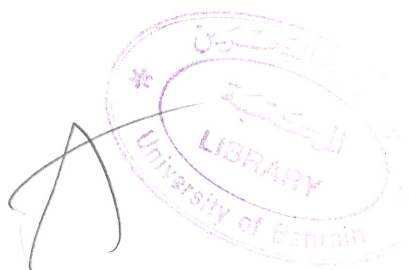


FLOOR PLAN DRIVEN DISTRIBUTED REDESIGN OF RTL CIRCUIT: A TRANSFORMATION APPROACH



by

Yusuf Aljazeeri



Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Thesis Advisor: Professor Christos A. Papachristou
Aljazeeri, Yusuf.

Floor plan driven distributed redesign
of RTL circuit : A transformation approach
TK7888.4.A1481998 c.2

Department of Computer Engineering and Science
CASE WESTERN RESERVE UNIVERSITY

January 1998

FLOOR PLAN DRIVEN DISTRIBUTED REDESIGN OF RTL CIRCUIT: A TRANSFORMATION APPROACH

Abstract

by

Yusuf Aljazeeri

In this dissertation we present redesign transformations that can be applied to high-level circuit descriptions. The redesign transformations transform the register transfer level (RTL) description of a finite state machine (FSM) description of a controller or a datapath description to another RTL datapath or FSM description. These transformations preserve the original circuit functionalities while altering the cost of the circuit area, delay, or average power consumption. The main objective of redesign is to achieve better circuit performance, smaller area, and/or less power consumption.

Two types of RTL transformations, FSM controller transformations and datapath transformations. will be presented.

The controller transformations are used to transform a centralized controller FSM into a distributed controller which has multiple FSMs. The concept of distributed controller is based on partitioning a datapath into functional blocks. The centralized controller is duplicated into multiple local controllers. Each local controller produces a subset of the original control signals in order to control one functional block. Each functional block is placed close to its local controller during the layout phase, which leads to shorter control wires. Activity is localized within the circuit as well, which reduces transistors switching and ultimately power consumption.

The centralized design and two types of distributed controller designs will be compared. In the first type of distributed controller design (NCLC), none of the local controllers communicate with each other, and in the second type (CLC), the local controllers do communicate. Experimental results show that circuit with distributed controllers have lower critical path delay and less power consumption than equivalent circuit with a centralized controller. Moreover, at sub-micron level where the gate delay is negligible and wire delay is dominant the distributed controller design has better performance than the centralized controller.

Datapath transformation will be presented also. An original design will be transformed into another design preserving the functionality of the circuit to let the designer explore different designs with different area, delay and power cost.

Copyright c (1998) by Yusuf Aljazeera

ACKNOWLEDGMENT

I am grateful to GOD, the merciful, the compassionate

Many thanks to my thesis advisor professor Christos A. Papachristou, whose support and guidance have been the most motivating factors for helping me in the completion of this work. Without his scholarly advice and unfailing help this work would not have been possible.

I would also like to extend my appreciation to professor Daniel Saab
I would also like to express my gratitude and special thanks to my family;
without their help and support I would never have been able to complete this
work.

Contents

1. Introduction.....	1
1.1 Background	1
1.2 Related Work	6
1.3 Motivation For Redesign	9
2. Datapath and Controller.....	11
2.1 Control Data Flow Graph (CDFG)	11
2.2 Datapath	17
2.2.1 Controller	18
3. Controller Transformations.....	30
3.1 Functional Block	30
3.1.1 Assignment Functional Block	32
3.2 Datapath Partitioning Algorithm	33
3.3 External Inputs	37
3.4 Distributed Controller	39
3.4.1 Non-Communicating Local Controller (NCLC)	40
3.4.2 Communicating Local Controller (CLC)	46
3.5 CLC Design Issues	50
3.5.1 Mealy vs. Moore Machines	50
3.5.2 <i>NS</i> Signal Code and Communication Bus width	52
3.5.3 Peer Controllers	57

4. Experimental Results	60
4.1 Distributed Controllers Generation and implementation	60
4.1.1 Software Structure	60
4.1.2 Floor planning:	63
4.1.3 Delay and Power	66
4.2 Cases Studies	72
4.2.1 Case study I	72
4.2.2 Case Study II	82
4.2.3 Case Study III	92
4.2.4 Experimental Results The Eight Behaviors	107
5. Datapath Transformations	117
5.1 Operation Attributes	118
5.2 ALU Reallocation Transformations (ART)	120
5.3 Register Reallocation Transformations (RRT)	127
5.4 ALU and Register Reallocation Transformation	134
5.5 Rescheduling Transformations (RT)	134
5.6 Experimental results	143
5.7 Rescheduling Transformations	148
5.7.1 Register Reallocation Transformation Solution	151

List of Figures

2.1	differential equation data flow graph (DFG).	12
2.2	Control data flow graph (CDFG) diagram.	13
2.3	Complex CDFG diagram.	14
2.4	If-else control data flow graph (CDFG).	16
2.5	MUX types.	19
2.6	a datapath for the if-else CDFG.	20
2.7	The Differential equation behavior FSM.	23
2.8	general controller diagram.	24
2.9	A controller FSM for the if-else CDFG in Figure fig7.	25
2.10	Scheduled CDFG for POST calculator behavior.	26
2.11	POST calculator basic blocks.	27
2.12	POST calculator datapath.	28
2.13	POST calculator controller FSM.	29
3.1	Functional blocks.	31
3.2	Assignment functional block types.	33
3.3	Functional block with no controller.	35
3.4	Datapath partitioning example.	36
3.5	External controller to load input latches.	38
3.6	If-else control data flow graph (CDFG).	42
3.7	datapath for If-else behavior.	43
3.8	Centralized Controller.	44

3.9	Non-Communicating Local Controllers (NCLC).	45
3.10	Communicating local controllers.	50
3.11	Moore and Mealy implementation with similar cost.	51
3.12	Moore and Mealy implementation with different cost.	52
3.13	Single bit wake up signal, case I.	53
3.14	Single bit wakeup signal, case II.	54
3.15	CLC design where NS signal requires more than one bit.	55
3.16	NS bus width.	56
3.17	Peer local controllers.	58
3.18	Peer local controller deadlock.	59
4.1	Distributed controller design and implementation.	62
4.2	circuit hierarchical layout procedure.	64
4.3	Two-level centralized controller design layout.	65
4.4	Three-level distributed controller design layout.	67
4.5	Wire delay is dominant at sub-micon level.	70
4.6	the control wire ratio relation $w_c/w_l > 1$ remain valid at sub-micron level.	71
4.7	DCT scheduled DFG.	73
4.8	DCT functional blocks.	74
4.9	DCT centralized controller.	75
4.10	DCT CLC master controller.	77
4.11	DCT CLC local controllers.	78
4.12	Scheduled CDFG for POST behavior.	84
4.13	postage basic blocks.	85
4.14	POST behavior datapath.	86

4.15	The FSM of the POST behavior controller.	87
4.16	POST CLC local controllers.	89
4.17	Scheduled CDFG for the FPADDS behavior.	94
4.18	PART I: Basic Blocks for the FPADDS behavior.	95
4.19	PART II: Basic Blocks for FPADDS behavior.	96
4.20	Datapath for the FPADDS behavior.	97
4.21	Functional Blocks for the Floating-Point Addition behavior.	99
4.22	Centralized controller for the FPADDS behavior.	100
4.23	The subtractor local controller (master) for the FPADDS behavior.	102
4.24	PART I:CLC distributed controllers for the Floating-Point Addition example.	103
4.25	PART II: CLC Distributed Controller for the Floating-Point Addition example.	104
5.1	An overview of both controller and datapath transformations.	119
5.2	Example for ALU reallocation transformations.	121
5.3	PART I: Effects of ART on a datapath.	122
5.4	PART II: Effects of ART on a datapath.	123
5.5	PART III: Effects of ART on a datapath.	126
5.6	The lifetimes of p and w overlap.	127
5.7	Output MUX elimination using RRT.	129
5.8	Example to show RTT effects on a datapath.	130
5.9	RRT move resulted with no extra connections.	131
5.10	RRT move resulted in saving hardware.	132
5.11	RRT move resulted in extra connection.	133

5.12	ART and RRT transformations.	135
5.13	An operation can be rescheduled within its slack time.	136
5.14	PART I: Effect of rescheduling transformation on the controller	140
5.15	PART II: Effect of rescheduling transformation on the controller	141
5.16	RT and ART transformations example.	142
5.17	Circuit after RT and ART transformations.	144
5.18	A CDFG for datapath transformation.	146
5.19	The original datapath for the datapath transformation.	147
5.20	controller for the previous CDFG.	148
5.21	the CDFG after rescheduling transformations	150
5.22	Datapath after rescheduling transformations.	150
5.23	Controller after rescheduling transformations.	151
5.24	Datapath after RRT transformations.	152
5.25	Controller after RRT transformations.	153
.1	Wire resistance and capacitance	163

Chapter 1

Introduction

1.1 Background

High-level synthesis is the process of converting an algorithmic behavioral description, which is usually represented in a control data flow graph (CDFG), to a Register Transfer Level (RTL) description [6] of the circuit. An RTL description consists of two parts, a datapath and a controller. A datapath is a set of connected digital components that execute the operations of the behavior. The controller is a Finite State Machine (FSM) description of the circuit activities for each cycle.

High level synthesis has two phases, scheduling and allocation. Both phases are highly interdependent. Techniques vary on which phase is performed first. Both scheduling and allocation are NP complete problems. Many heuristics have been developed, therefore, to arrive at good solutions for scheduling and allocation problems with different objectives.

The task of the scheduler is to assign each operation in a behavior to one or more time steps. This assignment must preserve data flow dependencies. An operation can be scheduled neither before its operands are available, nor after its first usage as an operand for other operations. Moreover, the scheduler must try to find an optimal schedule that meets time and resources constraints. The scheduler's choice between alternative optimal solutions is highly dependent on the heuristic algorithm used.

Many scheduling algorithms were developed. As Soon As Possible (ASAP) is a scheduling algorithm that tries to schedule each operation as early as possible, independent of the resource constraints. An operation is assigned to the time step immediately after all of its operands are available. The As Late As Possible (ALAP) algorithm schedules each operation as late as possible, independent of the resource constraints. An operation is assigned to the time step immediately before its first usage as an operand by other operations. Both ASAP and ALAP sort all operations in the data flow graph according to their data dependencies using depth-first search or depth-last search respectively. Since neither scheduling algorithm is concerned with resource constraints, they are not used as final scheduling techniques, but are used as a pre-scheduling step in some scheduling algorithms.

List scheduling [26] uses a local priority function to resolve resource conflicts. Some operations are deferred for later scheduling. Operations are sorted in topological order using control and data dependencies. Ready operations are scheduled first. If there are more ready operations than resources to execute them, some of these operations will be deferred to another time step. Which operations to defer depends on some local priority.

In force-directed scheduling [31], a global time constraint is specified, and the heuristic tries to minimize resources. It places similar operations in different time steps. A slack time for each operation is determined. A uniform probability is associated with placing an operation in any feasible time step. A probability distribution graph is created from the summation of the probabilities of each type of operation for each time step. A force is associated with each operation for each feasible time step. Each probability distribution will

have an associated force on all operations at a specific time step. An operation is selected to be assigned to a time step if it has the least force, which means less concurrency and fewer resources.

A scheduling algorithm can accommodate conditional branching and loops. The algorithm described in [18] performs scheduling and allocation for a behavior with conditions. It tries to exploit the potential of resource sharing between mutually exclusive operations. It also tries to reduce the critical path delay.

The allocation phase follows or preceeds the scheduling phase, or sometimes it is concurrent with the scheduling phase. Every operation in a data flow graph is allocated to a functional unit to execute it. Register allocation is done in this phase also. Variables are assigned registers where they will be stored. The allocator tries to minimize the amount of hardware needed for a given behavior. Functional units and registers are allocated such that every hardware unit is utilized as much as possible. Operations that need different functional units, operations that are in different branches of a conditional structure, and those that are scheduled at different time steps are mutually exclusive. Mutually exclusive operations do not cause functional unit allocation conflicts. Two variables are mutually exclusive, and thus do not cause register allocation conflict, if their life spans do not overlap.

A graph coloring algorithm can be used to obtain the optimal functional unit and register allocation solution. The data flow graph operations are treated as the coloring graph vertices. An edge is placed between any two operations that are not mutually exclusive. The solution is to find the minimum number of colors to color this graph. The same process can be applied

for register allocation.

A functional unit may have multiple operation allocated to it. Consequently, more than one register may need to be connected to each port of this functional unit. A multiplexer is used to select and feed the correct signal. A multiplexer that is connected to an input port of a functional unit is called input multiplexer. A multiplexer that is connected to a functional unit's output port is called an output multiplexer. A output multiplexer is generated when the results from different functional units need to be stored in a common register.

Many allocations algorithm were developed. MAHA in [32] is an early allocation algorithm. It allocates the critical path and divides it into n time step, where n is the number of nodes in the critical path. MAHA tries to allocate multiple operations to a single ALU, as long as they do not occur in the same time step. The other nodes that are not located in the critical path are allocated to idle resources. An operation is moved up and down within its slack time to find an idle resource. If there is no idle resource, then MAHA adds a new resource of that type.

The authors in [43], solve the allocation problem by solving a bipartite graph weighted matching problem. The problem is divided into three parts: register allocation, operation assignment, and connection allocation. The first two parts are modeled as a bipartite graph weighted matching problem and solved using the Hungarian Method. For register allocation, variables are sorted according to their birth times. The maximum number of needed registers equals the number of live variables in the most dense control step. A cluster of mutually unsharable variable is allocated at each time step by creat-

ing a bipartite graph. The first set of vertices in the bipartite graph represents the variables in the unsharable cluster and the other set of vertices represents the available registers. An edge is placed between two vertices, one from each set, only if a variable can be assigned to a register, and a weight is assigned to the edge. An optimal allocation solution can be obtained by solving the weighted bipartite matching problem. An ALU allocation is obtained by applying the same concept to the operation assignment problem. The third part connection allocation is solved using a greedy algorithm.

In [35] an allocation algorithm for clock period optimization is developed. Given a fixed number of resources, different assignments lead to different clock periods. The allocation algorithm was developed to find an allocation that leads to minimal clock period implementation.

Some algorithms are based on probabilistic searching mechanisms. These algorithms try to solve both scheduling and allocation problems at the same time. Tabu search is used in [18]. Given an initial schedule, nodes are moved from one control step to another. A node is moved within its stack time limit. During each iteration, a set of solutions in the neighborhood of the current solution is generated. The best solutions are kept in a new set. A list of some of the best solutions from the last k iterations are kept in the *tabu* list. Going back to these solutions is not allowed, to avoid cycles in the iterative process.

Simulated annealing is used in [33]. A two-dimensional placement problem is formed. Time forms one dimension and resources form the other dimension. Each operation is placed in a cell in the table. Two numbers are generated randomly. The first number represents the time step when an operation will be scheduled and the second number represents a specific ALU to which an

operation is allocated. A new state is generated by interchanging the placement of two operations or by interchanging the two operands of a symmetric operation. The cost function is evaluated and some states are accepted. The higher the temperature, the higher the chance of selecting a costly state. The number of states generated per temperature point is a multiple of the number of operations. For every iteration, the temperature is lowered by about 0.9 point. The annealing process terminates when the cost function does not change for three temperature points.

Genetic algorithms are used in [23, 34]. Scheduling and allocation in [23] are performed at the same time. A quadratic assignment problem is formulated. The cost function minimizes both time and resources. An initial population of assignments are evaluated. The fittest assignment solutions are crossed over and a mutation is applied to some of the new population. The new population becomes the current population. This process is repeated many times until the change in the cost function is below a certain threshold.

1.2 Related Work

Every high-level synthesis system, has its own design style, which affects how many datapath components are allocated and how they are interconnected. A high level synthesis system also affects how the controller FSM is constructed. Redesign is a process of transforming one design style to another while preserving the circuit functionality and meeting the design constraints. Transformations can be done to a datapath description or a controller FSM description.

Redesign is not a new concept. Previous work on redesign transformations

has been performed at different levels of the digital circuit design process, in order to achieve different objectives. At a lower level, gate level redesign is used in [10]. The authors try to reuse efficient combinational logic design in the design of a new circuit. A new circuit is designed, then at the logic level, boolean re-substitution is used to replace a part of the circuit logic gates with more efficient and less costly logic gates, which are defined and tested previously.

At the highest level, behavioral transformations are used in [14] to change the ordering and the grouping of the operations and conditional branches to arrive at a behavioral description with minimal cost. Behavioral transformations are also done in [26]. Multiplication and division with a power of two operands are changed to shift left and shift right operations respectively. Loop unrolling and operation moving are used to better utilize hardware at a high level of concurrency. Operation elimination is done for all testing operations, which depend on a constant external input condition. The constant external input condition is fed directly to the controller instead of the datapath.

Some transformations are performed on a digital circuit at the RTL level. In [11], fast scheduling and allocation are performed first, then a layout of the design is generated. Certain RTL patterns are searched for and transformed to other functionally similar patterns but more cost effective in terms of area and delay. This transformation indirectly performs rescheduling and reallocation. To minimize long cycle times, [12] develops a system that carries out some changes to the RTL source description of the circuit to improve the circuit performance or area. After the datapath is constructed in [13], a refinement phase follows. A cluster of datapath components are removed and reallocated.

A binding quality is defined. A component's current binding quality is compared to its worst binding quality from previous iterations. From the set of the components with the worst binding quality, a subset of tightly related components is selected for reallocation. This process is repeated until a better cost datapath emerges.

The datapath transformation in our research is based on [1]. In this paper RTL level reallocation transformations are developed. There are two phases for reallocation. The first phase is the reduce phase, where operations are moved from the most costly ALUs to other less costly ALU. An ALU is deleted after all operations that are allocated to it are removed. The second phase is interconnect reduction. A layout estimator is used to provide interconnect costs. Operations are moved between compatible ALUs within the same time step in order to reduce the interconnect cost.

In controller transformation, the author in [18] did some work on reducing the control path delay by decomposing a centralized controller into multiple local controllers. A layout for a datapath with its single controller is generated. From the layout the location of control sources and sinks and actual wire length are extracted. Maximal decomposition is done, where every control point has its own controller. A clustering phase follows, where groups of control points that are close to each other are controlled by a single controller. a single controller is replaced with multiple small local controllers. The original control points are partitioned according to their proximity to each other in the physical layout.

1.3 Motivation For Redesign

The motivation for the controller transformation is to transform an existing RTL circuit with a centralized controller to an RTL circuit with distributed controller to increase the circuit performance and reduce power consumption, specially at sub-micron technology.

Our approach also based on partitioning the controller output into groups of control points as Eppling work in [18]. However, the control points are grouped at the RTL level. Briefly, a datapath is partitioned around its constituent ALUs, so that all the control points that enable an ALU operation are placed in the same partition. The centralized controller output is partitioned to the control points which resulted from the datapath partitioning. Multiple local controllers are generated where each local controller controls one partition. Each partition is called a functional block and is synthesized separately. Each local controller, also, synthesized separately. By placing and routing each local controller closer to its functional block, control wires are shortened, thus reducing the wiring delay. Each local controller controls a part of the datapath. Partitioning the datapath indirectly partitions the the controller output, thus localizing the activities of the datapath to that specific functional block. This reduces the transistor switching activities, which in turn reduces the power consumption of the circuit. The datapath in the original design is active during all time steps, while some functional blocks are idle in some states. Thus, some local controllers do not change their output at every time step. This reduces more the circuit power consumption. Moreover, Each local controller generates signals for only the control points that are needed for a specific functional block. Consequently, the local controller output logic

is simpler, which leads to a shorter delay through the output logic and less power.

The motivation for datapath transformation is to allow the designer to explore different designs with different area, delay and power cost for the same behavior.

Basic terminologies are presented in chapter two. Transformation from centralized to distributed controller is discussed in chapter three, experimental results are presented in chapter four, and datapath transformations are discussed in chapter five.

Chapter 2

Datapath and Controller

In this chapter some terminologies will be explained.

2.1 Control Data Flow Graph (CDFG)

Data flow graph is a graphical representation showing a sequence of operations and their dependencies. To illustrate this we present the differential equations behavior written in SYNTTEST ¹ format .

```
mul0 := u * dx
mul1 := 5 * x
mul3 := 3 * y
mul2 := mul0 * mul1
add0 := x + dx
mul4 := dx * mul3
cmp := add0 > a
add1 := mul0 + y
sub0 := u - mul2
sub1 := sub0 - mul5
```

The data flow graph (DFG) for these operations is in figure 2.1. Each operation is represented by a node. Two nodes are connected if there is data dependencies between them. For example `mul4` uses `mul3` as operand, so there is an edge

¹SYNTTEST is a high level synthesis system. It is home developed. [48]

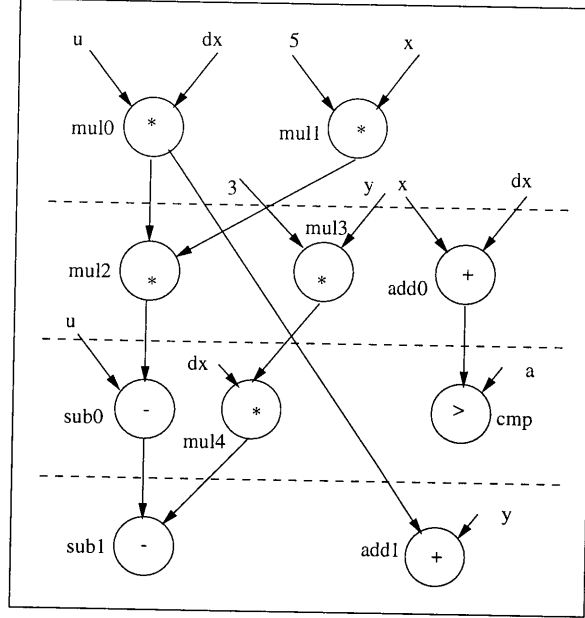
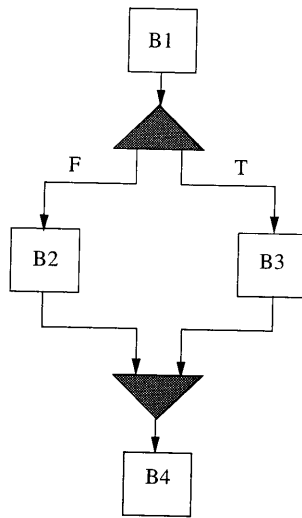


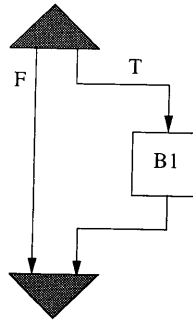
Figure 2.1: differential equation data flow graph (DFG).

from mul3 to mul4. When each operation is assigned to specific time step, then we have scheduled data flow graph as in Figure 2.1. A data flow graph is called a basic block and it does not have any control structure. A control data flow graph (CDFG) is a behavior which has control structure like conditional branch or loop. A diagram of a control data flow graph with an if-else structure is shown in figure 2.2(a).

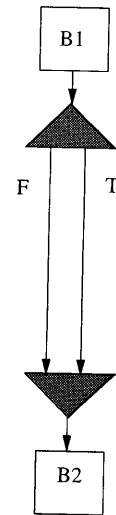
Each rectangle represents a basic block. The triangle represents the select symbol and the inverted triangle represents the join symbol. Basic block B_1 will be executed first. Only one basic block after the select symbol will be executed. Either basic block B_2 or B_3 will be executed according to the condition value false or true. All basic blocks after the select symbol join and the basic block after the join symbol is executed. So basic block B_4 will be executed after B_2 or B_3 . There must be at least one basic block after the select symbol. The diagram in figure 2.2(b) represents if without else part.



a: if structure cdfg digram



b: legal cdfg



c: illegal cdfg

Figure 2.2: Control data flow graph (CDFG) diagram.

The diagram in 2.2(c) is not allowed because there is no effect of the condition structure. CDFG can be complex, the CDFG in figure 2.3 is for the pseudo code below

```

if(cond1) then
    sequence of operations (B1)
    if (cond2) then
        if(cond3) then
            sequence of operations (B2)
        endif
    else
        if(cond4) then
            sequence of operations (B3)
        endif
    endif
endif
endif
  
```

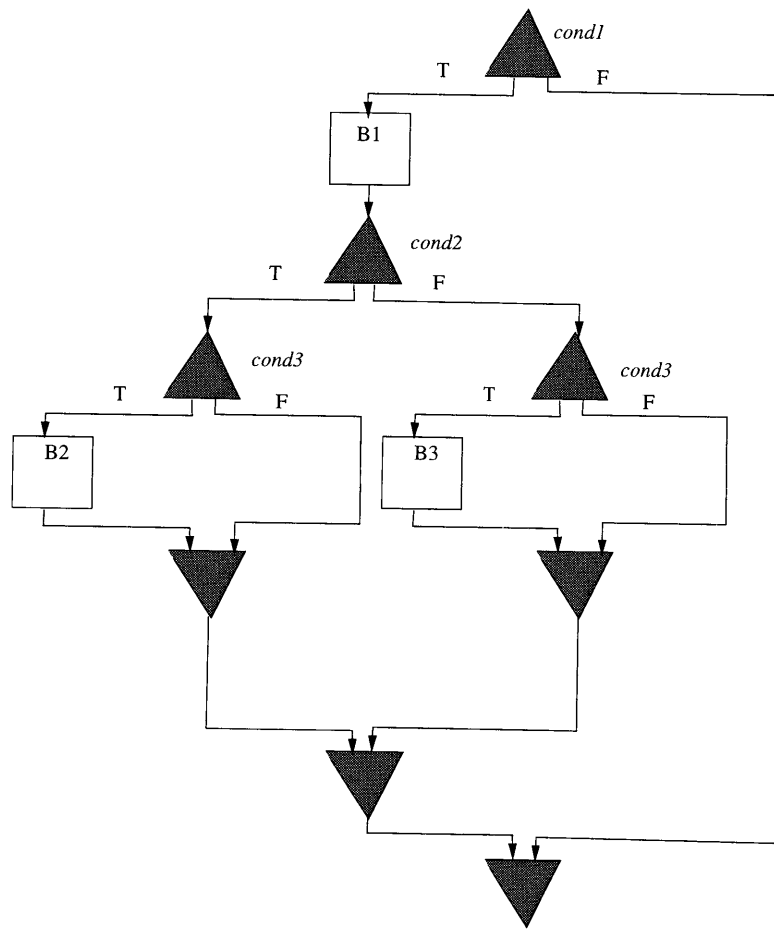


Figure 2.3: Complex CDFG diagram.

The CDFG in figure 2.4 is for the following behavior written in SYNTTEST format:

```

k := a * c
h := d + c
CONDITION k
DIVERGE 2
CASE True
x := k + a
y := x * h
ENDCASE
CASE False
q := k + b
ENDCASE
MERGE 1
JOIN y q INTO z
ENDCONDITION
w := z * c
p := a * w
v := a + w

```

There are four basic blocks. The first basic block can be executed in single time step. The value of the variable k will determine which branch will be taken. If $k \neq 1$ then the left basic block with single operation will be executed. Otherwise the other basic block will be executed. The last basic block will be executed after either the second or the third basic block get executed. The whole behavior can be executed on 5 or 4 time steps depending on which

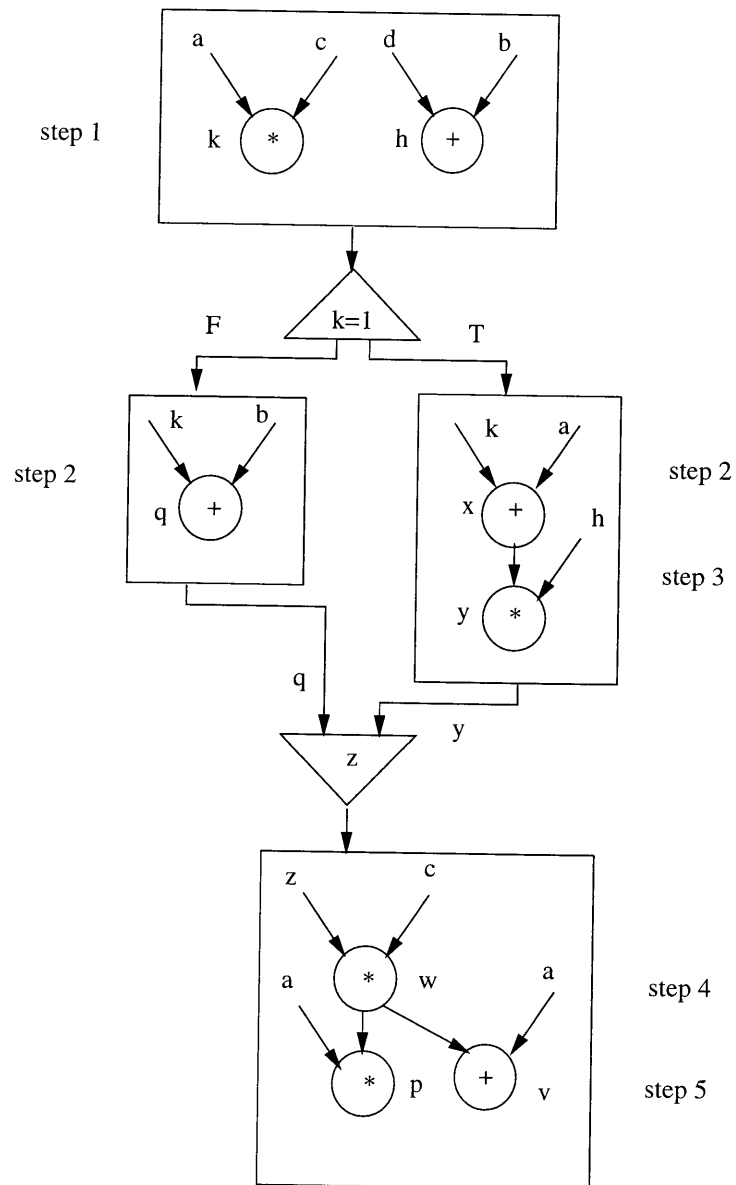


Figure 2.4: If-else control data flow graph (CDFG).

branch is taken. Each basic block in a CDFG can be scheduled separately.

2.2 Datapath

A datapath can be represented as a directed bipartite graph with two sets of nodes: a set of components and a set of connections. There are three types of components : register (REG), multiplexer (MUX) and functional unit (ALU). We assume the usual definition of these components in term of I/O ports and control ports [5]. Each component has connections from and to other components via a connection node from the connection set. A terminal component has only a from or a to connection. Every terminal component is a register but not every register is a terminal component.

We distinguish three types of MUXes according to their connections. An input MUX is connected from REGs to ALU as follows $REG \rightarrow MUX \rightarrow ALU$, where “ $A \rightarrow B$ ” means there is a connection from the output of A to the input of B . An output MUX has the connection $ALU \rightarrow MUX \rightarrow REG$. The connections to an output MUX is from ALUs to Register. An assignment MUX has the connection $REG \rightarrow MUX \rightarrow REG$. An assignment MUX is connected from and to REGs. A MUX which is connected to or from more than one ALU is a shared MUX. In Figure 2.5 MUX $m1$ is an input MUX and it is a shared MUX because it feeds both the multiplier and the subtractor. $m2$ is an output MUX and it is , also, a shared MUX because it is connected to both the adder and the multiplier. $m3$ is an assignment MUX.

An output register is a register which has the connection $ALU \rightarrow REG$ or $MUX \rightarrow REG$. All registers that are connected to a shared output MUX are, also, shared. In Figure 2.5 $r1$ is shared output register because it is

connected to MUX $m2$ which is a shared output MUX. An output register is not necessarily a terminal component. For example in Figure 2.5, $r1$, $r2$, and $r3$ are all output registers and only $r3$ is a terminal component.

The datapath in figure 2.6 is generated for the CDFG in figure 2.4 by SYNTTEST². This datapath has four input latches to store the four external inputs: a, b, c and d . It has only two ALUs, five MUXes and two output REGs.

In the first cycle, after the external inputs are loaded in the latches, both $k = a + c$ and $h = d + b$ are executed. The condition k is stored in register $r1$. The register $r1$ is fed to the controller as input. In the second cycle, depending on the value of k , either $q = k + b$ or $x = k + a$ is executed. In the third cycle, either $w = z + c$ or $y = x + h$ is executed. The result will be in register $r1$ after three cycles if the condition is false ($k \neq 1$). When the condition is true, the operation $w = z + c$ will be executed and the result is stored in register $r1$ after four cycles. the variables q and y are mutually exclusive, only one will be valid. The variable z represents the valid variable. So the operation $w = z + c$ is the same as $w = q + c$ when the condition k is false and it is the same as $w = y + c$ when the condition k is true.

2.2.1 Controller

An FSM can be represented by the structure $[S, I, O, f_s, f_o]$. S is the set of all states, I is the set of all inputs, O is the set of all outputs, $f_s : S \times I \rightarrow S$ is the transition function which maps the current state with an input to the next state and $f_o : S \rightarrow O$ is the FSM output function (Moore machine is used). A state in the controller represents a time step in the scheduled data flow graph.

²This datapath is generated by SYNTTEST, different high level synthesis system may generate different datapath for the same CDFG.

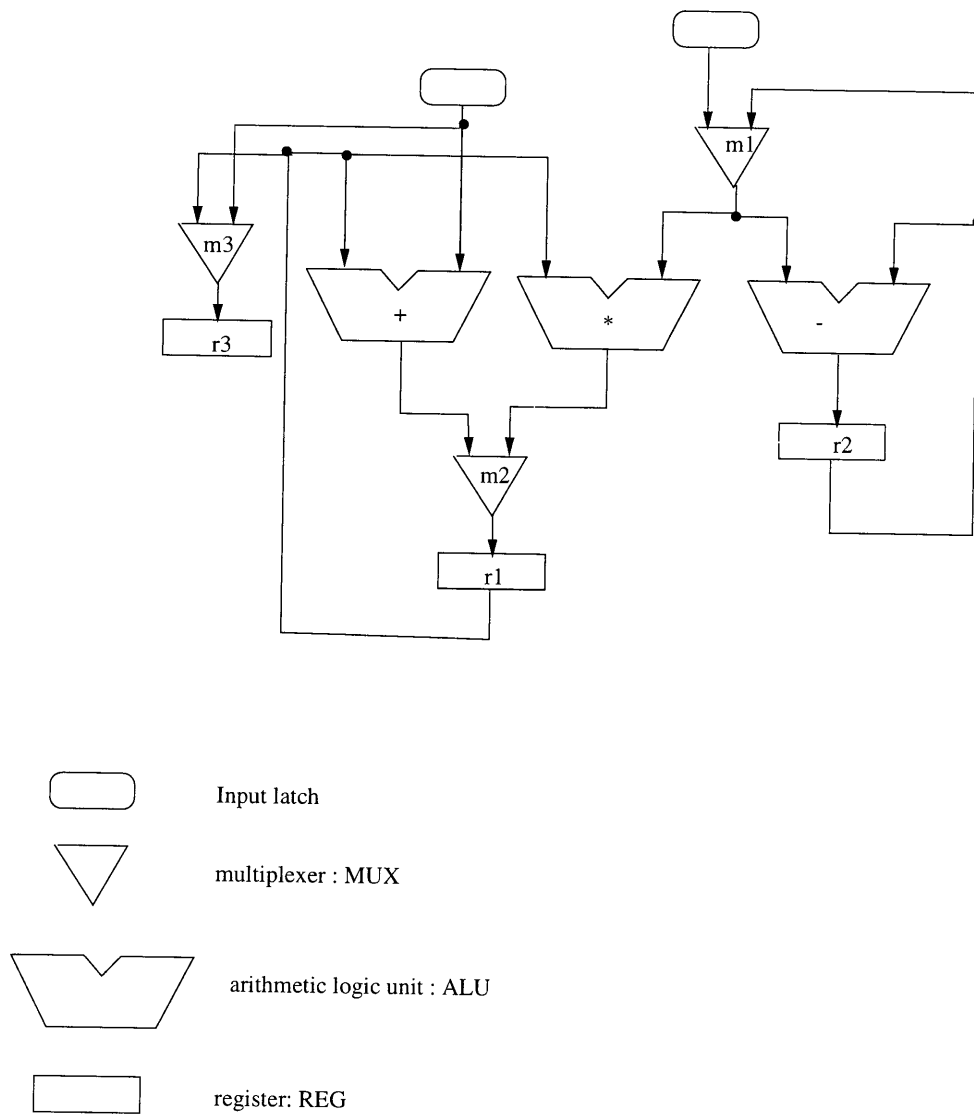


Figure 2.5: MUX types.

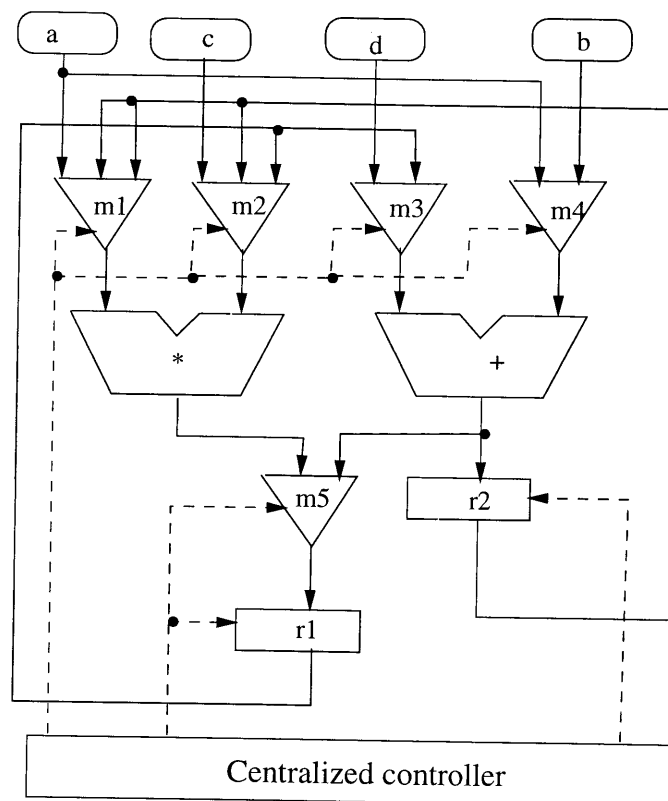


Figure 2.6: a datapath for the if-else CDFG.

At any cycle, only one state is active.

The input set I is the set of conditions in a control data flow graph. A condition can be an external input or generated by an operation executed in the datapath. If a datapath executes a non-conditional data flow graph, there are no inputs for the controller, so $I = \phi$ and the controller goes unconditionally to the next state on the next cycle. When a datapath executes a data flow graph with conditions, the input set consists of the conditions that controls the data flow of a CDFG. The CDFG in Figure 2.4 has one condition k , so the controller input is register $r1$ where the condition k is stored. A controller input bus width is $\sum_{i=1}^{i=n} r_i$, where n is the number of registers that store these conditions and r_i is number of bits of register i . A register may store more than one condition when the life spans of these conditions do not overlap. Generally this register is attached to a comparator, and conditions are generated by operations in the datapath.

