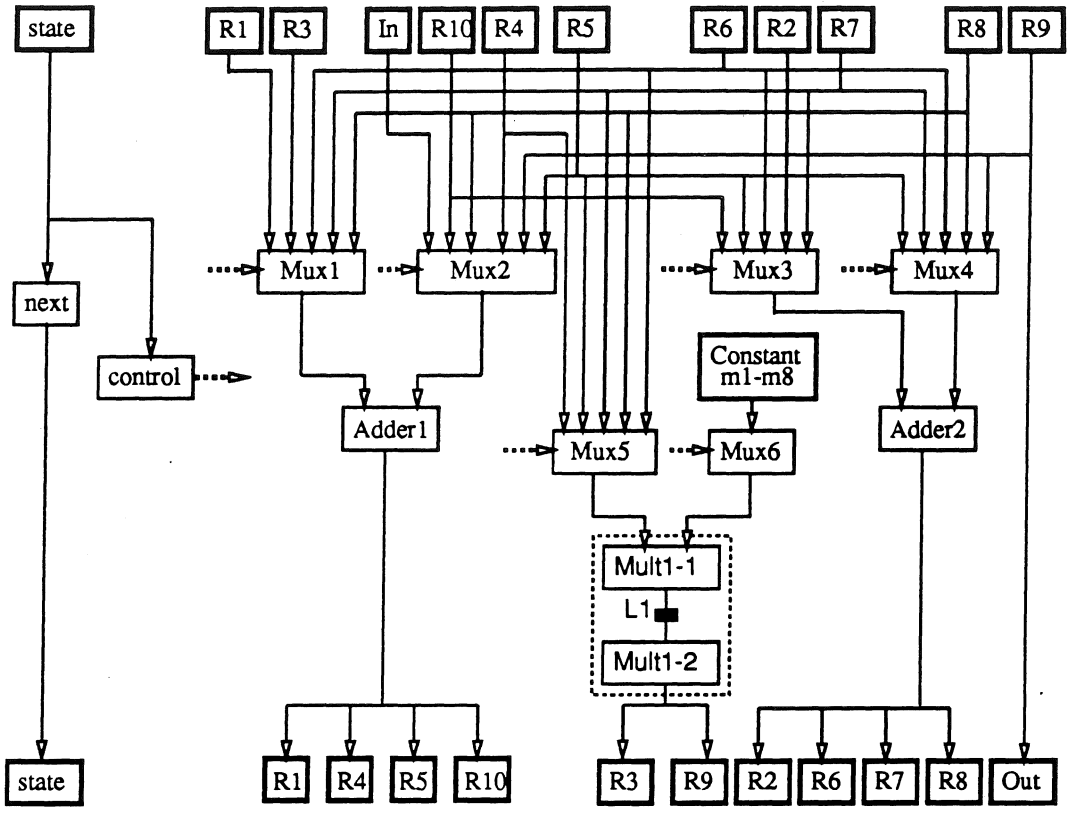


Figure 4-17. Pipelined Elliptic example with 2-stage multiplier with no mux output latch