The controller output vector is $\langle e_1, e_2, \dots, e_k \rangle$ where e_i is an enabling signal for the component i . Since we have three types of components in a datapath, The controller has three types of enabling signals, namely, MUX select, REG load and ALU function select. REG load is a single bit control signal. MUX select and function select signals can have single or multi-bit width, depending on a MUX's number of inputs and an ALU's number of functions. The width of the output bus of the controller is

$$W = R + \sum_{i=1}^{i=M} \log_2 f(MUX_i) + \sum_{i=1}^{i=F} \log_2 g(ALU_i)$$

where R , M and F are number of REGs, MUXes and ALUs respectively and $f(MUX_i)$ is number of inputs of MUX_i and $g(ALU_i)$ is number of functions of ALU_i .

There are three stages for executing an operation on a specific ALU. In the first stage, if the ALU has one or more input MUXes, the controller must generate MUX select signal, to feed the correct operands for the current operation to the ALU. If an ALU has no input MUXes, then these operands are always available and no controller enabling signals are needed. In the second stage, the controller must select the correct function from the set of functions in multi functions ALU. If the ALU is a single function ALU, then no controller function select signal is needed. In the third stage, after the ALU executes the operation, the result must be stored in a register. A register load signal is needed from the controller. If an ALU is connected to the output register indirectly through an output MUX, then a MUX select signal is needed to select the result of this ALU from the other inputs of the output MUX to be stored in the output register.

A controller FSM for a basic block is a simple sequencing counter. Every state has single transition to the next state. For example, the differential equation FSM is shown in figure 2.7, has four states and two extra states. The four states correspond to the four time steps in the DFG in shown in Figure 2.1.

Two states are added at the top of each FSM. State *zero* is the starting state. The output vector is initialized to zero at state *zero*. When the reset signal is high the controller will go to state *zero*. It will self loop in state *zero* until *input_data_ready* (*IDR*) signal is high where it will make transition to state *one*. The *input_data_ready* acknowledge the controller that the external input data for the datapath are ready to be read. At state *one* all external inputs are loaded in their specified input latches. Next cycle the controller

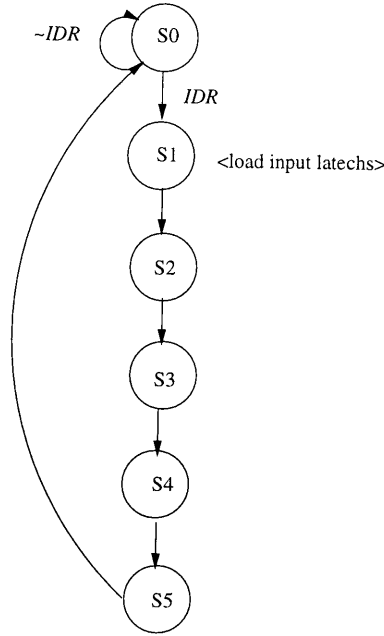


Figure 2.7: The Differential equation behavior FSM.

makes unconditional transition to state *two* which represent the first step of scheduled CDFG. Also there is a unconditional transition from the last states to state *zero*. The last state transition to state *zero* allows the circuit to operate continuously without the need for reset signal. It only wait for *input_data_ready* indicating the availability of new data. So every controller will have these two states added in the top of the first state of the scheduled CDFG and the last state(s) transition(s) to the *zero* state. Figure 2.8 show a general controller diagram.

When we have a CDFG with more than one basic block, first we generate the FSM of each basic block separately. These FSMs are connected in the same way their basic blocks are connected in the CDFG. Let B_d be the basic block before a select symbol; let B_i be a basic block after the select symbol and B_m be the basic block after the join symbol. A transition edge is made from the last state of the FSM of basic block B_d to the first state in the FSM

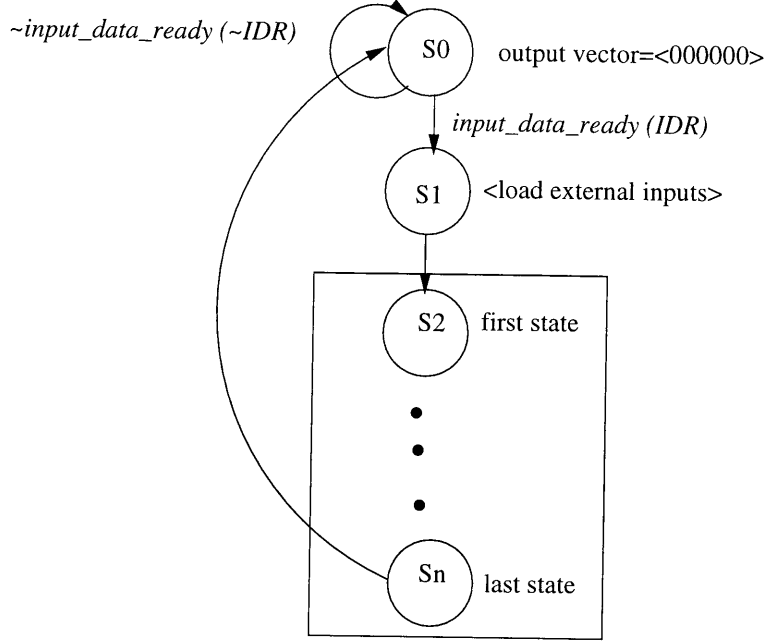


Figure 2.8: general controller diagram.

of basic block B_i . This is a conditional transition. The value of the condition in this edge is the same as the value required in the CDFG to transfer control from basic block B_d to B_i . A transition will be made from the last state of the FSM of basic block B_i to the first state of the FSM of basic block B_m . This is an unconditional transition. The same procedure will be applied to other basic blocks that come after the select symbol.

the number of state of any controller FSM can be estimated as $S = (\sum_{i=1}^{i=n} t(B_i)) + 2$ where n is the number of basic blocks in a CDFG. The function $t(B_i)$ give us the number of time steps in scheduled block B_i

The FSM shown in 2.9 is for the CDFG shown in figure 2.4. The FSM of the basic block $B1$ has one state $S2$. When the condition k is true the control is transfered to basic block $B3$ and when it is false the control is transfered to basic block $B2$. Therefore, there are two transitions from state $S2$. The

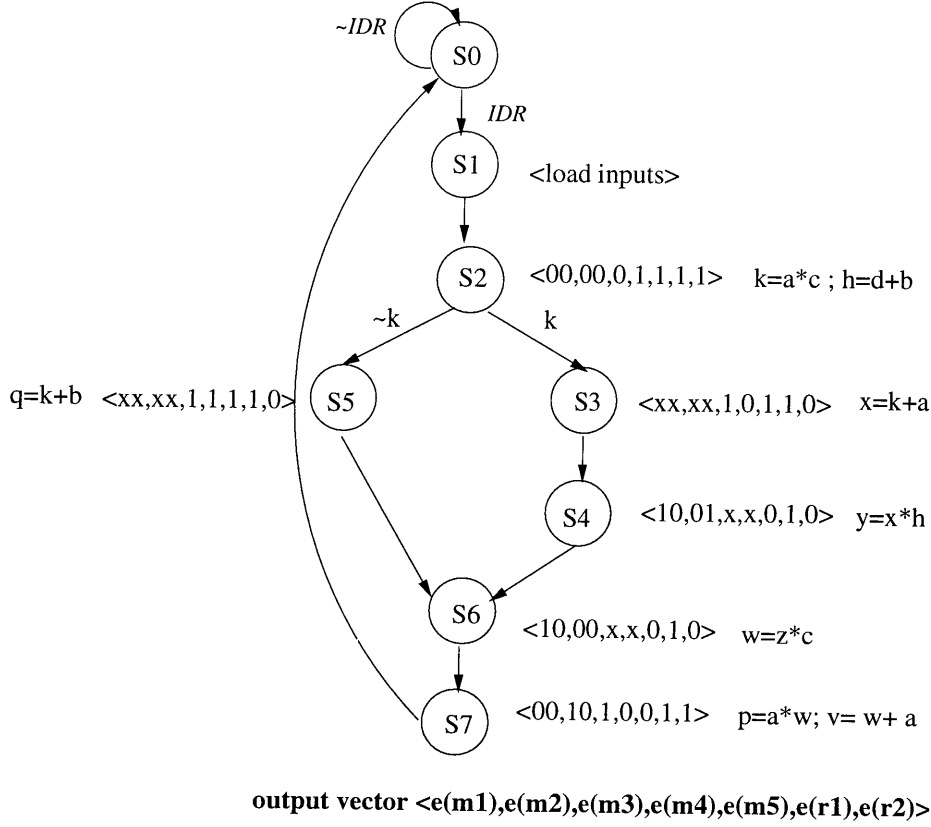


Figure 2.9: A controller FSM for the if-else CDFG in Figure fig7.

transition to state $S5$ is when k is false and the transition to state $S3$ is when k is true. The last states of both basic blocks $B2$ and $B3$ which are $S5$ and $S4$ have unconditional transition to state $S6$ which is the first state of basic block $B4$ coming after the join symbol.

This FSM is to control the datapath shown in figure 2.6. The datapath has two ALUs, five MUXes and two registers. Since both ALUs are single function, they do not need any controller signal. The controller will control the operation of the five MUXes and the two registers. Therefore, the output vector is $\langle e(m1), e(m2), e(m3), e(m4), e(m5), e(r1), e(r2) \rangle$ where $e(x)$ is the enabling signal for component x . The output bus width is equal to $2+2+1+1+1+1+1 = 9$ bits, where each number represents the width of each component in the

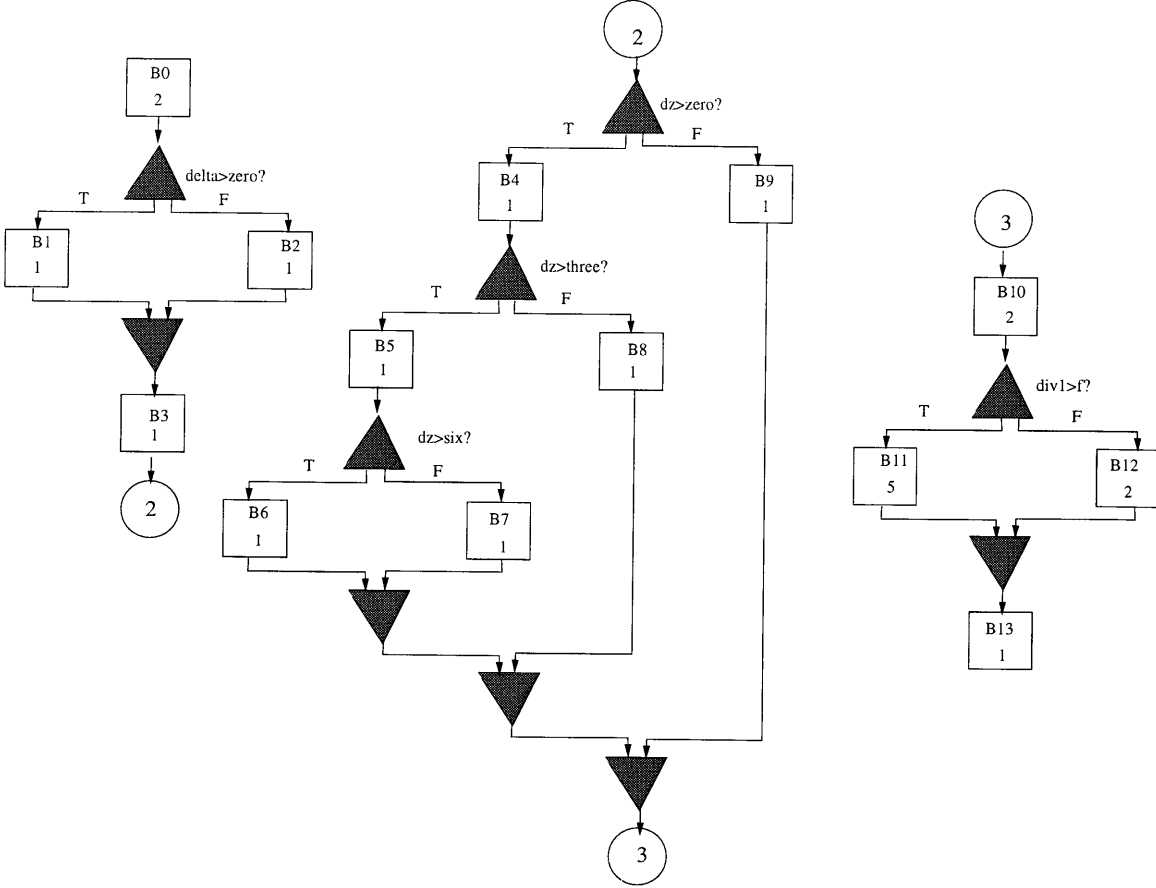


Figure 2.10: Scheduled CDFG for POST calculator behavior.

output vector in the same order.

The controller complexity increases with the number of condition and how these condition are nested. For example, the POST calculator behavior, shown in figure 2.10, has 13 basic blocks. The number of time steps for each basic blocks is written inside each block. Most basic blocks have a single time step. This behavior is control dominated. Figure 2.11 shows the data flow graph for each basic block. The post calculator example has 5 conditions. Three of these conditions are nested, so basic block *B7* is executed only if $dz > 0$, $dz > three$ and $dz \leq six$. The post calculator datapath is shown in Figure 2.12. The datapath has four ALUs, eight MUXes, and eight output

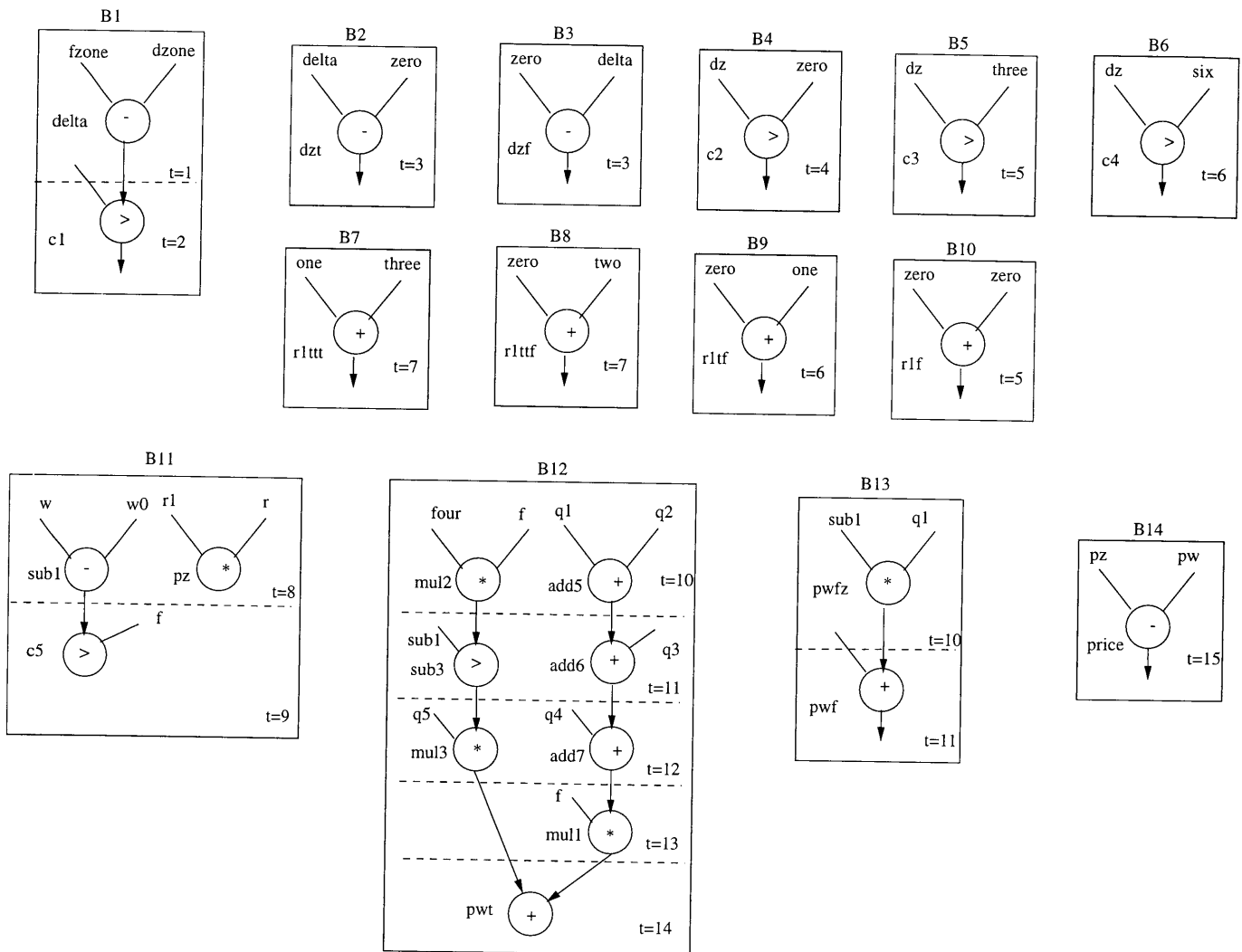


Figure 2.11: POST calculator basic blocks.

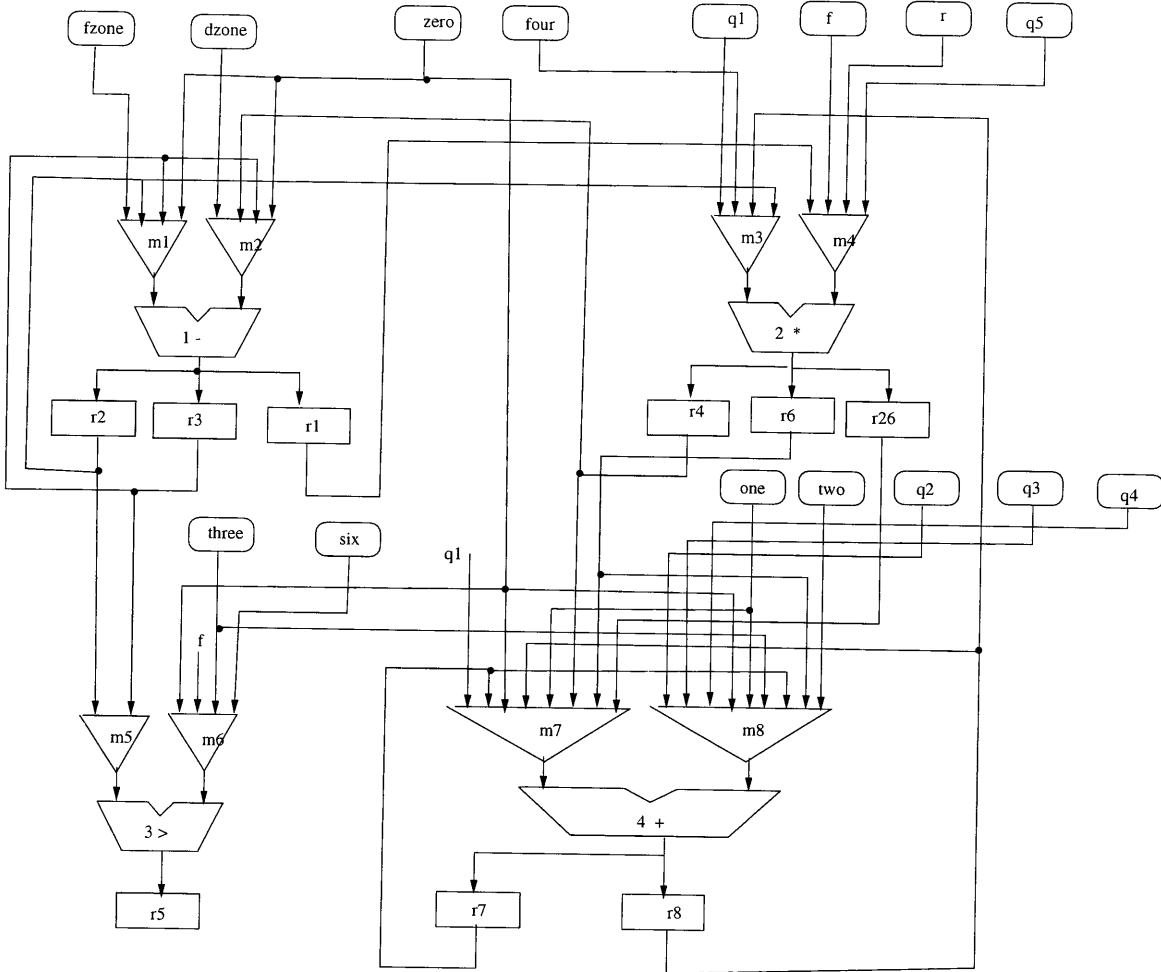


Figure 2.12: POST calculator datapath.

registers. The POST behavior controller is in Figure 2.13. All conditions ($c1, c2, c3, c4, c5$) are coming from register five, because their live span do not overlap. Even though, we have five conditions but only one bit, which is the size of register five, feed the controller as input. The output vector is 9+18 bits wide. 9 bits are register load and 18 bits for MUXes select signals. The Postage FSM has 23 states.

Chapter 3

Controller Transformations

When a behavior is given to a high-level synthesis system, it generates an RTL circuit description for its datapath and an FSM description for its controller. The distributed controller design is generated, first by partitioning a datapath into functional blocks. Then the controller FSM is decomposed into multiple local controllers, each controlling a single functional block. The datapath partitioning, therefore, indirectly partitions the centralized controller output into fields, with each local controller producing the fields specific to its functional block.

3.1 Functional Block

An RTL path in a datapath follows a register transfer flow, thus, an RTL path is described $REG_{input} \rightarrow MUX_{input} \rightarrow ALU \rightarrow MUX_{output} \rightarrow REG_{output}$. REG_{input} can be an external input latch or a register. Both MUX_{input} and MUX_{output} may or may not exist. Clearly, the functional component of an RTL transfer is an ALU. Given an ALU, the RTL block associated with the ALU consist of all RTL paths involved with this ALU. An RTL block can have shared components. For example, in Figure 3.1 the partitions $\{m1, *, m2, r1\}$ and $\{m1, -, r2\}$ are RTL blocks and the MUX $m1$ is shared between them. A functional block is an RTL block with no shared components. It consist of an ALU with all MUXes and registers that are connected to it and not

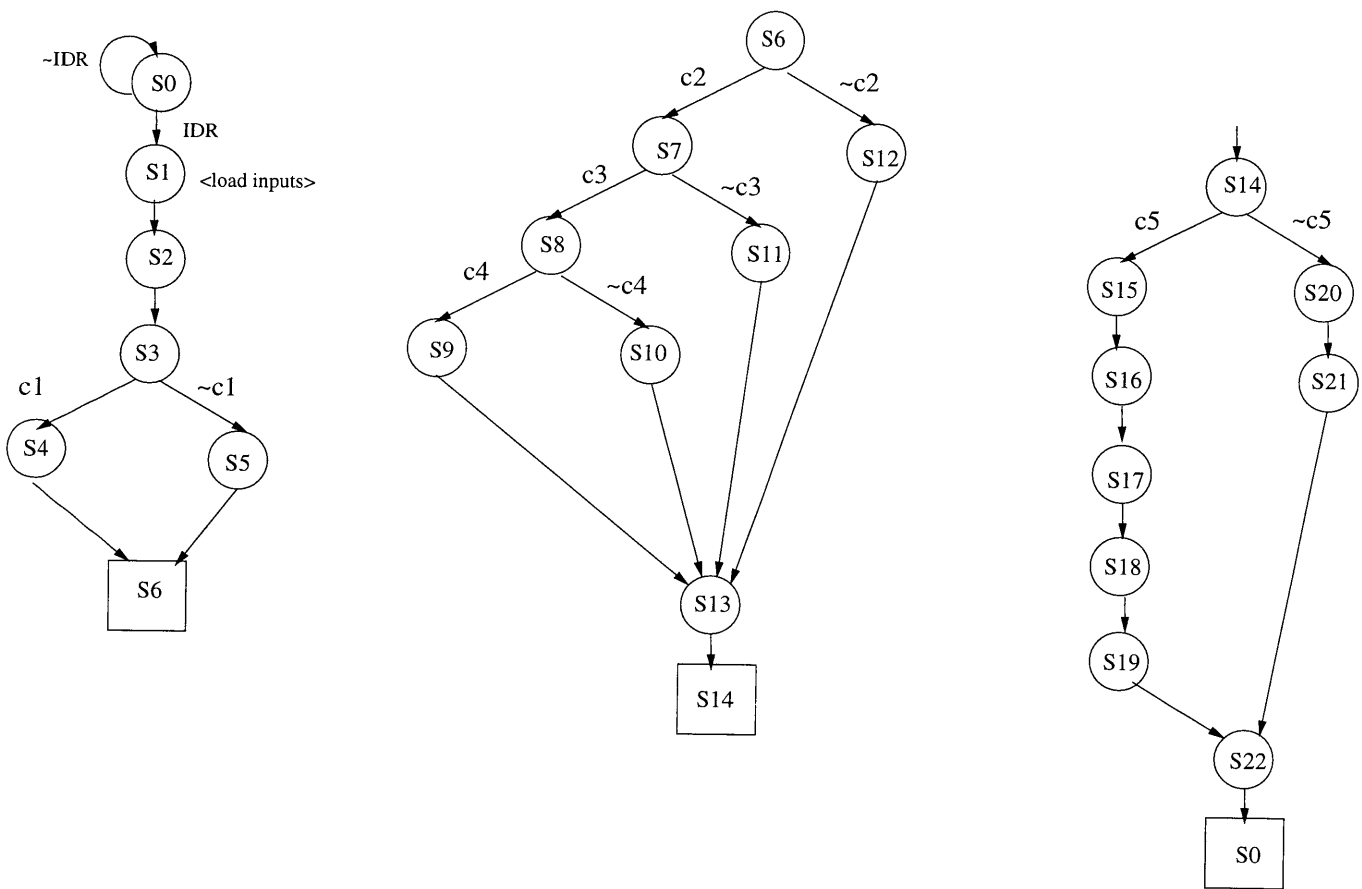


Figure 2.13: POST calculator controller FSM.

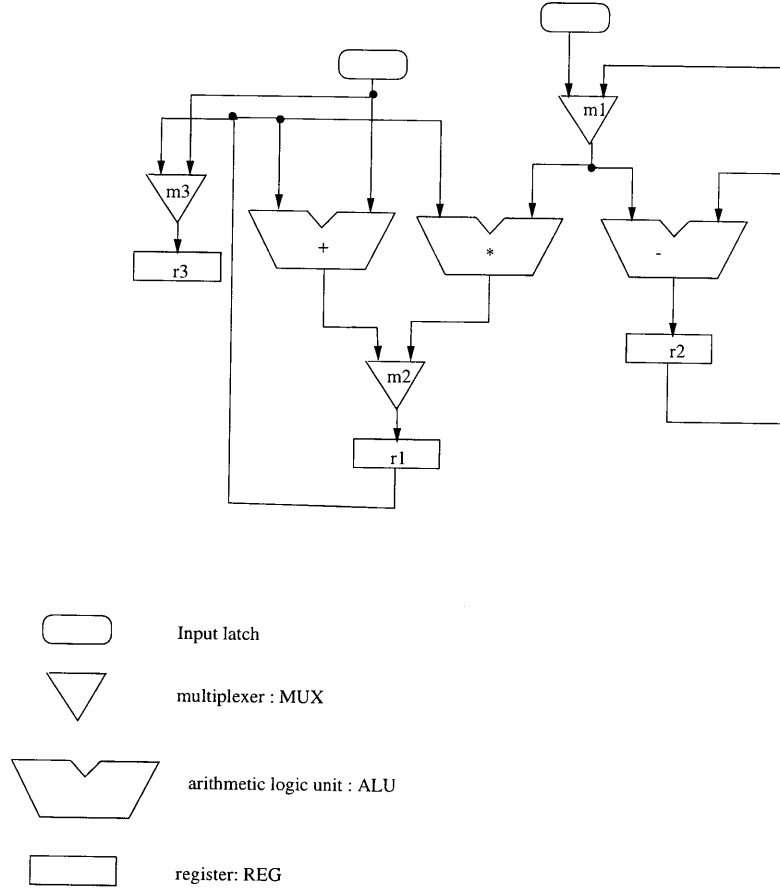


Figure 3.1: Functional blocks.

shared. A shared components between ALUs, is placed only in one functional block. The the datapath in Figure 3.1, the partition $\{m1, *, r2\}$ does not comprise a functional block because register $r2$ is not connected to the multiplier or the MUX $m1$. However, the partition $\{m1, *, m2, r1\}$ constitutes a valid functional block. The functional blocks that are generated by the datapath partitioning algorithm are disjoint. Every component in the original datapath belongs to one and only one functional block, and every ALU will be in a distinct functional block. The activity of an ALU, which is defined as the number of operation instances associated with ALU, is the criteria used to decide in

which partition a shared component will be placed.

3.1.1 Assignment Functional Block

The assignment operation transfers data from one register to another without any modification. The assignment operation does not require a physical ALU. Figure 3.2 shows three types of assignment functional blocks. The first type of assignment functional block is a data transfer from one register to another register. For example, an assignment functional block as the one in Figure 3.2 (a) of Figure 3.2 is generated for the assignment statements: $A := B, C := F$, where A and C are stored in register $r2$ and B and F are stored in register $r1$.

An other type is constructed when we have a data transfer from many registers to one register. An input MUX is needed to select which input register will be transferred to the output register. For the previous assignment operations, a type two assignment functional block is generated if both A and C are stored in register $r2$ and B is stored in register $r1$ and F is stored in $r3$ as shown shown in figure 3.2(b).

The third type of assignment functional block is a data transfer from many registers to many registers. It is similar to type two, except there can be several output registers. A type three functional block can be generated, for the previous two assignment operations, if A is stored in register $r2$ and C is stored in register $r4$, B is stored in register $r1$, F is stored in $r3$ as shown in figure 3.2(c). Only type two and type three assignment functional blocks are allowed in our datapath.

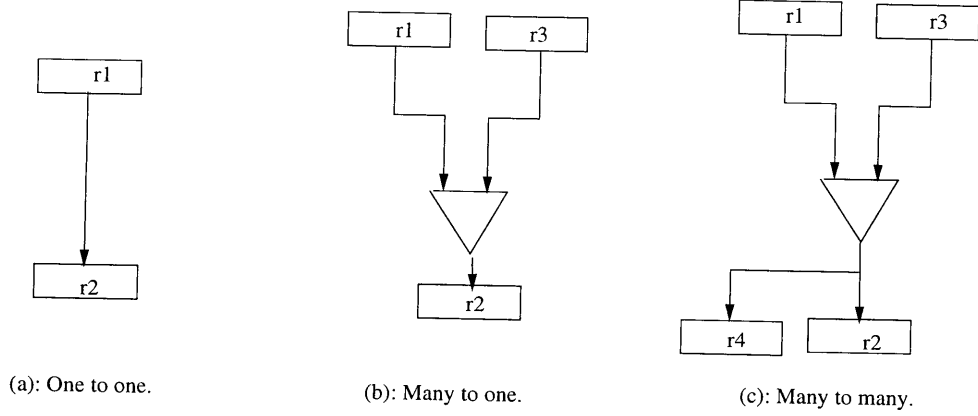


Figure 3.2: Assignment functional block types.

3.2 Datapath Partitioning Algorithm

The datapath partitioning algorithm places each ALU in the datapath in a separate partition. All non shared components that are attached to an ALU are placed in the same partition as that ALU. A shared component is placed in only one partition. If a MUX is shared between many ALUs, then it will be placed with the most active ALU from the set of ALUs that share it. Any register that is attached to an output MUX is placed in the same partition where that output MUX is placed. Each MUX that is connected only to and from registers (assignment MUX) is placed in a new partition with its output registers. The detailed algorithm is in the Appendix A.

After partitioning the datapath, each component of the datapath (except input latches) will be in one and only one partition. Most functional blocks have more than one component. There may be functional blocks that have only a single component. This single component is an ALU. The other components that are connected to this ALU must be shared with other ALUs, therefore, they are placed on other partition. The input ports of that single ALU must

be connected to a single register or a single latch (no input MUXes), or its input MUXes are shared with other ALUs and are placed in the partition of the most active ALU of these ALUs. Every ALU must be connected to one or more output registers directly or indirectly by an output MUX. Only one ALU can be connected directly to an output register. If the output register is shared between many ALUs, an output MUX is needed to connect these ALUs to that output register. A single ALU in a partition means an output register of this ALU is shared and it is connected to the register through an output MUX. This output register is placed in the same partition with the output MUX.

In Figure 3.3(a), the components $m1, m2$ and $r1$, in the datapath, are shared between the adder and the multiplier. These three shared components are placed in the multiplier partition as shown in figure 3.3(b) because the multiplier has more operation to execute than the adder. So, the adder is placed in the other partition with no other components. The adder will not have a local controller because it is a single function ALU.

To show how the partitioning algorithm works, we will apply it to the datapath in Figure 3.4(a). The number of operations allocated to each ALU are shown in the same Figure. Since this datapath has three ALUs, three functional block partitions are created and each ALU is placed in one of these partitions: $FB_1 = \{+\}$, $FB_2 = \{-\}$ and $FB_3 = \{*\}$. The input MUX $m1$ is shared between the multiplier ($*$) and the subtractor ($-$). It is placed with the multiplier because it has five operations to execute and the subtractor has only three operations. The output MUX $m2$ is placed with multiplier as well because it is more active than the adder ($+$) which has only four operations

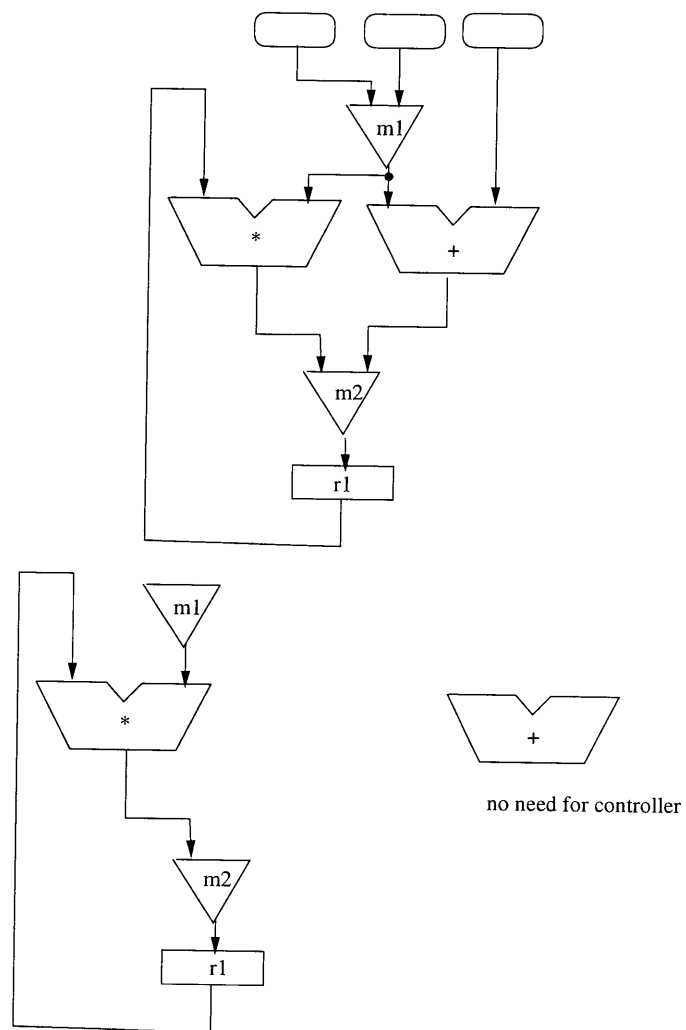


Figure 3.3: Functional block with no controller.

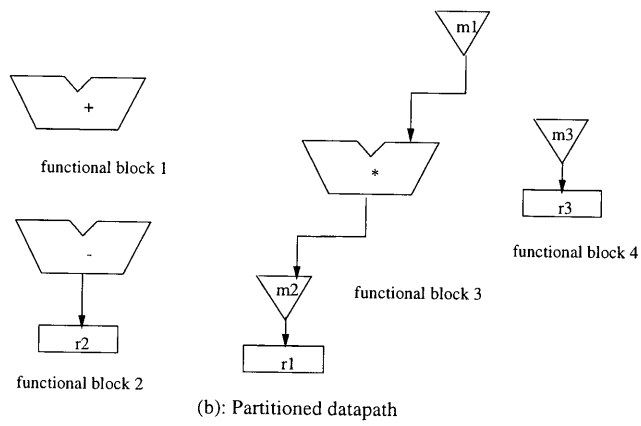
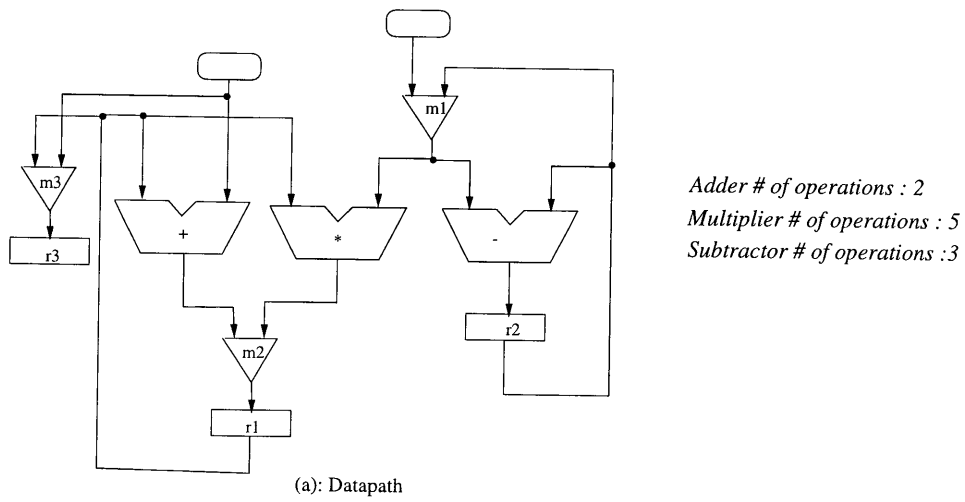


Figure 3.4: Datapath partitioning example.

to execute. The output register $r1$ is placed in the same partition with $m2$ because all output registers that are connected to an output MUX are placed in the same partition as that output MUX. Register $r2$ is connected directly to the subtractor, so it will be with the subtractor. Since MUX $m3$ is an assignment MUX, it is placed in new functional block with its output register $r3$.

At the end we have four functional blocks. The first one is $FB_1 = \{+\}$. The adder does not have input MUXes and its output MUX $m2$ and output register $r1$ are placed with the multiplier. Therefore, the adder functional block does not have a local controller, because it does not have any control points. The second functional block is $FB_2 = \{-, r2\}$ and it has a single control point for register $r2$. The third functional block is $FB_3 = \{m1, *, m2, r1\}$. The multiplier local controller will control both MUXes $m1$ and $m2$ and the output register $r1$. The multiplier local controller must enable these components even when they are needed by another ALU. The fourth functional block is $FB_4 = \{m3, r3\}$, which does not have an ALU, but does have an assignment MUX $m3$. Its local controller will control $m3$ and $r3$.

3.3 External Inputs

External inputs latches are registers without a clock signal. All external input data needed by a circuit to perform its function are stored in these latches. Only a single loading signal is needed to load all input latches. The *input_data_ready* signal tells the circuit that the external input data are ready to be read. External inputs latches may be connected to more than one ALU or MUX. These components may be located in different functional blocks, so

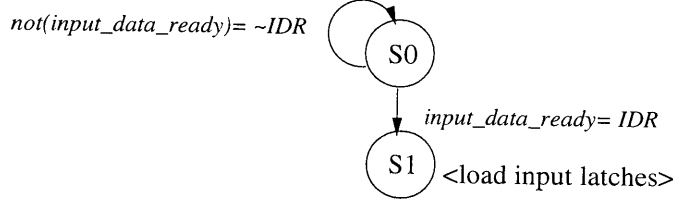


Figure 3.5: External controller to load input latches.

external input latches do not belong to any functional block. Therefore, no local controller is responsible for generating the external inputs load signal.

One solution to load external inputs data is to generate a special local controller as shown in Figure 3.5 which does not belong to any functional block. It has only two states. It starts at state *zero*, when the reset signal goes high. It loops in state *zero* until the *input_data_ready* signal is high, then it makes the transition to state *one*. At state *one*, a load signal is generated to load all latches with external inputs. This solution has an extra local controller added to the circuit design cost.

Another solution to load these external inputs data is to duplicate input latches into all functional blocks that need them as input. All local controllers start at state *zero*, and loop in state *zero* until the *input_data_ready* signal is high. Then, all of the local controllers make the transition to state *one* in the same cycle. All input latches are loaded in state *one*. There is extra cost, because most input latches may be duplicated as many times as the number of functional blocks. For example, if there are five external input data and there are ten functional blocks and all of them needed these five external input data, then 45 extra input latches are added to the circuit cost.

The solution that we use is to let the *input_data_ready* signal be the loading signal for all input latches. When *input_data_ready* goes high, all input

data are loaded into input latches. In this solution all input latches are independent of all functional blocks, there is no external local controller, and input latches are not duplicated. The signal *input_data_ready* is needed to load all input data and to synchronize the operations of all local controllers in both distributed controller design models. All local controllers start at state *zero*, when the reset signal goes high. All of them loop in state *zero* until the *input_data_ready* signal is high, then all local controller will make transition to state *one*. Nothing will happen at state *one* in all local controller. At the end of one clock cycle, all input data are ready in the input latches, so all local controllers can start from state *two* correctly at the beginning of next clock cycle.

3.4 Distributed Controller

In this section an overall view of how a centralized controller FSM is transformed into a distributed controller is presented. As mentioned earlier, a distributed controller consists of multiple local controllers. Each local controller controls one functional block derived from the partitioning algorithm. For consistencies, the distributed controller must produce the same output in the same time cycle as the centralized controller.

For a distributed controller to perform the same task as the centralized controller, synchronization between all local controllers is needed. The two design methods differ on how synchronization is achieved. Two methods of designing a distributed controller will be presented in subsequent sections. **Both design methods are based on partitioning the centralized controller output into control points according to the datapath partitioning.**

According to the datapath partitioning, the centralized controller output is partitioned into fields. Each field contains the enabling signals for all control points in a specific functional block. These fields are disjoint because every component in the original datapath is placed in one and only one functional block. A local controller is generated for each functional block that has control points. The distributed controller output content and the output timing are the same as the centralized controller output content and timing.

In the next sections, the two design methods will be presented.

3.4.1 Non-Communicating Local Controller (NCLC)

We discuss, here, how a centralized controller is transformed into Non-Communicating Local Controllers (NCLC).

Let $[S, I, O, f_s, f_o]$ be the centralized controller finite state machine. The output O contains all control signals for all control points in a datapath. Let O_i be the control signals for the functional block FB_i . $\bigcup_{i=1}^n O_i = O$ and $\bigcap_{i=1}^n O_i = \phi$, where n is the number of functional blocks generated by the datapath partitioning algorithm. The approach taken is to duplicate the centralized controller into at most n local controllers, with each local controller having the same state set S , input set I and the same transition function f_s as the centralized controller. The local controller for the functional block FB_i is $[S, I, O_i, f_s, f_{oi}]$.

The output function $f_{oi} : S \rightarrow O_i$ is the same as the f_o function except its range is O_i instead of O . The output produced by a local controller at state s is a field of the output produced by the centralized controller at state s . The synchronization among the local controllers is done by making all local

controllers have the same state transitions as the centralized controller. thus, a local controller must traverse every state that a centralized controller would traverse, even though its functional block may be idle in some of these states.

To illustrate the two distributed controller design, the datapath in Figure 3.7(a) is generated for the control data flow graph shown in Figure 3.6.

Figure 3.7(b), shows the two functional blocks generated after the datapath is partitioned and their local controllers. The multiplier functional block is $\{m1, m2, m5, r1, *\}$ and the adder functional block is $\{m3, m4, r2, +\}$.

The original centralized controller for the if-else CDFG is shown in Figure 3.8. State 2 represents the first time step in the control data flow graph. The output vector is $\langle e(m1), e(m2), e(m3), e(m4), e(m5), e(r1), e(r2) \rangle$, where $e(x)$ is the enabling signal for component x . The output bus is nine bits wide.

Figure 3.9 shows the NCLC local controllers generated from the centralized controller of the if-else behavior. The centralized controller control vector $\langle e(m1), e(m2), e(m3), e(m4), e(m5), e(r1), e(r2) \rangle$ is partitioned between the two local controllers. For example, the multiplier functional block has three MUXes namely $m1$, $m2$, and $m5$ and registers $r1$, so the multiplier local controller output control vector is $\langle e(m1), e(m2), e(m5), e(r1) \rangle$. The adder functional block local controller output control vector is $\langle e(m3), e(m4), e(r2) \rangle$. The combined output of the two local controllers is the same as the centralized controller output.

The output MUX $m5$ and the register $r1$ are shared between the multiplier and the adder, however, the multiplier local controller controls both $m5$ and $r1$. Even though the multiplier is idle in states $S3$ and $S5$, control signals are

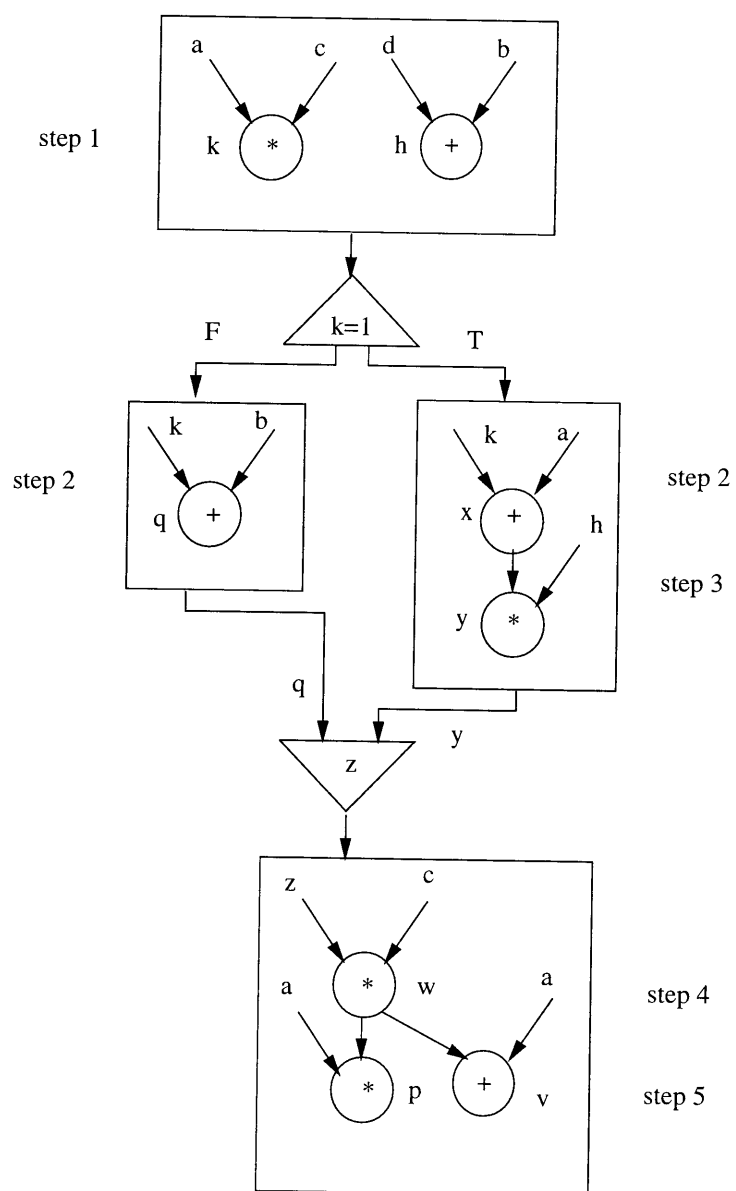
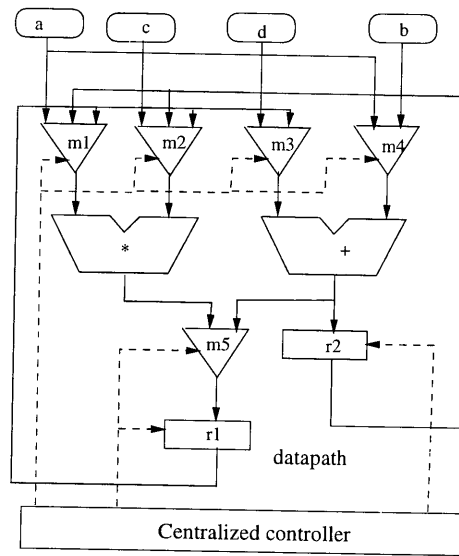
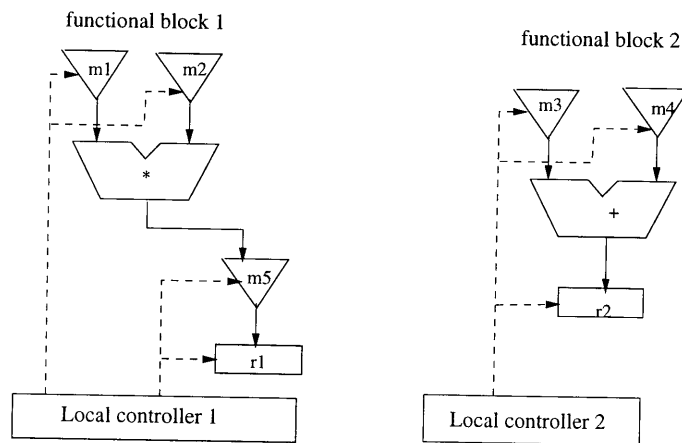


Figure 3.6: If-else control data flow graph (CDFG).

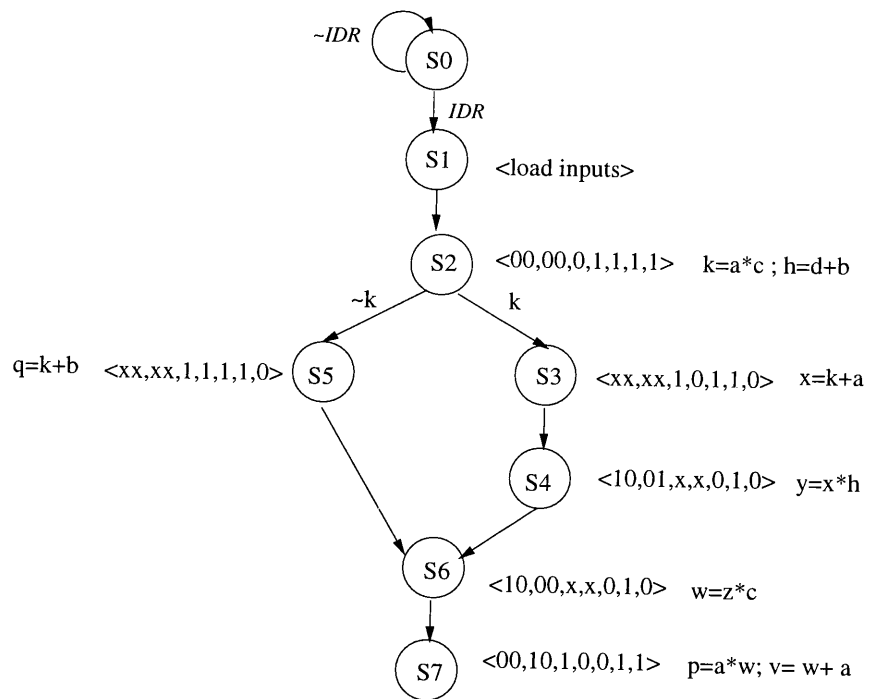


(a): Centralized controller design



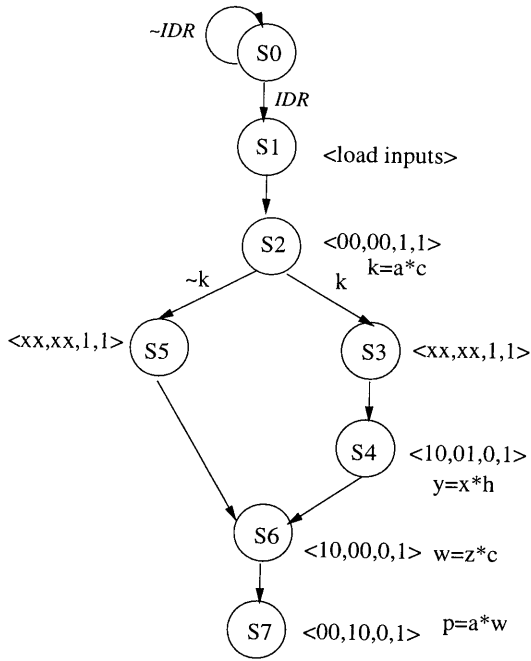
(b): Distributed controller design

Figure 3.7: datapath for If-else behavior.



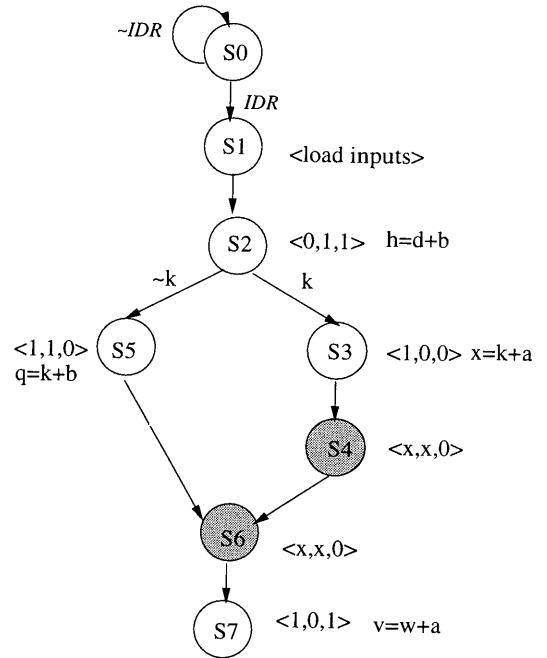
output control vector $\langle e(m1),e(m2),e(m3),e(m4),e(m5),e(r1),e(r2) \rangle$

Figure 3.8: Centralized Controller.



Multiplier local controller.

output control vector $\langle e(m1), e(m2), e(m5), e(r1) \rangle$



Adder local controller.

output control vector $\langle e(m3), e(m4), e(r2) \rangle$

Figure 3.9: Non-Communicating Local Controllers (NCLC).

needed to select and store the adder results from the two operations $x = k + a$ and $q = k + b$. The adder must be selected from the MUX $m5$ inputs and the register $r1$ must be loaded with the result from the adder. Therefore, the multiplier local controller is active in states $S3$ and $S5$ to enable the shared MUX $m5$ and register $r1$ for the adder. As a result, the multiplier local controller is active in more time steps than the multiplier ALU. There are 4 multiplication operations, so the multiplier ALU is active in four states. The multiplier local controller is active in six states. The adder is not active in states $S4$ and $S6$. The adder local controller has to make transitions to these states, however, to remain synchronized with the multiplier local controller. The total number of states in the NCLC design is double, but the output function of each local controller is simpler.

3.4.2 Communicating Local Controller (CLC)

In the previous design, all local controllers are independent of each other and there is no communication between them. Synchronization is achieved simply by duplicating the whole centralized controller FSM and its transition function in every local controller.

An alternative design is to eliminate all idle states from each local controller, where an idle state is one in which no operation is executed. In this design, synchronization is achieved by communicating next state information between local controllers.

Definition Active State: A state in which there is at least one operation executed: $s \rightarrow o_1$, where o_1 output control vector enables at least one MUX_{select} , ALU_{select} , or REG_{load} .

Definition Idle State: A state in which there is no operation executed: $s \rightarrow o_0$, where o_0 output control vector does not enable any MUX_{select} , ALU_{select} , or REG_{load} .

Definition Sleeping State: Sleeping state s is the last active state before an idle state, such that $s \times i \rightarrow q$ where q is an idle state.

Definition $IDLE_PATH(s)_x$ set: Given a sleeping state s the $IDLE_PATH(s)_x$ is a set of connect idle states which are reachable from the sleeping state s .

$IDLE_PATH(s) = \{s_i | i = 1..n\}$ and

$s \times i_1 \rightarrow s_1 \times i_2 \rightarrow s_2 \times i_3 \dots s_n \times i_n \rightarrow s_{n+1}$

where s_{n+1} is an active state.

Definition $IDLE(s)$ set: Given a sleeping state s the $IDLE(s)$ is the union of all $IDLE_PATH(s)$:

$IDLE(s) = \bigcup_{i=1}^{i=m} IDLE_PATH(s)$ where m is number of $IDLE_PATH(s)$

Definition $ALARM(s)$: Given a sleeping state s the $ALARM(s)$ is a set of non connected active states which are reachable from the sleeping state s via all $IDLE_PATH(s)$.

There are four steps to transform a centralized controller to a CLC distributed controller. 1) an NCLC distributed controller is generated. 2) From the NCLC local controllers, the one that control the most active ALU is designated as the master local controller. 3) All idle states from all local controllers, not including the master local controller, are eliminated. 4) The remaining active states for each local controller are reconnected using new transition function.

A local controller associated with the most active ALU is designated as the master controller. A master controller has the same structure and transitions

as NCLC local controller. However, the master controller has the next state code as an additional output field, called an NS ¹ signal, which is used to wakeup the sleeping (idle) controllers. At every cycle the master controller broadcasts the next state as an NS signal.

All idle states from other local controllers are eliminated. As a result, the local controller FSM may be disconnected. For example, the adder local controller in Figure 3.9 has two idle states $S4$ and $S6$.

For example, in the adder local controller in Figure 3.9, state $S3$ is sleeping state because state $S4$ and $S6$, which are the next states are idle states. Also, state $S5$ is sleeping state because the states $S6$ is an idle state. The only state in the alarm set of the sleeping state $S3$ is $S7$ which is the next active state after $S4$ and $S6$. State $S7$, also, is the only state in the alarm set of the sleeping state $S5$. A sleeping state will loop to itself until the NS signal broadcasted from the master controller matches one of the next states in the alarm set. The sleeping state will then make a transition to an active state.

The next state signal NS is generated by the master controller and fed to all local controllers that have at least one sleeping state. Assuming that every state has a unique code, the communication bus width is $\log_2 k$ where k is the number of states. Some local controllers do not sleep, because they have no idle states. A local controller without any sleeping states is called fully active local controller, and it need not be connected with the master controller via the communication bus. Obviously, the master controller is also a fully active controller as it has no sleeping states.

The following algorithm transforms a centralized controller to CLC distributed controller:

¹ NS signal is one or more bits wide

Input: A centralized controller FSM. Output: n local controller FSMs; n is the number of functional blocks.

1. Transform a centralized controller to an NCLC (Non-Communicating Local Controller) design.
2. From all local controllers in the NCLC design, let the local controller which is associated with most active ALU be the master local controller: $[S, I, O_a, f_s, f_o]$
3. Add the next state (NS) output function to the master local controller output. Let the master local controller original transition function be $f_s : s \times i \rightarrow s_i$ and the output function be $f_o : s \rightarrow o_j$. Change the output from f_o to $f_{oa} : s \rightarrow o_j, s_i$.
4. Traverse all local controllers except the master controller and eliminate all idle states.
5. For every disconnected local controller, connect the remaining FSM as follows:
 For every sleeping state s , find $ALARM(s) = \{s_i : i = 1..k\}$.
 Split the sleeping state transition function to two transitions.
 The self loop transition is $f_s : s \times \text{not}(s_1 + s_2 + \dots + S_k) \Rightarrow s$. The other transition is $f_s : s \times s_i \Rightarrow s_i$.

An example in Figure 3.10 shows the CLC local controller generated from the centralized controller for the if-else example in figure 3.8. The multiplier local controller will be the master local controller because it is the most active controller. At state $S2$, the master controller will broadcast $S5$ or $S3$ depending on the value of the condition k . If the condition is true, then $S3$ will be broadcasted, otherwise will be $S5$. The adder local controller has two sleeping states $S5$ and $S3$. However, only one of these states will be a valid

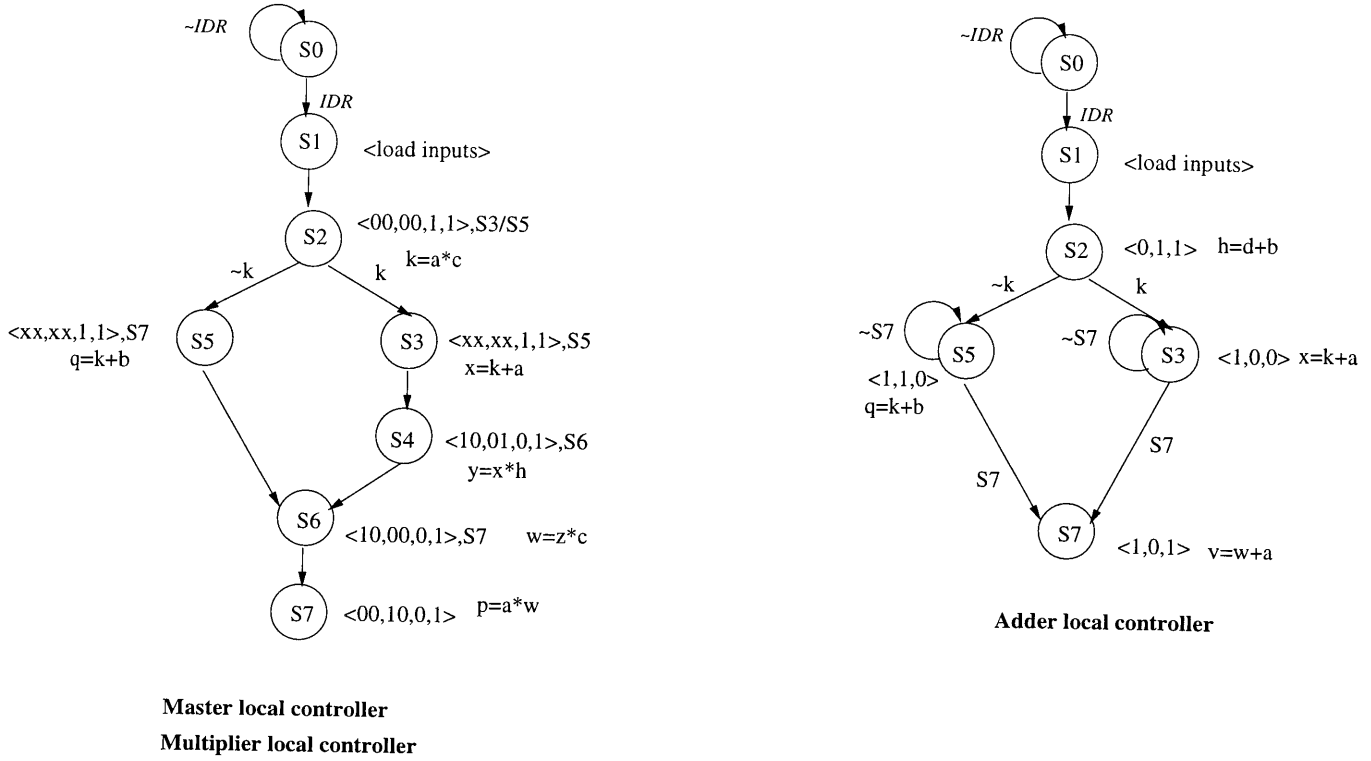


Figure 3.10: Communicating local controllers.

sleeping state depending on the condition value. The next active state is $S7$, so the alarm set contain only $S7$. The valid sleeping state will self-loop until the master controller broadcasts $S7$ as the next state.

3.5 CLC Design Issues

We faced a few problems when designing the communicating distributed controller (CLC).

3.5.1 Mealy vs. Moore Machines

The first problem is related to the CLC FSM implementation as a Mealy or Moore machine. Both Mealy machine and Moore machine are an FSM

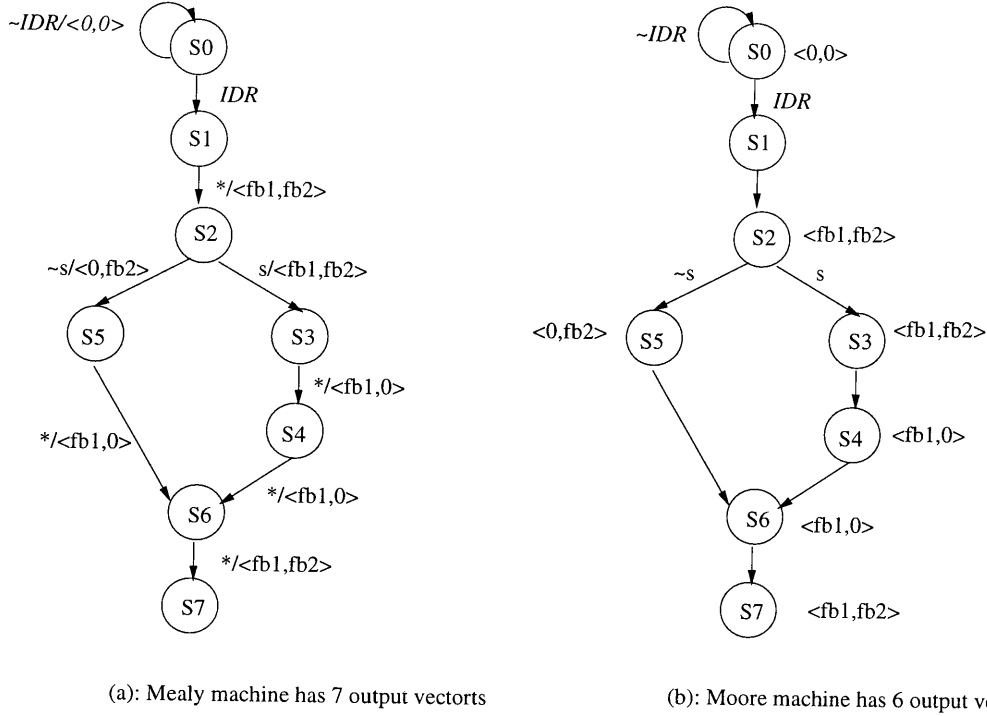


Figure 3.11: Moore and Mealy implementation with similar cost.

structure $[S, I, O, f_s, f_o]$ [5]. They differ only one the output function. Mealy machine FSM output depends on both the current state and the current input: $f_o : S \times I \rightarrow O$, where Moore machine output depends only on the current state: $f_o : S \rightarrow O$.

In Figure 3.11, an FSM is implemented as a Moore machine with six output vectors as shown in part (a) and as Mealy machine with seven output vectors as shown in part (b). However, the Mealy implementation of the FSM in Figure 3.12(b) has ten output vectors, compared to six output vectors in its Moore implementation. As the number of transition edges increase, so does the difference in implementation cost between Mealy and Moore machines. Both the centralized controller and the NCLC distributed controller do not have self loops in any state except the *zero* state, so the cost difference between Moore

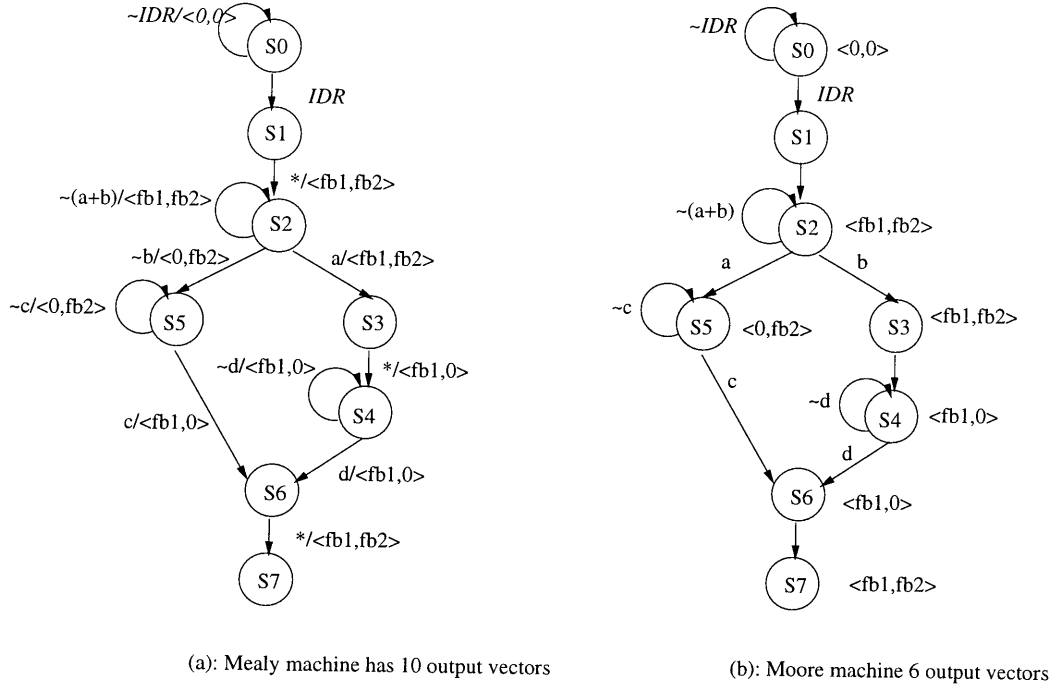


Figure 3.12: Moore and Mealy implementation with different cost.

and Mealy implementation for NCLC is not noticeable. However, the communicating local controller (CLC) has more self loops (more transition edges) due to sleeping states. Also, the CLC local controllers need to communicate NS to achieve synchronization. The NS signal is treated as input by other CLCs. This information affects the CLC state transition function and increases the number of edges in a CLC FSM, so the implementation of a CLC as a Moore machine costs less than the implementation as a Mealy machine.

3.5.2 NS Signal Code and Communication Bus width

A communication bus is used to communicate the next state signal NS . A minimum number of bits is desired to wake up a sleeping local controller. Let there be k states in a centralized controller. Suppose there are n local controllers and each local controller needs w NS wakeup signals during its

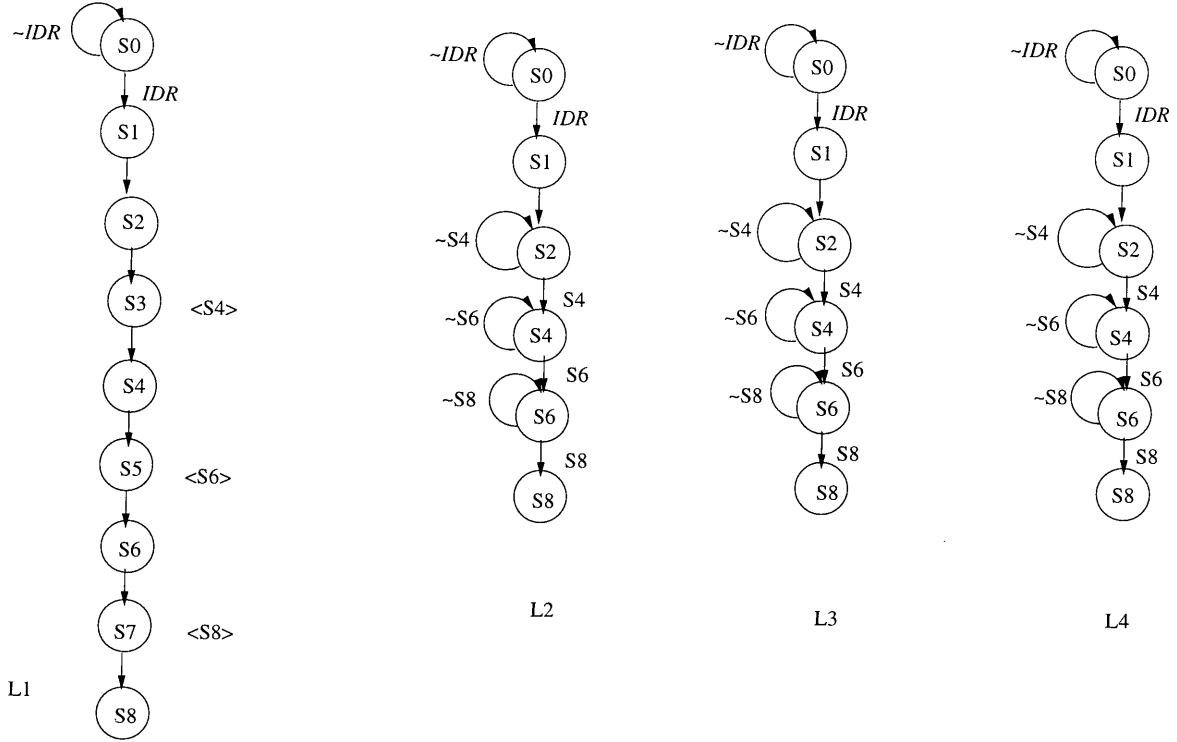


Figure 3.13: Single bit wake up signal, case I.

operation. If all n sleeping controllers need to be waken up by all of the w wakeup signals, a single bit communication wire is sufficient. For example, in Figure 3.13 the local controllers L2, L3, and L4 need to be waken up in states $\{S4, S6, S8\}$. L1 will be the master controller. In this design, the master controller supplies the NS signal only when it is needed. When no controller needs a wakeup signal the one bit bus will be low. When the bus goes high, all sleeping local controllers will make the transition to the next active state. A sleeping local controller does not need to know which state the NS signal represents because the master controller supplies the NS signal when all local controllers are asleep and all sleeping local controllers need to be waken up at the same time.

Another case where a single bit communication wire is sufficient, is when

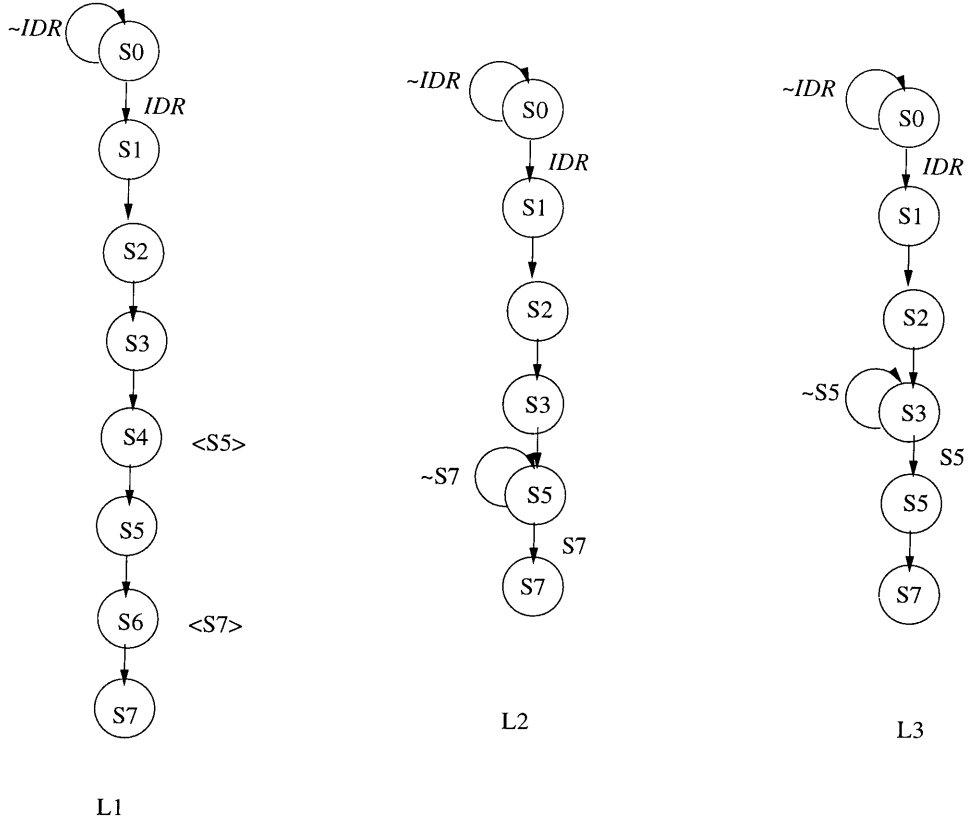


Figure 3.14: Single bit wakeup signal, case II.

we have at any time only one local controller which is asleep. When the master controller broadcast the NS , it will be for that sleeping controller. For example, in Figure 3.14, L3 sleeps in state $S3$ and wakes up at state $S5$. From state $S3$ until state $S4$ only L3 is a sleep. When L1 at state $S4$ broadcasts the single bit NS signal, L3 will wake up. From state $S5$ until state $S6$, L2 will be sleeping. The master controller will wake it up when NS goes high at state $S6$.