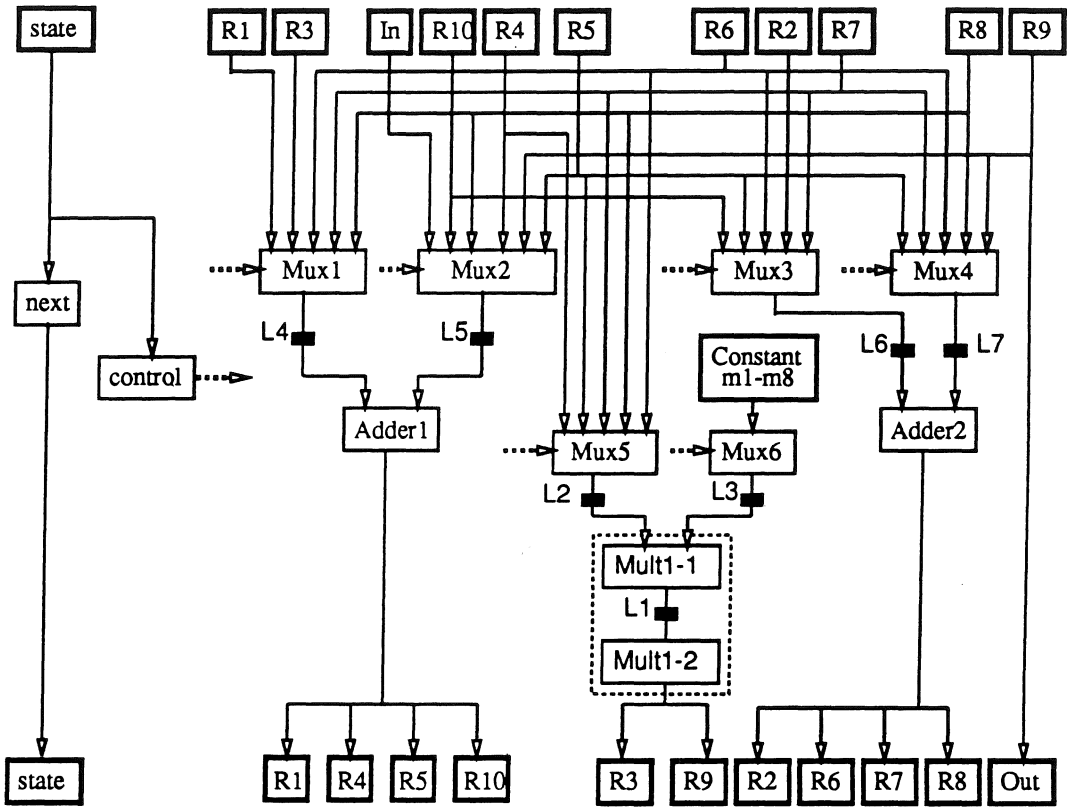


Figure 4-18. Pipelined Elliptic example with 2-stage multiplier and mux output latch