In some situations, not all sleeping local controllers need to be waken up at the same time. Their sleeping periods may overlap. A sleeping local controller must know for which next state a wakeup signal is generated. For example, in Figure 3.15, L2 goes to sleep in state $S2$ and L3 goes to sleep in $S3$. L2

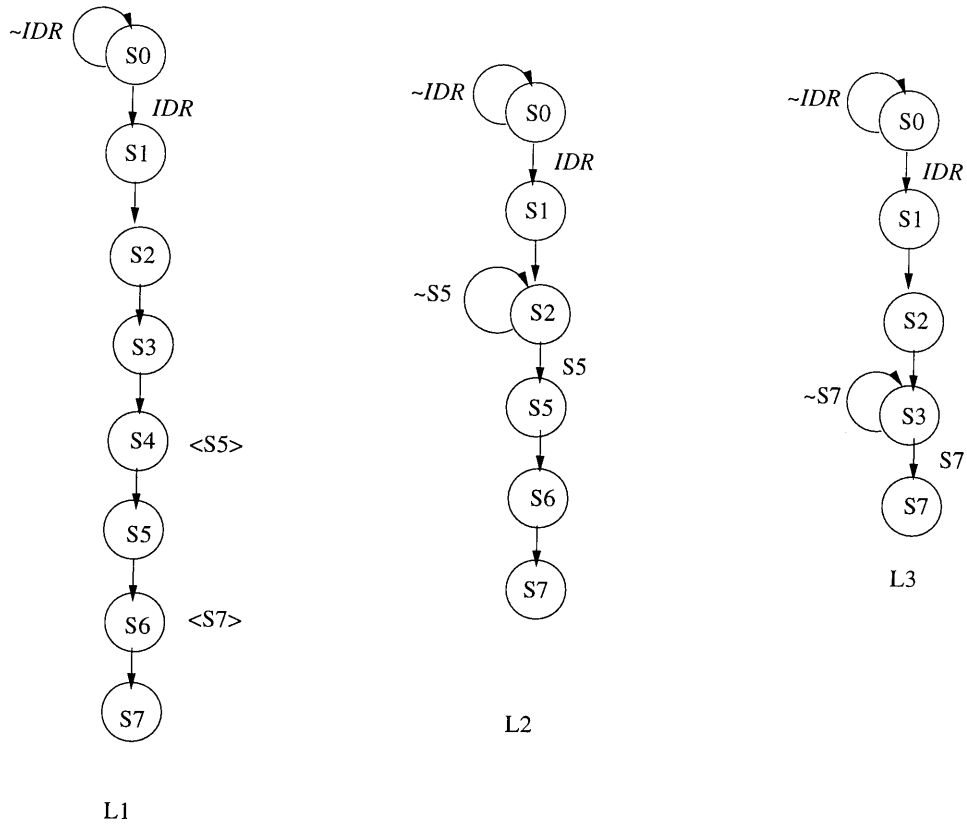
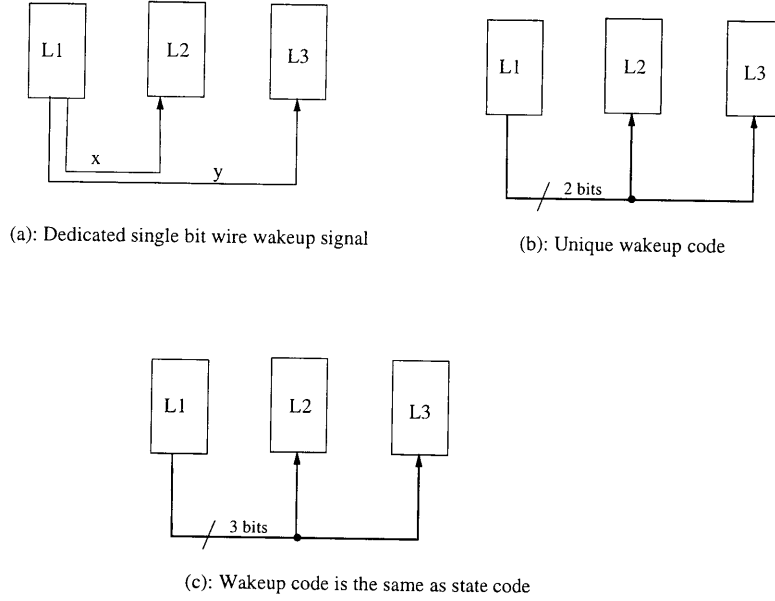


Figure 3.15: CLC design where NS signal requires more than one bit.

sleeps from state $S2$ until $S4$ and $L3$ sleeps from state $S3$ until $S6$. $L2$ and $L3$ sleep period overlap, at state $S4$ both controllers, $L2$ and $L3$ are asleep. $L2$ needs to wake up in state $S5$, but $L3$ need to wake up in state $S7$. $L1$, the master local controller, generates both NS signals. However, $L2$ and $L3$ need to distinguish between the NS signal for state $S5$ and the NS signal for state $S7$, so that each controller wakes up at the correct time.

One solution is to assign for every controller that goes to sleep its own dedicated wakeup wire. The master controller is connected to all of the bits of the communication bus and every local controller is connected to just a single wire. The master controller generates many NS signals. Each signal is one bit wide and is for a specific local controller. When a controller receives a

Figure 3.16: NS bus width.

wakeup signal it knows it must wake up. The communication bus width for this solution is n bits, where n is the number of local controllers with at least one sleeping state. Figure 3.16(a) shows the local controller connections for the previous CLC design shown in Figure 3.15. The x wire connects physically L1 to L2 and the y wire connects L1 to L3. The master controller L1 broadcasts $S5$ through x only to L2. Also, it broadcasts through y only to L3.

Another solution is to assign each state a unique code (key) and broadcast the next state code to all local controllers. If s is the number of unique sleeping states among all of the local controllers, the communication bus width is $\log_2 s$. If the bus width is less than $\log_2 k$ where k is the total number of states in the centralized controller, the wakeup signal code cannot be the same as the state code for the FSM. Figure 3.16(b) shows the local controllers connections with unique NS code. Two wakeup signals are needed, one for $S5$ and one for $S7$. Only two bits are needed to distinguish between $S5$ and $S7$. We can say that

code “00” represents no signal on the bus. The NS code “10” is a wakeup signal for $S5$, even though the state code is “101”. the wakeup code “01” is a wakeup signal for state $S7$, even though the state code is “111”. The FSM state codes for these states are three bits wide because the FSM has eight states. Since the NS wakeup signal is different than the state code, a new combinational logic circuit is needed to generate these NS signals.

The third solution is to let the state code, be itself the wakeup signal code. This is the solution we used. The master controller, broadcasts the NS signal in every state. So, there is no need for new logic circuit to generate the NS signal, the output of next state logic which generate the FSM next state is used as the code for the NS signal. Figure 3.16(c) shows the connections between previous local controllers when the NS code is the same as the state code.

3.5.3 Peer Controllers

Instead of designating one local controller to be the master controller, a peer controller design can be used. In this design all local controllers are treated equally and all non-active states are eliminated for all local controllers. As a result, we may need more than one local controller to provide the NS signals. NS signals are generated only when they are needed, so if all local controllers are active at a specific state, no NS signal is generated for that state. A local controller can generate the NS signal if its next active state is in the alarm set of any sleeping local controller. For example, in figure 3.17 the controller L1 needs to wake up at state $S4$ and $S6$, so its alarm set is $\{S4, S6\}$. Controller L2 can provide the NS signal for state $S4$, because $S4$ is an next active state after state $S3$. However, state $S6$ is not an active next state for controller L2.

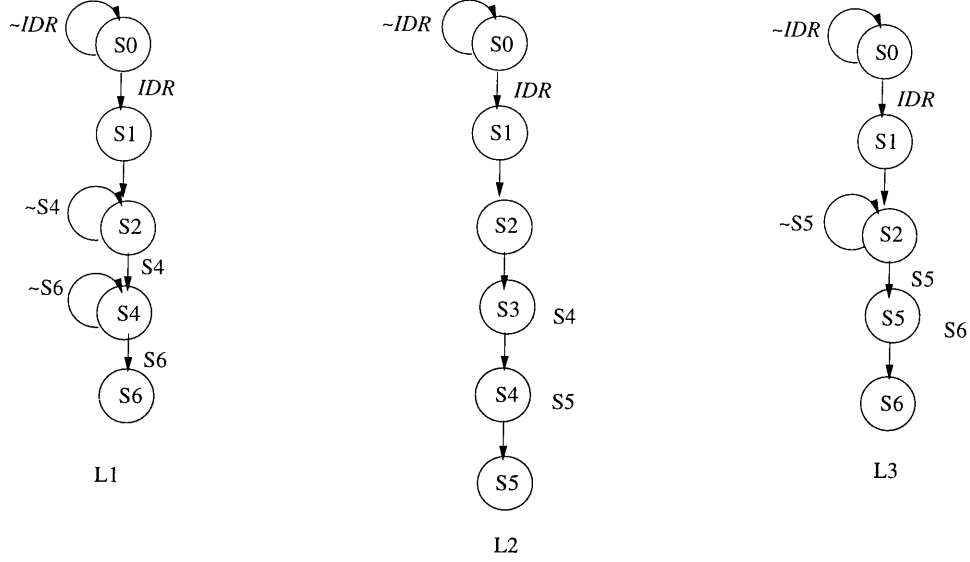


Figure 3.17: Peer local controllers.

Controller L3, on the other hand, can provide the NS signal for state $S6$ but not for state $S4$. Moreover, L2 can provide the NS signal for $S5$ for L3. If more than one local controller can generate the NS signal, only one of these controller is used to generate this signal.

Peer controller design has a deadlock problem. A deadlock occurs when all local controllers are sleeping and each waiting for the other to wake it up. To illustrate this problem we use the example shown in Figure 3.18. L1 sleeps in state $S2$ and its alarm set is $\{S6\}$. L2 sleeps in state $S5$ or $S4$ depending on the condition evaluation, and its alarm set is $\{S7\}$. Only L1 can generate NS for state $S7$, but it is sleeping. L2 never goes to state $S6$. so it can not wake up L1. To void this deadlock problem, the peer controller design is not used. Instead, one local controller is designated as the master local controller.

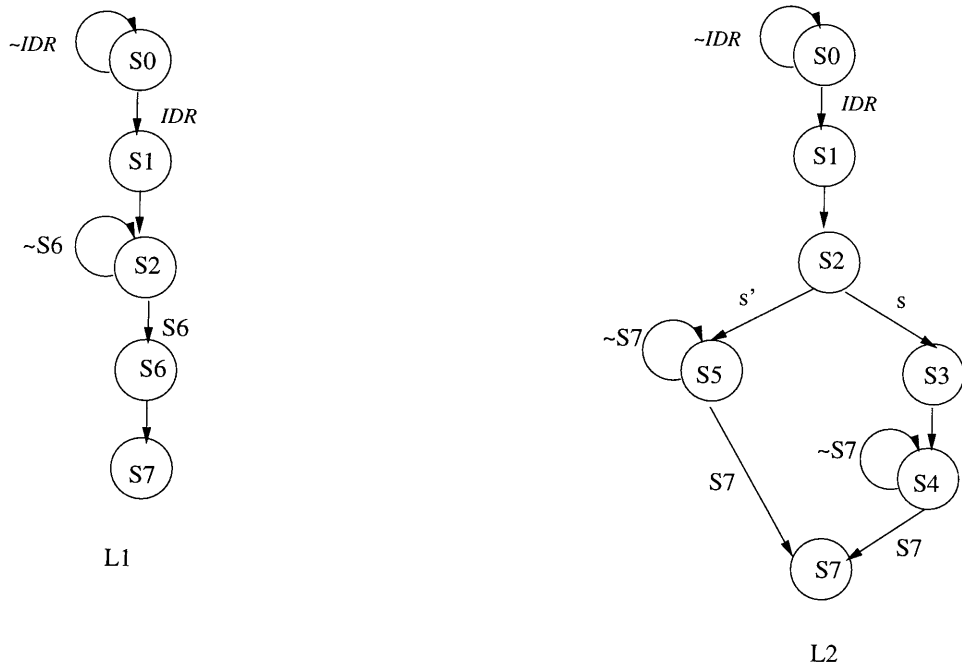


Figure 3.18: Peer local controller deadlock.

Chapter 4

Experimental Results

We present how the VHDL description of a centralized controller circuit, NCLC circuit and CLC circuit are generated using our Transformation Tool. We also, present how COMPASS [49] is used to synthesize the VHDL codes, generate hierarchical layout and analyze the three designs.

Eight behaviors are implemented and analyzed. These behaviors differ in the number of hardware components, the number and type of ALUs, the size of the MUXes, the number of time steps, the conditional structure, assignment operations, and shared components. Three behaviors will be discussed in detail: Discrete Cosine Transform (DCT), the Postage stamp calculation behavior (POST), and the floating-point addition.

The next section begins with a description procedure for generating and designing distributed controller.

4.1 Distributed Controllers Generation and implementation

4.1.1 Software Structure

Our RTL transformation software contains the datapath partitioning algorithm and the distributed controller design algorithms. It is implemented in C on a SUN SPARC-5 workstation. It needs for input an RTL circuit description

in term of its operations schedule, register allocation, and control flow.

For our experimental purposes, a circuit behavior is first scheduled and resource are allocated for it by a high level synthesis tool such as SYNTTEST which is developed at CWRU [48] ¹. The behavior is written in the `*.dfg` file in SYNTTEST format and the circuit resource and delay constraints are written in the `*.cons` file . The `*.dfg` file contains the control flow structure. These two files are the input to SYNTTEST which performs scheduling and allocation that satisfy delay and resource constraints. SYNTTEST generates many files , from them is the `*.rtl` file, which contains scheduling and allocation information and the `*.attribute`, which contains register allocation information. The three files: `*.dfg`, `*.rtl` and `*.attribute` are used as input for our transformation Tool software, which produces both distributed controller designs.

The chart in Figure 4.1 shows an overview of our distributed controller design and implementation. Our Transformation tool produces VHDL codes for NCLC circuit and CLC circuit. For our experimental purposes, VHDL code for the centralized controller circuit is, also, produced. For the centralized controller design, three files of VHDL code are generated: datapath RTL description, controller FSM description, and the whole circuit description in term of how the datapath and the controller entities are connected.

For both distributed controller designs, $3 * n + 1$ files of VHDL code are generated, where n is the number of functional blocks. There is a VHDL code file for every functional block and every local controller, assuming every block needs a controller. Every functional block entity is connected with its

¹Other high level synthesis tools can be used.

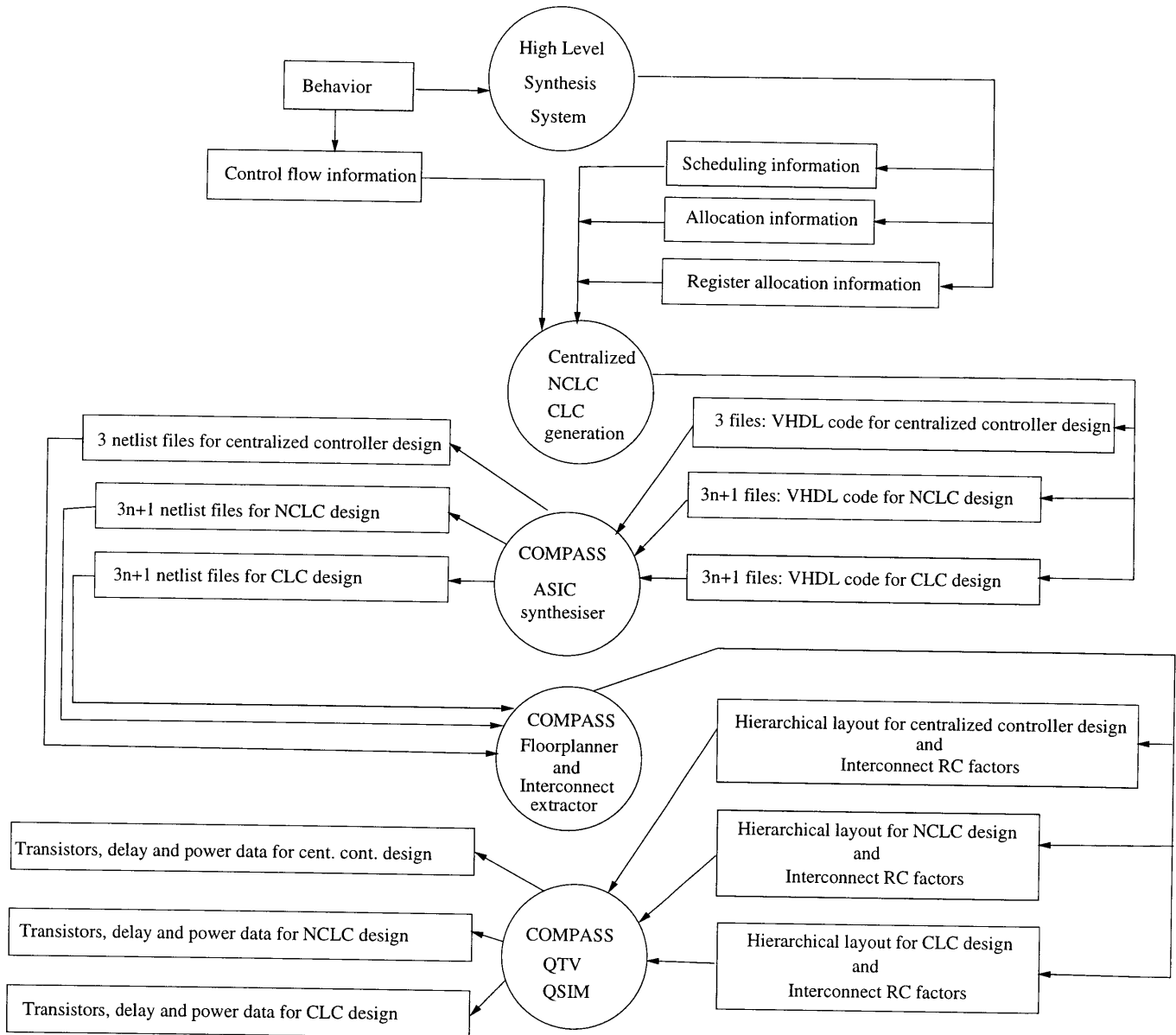


Figure 4.1: Distributed controller design and implementation.

local controller entity in a unit VHDL code. file. One VHDL code describes the entire distributed controller design, in term of how units are connected. The files for the CLC distributed controller design are similar to the files for the NCLC distributed controller design, except the local controllers FSM descriptions are different.

All three designs are synthesized to netlists using the COMPASS ASIC synthesizer which generate the same number of netlist files (*.nls) as the VHDL code files.

4.1.2 Floor planning:

The COMPASS floor planner is used to generated Hierarchical layout. A hierarchical layout scheme is used. The chart in Figure 4.2 shows the layout procedure.

A layout is made of each netlist (*.nls), which corresponds to one VHDL code file. the floor planner generate a layout (*.ly) for for all intermediate netlist structure (i.e centralized controller netlist). COMPASS MAKEPHANTOM utility is used to generate a macro cell (*.mcp) for each layout (*.ly). Macro cell is a digital logic circuit which can be treated as a cell in a hierarchical complex design. using the macro cells as building blocks, the entire circuit layout was done by placing these macro cells and performing block routing among them. The floor planner is used in the second and third level to generate a hierarchical layout (*.cp). A two-level hierarchical layout was used in the centralized controller design as shown in Figure 4.3. 1) At the first level, two layouts and macro cells are generated for the datapath and the centralized controller. 2) The datapath and the centralized controller macro cells

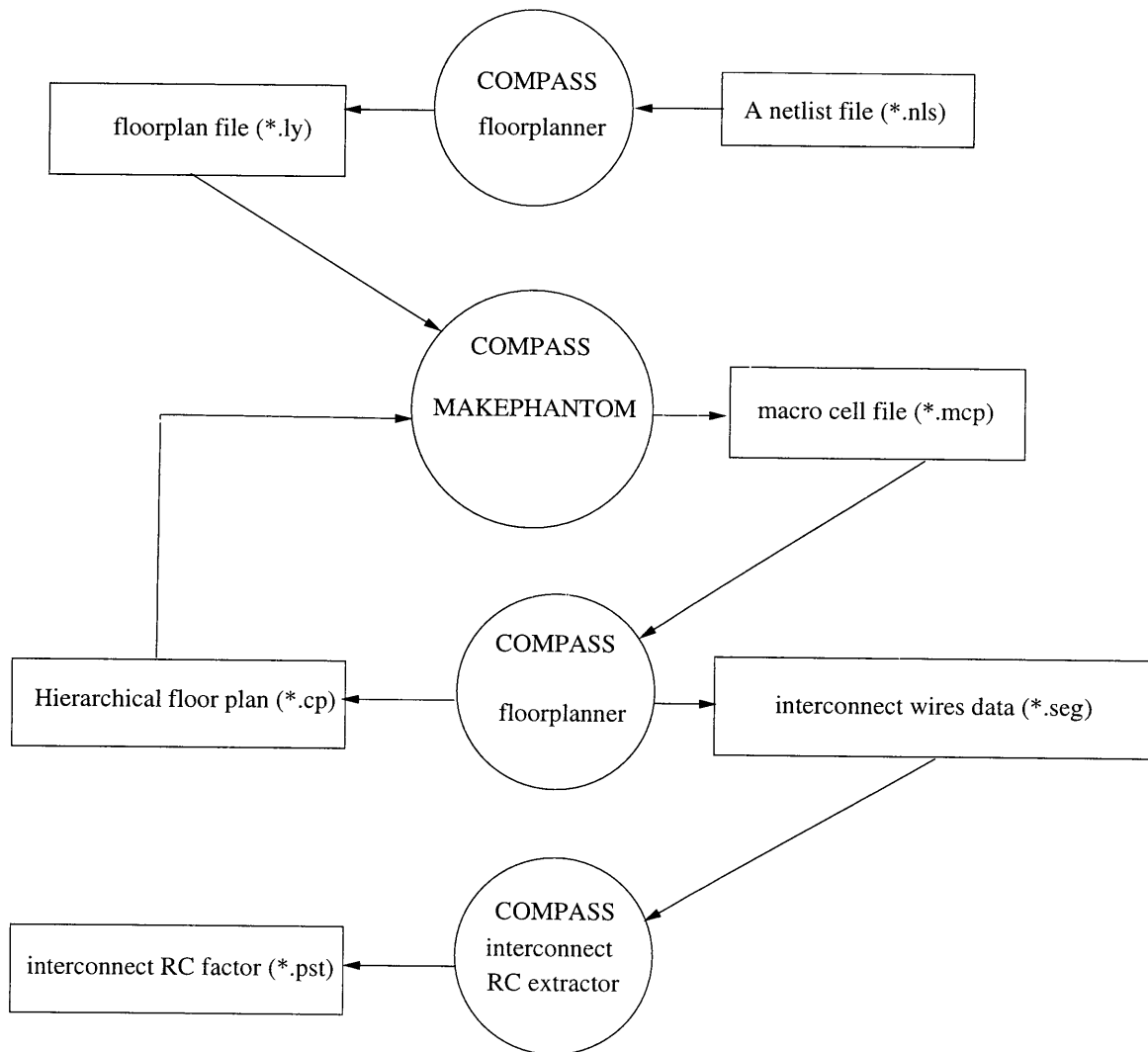


Figure 4.2: circuit hierarchical layout procedure.

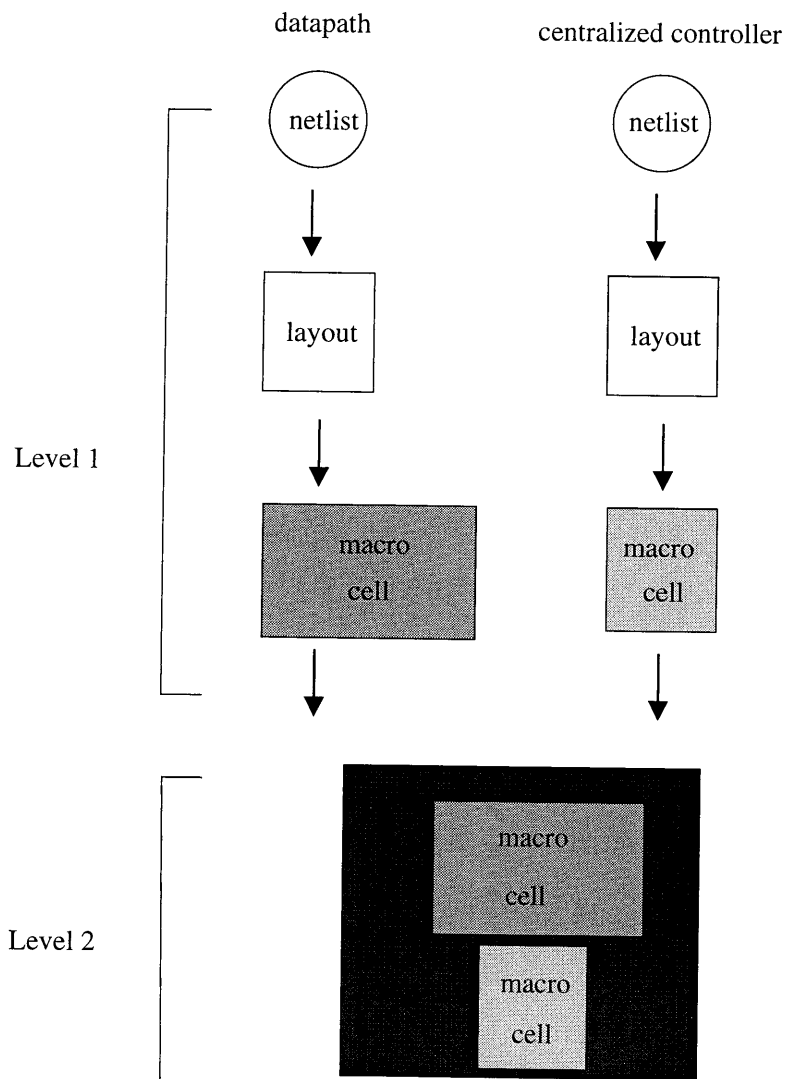


Figure 4.3: Two-level centralized controller design layout.

are placed and block routed to create the entire circuit. On the other hand, a three-level hierarchical layout was used in each distributed controller design as shown in Figure 4.4. 1) Each functional block and each local controller are placed and routed separately. Macro cells were generated for each functional block, and each local controller layout. 2) Each functional block macro cell and its local controller macro cell were placed and block routed together. The layout of each pair of functional block and local controller is called a unit layout. Unit macro cells were generated using MAKEPHANTOM utility. 3) The entire circuit was generated by placing all unit macro cells and block then routing them together.

4.1.3 Delay and Power

Delay and power are two major issues for designing circuit in sub-micron technology. Due to the small devices size at sub-micron level, gate delay become negligible, and wiring delay become dominant. Thus reducing the wire length is essential. Generally, circuit fabricated at sub-micron level operate at high frequencies, so dissipating more power, so reducing the transistors switching activities is, also, essential.

In the floor planner, interconnect wire lengths are saved from each final layout in a (*.sig) file. COMPASS “interconnect” was used to extract the interconnect wires RC factors in (*.pst) file for timing analysis. Control wires are among the interconnect wires. In the centralized controller design, the control wires interconnect the datapath macro cell and the centralized controller macro cell. In the distributed controller design, the control wires connect each pair of a local controller and a functional block. Thus the control

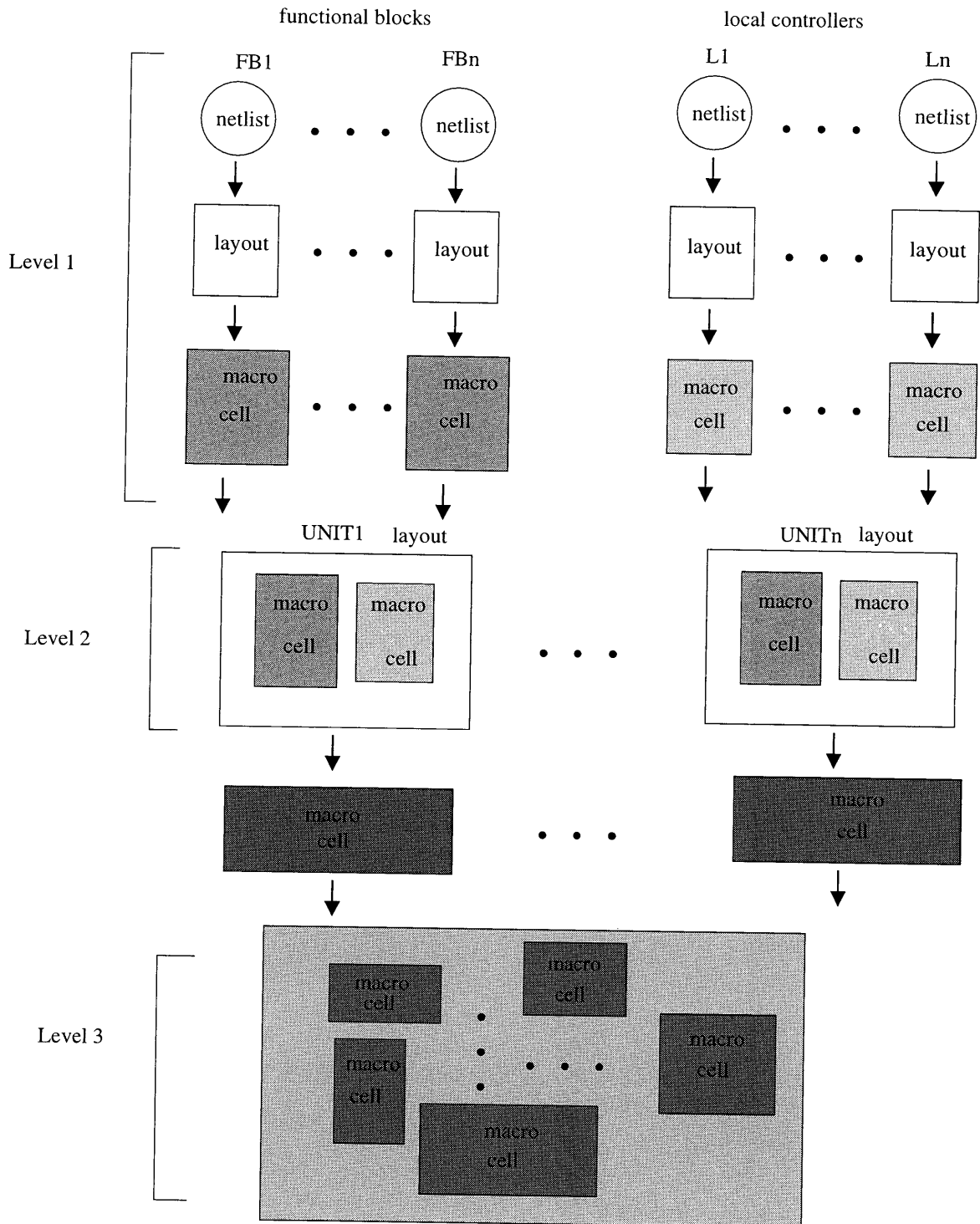


Figure 4.4: Three-level distributed controller design layout.

wires RC factors need to be extracted and their delays be measured.

COMPASS QTV is used to compute the critical path delay at for the three designs. Both the whole design (*.nls) and (*.pst) files are inputted to the QTV utility. The (*.pst) file contains the interconnect wiring RC factors, thus the wiring delay is included in the critical path delay calculation. The critical path delay consists of three components: delay through the controller (centralized or local), interconnect delay through the wire and delay through the datapath (functional block). Information on wire delay calculation see appendix C.

Since our design is implemented using $\lambda = 0.6$, the performance of the distributed controller design at sub-micron level $\lambda < 0.5$ is estimated. At sub-micron level [7] and [8] show that gate delay is negligible and interconenct delay is dominant. In [3] is shown that scaling down the gate size by α , scales down the gate delay by α^2 . Also, scaling down the wire size by α , increase the wire resistance R by α and reduces the wire capacitance C by α . So there is no effect of scaling down the wire size on RC value. However, the delay depends on both RC value and the wire length l

$$\text{wire delay} = 0.7RC \frac{l^2}{2}.$$

Scaling down the wire size my reduce its length by small factor. However, circuits which are designed at sub-micron may have long wires because die area stay the same and wire length depends devices location not size, there may be a wire from on corner of the die area to the opposite corner. Therefore, any small change in the wire length will have big effect on the delay because the delay is proportional with the square of the wire length.

The following example is to illustrate gate delay and wire delay ratio in

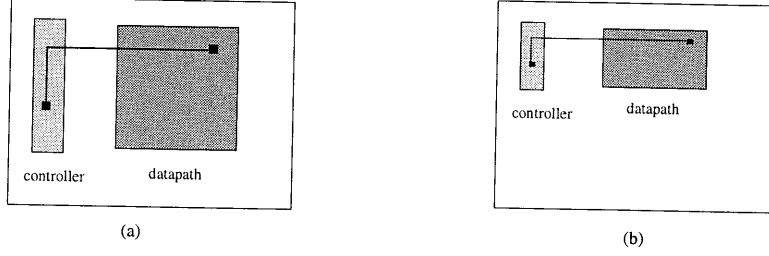
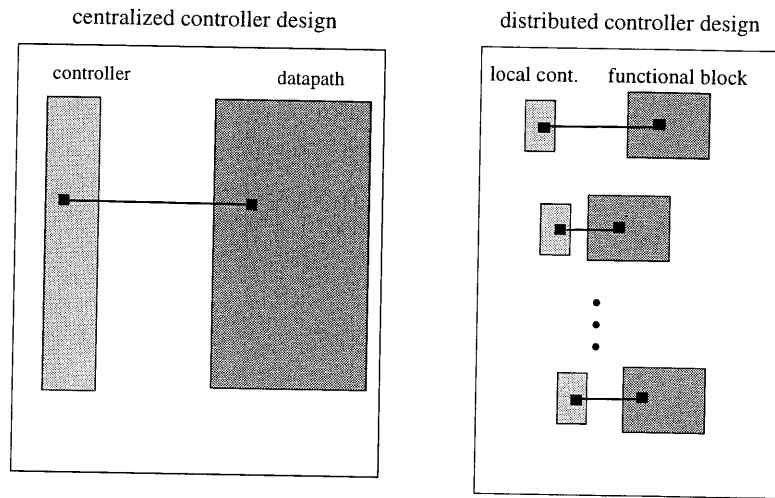


Figure 4.5: Wire delay is dominant at sub-micon level.

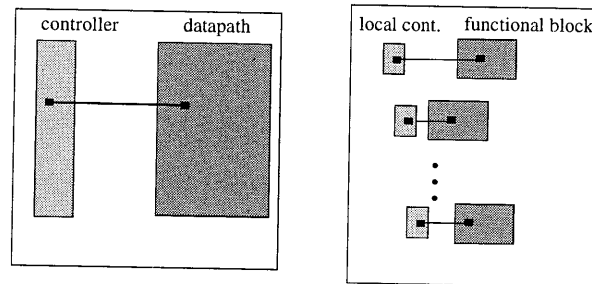
a delay at sub-micron level. Suppose that the critical path delay is between the controller and the datapath as shown in Figure 4.5(a) and the critical path delay is $16ns$ from which $5n$ delay through the controller logic, $1ns$ delay through the interconnect wire, and $10ns$ delay through the datapath logic. If all gates and interconnect wires are scaled down by $\alpha = 7$, the critical path delay will be $1.3ns$ from which $5/49 = 0.1$ delay through the controller logic, $1ns$ delay through the interconnect wire, and $10/49 = 0.2$ delay through the datapath logic. The wire delay will stay within $1ns$ assuming that the location of the devices that are connected to the wire terminals stay the same (see Figure 4.5). So Implementing a circuit in sub-micron technology, makes the gate delay negligible and the wire delay dominant.

The layout topology of a circuit implemented using $\lambda < 0.5$ is unknown to us, but our hierarchical layout design style gurantee that distributed controller design will have shorter control wires compared with a centralized controller design. The control wire length ratio $\frac{w_c}{w_l} > 1$, where w_c is the control wires between a datapath and the centralized controller design and w_l is the control wires between a local controller and a functional block. This relation is valid even when the circuit is implemented in sub-micron technology (see Figure 4.6).

The average power estimation is accomplished by simulating the circuit in



(a): circuits are designed in micron technology



(b): circuits are designed in sub-micron technology

Figure 4.6: the control wire ratio relation $w_c/w_l > 1$ remain valid at sub-micron level.

the COMPASS simulator QSIM (with the power option enabled). The power estimation works by counting the transitions on each node and computing an average frequency f over a given amount of time T , using the formula $P = fC_L V^2$ with $V = 4.65\text{V}$, $f = 50\text{ MHz}$ and C_L is the loading capacitance on the particular node. The power tool finds the power for each node. Then summing over all nodes in the circuit after time T , number of cycles needed for a particular behavior, the simulator reports the overall average power consumed by the circuit.

The average power is computed by simulating the circuit with a large number of random inputs, and having the tool report the power after all patterns were executed. In other words, we allowed the simulator to find the average transitions on nodes over a long period of randomly inputted data. The random patterns were fed only to the data lines. As the number of patterns fed to the circuit increases, the average power reported by the simulator begins to converge. Even though the accuracy of average power consumption can vary according to input behavior and number of inputs, a 3000 patterns input set is accurate enough. This information is important, because checking all transitions on all inputs would require $(2^{\sum_{i=1}^n w_i})!$ patterns, where n is the number of inputs and w_i is the width of input i . However, any large set of random patterns will converge to the same result.

4.2 Cases Studies

In this section three behaviors : DCT, POST and FPADDs will be discussed in detailed. All the data are placed in a table at the end of this section.

4.2.1 Case study I

The Discrete Cosine Transform (DCT) is a data dominated with no conditions in its behavior and it has 47 operations. It has nine inputs and nine outputs. It has the longest schedule which is 31 time steps. Also, it has the biggest MUXes (i.e. a 19-to-1 MUX). Its scheduled data flow graph is shown in Figure 4.7. The files `cosine.rtl` and `cosine.attribute`, generated by SYNTEST, as well as the SYNTEST input file `cosine.dfg`, are the input to our transformation Tool. Our transformation tool then generated VHDL codes for the centralized controller design, the NCLC design and the CLC design.

The datapath is shown in Figure 4.8. It has 3 ALUs, 5 MUXes, and 23 registers and it is partitioned into three functional blocks. One block contains the adder, another block contains the subtractor and the third block contains the multiplier. This partition is trivial because there are no shared components between ALUs.

The multiplier functional block includes one MUX and seven output registers: $FB1 = \{m1, *, r1, r2, r3, r4, r5, r6, r7\}$. MUX $m1$ is an 11-to-1 MUX and has four bits select line. The adder functional block has two big MUXes and 11 registers: $FB2 = \{m2, m3, +, r8, r9, r10, r11, r12, r13, r14, r15, r16, r17, r18\}$. both MUXes $m2$ and $m3$ are a 19-to-1 MUX and have five-bit wide select signal. The subtractor functional block contains two MUXes and five registers: $FB3 = \{m4, m5, -, r19, r20, r21, r22, r23\}$ Both MUXes $m4$ and $m5$ are 7-to-1 MUXes which needs three-bit wide select signal.

The DCT centralized controller is shown in figure 4.9. The controller does not have any input because this behavior has no conditions. Thus, the controller is a sequencer, at every clock cycle an unconditional transition oc-

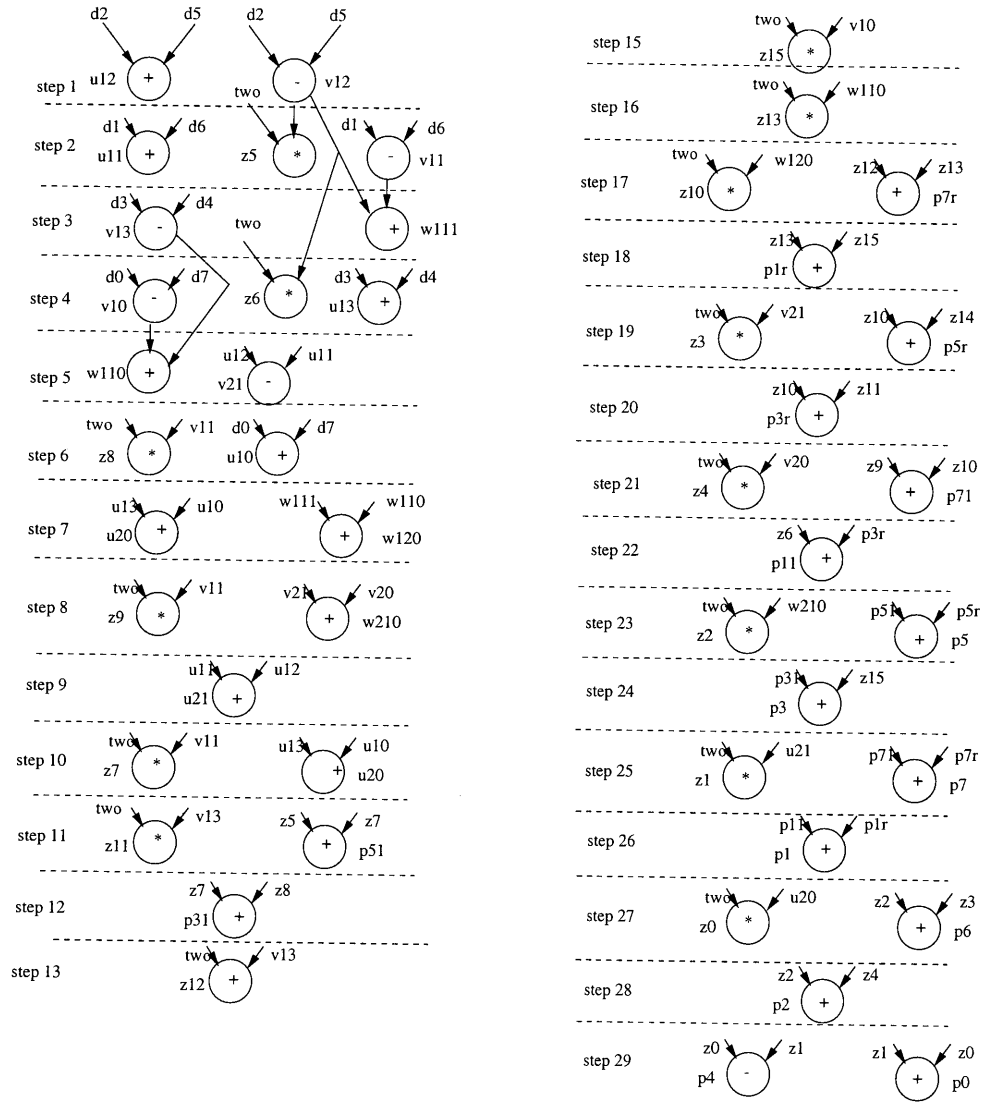


Figure 4.7: DCT scheduled DFG.

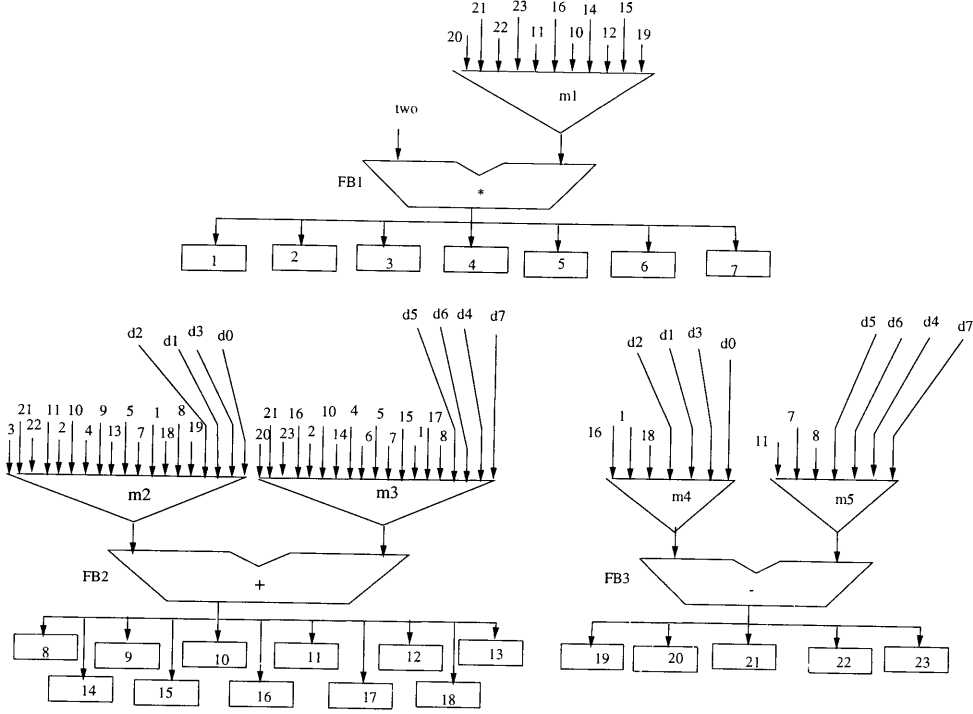


Figure 4.8: DCT functional blocks.

curs to the next state. The controller output bus width is 43 bits from which 23 bits are register load signals and 20 bits are MUX select signals. The centralized controller output control vector is enable MUXes five MUXes $\langle m1, m2, m3, m4, m5 \rangle$ and 23 registers from $r1$ to $r23$.

The NCLC distributed controller consists of three Non-communicating local controller. Each controller has the same FSM as the centralized controller with different output. The NCLC distributed controller is generated by duplicating the centralized controller FSM into three local controllers. The output from the centralized controller is partitioned into field of output control vector for each functional block. The multiplier local controller output control vector is $\langle e(m1), e(r1), e(r2), e(r3), e(r4), e(r5), e(r6), e(r7) \rangle$. The adder local controller output control vector is $\langle e(m2), e(m3), e(r8), e(r9), e(r10), e(r11),$

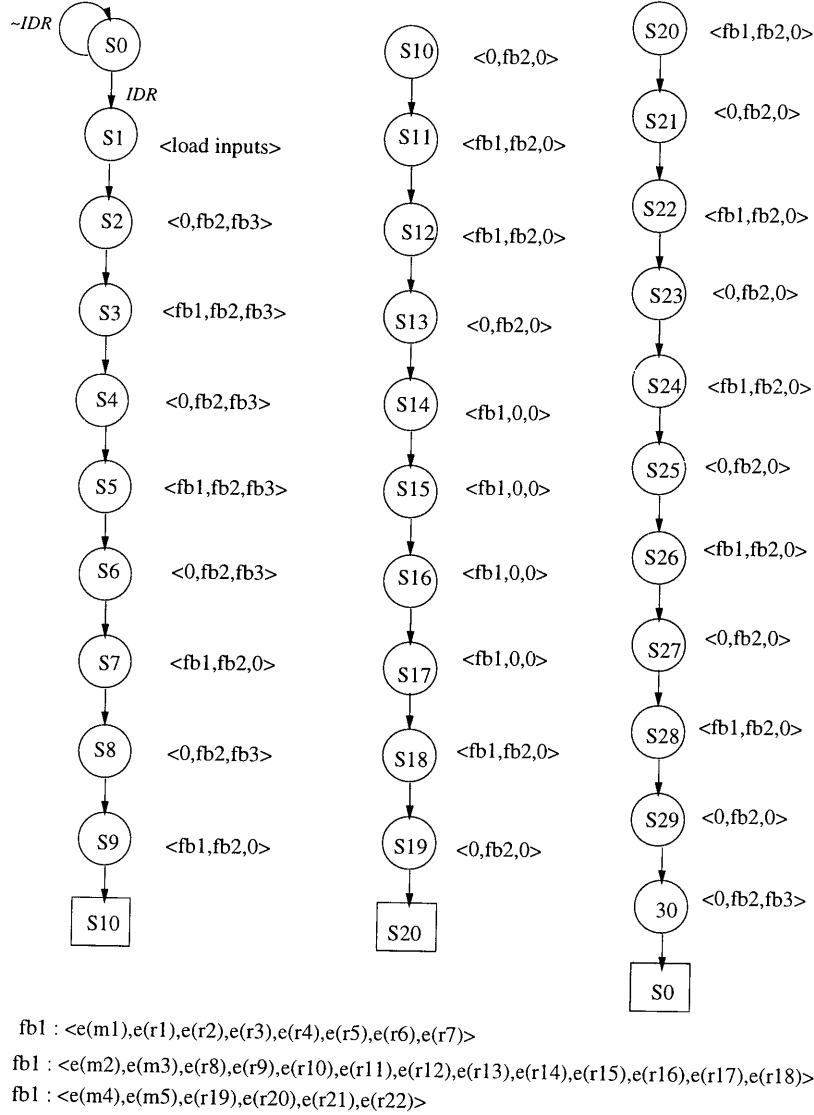


Figure 4.9: DCT centralized controller.

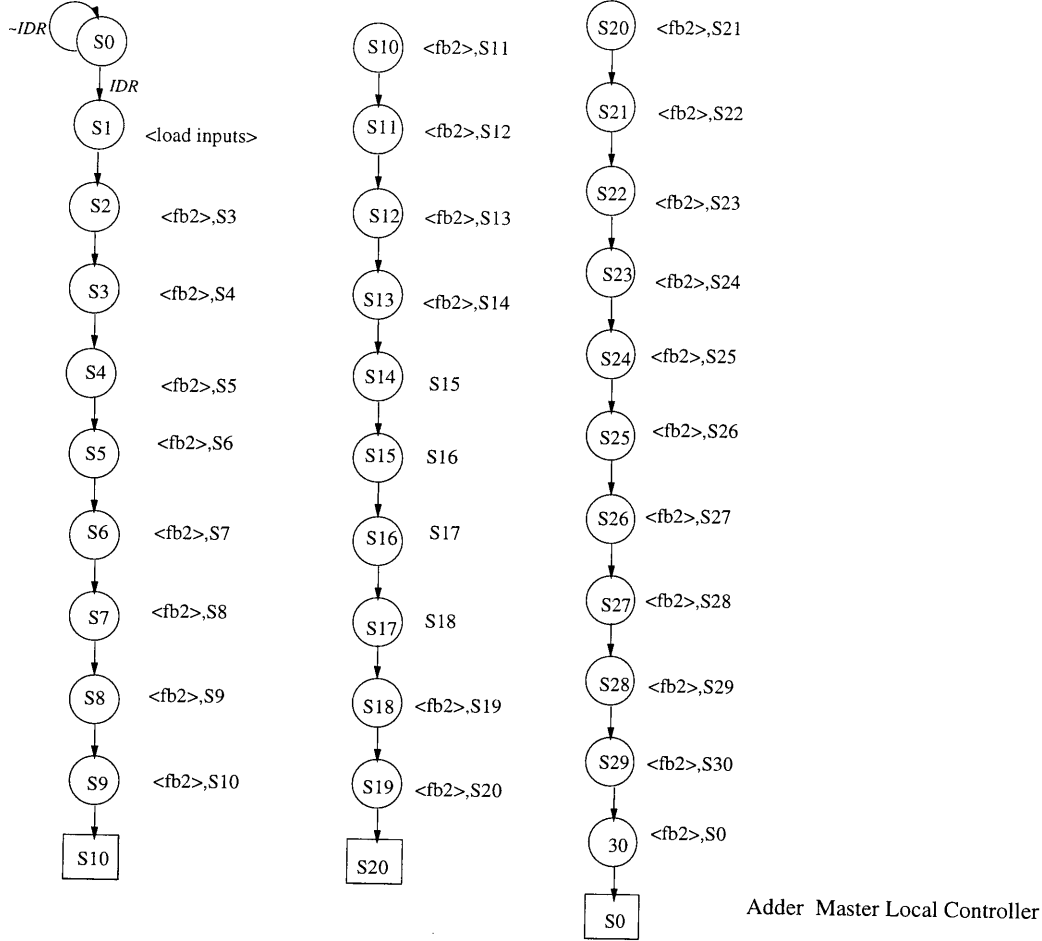


Figure 4.10: DCT CLC master controller.

$e(r12), e(r13), e(r14), e(r15), e(r16), e(r17), e(r18)>$. The subtractor output control vector is $<e(m4), e(m5), e(r19), e(r20), e(r21), e(r22), e(r23)>$.

For the CLC design, the adder is chosen to be the master local controller because it executes 25 operations where the multiplier executes only 16 operations and the subtractor executes only seven operations. The adder local controller is shown in Figure 4.10.

For the multiplier and the subtractor and NCLC local controller are generated, then all idle states (no operation is executed) are removed. The disconnected FSM is reconnected using the new transition function. Reconnection

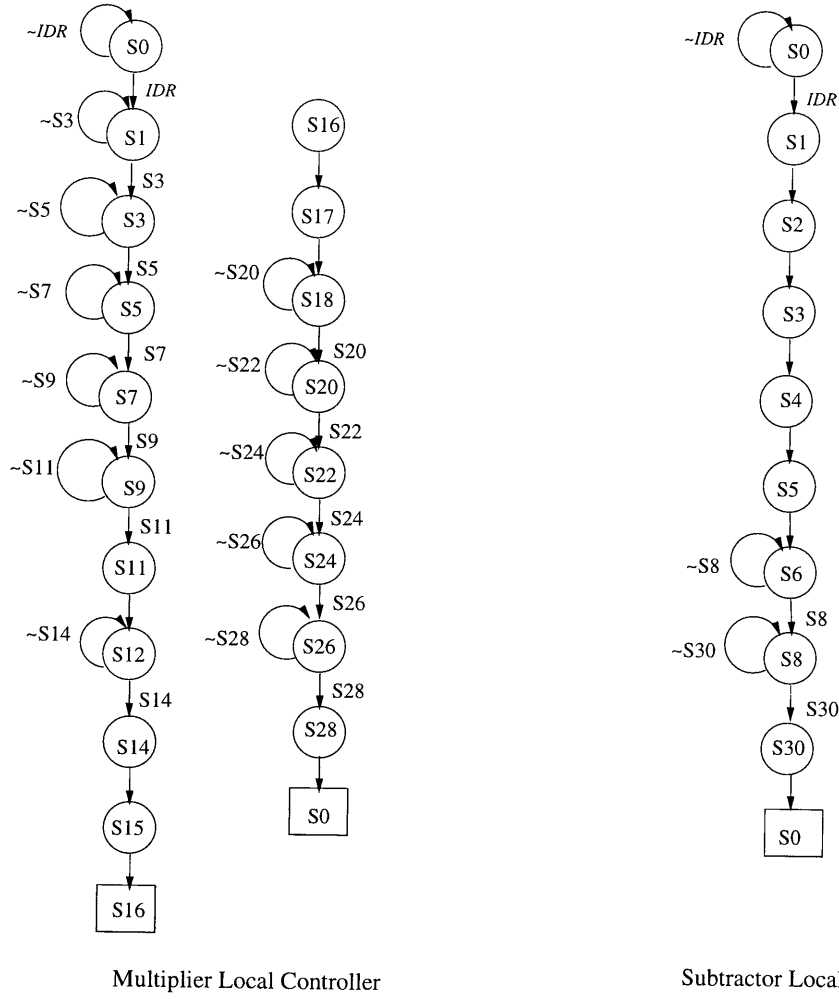


Figure 4.11: DCT CLC local controllers.

between the remaining states is done conditionally. A controller will self-loop in a sleeping state until the NS signal which is supplied by the master controller, equals the next active state for the controller. The multiplier and the subtractor local controllers are shown in Figure 4.11. For example, state $S8$ of the subtractor local controller is a sleeping state. It will self-loop until the NS signal from the adder local controller is $S30$.

Our transformation tool generated 23 VHDL code files. Three are for the centralized controller DCT circuit, 10 for the NCLC circuit and 10 for CLC

circuit. The VHDL codes for the three designs are inputted to the COMPASS ASIC synthesizer to generate the 23 netlists.

A hierarchical layouts for the three circuits are generated by COMPASS floor planner. The wiring RC factors are extracted. Then, the critical path delay, the average power consumption and the number of transistors are computed for each circuit design.

Five charts are produced from the designs data. The critical path delay is shown in chart 3.1. The centralized controller circuit has longer delay than both distributed controller circuits. The critical path delay for the centralized controller is 41.93 ns . However, the critical path delay for NCLC and CLC circuits are 39.39 ns and 38.72 ns respectively. The centralized controller critical path is $controller \rightarrow m1 \rightarrow ALU1 \rightarrow r1$. $ALU1$ is the multiplier ALU (see Figure 4.8). The critical path for both distributed controller circuits is $FB1.controller \rightarrow FB1.m1 \rightarrow FB1.ALU1 \rightarrow FB1.r1$. The critical path delay has three components: delay through the controller, control wire delay and delay through the datapath. Chart 3.2 shows the delay for each component. The delay reduction is due to shorter control wires between the multiplier local controller and functional block. Moreover, there are fewer transistors in the multiplier local controller output logic than the centralized controller output logic, which contributes to reducing the controller delay.

At sub-micron level, the gate delay is negligible, but wire delay is dominant. Since, the wire RC factor is not effected by scaling down the wire size, the wire length is the main factor which contribute to the delay of a circuit implemented in sub-micron technology. As mentioned earlier, the critical path delay has three components. To have an indication of how effective is the dis-

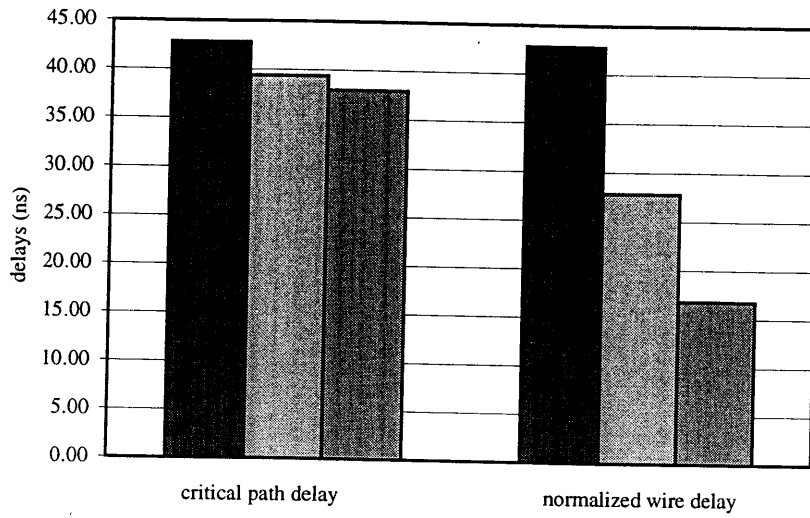


Chart 3.1: DCT Critical path and wire delays.

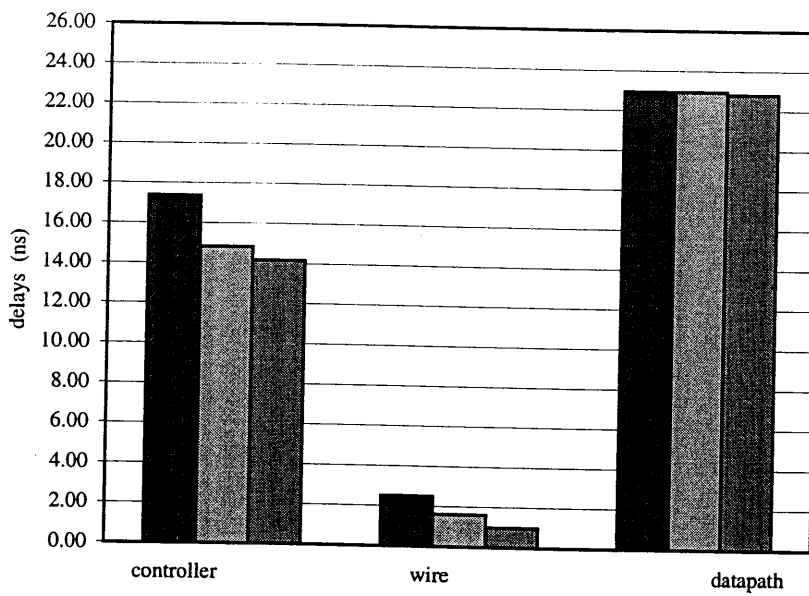
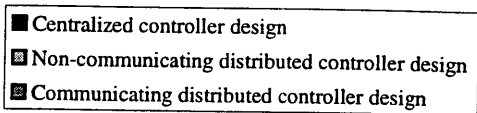


Chart 3.2: DCT critical path delay factors.



tributed controller design compared with centralized controller design if the circuit are implemented in sun-micron technolog, the delay through the controller and the datapath is ingore (these delays are made mainly of gate delays) and only wire delay is used. The wire delay is shown in chart 3.1 beside the critical path delay. To compare the size of reduction the critical path delay and the wire delays, the wire delay is normalized.

$$w_n = w_d \text{ cpd}_c / w_c$$

where w_n is the normalized wire delay, w_d is the wire delay, cpd_c is the centralized controller design critical path delay, and w_c is the centralized controller design wire delay.

Distributed controller design style can result in better perfomance than the centralized controller design style which is essential for the circuit to operate at high frequencies.

The distributed controller design , also, consumes less power. The average power consumption 6524.74 uw , 5804.72 uw , and 5797.33 uw are for the centralized controller design, NCLC design and CLC design respectively as shown in chart 3.3. The operation execution is localized in the distributed controller design. For each register transfer there are less transistors switching activities in the distributed controller circuit than in the centralized controller circuit. Charts 3.4 and 3.5 shows number of transistors of each unit (functional block and its local controller), number of active time steps and average power consumption of each functional block. The results indicate that the power consumption is related to the unit number of transistors and activities.

The centralized controller design has 8843 transistors where NCLC design has 8173 transistors and CLC design has 8694 transistors as shown in chart

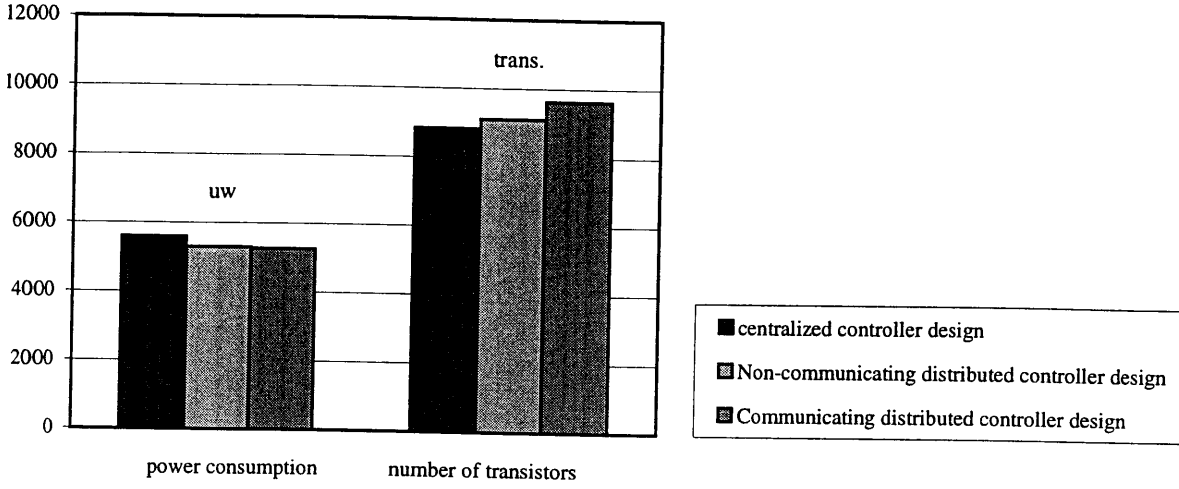


Chart 3.3: DCT power and transistors

3.3. Even though, distributed controller has more transistors than the centralized controller design, the distributed controller activities is localized. Thus, distributed controller design consume less power.

4.2.2 Case Study II

The second behavior, POST, calculates postage price based on package weight and distance from source to destination. The post behavior is control dominated and has five conditions, three of which are nested. It has 17 inputs and one output. It is scheduled in 16 time steps. The POST CDFG is shown in Figure 4.12 and each basic blocks is shown in Figure 4.13. Most basic blocks have are scheduled in a single time step.

The datapath is shown in Figure 4.14. It has 4 ALUs, 8 MUXes, and 9 registers and it is partitioned into four functional blocks. The subtractor functional block is $FB1 = \{m1, m2, -, r1, r2, r3\}$. Both MUX $m1$ and

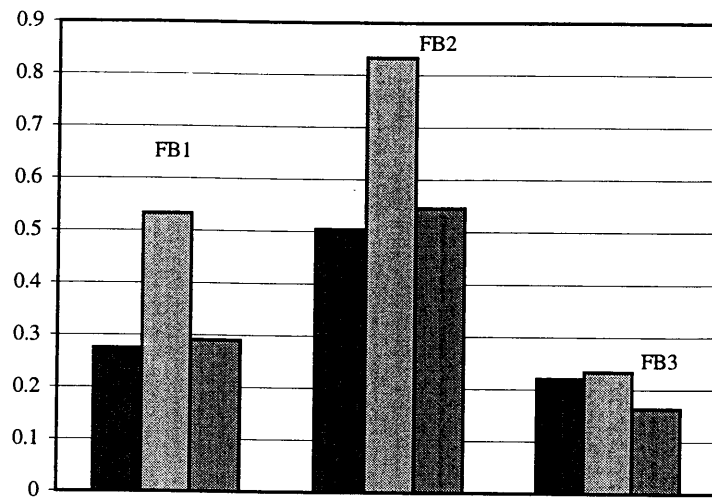


Chart 3.4: DCT NCLC functional blocks data.

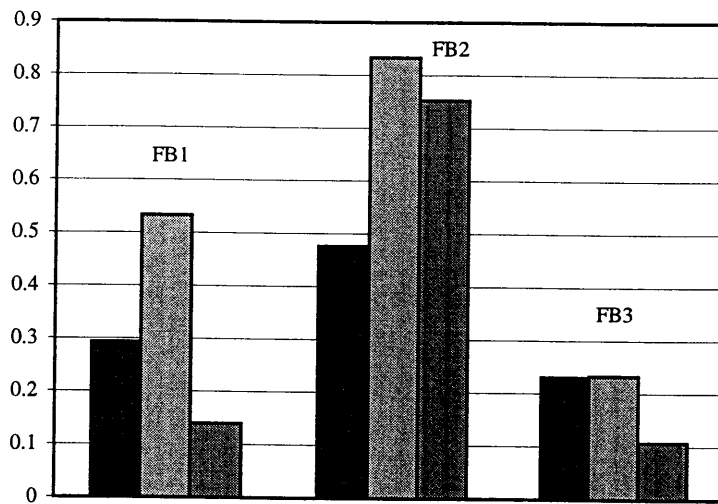
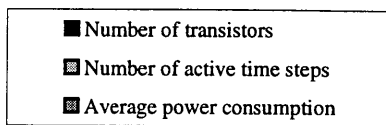


Chart 3.5: DCT CLC functional blocks data.



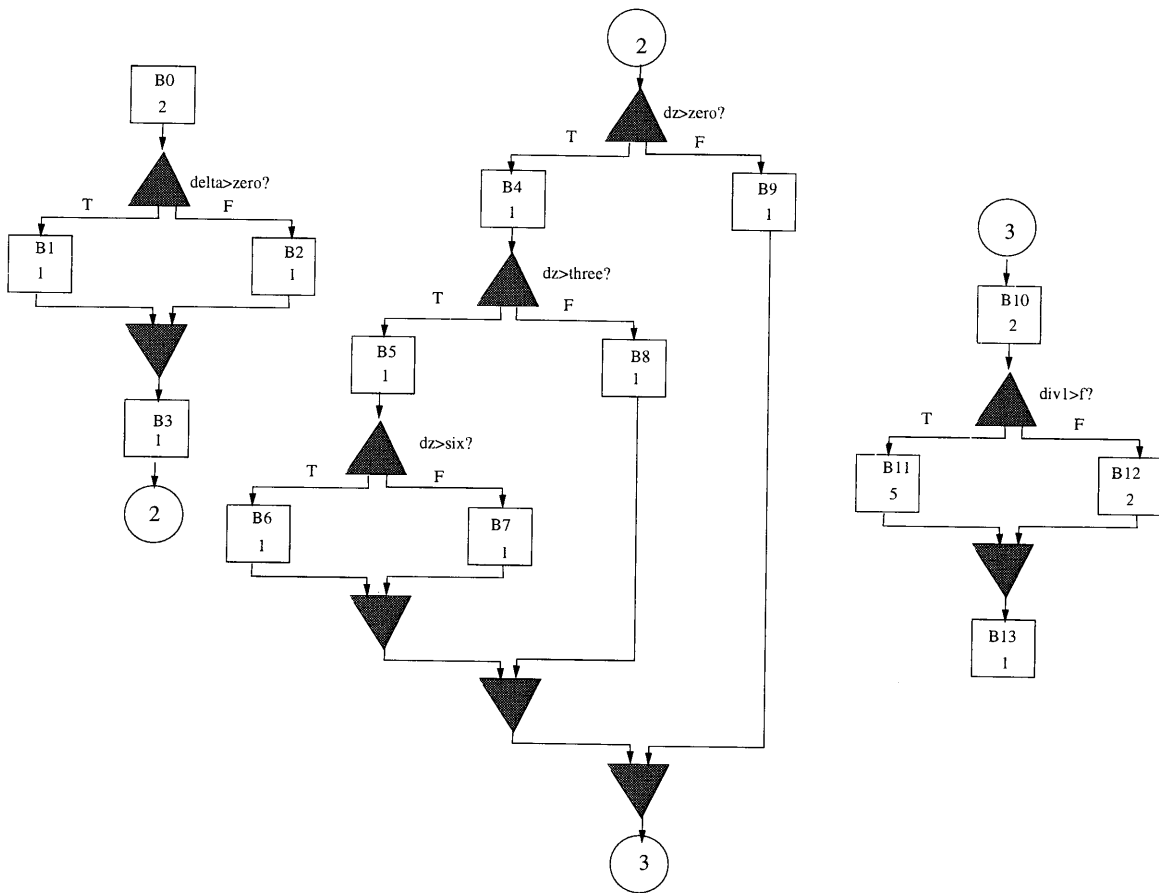


Figure 4.12: Scheduled CDFG for POST behavior.

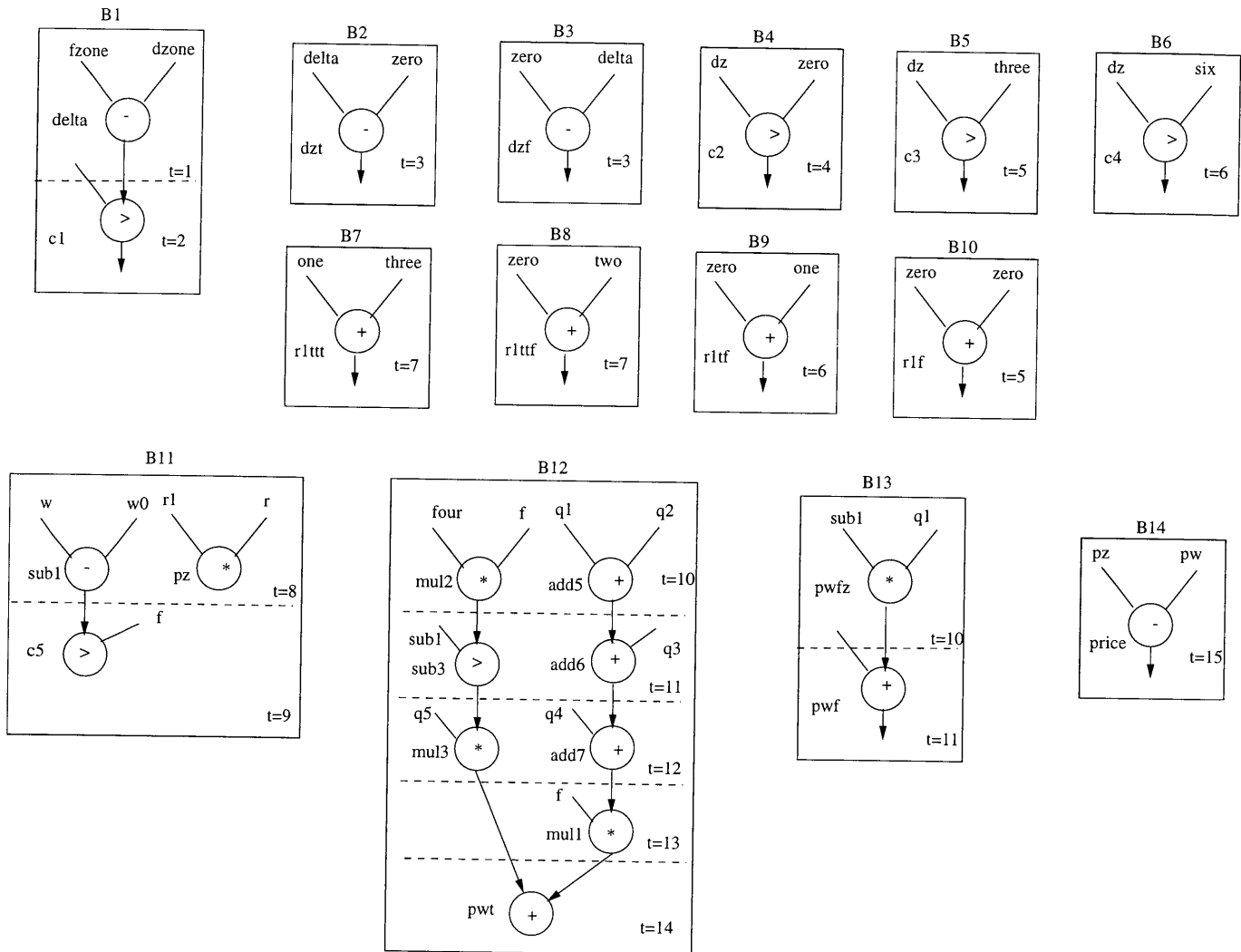


Figure 4.13: postage basic blocks.

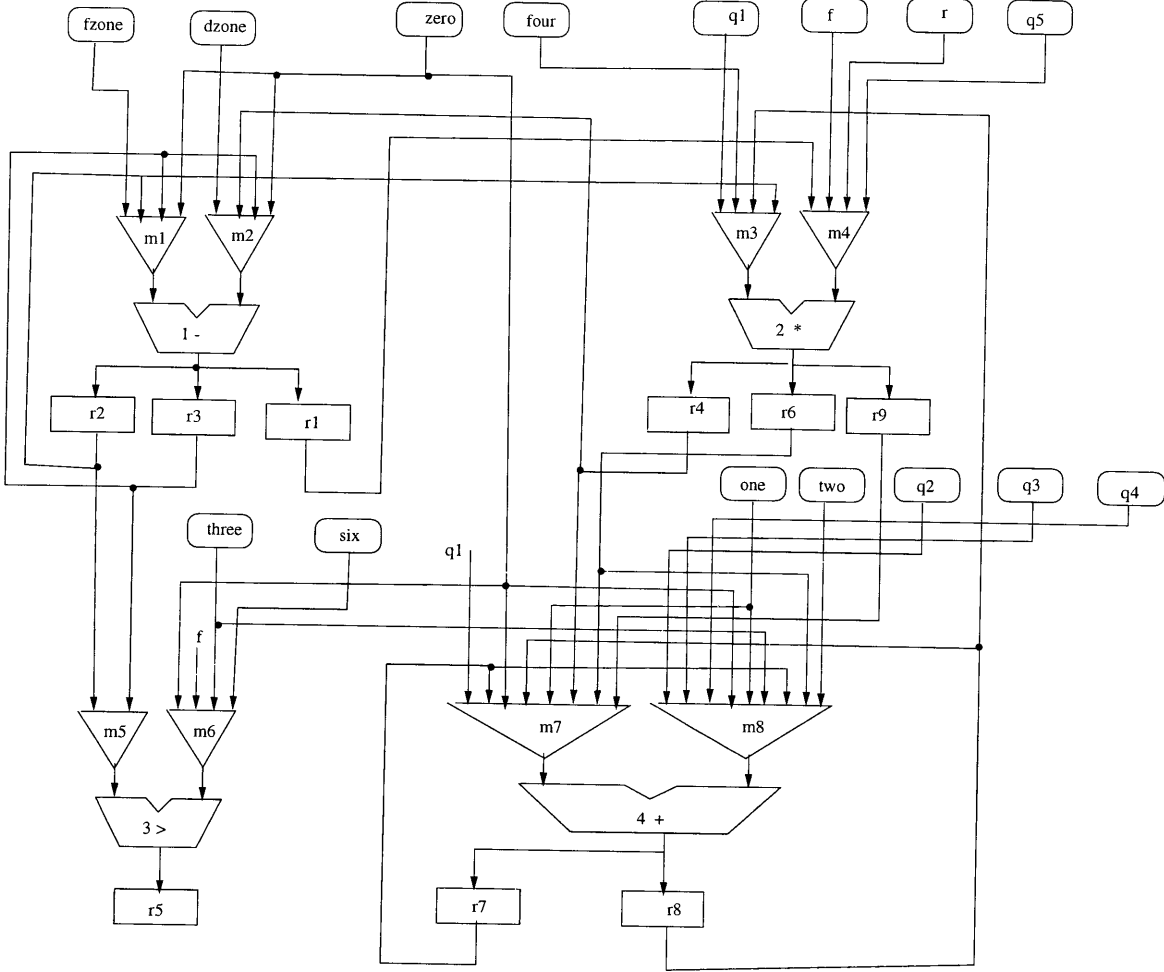


Figure 4.14: POST behavior datapath.

$\{m3, m4, *, r4, r6, r9\}$ where MUX $m3$ and MUX $m4$ are 4-to-1 MUXes. The “>” comparator functional block is $FB3 = \{m5, m6, >, r5\}$ where MUX $m5$ is a 2-to-1 MUX and MUX $m6$ is a 4-to-1 MUX. The adder functional block is $FB4 = \{m7, m8, +, r7, r8\}$ where MUX $m7$ is an 8-to-1 MUX and MUX $m8$ is a 9-to-1 MUX. $FB4$ is the largest functional block and the most active. This partition is trivial because there are no shared components.