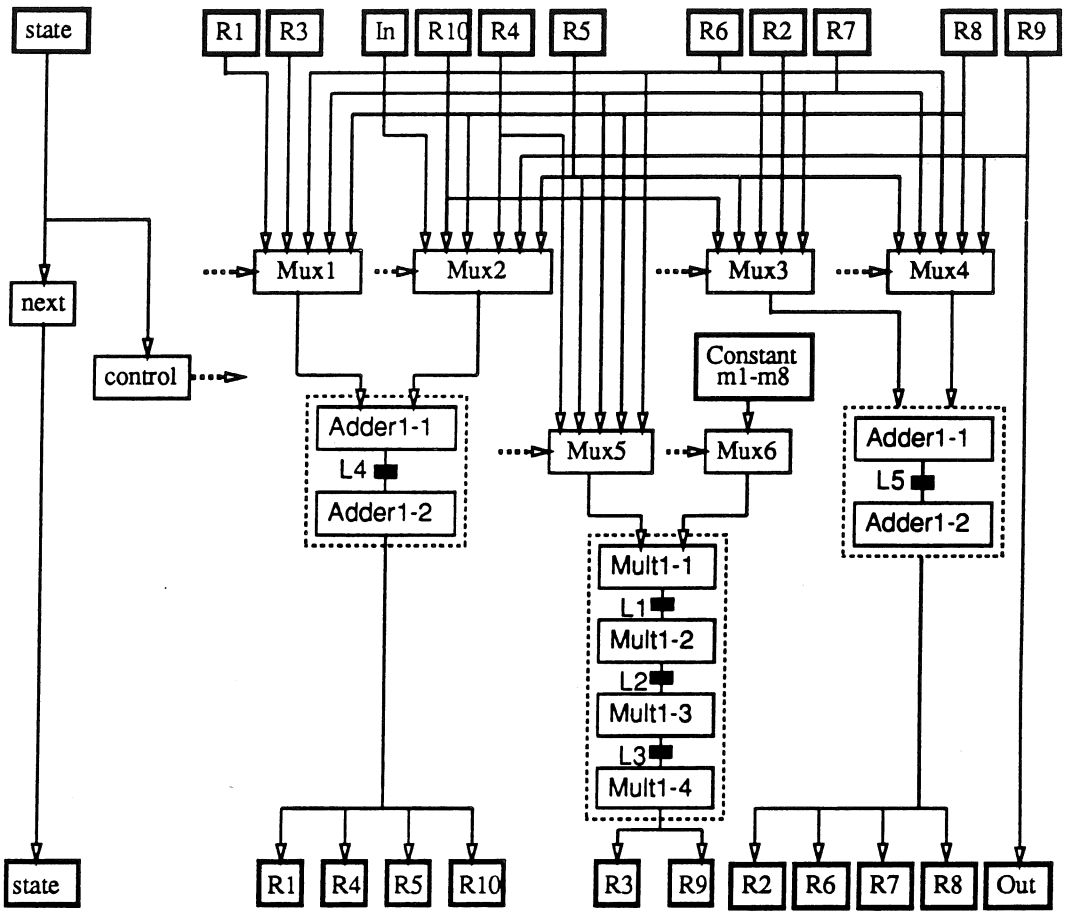


Figure 4-19. Pipelined Elliplitic example with 4-stage multiplier, 2-stage adder and no mux output latch