The centralized controller is shown in Figure 4.15. The POST controller has 22 states and an input bus which is one bit wide. The results of the conditions

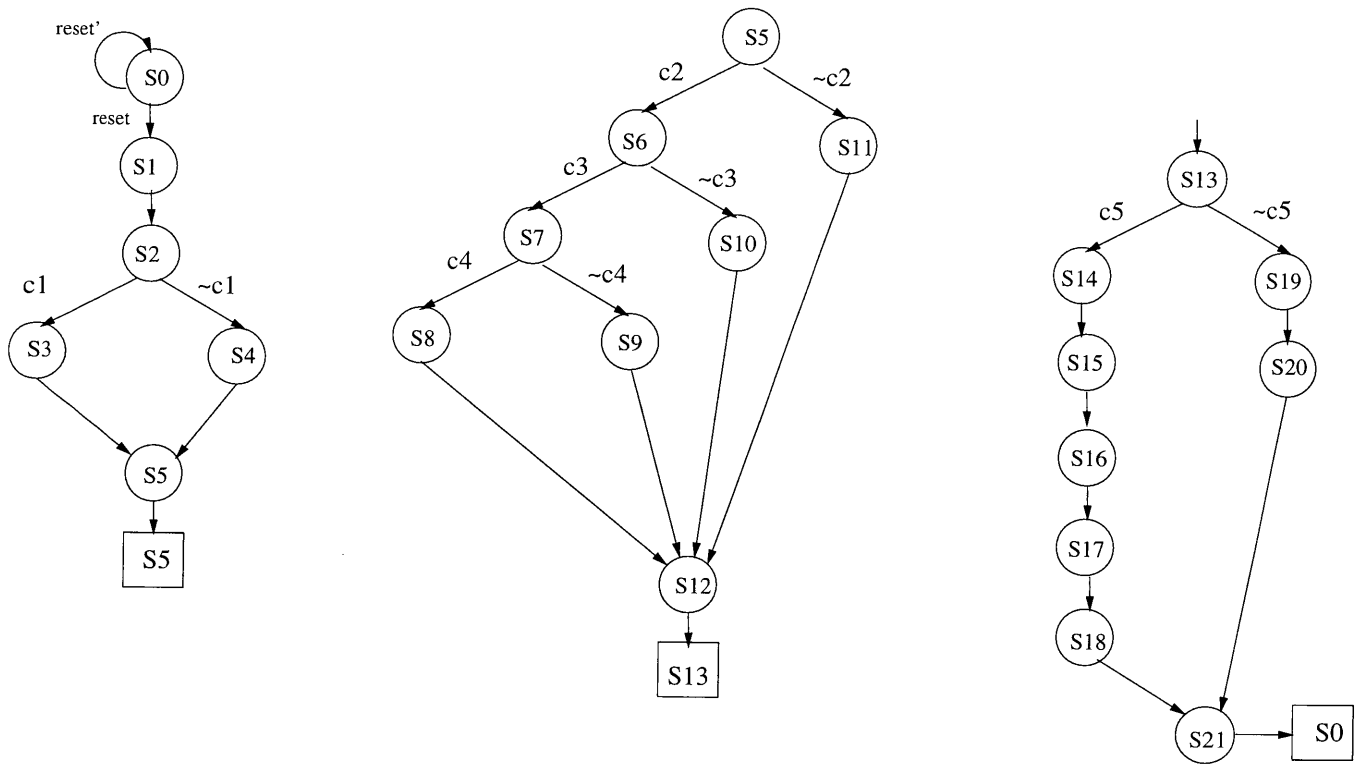


Figure 4.15: The FSM of the POST behavior controller.

$c1, c2, c3, c4$, and $c5$ are all stored in register $r5$ which feeds the controller. The POST controller output bus is 27 bits wide, nine of these bits are for register loads and 18 bits are for MUX select. The POST centralized controller output control vector is enable eight MUXes $\langle m1, m2, m3, m4, m5, m6, m7, m8 \rangle$ and nine registers from $r1$ to $r9$.

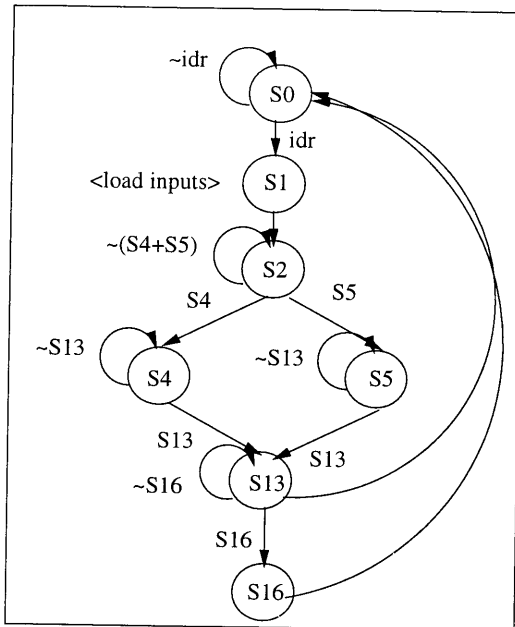
The NCLC distributed controller consists of four Non-communicating local controllers. Each local controller has the same FSM as the centralized controller with different output. The output from the centralized controller is partitioned into field of control output vector for each functional block. The subtractor local controller output control vector is $\langle e(m1), e(m2), e(r1), e(r2), e(r3) \rangle$. The multiplier local controller output control vector is $\langle e(m3), e(m4), e(r4), e(r6), e(r8) \rangle$. The comparator output control vector is $\langle e(m5), e(m6), e(r5) \rangle$. The adder output control vector is $\langle e(m7), e(m8), e(r7), e(r8) \rangle$.

The adder, in the CLC design, is chosen to be the master local controller because it executes 8 operations where the subtractor executes only four operations, the multiplier executes only four operations, and the comparator execute five operations. The master local controller (adder local controller) is similar to the centralized controller which is shown in Figure 4.15.

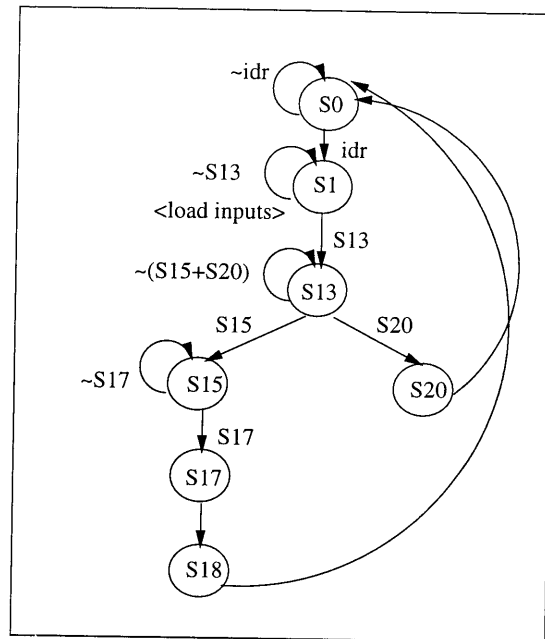
The other CLC local controllers are shown in Figure 4.16.

After the three designs are synthesized, their hierarchical layouts are generated and the number of transistors, the average power, and the critical path delay are measured. The results are presented in the next five charts.

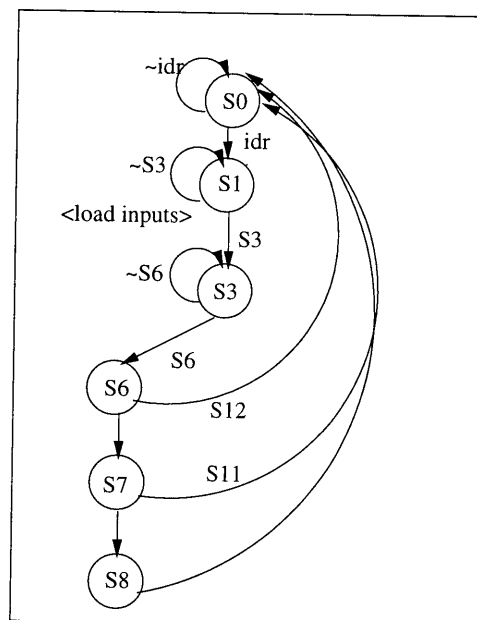
The results in chart 3.6 confirm the claim that distributed controller design has less critical path delay, less power consumption than the centralized controller.



Subtractor Local controller



Multiplier Local controller



Comparator Local controller

Figure 4.16: POST CLC local controllers.

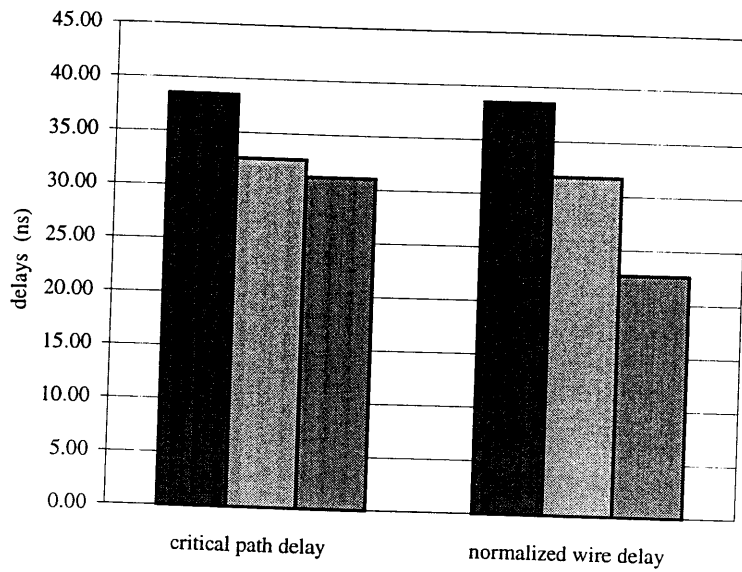


Chart 3.6: POST critical path and wire delays.

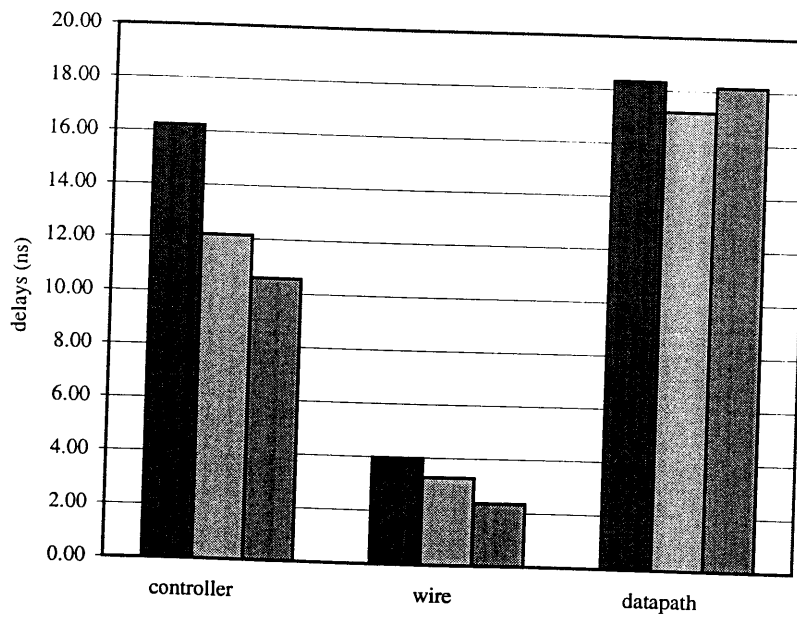


Chart 3.7: POST delay factors

■ Centralized controller design
 ■ Non-communicating distributed controller design
 ■ Communicating distributed controller design

The critical path delay for the centralized controller is 35.8 *ns*. However, the critical path delay for the NCLC and CLC circuits are 33.17 *ns* and 31.59 *ns* as shown in chart 3.5. The centralized controller critical path is $controller \rightarrow m3 \rightarrow ALU3 \rightarrow r4$. *ALU3* is the multiplier ALU (see Figure 2.12). The critical path for both distributed controller circuits is $FB3.controller \rightarrow FB3.m3 \rightarrow FB3.ALU3 \rightarrow FB3.r4$. The critical path delay has three components: delay through the controller, control wire delay and delay through the datapath as shown in chart 3.7. The delay reduction is due to shorter control wires between the multiplier local controller and functional block. Moreover, there are less transistors in the multiplier local controller output logic than the centralized controller output logic, which reduces the controller delay.

The second group in chart 3.6 shows normalized wire delay. The results suggest strongly that distributed controller design are desirable in circuits operating at high frequencies.

The centralized controller design has 6381 transistors where NCLC design has 6022 transistors and CLC design has 5596 transistors as shown in chart 3.8.

Moreover, The distributed controller design consumes less power. The average power consumption 3767.23 *uw*, 1920.32 *uw*, and 2778.35 *uw* are for the centralized controller design, NCLC design and CLC design respectively as shown in chart 3.8. The data in charts 3.9 and 3.10 shows the power consumption for each unit (functional block and its local controller) which has strong relation to the number of transistors of the unit and its activities.

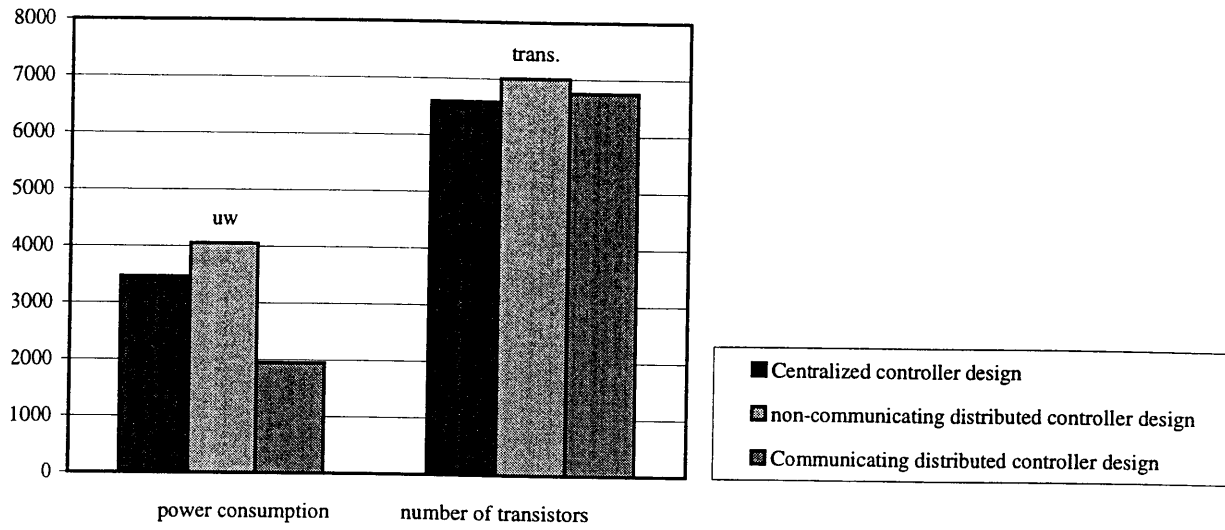


Chart 3.8: POST power and transistors

4.2.3 Case Study III

The floating-point addition behavior (FPADDS) is a complex control dominated behavior. It has assignment operations, it has seven shared components. It has four conditions as shown in the CDFG in Figure 4.17. It has seven inputs and four outputs. two of the inputs a, b are eight bits wide, p, q, MIN are four bit wide, and $ONE, ZERO$ are one bit wide. The output d is eight bits wide, s is four bit wide, and $expov, expun$ are one bit wide. The data flow graph for each basic block is shown in Figures 4.18 and 4.19.

The datapath is shown in Figure 4.20. The floating-point addition datapath has 13 ALUs, 11 MUXes and 15 registers. Some ALUs are specialized. For example, the “++” ALU counts the number of leading sign bits. The “= 7” ALU is a single input ALU that compares the input operand with the number 7. The single input ALU “= 0” compares the input operand with zero. The “<<” ALU performs the shift left operation and the “>>” ALU performs

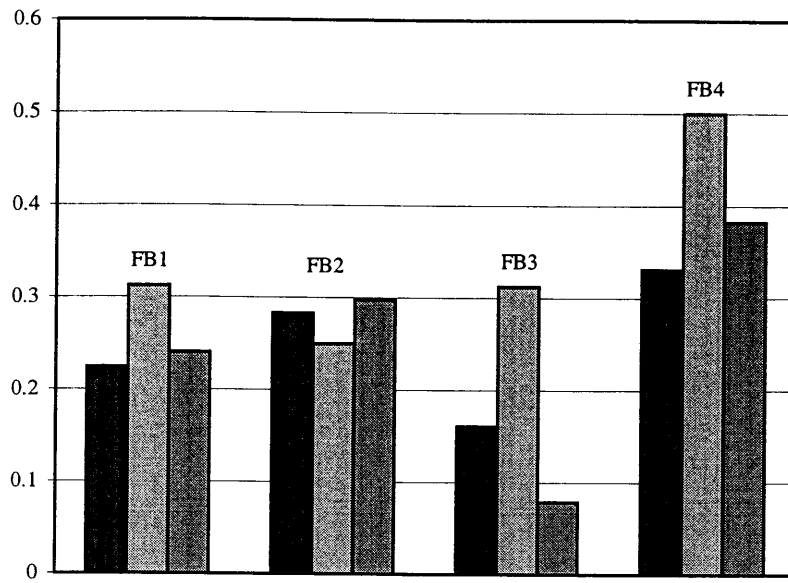


Chart 3.9: POST NCLC functional blocks data.

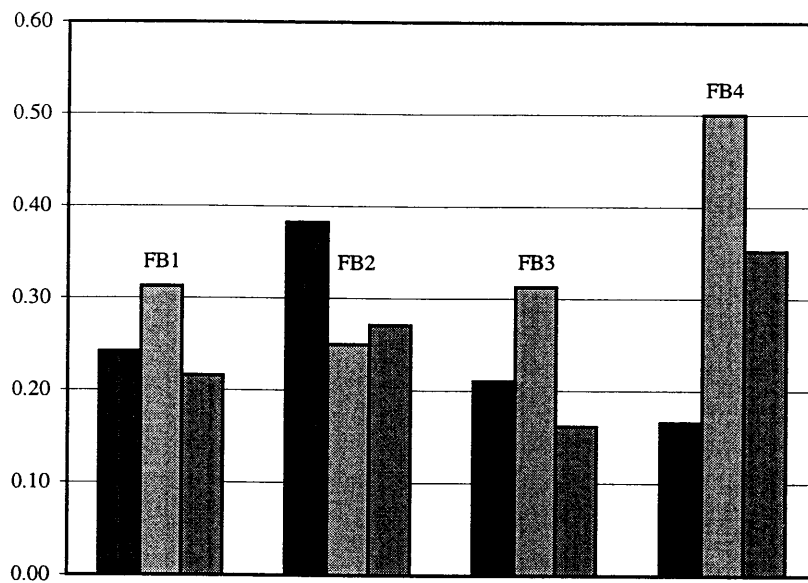
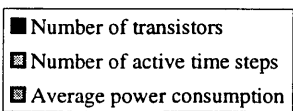


Chart 3.10: POST CLC functional blocks data.



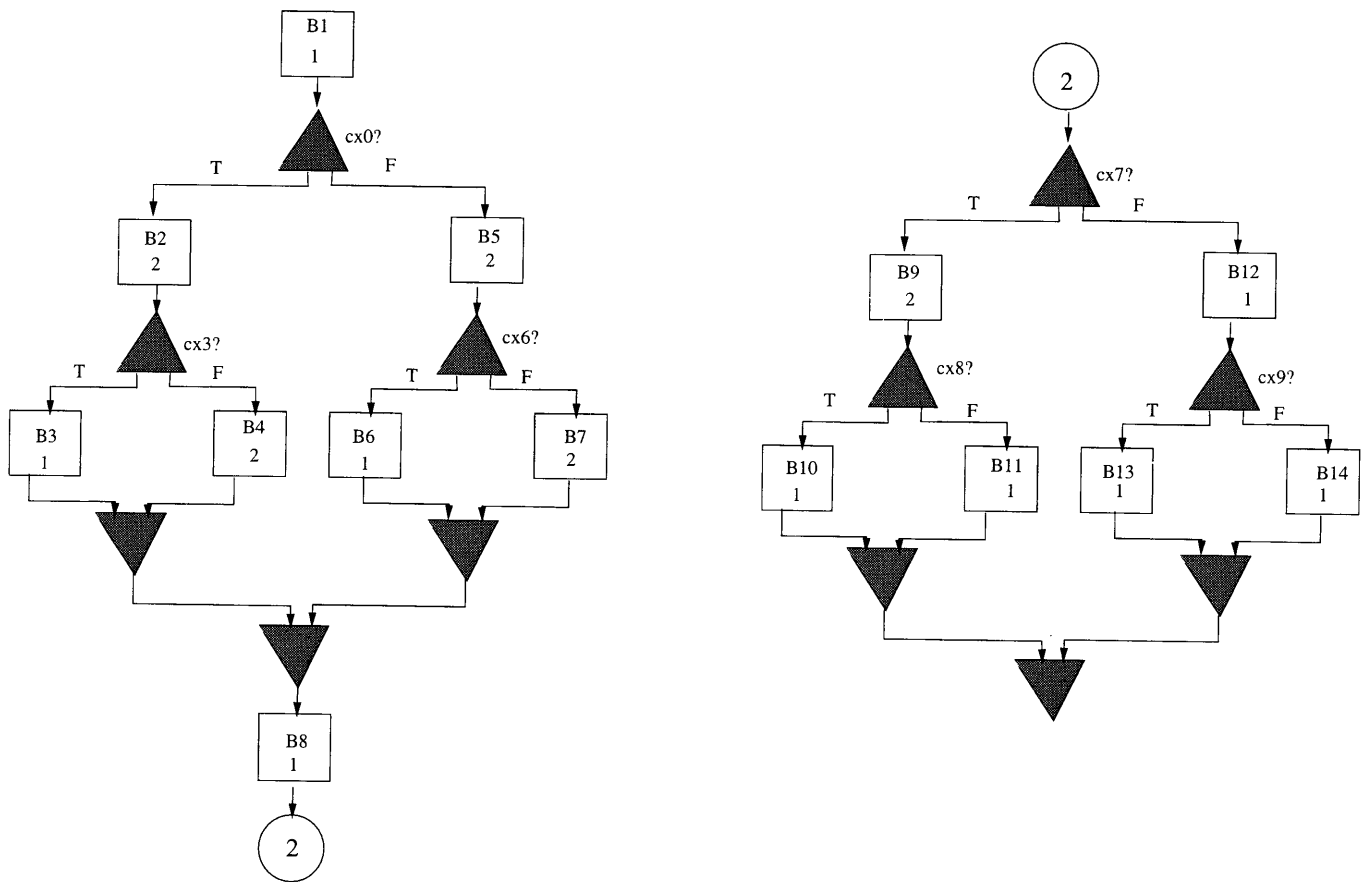


Figure 4.17: Scheduled CDFG for the FPADD behavior.

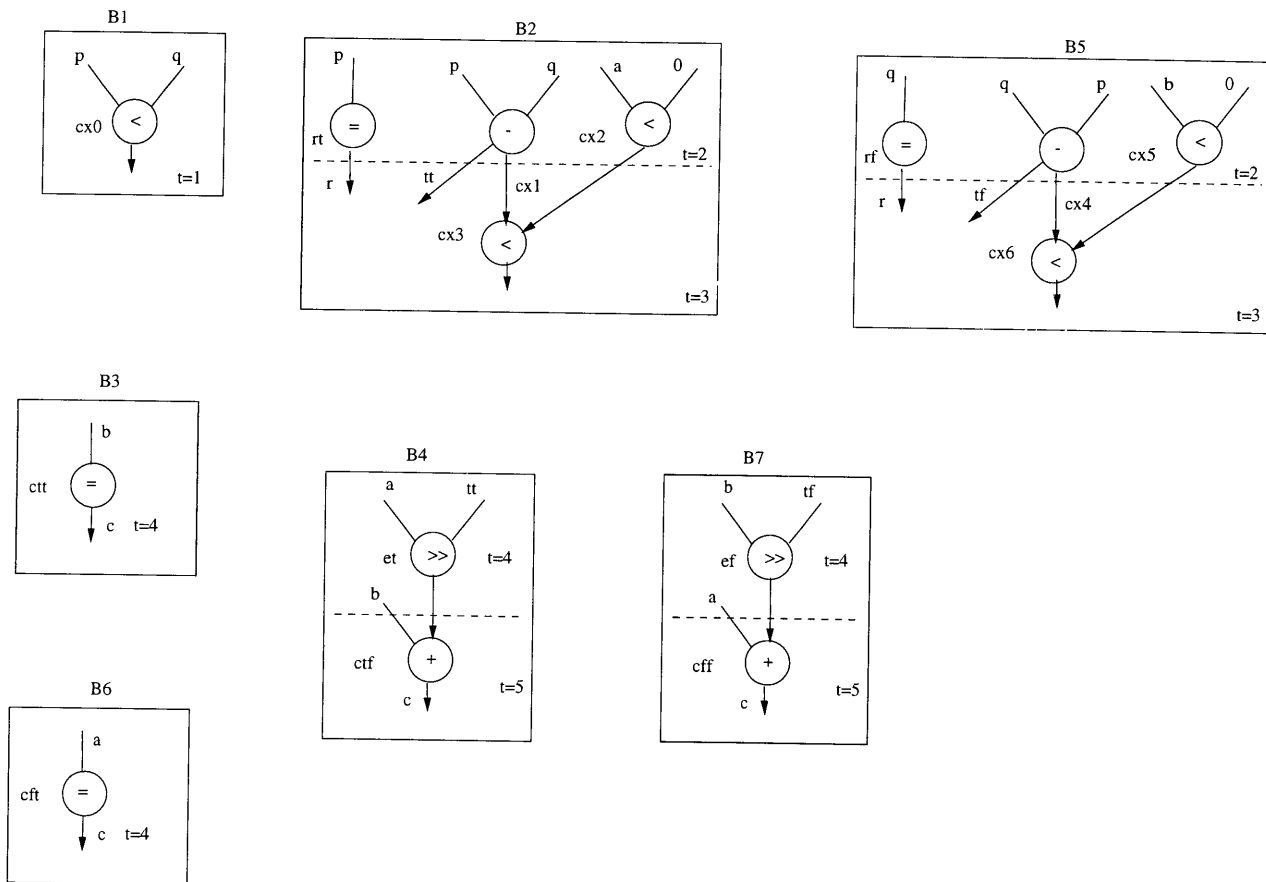


Figure 4.18: PART I: Basic Blocks for the FPADDs behavior.

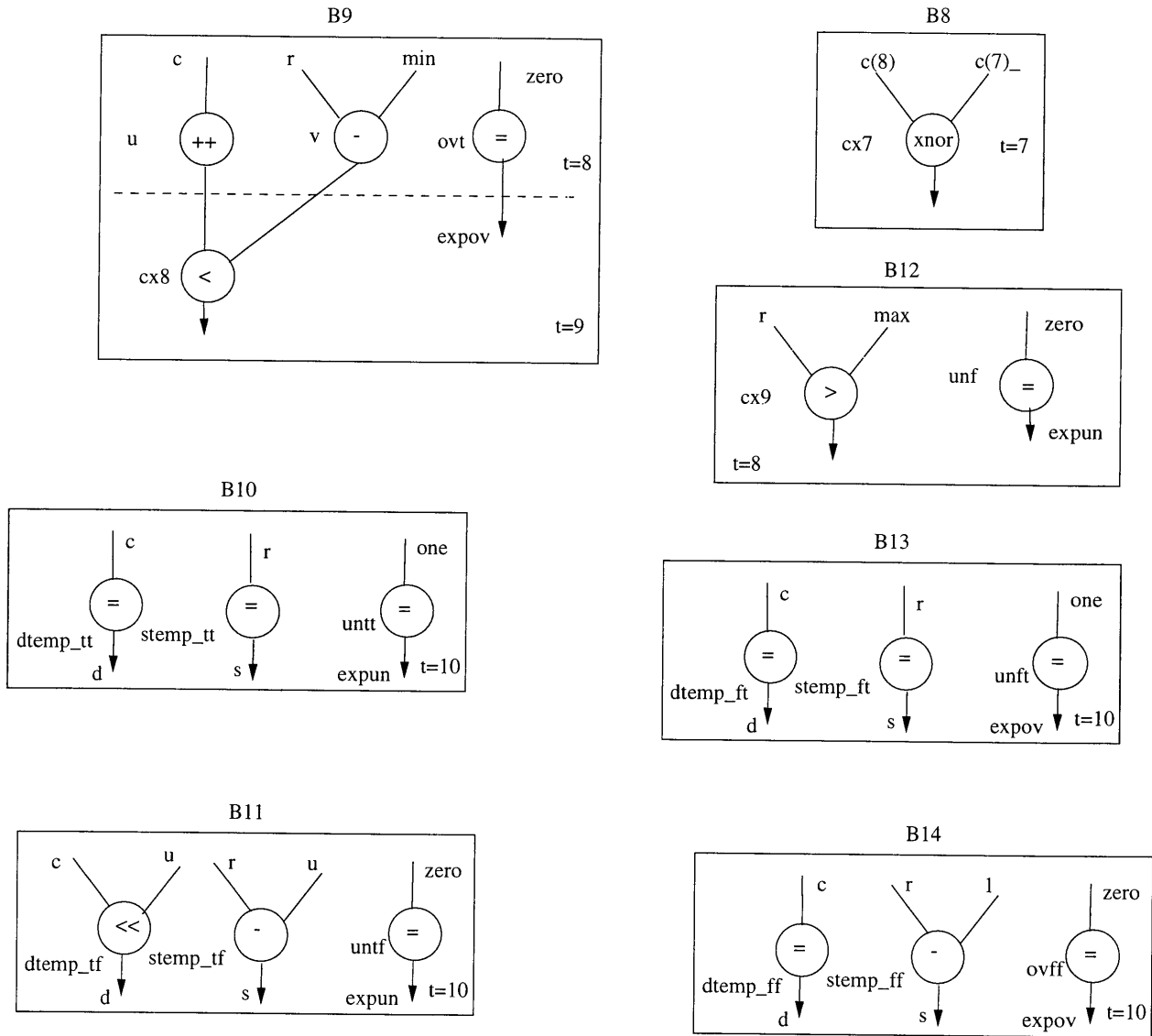


Figure 4.19: PART II: Basic Blocks for FPADDS behavior.

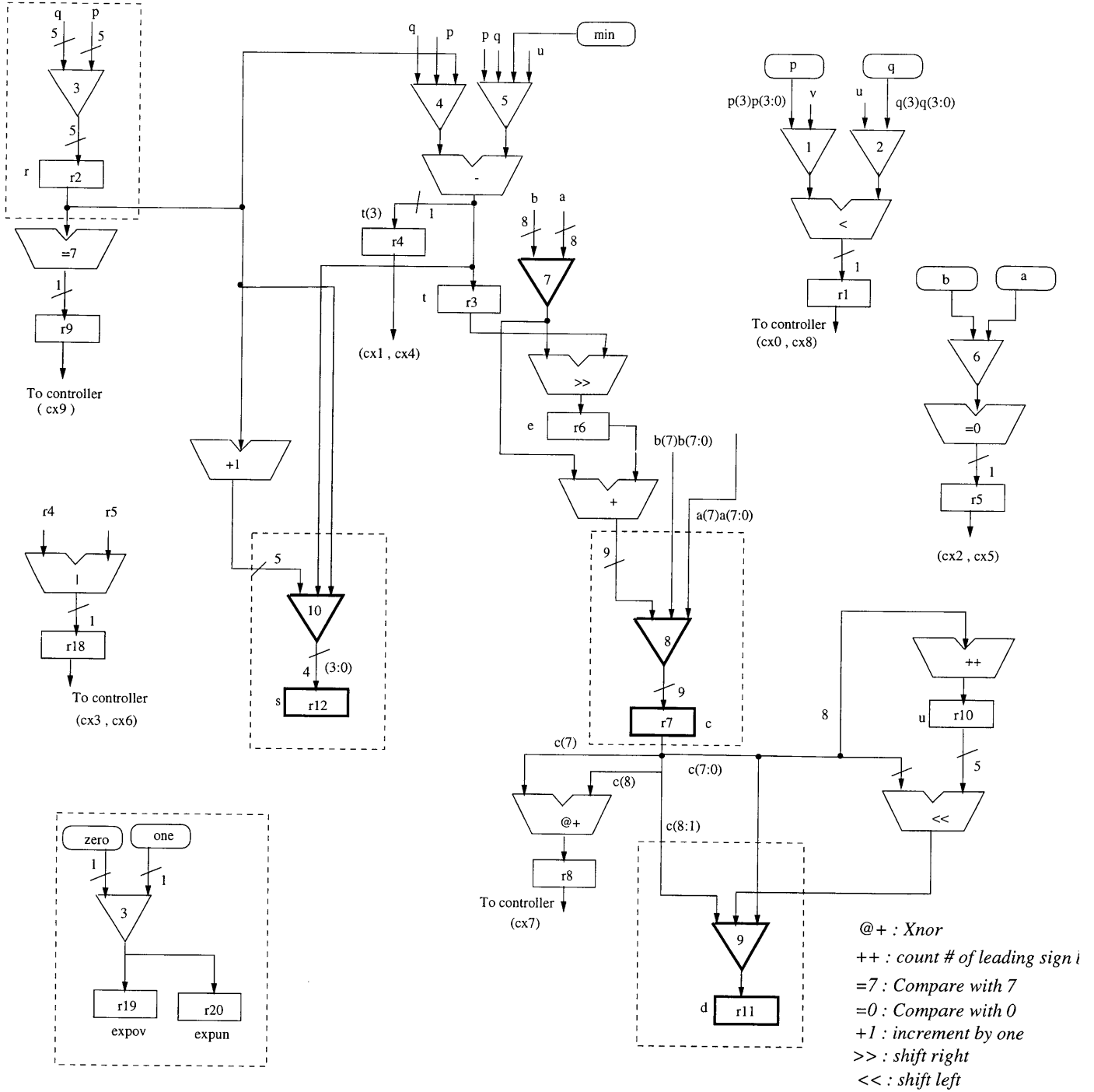


Figure 4.20: Datapath for the FPADDS behavior.

the shift right operation. The “+1” ALU increments the input operand by one. The assignment operation is done by transferring data from register to register without any modification of the data, so no special assignment ALU is shown in the datapath. All assignment operations are executed within dashed rectangles in Figure 4.20 and shared components are shown with thicker lines. There are 5 assignment MUXes.

For example, the operation $stemp_ft := r$ transfers the content of register $r2$ to register $r12$ through the assignment MUX $m10$. The variable r value does not change through any ALU manipulation.

the datapath is partitioned into 13 functional blocks. A shared component is placed with only one ALU. The input MUX $m7$ is shared between the shift right ALU and the adder ALU. The most active ALU should control MUX $m7$. Since both the shift right ALU and the adder ALU execute two instructions, MUX $m7$ is chosen randomly to be with the adder ALU. MUX $m10$ and register $r12$ are shared between the subtractor, and the “+1” ALU. MUX $m10$ and register $r12$ will be controlled by the subtractor local controller because it has more activities. “+1” ALU does not have a local controller because it is the only component in its functional block. Figure 4.21 shows the functional blocks for the partitioned datapath. A shared component may be active during some time steps when the functional block ALU is idle because a shared component is needed by other ALUs which share it. Therefore, the local controller that controls a shared component may be active during some states to execute other operations not allocated to its ALU.

The centralized controller is shown in Figure 4.22. It has 21 states. The six conditions in the CDFG: $cx0$, $cx3$, $cx6$, $cx7$, $cx8$, and $cx9$ are inputted to

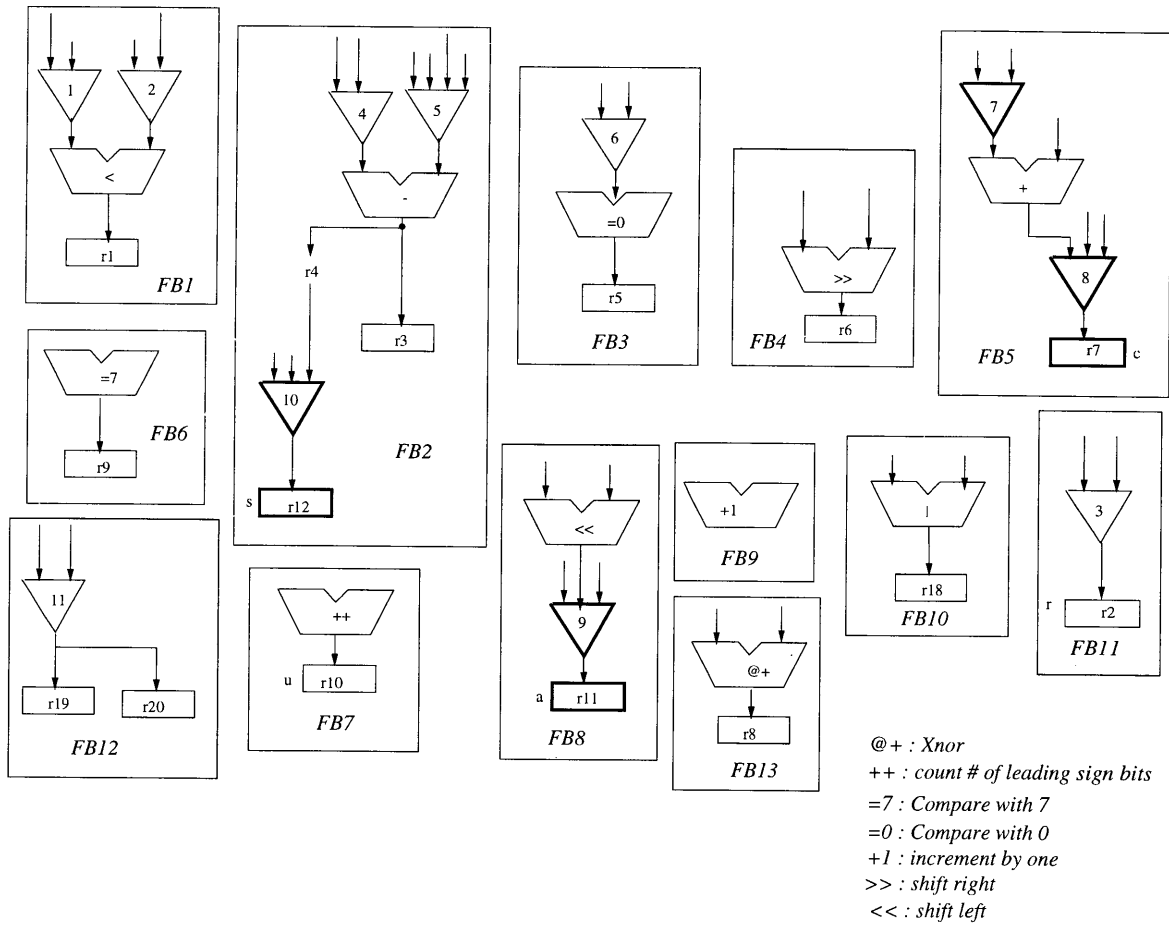
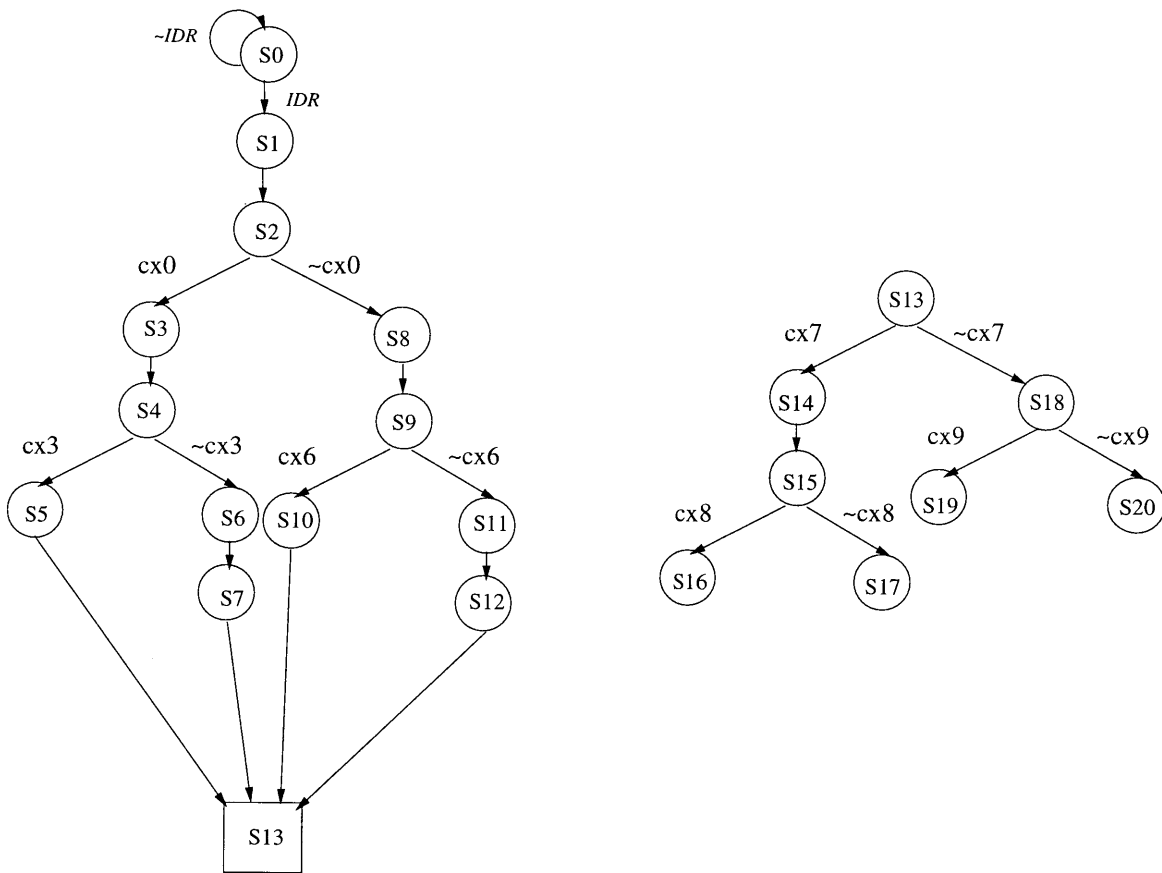


Figure 4.21: Functional Blocks for the Floating-Point Addition behavior.



output is not shown because it is large

Figure 4.22: Centralized controller for the FPADDs behavior.

the controller. Conditions $cx0$ and $cx8$ are stored in register $r1$. Register $r18$ stores conditions $cx3$ and $cx6$. Register $r8$ stores condition $cx7$ and register $r9$ stores condition $cx9$. All four registers $r1$, $r18$, $r8$, and $r9$ are one bit wide, so the controller input bus is 4 bits wide. The states $S16$, $S17$, $S19$ and $S20$ are final states. The controller output bus is 31 bits wide where 15 bits are for register loads and 16 bits are for MUX selects.

the transformation tool generated the NCLC design and the CLC design. Only 12 local NCLC local controllers are generated for the 13 functional block because the functional block $FB9 = \{“+1”\}$ has no control points. All NCLC local controller are similar to the centralized controller in Figure 4.22 except the output is simpler.

The subtractor local controller is selected to be the master local controller for the CLC design. Figure 4.23 shows the master local controller which is the same as the centralized controller except the output vector is different. The master local controller output vector, enables the subtractor functional block and broadcasts the next state NS signal. Figures 4.24 and 4.25 show the other CLC local controllers. State $S1$ is a sleeping state for the adder local controller. It will self-loop until the NS signal from the subtractor local controller is $S7$, $S5$, $S12$ or $S10$.

After synthesis and layout, the critical path delay, average power and transistor counts are measured for the three designs. The data are shown in the charts 3.11 until 3.15.

These data suggest strongly that distributed controller circuit has better performance and power consumption than the centralized controller circuit.

The centralized controller circuit has longer delay than both distributed

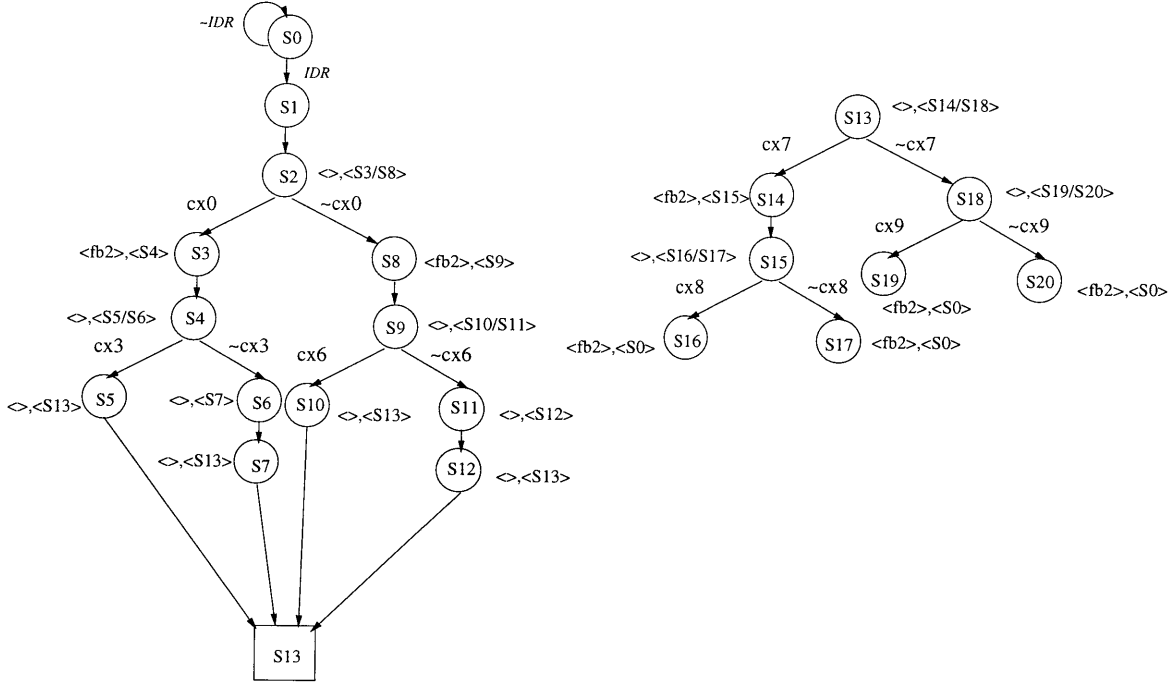
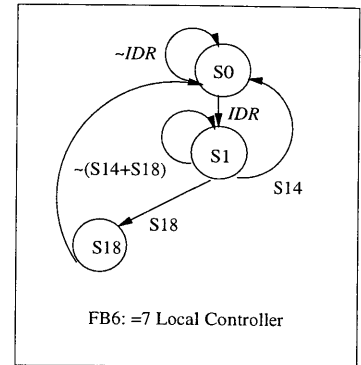
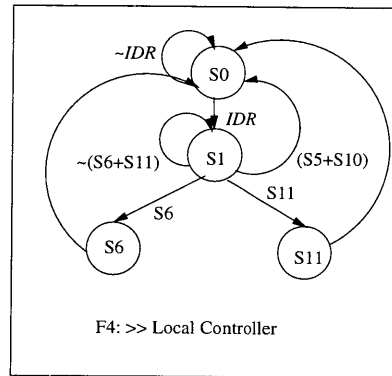
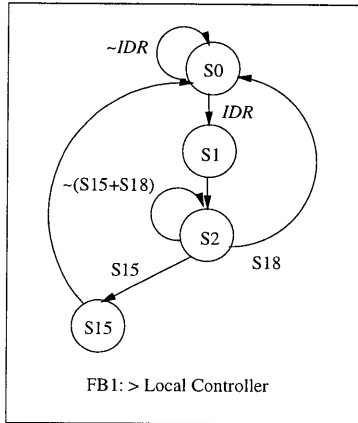


Figure 4.23: The subtractor local controller (master) for the FPADDS behavior.

controller circuits. The critical path delay for the centralized controller is 31.52. However, the critical path delay for NCLC and CLC circuits are 28.56 ns and 28.25 ns respectively. The centralized controller critical path is *controller* $\rightarrow$ *m4* $\rightarrow$ *ALU2* $\rightarrow$ *m10* $\rightarrow$ *r12*. *ALU2* is the subtractor (see Figure 4.20). However, the distributed controller critical path is *FB5.controller* $\rightarrow$ *FB5.m7* $\rightarrow$ *FB5.ALU5* $\rightarrow$ *FB5.m8* $\rightarrow$ *FB5.r7* where *ALU5* is the adder. Even though the critical path is different between the centralized and the distributed controller, the distributed controller critical path delay is less than the centralized controller critical path delay. The critical path delay has three components: delay through the controller, control wire delay and delay through the datapath. chart 3.12 shows the delays for each component. The delay reduction is due to shorter control wires between the adder local controller and



FB2: - Master Local Controller

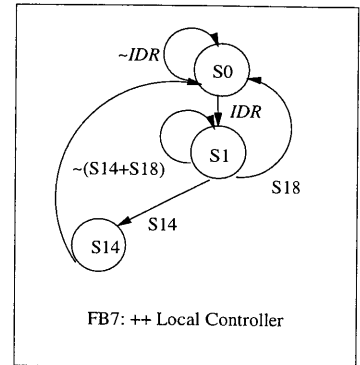
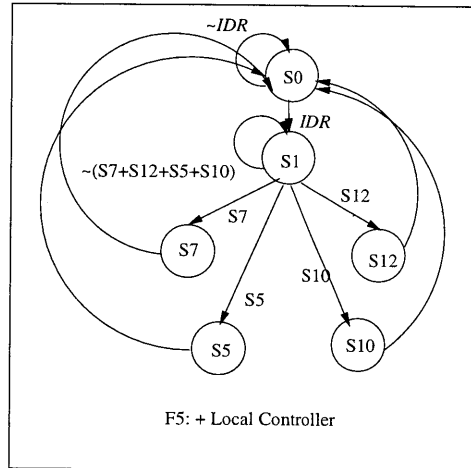
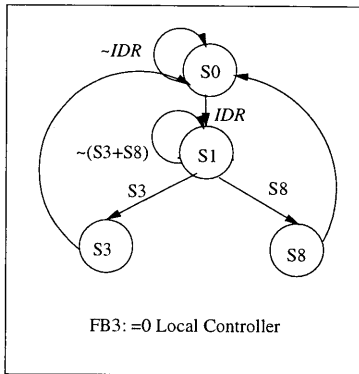
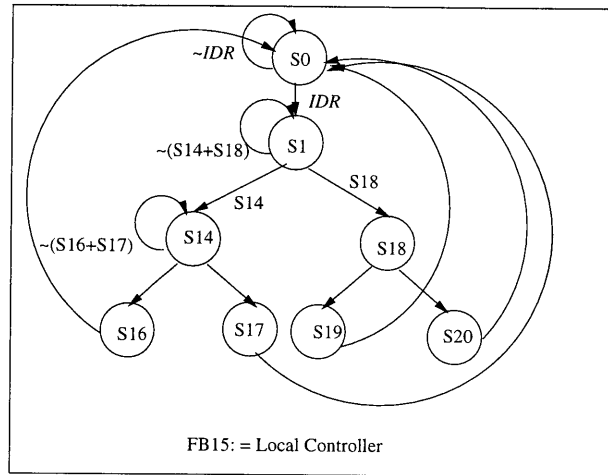
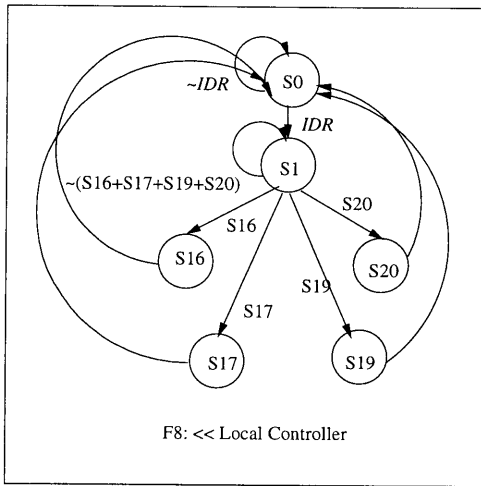


Figure 4.24: PART I:CLC distributed controllers for the Floating-Point Addition example.



F9: +1 No local controller

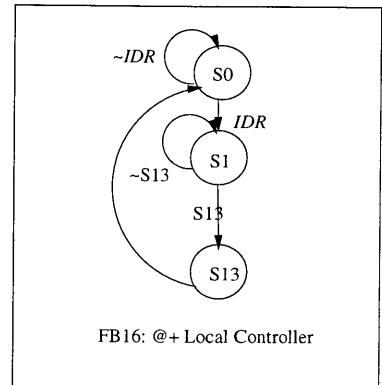
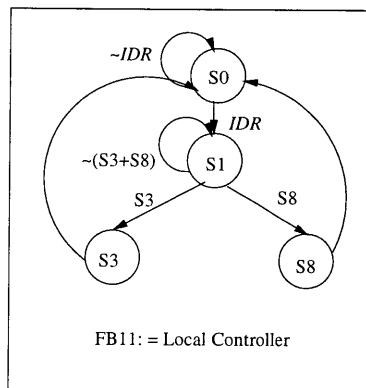
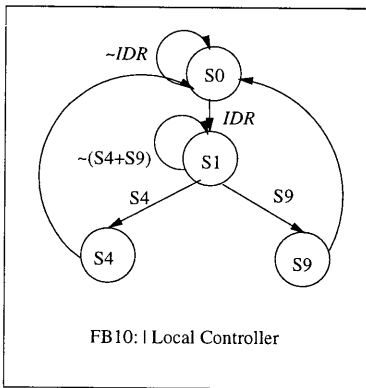


Figure 4.25: PART II: CLC Distributed Controller for the Floating-Point Addition example.

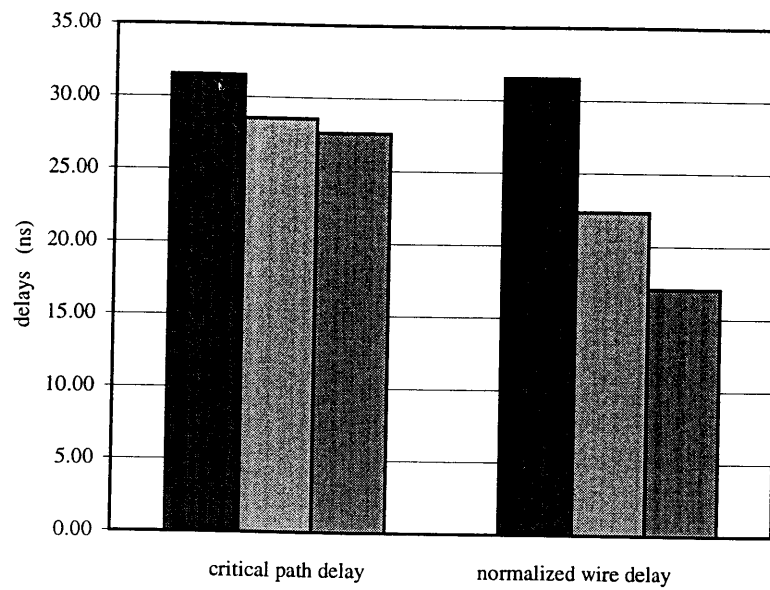


Chart 3.11: FPADDS critical path and wire delays.

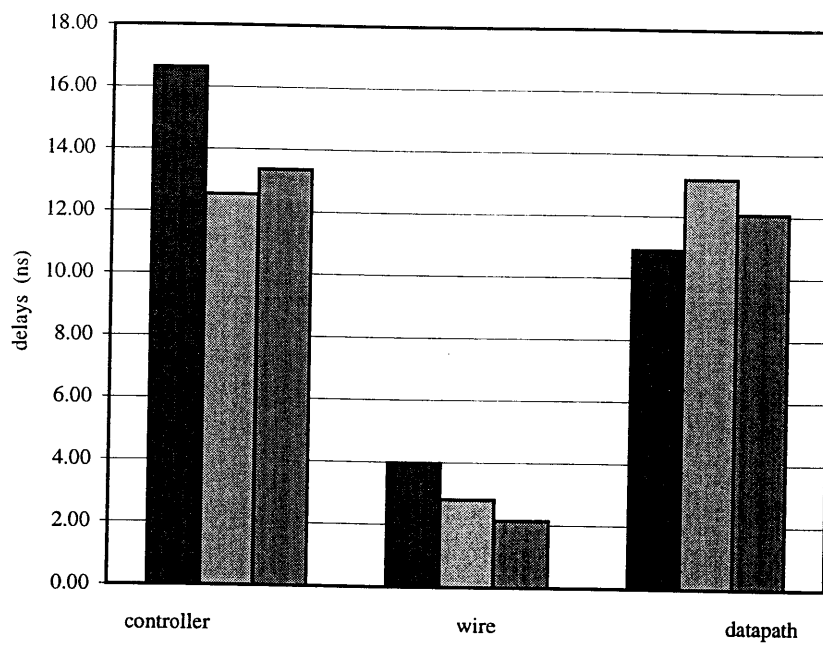


Chart 3.12: FPADDS delay factors.

Centralized controller design
 Non-communicating distributed controller design
 Communicating distributed controller design

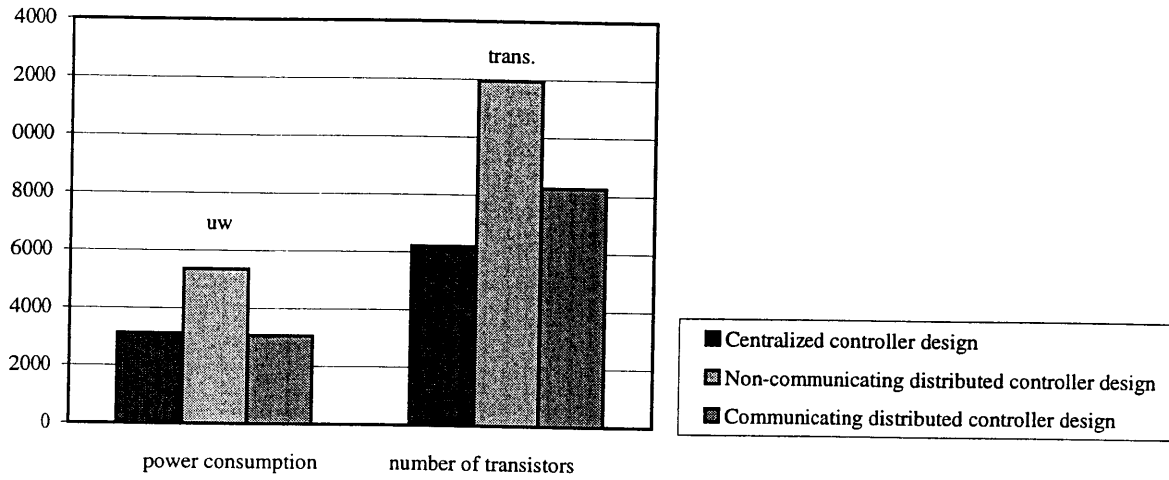


Chart 3.13: FPADDS power and transistors.

functional block.

The second group in chart 3.11 shows the normalized wire delay. The difference between the wire delays for distributed controller design and centralized controller design is more clear. Shorter control wires are essential for the circuit to operate at high frequencies.

The distributed controller design, also, consumes less power. The average power consumption 3382.85 *uw*, 3577.79 *uw*, and 3021.58 *uw* are for the centralized controller design, NCLC design and CLC design respectively as shown in chart 3.13. The RTL signals are localized in the distributed controller design. For each register transfer there are less transistors switching activities in the distributed controller circuit than in the centralized controller circuit. Charts 3.14 and 3.15, show number of transistors of each unit (functional block with its local controller), number of active time steps and average power consumption of each functional block. The results indicate that the power consumption is related to the unit number of transistors and activities.

Even though, distributed controller has more transistors than the centralized controller design. the distributed controller activities is localized. design has more transistor. The centralized controller design has 6213 transistors where NCLC design has 10497 transistors and CLC design has 6751 transistors. The NCLC design has %41 more transistors than the centralized controller and

CLC design has %8 more than the number of transistors of the centralized controller design. The increase in number transistors in both distributed controller designs is due to the complexity of the floating-point addition FSM which is duplicated 12 times. All the 12 local controller FSMs have the same structure and the same next state transition function as the centralized controller except their output is simpler than that of the centralized controller. However, the distributed controller design activities are less than the centralized controller design.

4.2.4 Experimental Results The Eight Behaviors

In this section we present the results for eight behaviors. The first Five behavior, Hough filter, Arfilter filter, Bandpass filter, Wave filter, and Discrete Cosine Transform (Case study I). do not have any conditions in their CDFG. The last three behaviors, Fast Fourier Transform, Post Calculator (case study II) and Floating-Point Addition (case study III), have conditions in their CDFG. these eight behaviors are ordered from the simple to the complex. All these behaviors are shown in the Appendix C.

Chart 3.16 shows the critical path delay. For each behavior, the critical path delay for centralized controller design, NCLC design, and CLC design are shown. The data in chart 3.16 suggest strongly that the distributed controller

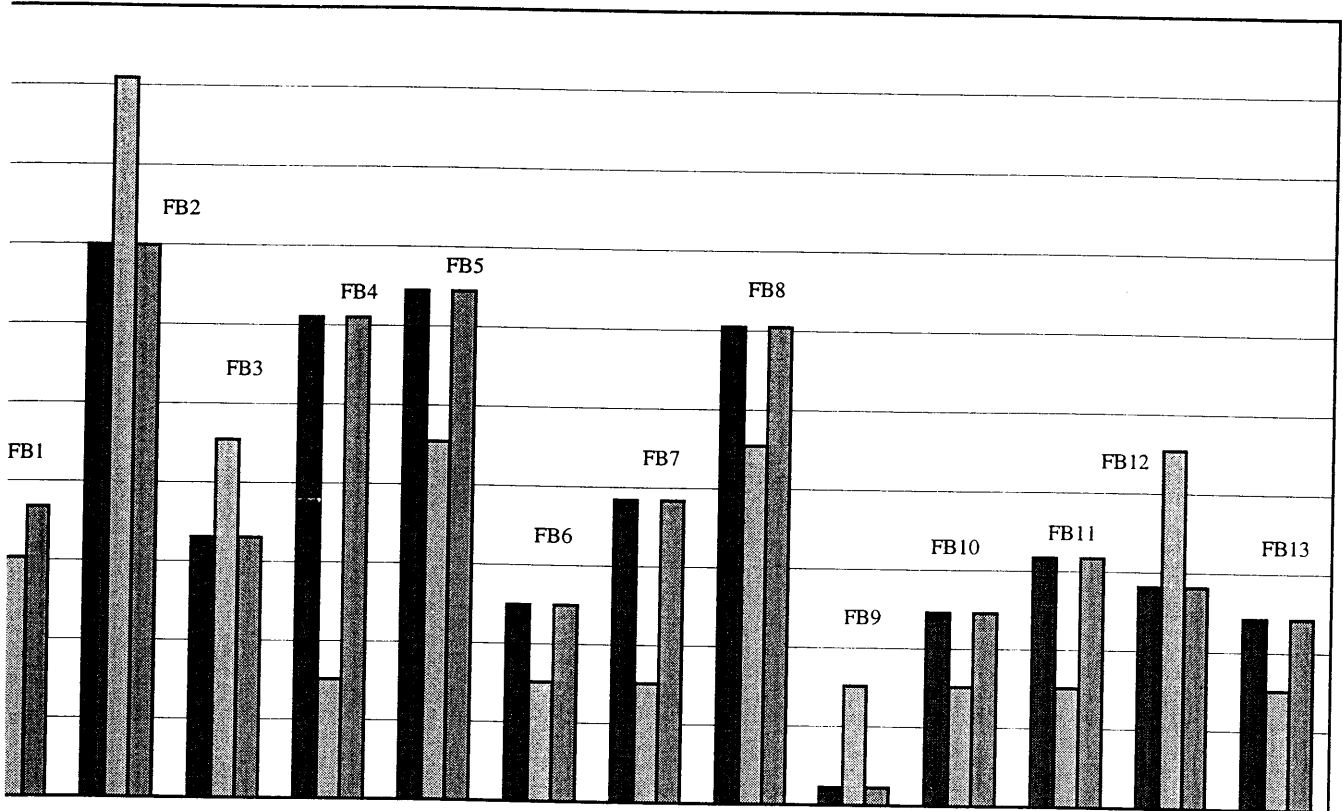


Chart 3.14: FPADDS/CLC functional blocks data

- Number of transistors
- Number of active time step
- ▒ Average power consumption

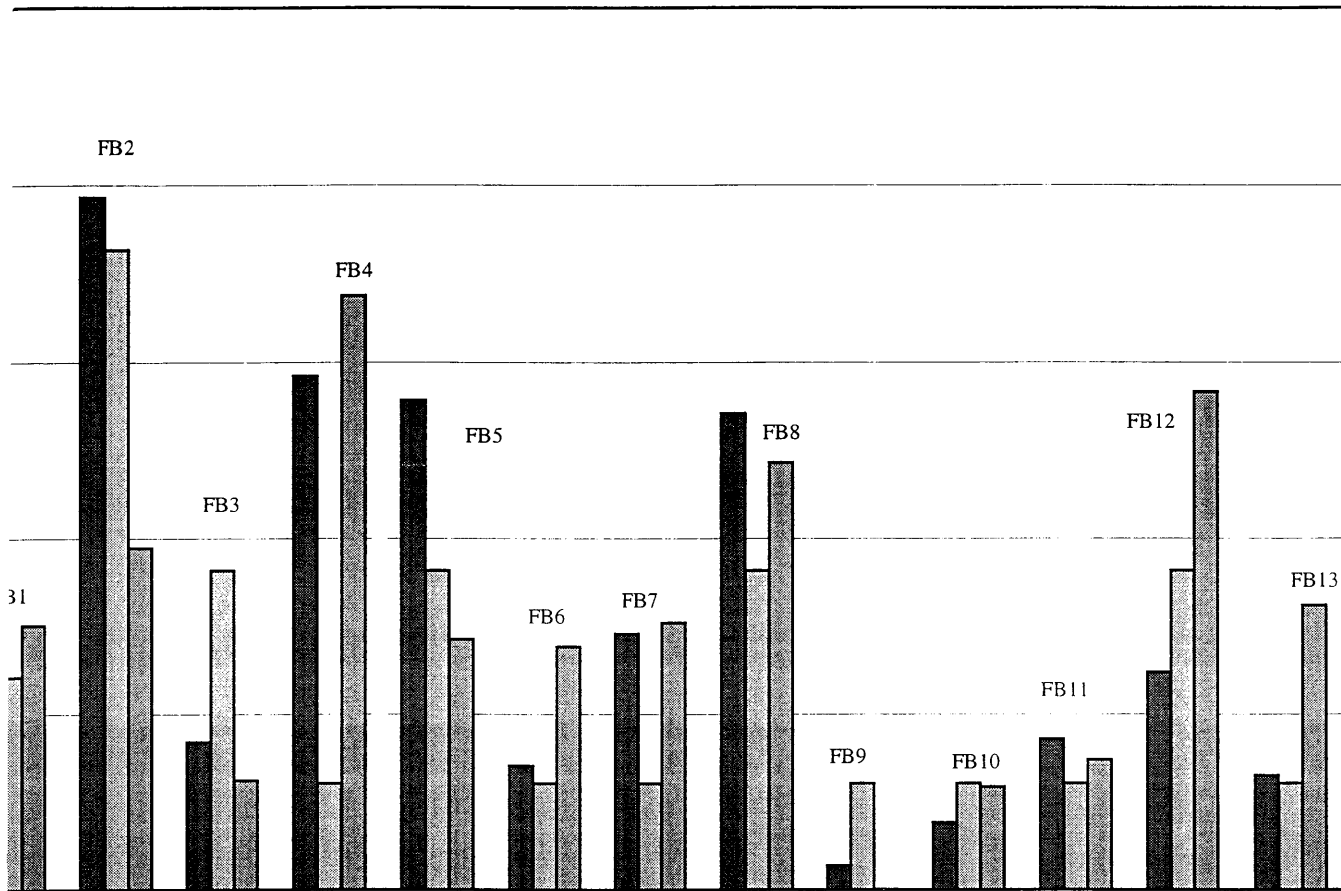


Chart 3.15 : FPADDS CLC functional blocks data.

■ Number of transistors
 ▨ Number of active time step
 ▩ Average power consumption

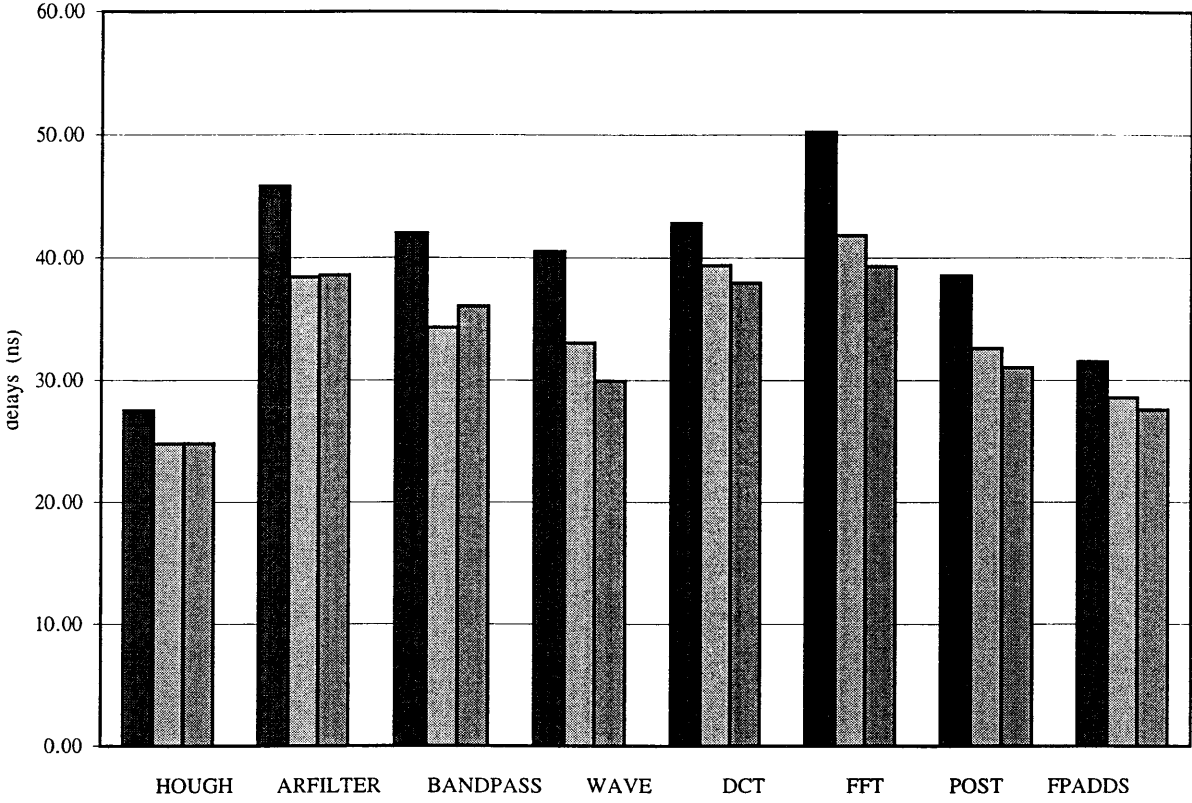
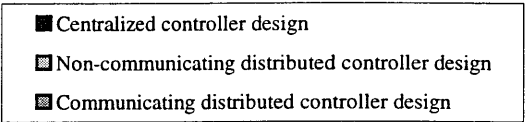


Chart 3.16: Critical path delays



design has less critical path delay than the centralized controller. There is a reduction of 8-18% in the critical path delay between the centralized controller design and the NCLC distributed controller design and a reduction of 10-22% between the CLC distributed controller design and the centralized controller design. The CLC design has less delay than the NCLC design. In all behaviors except the Bandpass Filter,

Chart 3.17 shows the normalized wire delays which is an indication of the performance of distributed controller design circuit compared with centralized controller design if all circuits are fabricated in sub-micron technology. In all eight behaviors, the distributed controller design can have better performance than the centralized controller design for in sub-micron level where the gate delay is negligible and wire delay is dominant.

Shorter control wires are essential for the circuit to operate at high frequencies. The reduction in wire delay between the centralized controller design and the NCLC distributed controller design is about 9-35% and between the centralized controller design and the CLC distributed controller design between the centralized 17-65%. Distributed controller design style is essential in sub-micron technology.

The chart in 3.18 confirm that distributed controller consumes less power than the centralized controller. There is a reduction of 1-36% in the average power consumption between the centralized controller design and the NCLC distributed controller design and a reduction of 6-51 % between the centralized controller design and the CLC distributed controller design. The POST CLC distributed controller design consumes 17% more power than the centralized controller design. The NCLC Floating-Point Addition design has 71% more

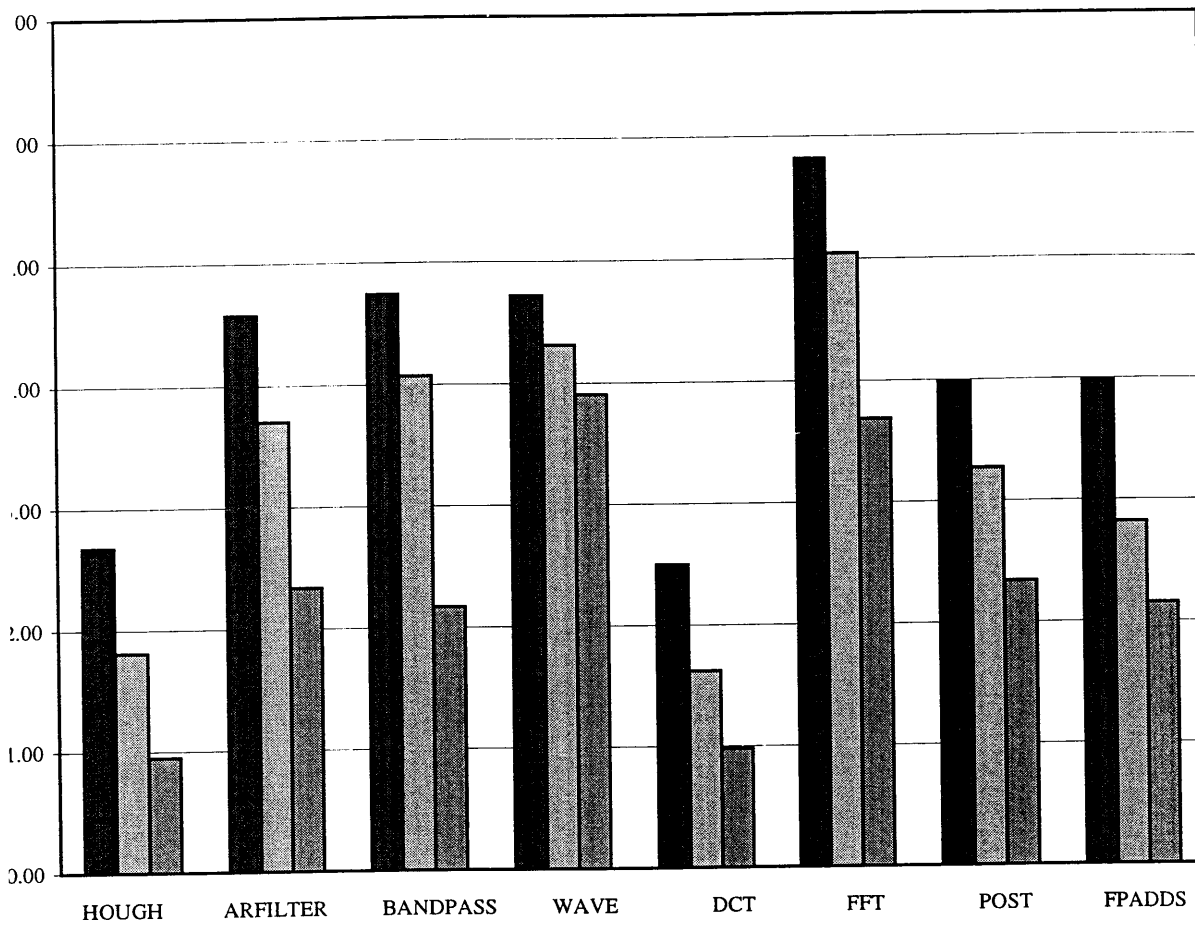
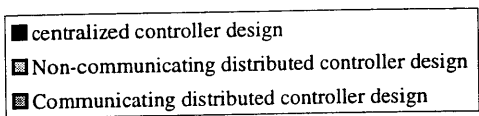


Chart 3.17: Normalized wire delay.



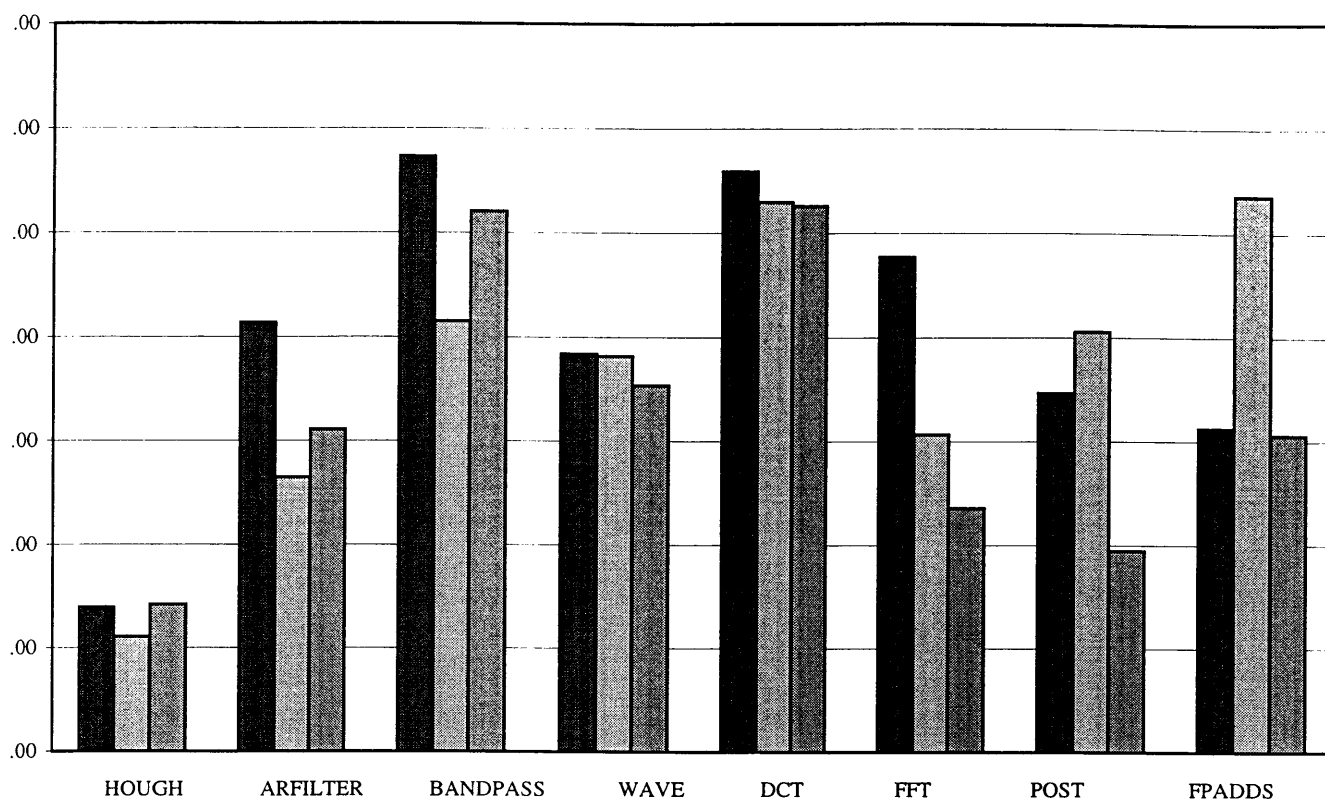


Chart 3.18: Average power consumption.