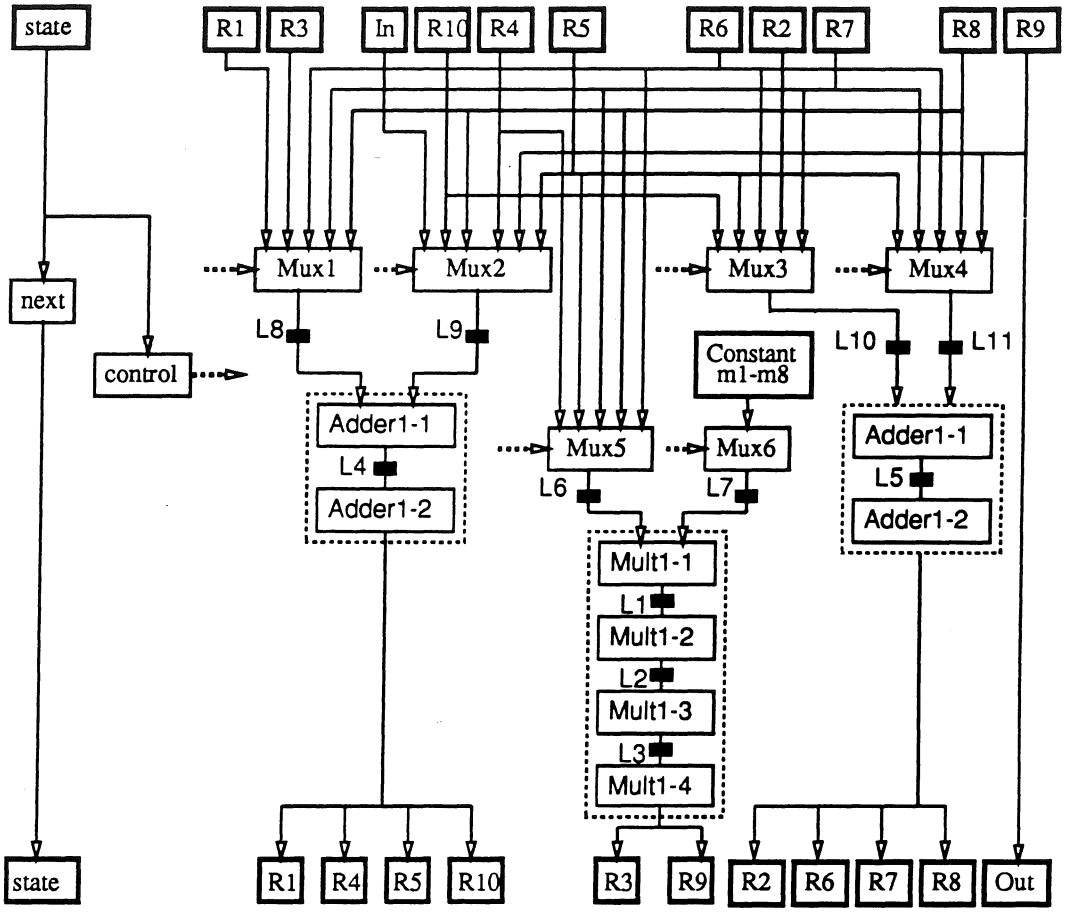


Figure 4-20. Pipelined Elliplitic example with 4-stage multiplier, 2-stage adder and mux output latch

Elliptic filter design	no. of latches	clock period (nsec)	no. of states	clock * states	%
without pipelining	0	495	19	9405	100.0
no multi stage unit, mux output latch	2	400	23	9200	97.8
2 stage multiplier, no mux output latch	1	295	23	6785	72.1
2 stage multiplier, mux output latch	7	201	40	8040	85.5
2 stage adder, 4 stage multiplier, no mux output latch	5	195	46	8970	95.4
2 stage adder, 4 stage multiplier, mux output latch	11	101	63	6363	67.7

Figure 4-21. Performance comparison of Elliptic example

5. Conclusion

A resynthesis method for pipelining register-to-register netlists has been proposed. Latch insertion and rescheduling tools have been developed and are available in UCI suite of synthesis tools. Experiments have shown that such pipelining can reduce total execution time by 29% - 41%.

Area cost increased by a latch insertion has not been studied. It is expected that inserted latches and routing will increase total chip area but not substantially. If dynamic charge latches are used instead of flip-flops, the area cost will be even smaller.

Routing for inserted latches might also change propagation delays and reduce the throughput gain. It also requires further study.

Current state transition table compaction is only applied to straight line code segments. If compaction is applied beyond branches, pipeline efficiency will be increased even further.

6. References

[BrGa86] F. D. Brewer, D. D. Gajski, "An Expert System Paradigm for Design", 23rd DAC, 1986.

[DuHa89] Nikil D. Dutt, Tedd Hadley, Daniel D. Gajski, "BIF: A Behavioral Intermediate Format For High Level Synthesis", Technical Report 89-03, Department of Information and Computer Science, University of California Irvine, January 1989.

[KuWK85] S. Y. Kung, H. J. Whitehouse and T. Kailath, "VLSI and Modern Single Processing", Englewood Cliffs, NJ: Prentice Hall. 1985, pp.258-264

[LiGa89] Joseph S. Lis, Daniel D. Gajski, "VHDL SYNTHESIS USING STRUCTURED MODELING", 26th DAC, 1989, pp.606-609

[MaSe91] Sharad Malik, Ellen M. Sentovich, Robert K. Brayton, Alberto Sagiovanni-Vincentelli, "Retiming and Resynthesis: Optimizing Sequential Networks with Combinational Techniques", IEEE Trans. Computer-Aided Design, vol. 10, No. 1, January 1991, pp.74-84.

[MaSi90] Sharad Malik, Kanwar Jit Singh, R. K. Brayton, Alberto Sagiovanni-Vincentelli, "Performance Optimization of Pipelined Circuits", ICCAD, 1990, pp.410-413.

[McCa90] Kristen N. McNall, Albert E. Casavant, "Automatic Operator Configuration in the Synthesis of Pipelined Architectures", 27th AC, 1990, pp.174-179.

[NoCa90] Stefaan Note, Francky Catthoor, Gert Groossens, Hugo De Man, " Combined Hardware Selection and Pipelinning in High Performance Data-Path", ICCD, 1990, pp.328-331.

[PaKG86] P. G. Paulin, J. P. Knight, E. F. Girczyc, "HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis", 23rd DAC, 1986.

[VaGa88] Nels Vander Zanden, Daniel Gajski, "MILO: A Microarchitecture and Logic Optimizer", 25th DAC, 1988, pp.403-408.