Centralized controller design
 Non-communicating distributed controller design
 Communicating distributed controller design

power than the centralized controller which is due to twice the number of transistors in the NCLC design compared with centralized controller.

The chart in 3.19 shows the number of transistors. Even though the controller is duplicated many times in the distributed controller design, there is no noticeable increase in the number of transistors compared with centralized controller design. Indeed, the first three behaviors have less transistors in the distributed controller design than in the centralized controller design. There is an increase of 3-15% in the number of transistors between the centralized controller design and both distributed controller designs. The reduction in the number of transistors in the distributed controller design is due to simple output logic for each controller. The NCLC Floating-Point Addition design has the largest number of transistors. The Floating Point Addition circuit is control dominated. It has 13 functional blocks, so the centralized controller is duplicated 12 times. However, the distributed controller critical path delay for the Floating-Point Addition circuit is less than that for the centralized controller design as discussed earlier.

The data shows the importance of distributed controller design specially for future circuit design where wire length plays a dominant role in the circuit delay and power consumption is an essential issue for circuit operating at high frequencies.

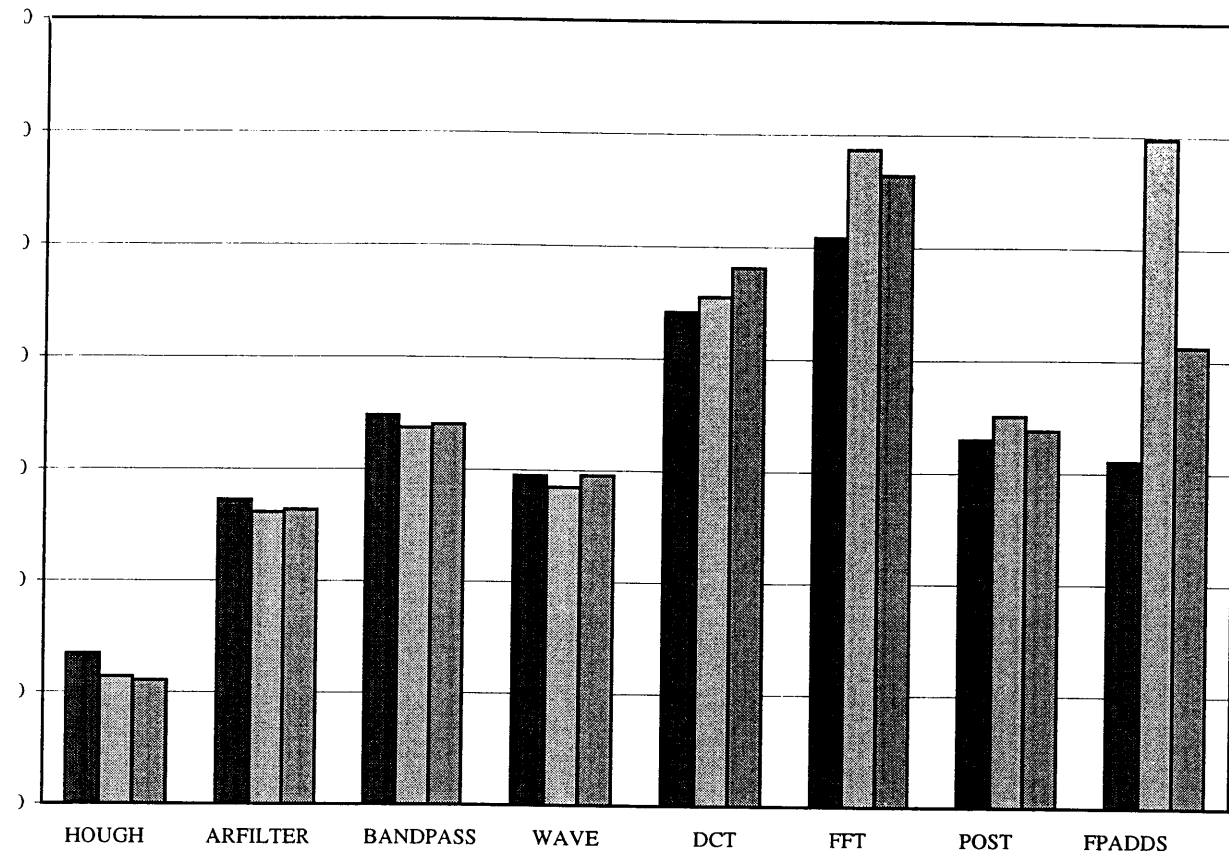


Chart 3.19: Number of transistors.

- Centralized controller design
- Non-communicating distributed controller design
- ▨ Communicating distributed controller design

The following are the data for the eight behaviors:

circuit		CPD (<i>ns</i>)	Normalized wire delay (<i>ns</i>)	Ave. power (<i>uw</i>) in <i>uw</i>	num. tran.
HOUGH	Centralized	27.53	27.53	1756.47	2692
	NCLC	24.8	18.61	1098.54	2274
	CLC	24.83	9.74	1167.49	2211
ARFILTER	Centralized	45.85	45.85	4696.30	5446
	NCLC	38.43	37.08	2994.14	5229
	CLC	38.58	23.42	3489.64	5275
BANDPASS	Centralized	42.01	42.01	6549.71	6961
	NCLC	34.31	36.09	4575.27	6742
	CLC	36.04	19.24	5749.40	6807
WAVE	Centralized	40.53	40.53	4637.56	5898
	NCLC	33.07	37.03	3871.35	5696
	CLC	29.93	33.54	3711.08	5911
DCT	Centralized	42.82	42.82	6524.74	8843
	NCLC	39.39	27.81	5804.72	9109
	CLC	37.95	16.78	5797.33	9630
FFT	Centralized	50.23	50.23	4571.27	10176
	NCLC	41.82	43.53	2948.70	11740
	CLC	39.31	31.82	2312.53	11282
POST	Centralized	38.53	38.53	3767.23	6596
	NCLC	32.61	31.71	1920.32	7014
	CLC	31.04	22.67	2778.35	6758
FPADDS	Centralized	31.52	31.52	3382.85	6213
	NCLC	28.56	22.33	3577.79	11953
	CLC	27.58	17.05	30211.58	8259

CPD : Critical Path Delay.

trans. : Transistors.

Chapter 5

Datapath Transformations

This chapter discusses another type of transformations which deal with operations and variables of a behavior. The basis of these transformations are in [1].

During allocation, every operation in a CDFG behavior is assigned to one ALU where it will be executed and every intermediate and final variable is assigned to one of the available registers in the datapath. During scheduling, every operation is assigned a time when it will be executed.

There are three types of transformations. ALU Reallocation, where operations are moved between compatible ALUs within the same time step. REG reallocation, where variables are moved between registers. The last type is Rescheduling transformation, where operations are moved to different time step but still executed in the same ALU. These transformation are done after scheduling and allocation. Reallocation transformations Modify the number of ALUs, MUXes, REGs in a datapath and they, also, modify the controller output. Rescheduling transformations modify the controller output and number of states.

The objective of datapath transformations is to alter the number of components in a datapath and allow the designer to choose between different feasible designs. Datapath transformations can be performed before or after distributed controller transformations. Figure 5.1 show an overview of a

transformation process.

5.1 Operation Attributes

Before scheduling and allocation, each operation has one or more attributes associated: names of its operands. After scheduling and allocation, three extra attributes are added for each operation: where it will be executed, when it will be executed, and where its result will be stored. ALU reallocation transformations modify the attribute where an operation is executed. Register reallocation modify the attribute where an operation result is stored and rescheduling transformations modify the attribute when an operation will be executed. Two types of table structure are used to present these attributes for the transformation Tool. The scheduling and allocation table (SAT) store where an operation will be executed and when an operation will be executed. The numbers of columns in a SAT is the number of ALUs allocated for a behavior and number of the rows is the number of time steps needed to execute the behavior. An operation is placed in a cell (t, a) where t is the row number which is the time when it will be executed and a is the column number which is the ALU name where it will be executed. An empty cell indicates an idle ALU at that time step. The other two attributes: the operands of an operations and where its result will be stored are presented in one or more operation enable table(s) (OET). Every ALU in the datapath has an OET table. For each ALU, number of input MUXes and their sizes can be deduced from the number of different column in its OET table. SAT and OET tables are used to facilitate the datapath transformations and to show the effect of these transformations on the datapath. SAT and OETs tables usage will be

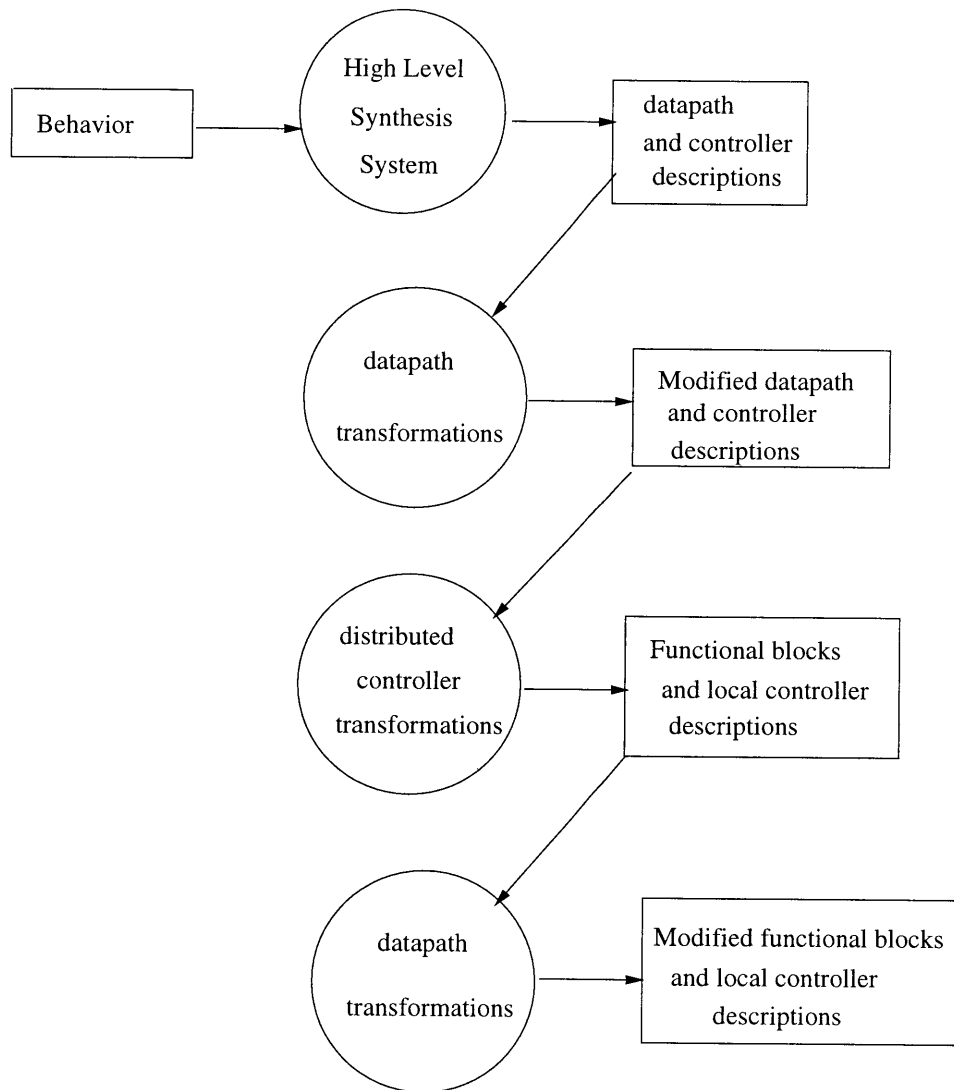


Figure 5.1: An overview of both controller and datapath transformations.

shown in the next section.

5.2 ALU Reallocation Transformations (ART)

The first type of transformation performs ALU reallocation. These transformations do not affect the schedule of a behavior. An operation can be moved to another ALU, but it will be executed at the same time step as before. In multi basic blocks CDFG, an operation cannot be moved to another basic block, because this move may lead to rescheduling the operation or modifying the behavior functionalities. The following are the four ALU reallocation transformations:

1. $A_MOVE(p, ALU_i, ALU_j)$

Assume an operation p is executing at time step t . Move p from ALU_i to ALU_j at time step t and ALU_j must have the functionality to execute the operation p .

2. $A_SWAP(p, q, ALU_i, ALU_j)$. Assume operation p is executing in ALU_i and operation q is executing on ALU_j at time step t . Swap p from ALU_i to ALU_j and q from ALU_j to ALU_i at time step t . ALU_i must be able to execute q and ALU_j must be able to execute p .

3. $A_DELETE(ALU_i)$

Remove ALU_i from the datapath. ALU_i must be idle at all time steps.

4. $A_ADD(ALU_i)$

Add new ALU_i to the datapath. Some operations eventually must be moved to ALU_i .

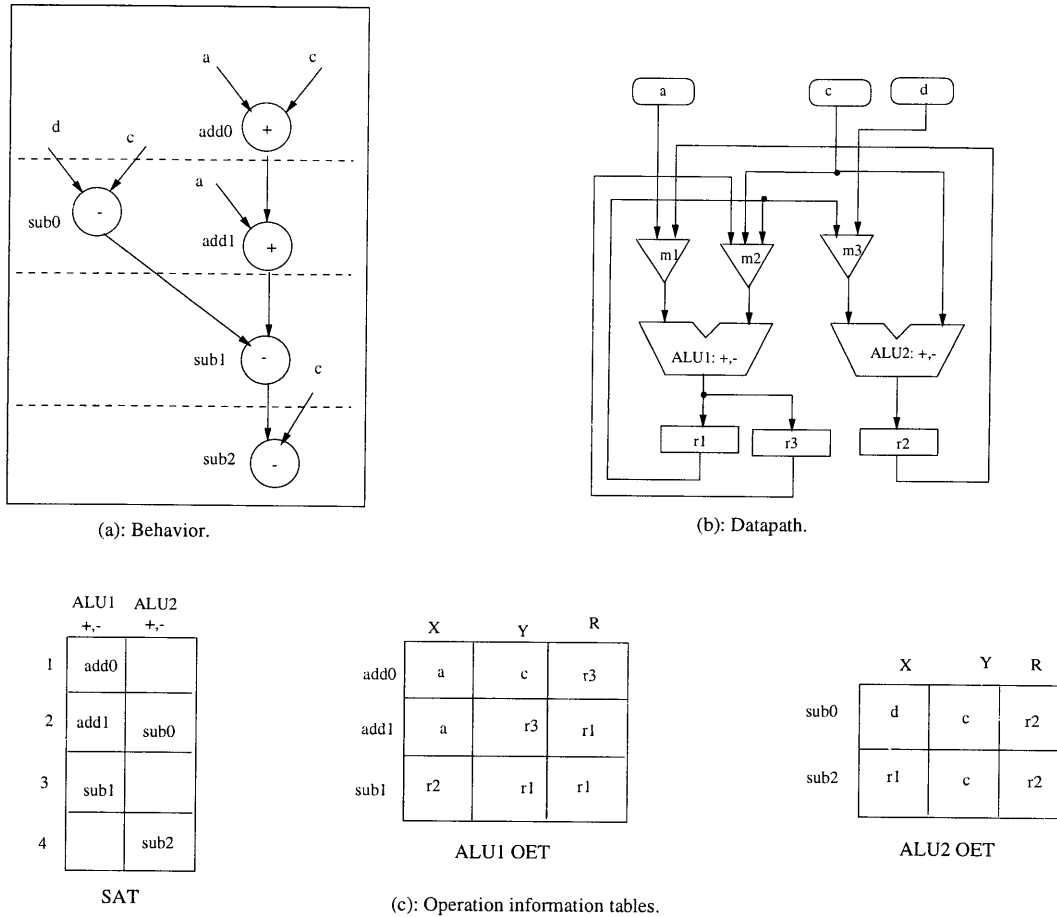
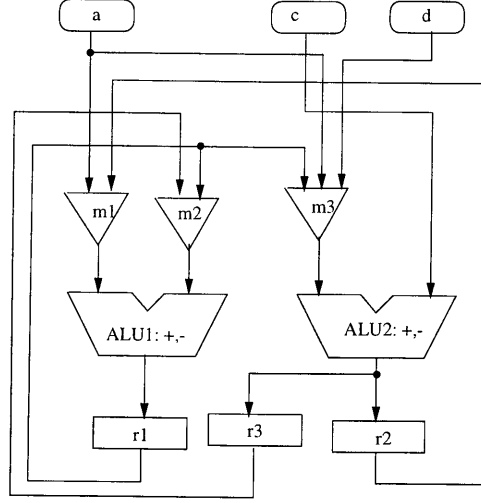


Figure 5.2: Example for ALU reallocation transformations.

Reallocating an operation to another ALU affects the datapath structure. Connections may be changed to reflect the transformation.

The behavior in Figure 5.2(a) and its datapath in (b), are used to illustrate the possible changes. SAT and two OET tables are shown in Figure 5.2 (c). SAT table shows to which ALU an operation is allocated and when it will be executed. The OET tables show the operands for each operation and where the result will be stored. The datapath has two “+−” ALUs, three MUXes and three registers.



A_MOVE(add0,ALU1,ALU2)

Figure 5.3: PART I: Effects of ART on a datapath.

When an operation is moved to a another ALU, the operation's operands must be added to the input list of the that ALU. One of three cases could occur. The first case is when the ALU is already connected to the register where an operation's operand is stored. No new connection is needed in this case. For example, the move *A_MOVE(add0, ALU1, ALU2)*, where the operation *add0* is moved from *ALU1* to *ALU2*. *add0* needs *a* and *c* as inputs. The operand *c* is already connected to the right input port of *ALU2*, so no new connection is needed to the right input port of *ALU2* as shown in figure 5.3.

The second case is when the ALU is not connected to the register where an operand is stored and it already has an input MUX. The size of the input MUX must be increased. For example, in the previous move, the operand *a* is not connected to the left input MUX *m3*, so it is added and the size of *m3* is increased from a 2-to-1 MUX to a 3-to-1 MUX as shown in figure 5.3.

The third case is when the ALU is not connected to the register where an operand is stored and the ALU does not have an input MUX. New MUX

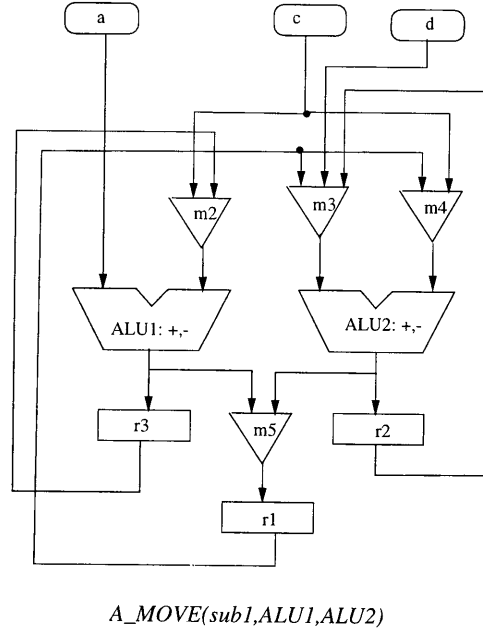


Figure 5.4: PART II: Effects of ART on a datapath.

is added. For example, an input MUX is added, to the right port of $ALU2$ after the move $A_MOVE(sub1, ALU1, ALU2)$ as shown in Figure 5.4. The operation $sub1$'s operands are $sub0$ and $add1$ which are stored in $r2$ and $r1$ respectively. The right port of $ALU2$ is connected only to c , so a MUX is added to connect the right port to $r1$ also.

The operands connections of a moved operation must be removed from the input list of the old ALU. Removing the inputs from the old ALU results in three cases. The first case occurs when the register that store an operand is needed by other operations performed by the old ALU. In this case no changes take place for the old ALU input port. For example, after the move $A_MOVE(add0, ALU1, ALU2)$, no change is made for the left port of $ALU1$ because the connection to register a is not only needed by $add0$ but also by $add1$ operation as shown in Figure 5.3.

The second case is when the old ALU has an input MUX and removing the moved operation operand connection reduces the size of the input MUX. For example, both $A_MOVE(add0, ALU1, ALU2)$ and $A_MOVE(sub1, ALU1, ALU2)$ reduce the right input MUX $m2$ size, as shown in Figure 5.3, from a 3-to-1 MUX to a 2-to-1.

The third case is when the old ALU has an input MUX and removing a moved operations operand connection reduces the input of the MUX to one input. Therefore, this input MUX must be removed. For example, in the move $A_MOVE(sub1, ALU1, ALU2)$, the MUX $m1$ is reduced to one input MUX, so it is removed as shown in Figure 5.4.

The effect of the A_SWAP is similar to the A_MOVE . The A_SWAP is equivalent to two A_MOVE . addition or deleting an ALU does not effect the rest of the datapath because in both transformation no operations are allocated for the added or deleted ALU.

There is another indirect effect of reallocation on the datapath and the controller. Suppose, an operation result is stored in one of the registers attached to the ALU that executes that operation. If an operation is moved to another ALU, there may be no connection between the new ALU and the register where the result of the operation is stored.

Should the storage of an operation result be reallocated to another register attached to that ALU?, or should no reallocation be performed and an output MUX is added to connect the old ALU and the new ALU to that register?. In our transformations, we adapted the rule that an operation variable is stored in the original register unless it is moved explicitly by a register reallocation transformation move. Let the register where the operation result is stored be

REG_o , and suppose, it is connected to the old ALU. There are three possible cases.

The first case occurs when the old ALU is connected directly to REG_o which is used to store only this moved operation variable. In this case, REG_o is disconnected from the old ALU and reconnected to the new ALU. For example in the $A_MOVE(add0, ALU1, ALU2)$, which is shown Figure 5.3, $add0$ result is stored in $r3$ which is connected to $ALU1$ originally. $r3$ is used to store only $add0$ and it is not connected to $ALU2$, so $r3$ is reconnected to $ALU2$ instead of $ALU1$ after the move.

The second case occurs when REG_o is connected directly to the old ALU but REG_o is used for storing other variables in addition to the moved operation variable. In this case the connection to the old ALU must remain. However, REG_o must also be connected to the new ALU. A new output MUX is added to connect both the old ALU and the new ALU to REG_o . For example, in $A_MOVE(sub1, ALU1, ALU2)$, as shown in Figure 5.4, $sub1$ result is stored in $r1$ which is, also, needed to store the result of $add1$. A new output MUX is added to connect both $ALU1$ and $ALU2$ to $r1$.

The third case occurs when the old ALU is connected to register REG_o indirectly through an output MUX. If the new ALU is connected to register REG_o through that output MUX, no changes are needed, otherwise, the output of the new ALU must be added as input to the output MUX. For example, If both $sub1$ and $sub2$ are stored in register $r1$, there will be an output MUX $m5$ in the original datapath as shown in figure 5.5(a). After $A_MOVE(sub1, ALU1, ALU2)$, $r1$ already connected to $ALU2$, so no change is needed as shown in Figure 5.5(b).

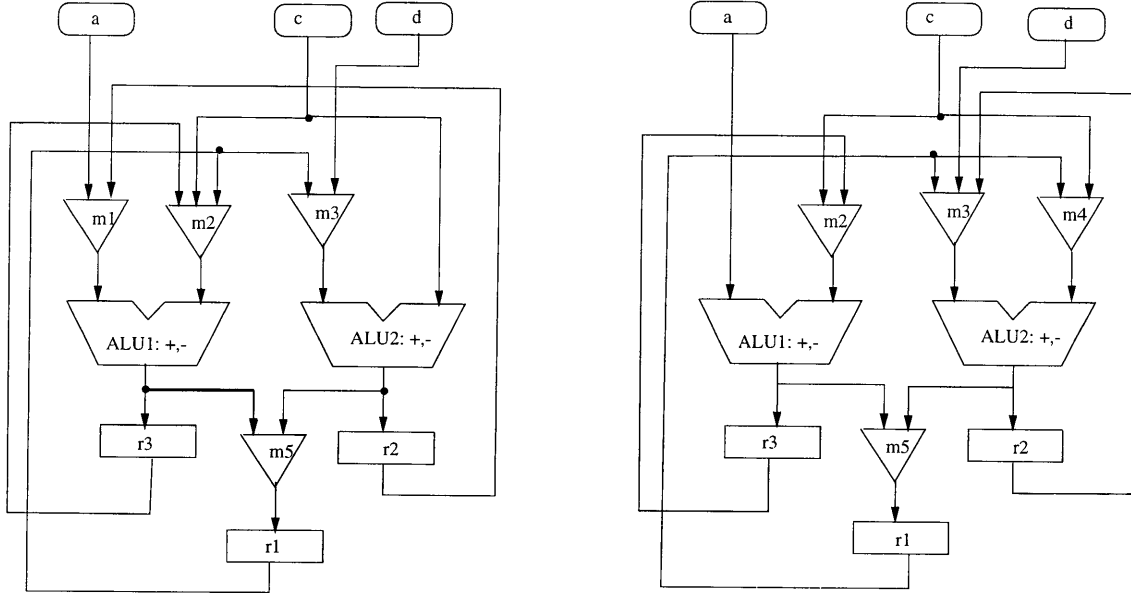
(a): Original datapath with MUX $m5$.(b): Datapath after $A_MOVE(sub1, ALU1, ALU2)$

Figure 5.5: PART III: Effects of ART on a datapath.

The ALU reallocation transformation affects the controller output function only. It does not affect the controller FSM transition function or number of states. When an operation is moved from one ALU to another, the controller output vector at particular state is changed to reflect this move. Since the ART transformation does not effect the schedule of a behavior, the output of only one state is modified. For example, to execute $add0$ in $ALU1$, the first input of MUX $m1$ and the second input of MUX $m2$ must be selected. A load signal, also, needed to load register $r3$ with the result of $add0$. After $A_MOVE(add0, ALU1, ALU2)$ which is shown in Figure 5.3, the enabling signals to execute $add0$ are changed. So, now, a signal to select the second input of MUX $m3$ is needed instead of both $m1$ and $m2$ MUX select signals. The enabling signals for $sub1$, originally, are $m1$ and $m2$ MUX select signal and register $r1$ load. After the $A_MOVE(sub1, ALU1, ALU2)$ which is shown

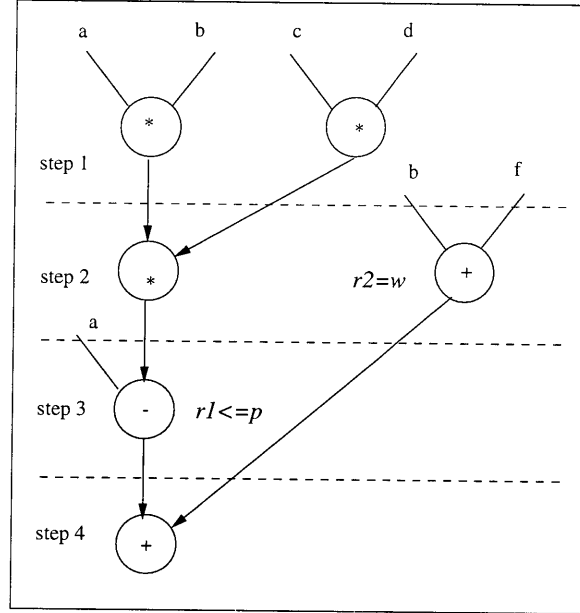


Figure 5.6: The lifetimes of p and w overlap.

in Figure 5.4, two new MUX $m4$ and $m5$ are added. The *subl* operation needs MUX select signals for $m3$, $m4$ and $m5$ and load register signal for $r1$ to be executed in $ALU2$.

5.3 Register Reallocation Transformations (RRT)

The second type of transformation deals with register reallocation. Each operation is associated with a variable in the data flow graph. The life span of a variable begins when it is first computed (WRITE) and end when it is last used (READ) in the behavior. From the perspective of operation p , a variable is alive in the register REG_i at time step t if it is produced by an operation scheduled before p and it will be used by an operation scheduled after p . A register cannot store two variables whose lifetimes overlap.

For example, in the data flow graph shown in 5.6, the variable p cannot be

moved from register $r1$ to $r2$ because $r2$ stores the variable w which is alive from the addition operation in time step 2 to the addition operation in time step 4. The lifetimes of p and w overlap because p is alive from time step 3 to time step 4. A variable cannot live in two registers unless it is copied. During its entire lifetime a variable must be stored in one register.

The following are the four register reallocation transformations:

1. $R_MOVE(p, REG_i, REG_j)$

Assume operation p is executing at time step t . Move the result of the operation p from REG_i to REG_j . Register REG_j must be free for the lifetime of the result of p .

2. $R_SWAP(p, q, REG_i, REG_j)$. Swap the variables p and q between the two registers REG_i and REG_j . Register REG_i must be free during the life of q and register REG_j must be free during the life of p .

3. $R_DELETE(REG_i)$

Remove register REG_i from the datapath. It must be free during the entire schedule time.

4. $R_ADD(REG_i)$

Add a new register REG_i . Some variable must be moved to this register.

To eliminate the need for adding an output MUX which occur as side effect of ART transformation, register reallocation transformation can be applied to move an operation variable to a register which is already connected to the new ALU. To illustrate this, the ART move $A_MOVE(sub1, ALU_1, ALU_2)$, which is shown in Figure 5.4, resulted in datapath with an output MUX. This output

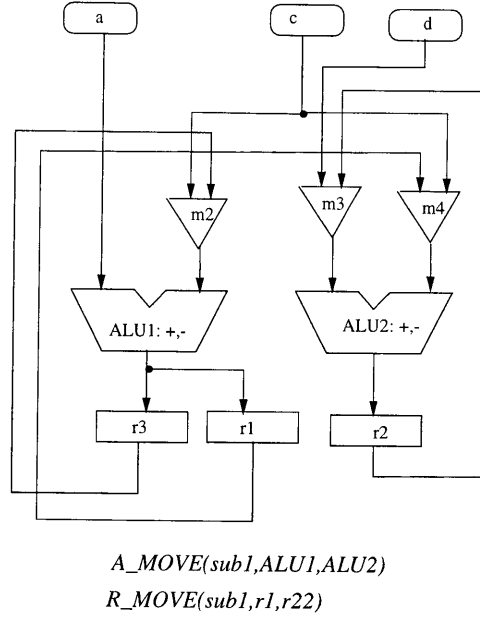
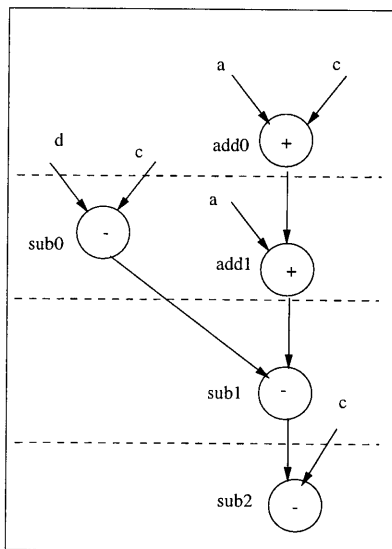


Figure 5.7: Output MUX elimination using RRT.

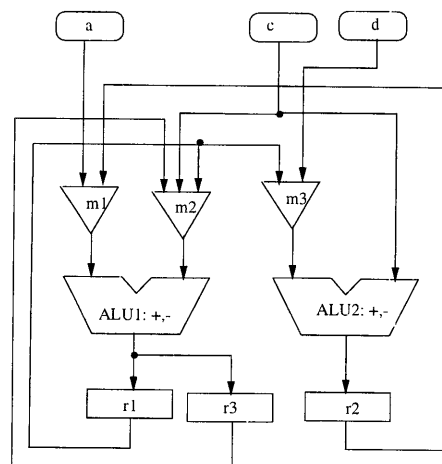
MUX is eliminated after applying, $R_MOVE(sub1, r1, r2)$ as shown in Figure 5.7.

Generally, an output MUX is added when a variable is moved to register which is not connected to the ALU where it is executed. However, moving a variable to another register has additional effects on the datapath, if the result of an operation is needed by another operation as input.

There are three cases, which depend on the dependencies between the operations and the datapath connections. Let $R_MOVE(p, REG_i, REG_j)$ be a RRT move, and let ALU_k execute operation y which uses the result of operation p as operand. Each of the three cases will be demonstrated using the previous example shown in Figure 5.2 which is repeated in Figure 5.8 for clarity. First case occurs when the connection from REG_i to ALU_k is needed only for the execution of y . After the RRT move, the connection from REG_i to ALU_k is changed so that ALU_k is reconnected to REG_j . Thus there is no



(a): Behavior.



(b): Datapath.

	ALU1 +,-	ALU2 +,-
1	add0	
2	add1	sub0
3	sub1	
4		sub2

SAT

	X	Y	R
add0	a	c	r3
add1	a	r3	r1
sub1	r2	r1	r1

ALU1 OET

	X	Y	R
sub0	d	c	r2
sub2	r1	c	r2

ALU2 OET

(c): Operation information tables.

Figure 5.8: Example to show RTT effects on a datapath.

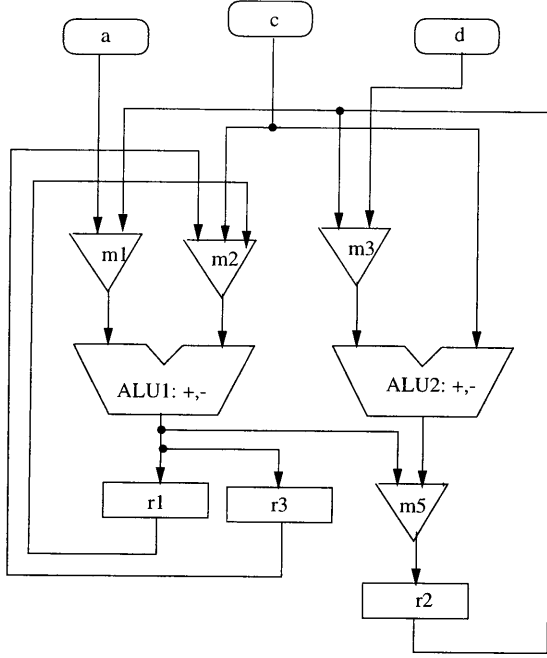
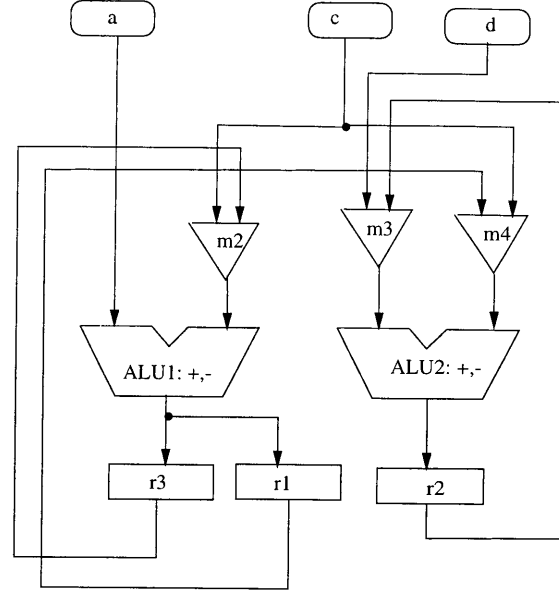
(a): $R_MOVE(sub1, r1, r2)$ (b): $R_MOVE(sub1, r1, r2)$
 $A_MOVE(sub1, ALU1, ALU2)$

Figure 5.9: RRT move resulted with no extra connections.

change in the datapath hardware cost. For example, $sub2$ uses the result of $sub1$ as operand. Originally $sub1$ is stored in $r1$. Since $sub2$ is executed at $ALU2$, there is a connection from $r1$ to MUX $m1$ to supply $sub1$ for $sub2$. The connection to $r1$ is needed only to supply $sub1$. It is not needed by other operations. When $sub1$ is moved to register $r2$ after $R_MOVE(sub1, r1, r2)$, MUX $m1$ is connected to $r2$ instead of $r1$ to supply $sub1$ for $sub2$ as shown in Figure 5.9(a) and no extra hardware is needed. The output MUX $m5$ is needed if $sub1$ result is moved to $r2$ which is attached to $ALU2$ without reallocating $sub1$ to $ALU2$ as in 5.9(a). However, if $sub1$ is, also, moved to $ALU2$ the output MUX is eliminated as in Figure 5.9(b).

The second case occurs when the connection from REG_i to ALU_k is only

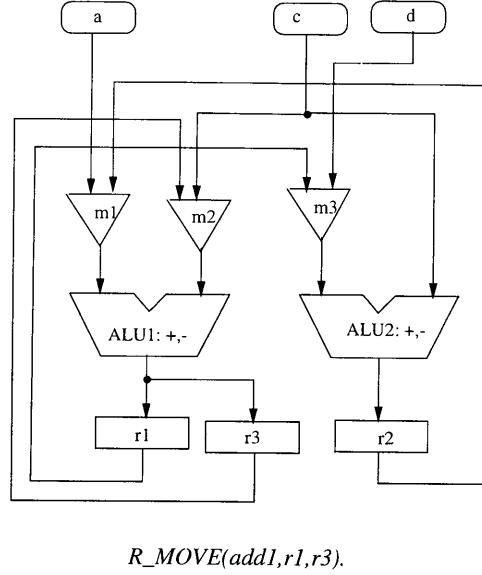


Figure 5.10: RRT move resulted in saving hardware.

needed to execute y and there is already a connection from REG_j to ALU_k (which is needed by other operations). We just have to eliminate the connection from REG_i to ALU_k and save some hardware. For example, $sub1$ uses the result of $add1$ as operand and originally $add1$ is stored in $r1$. The MUX $m2$ is connected to both $r1$ and $r3$. The connection to $r1$ is needed only to supply $add1$ for $sub1$. After the move $R_MOVE(add1, r1, r3)$, which move the result of $add1$ from $r1$ to $r3$. Since $m2$ is already connected to $r3$ and there is no need for $r1$ connection after moving $add1$ to register $r3$, the connection between MUX $m2$ and $r1$ is eliminated as shown in Figure 5.10.

The last case occurs when the connection from REG_i to ALU_k is needed by other operations besides y . We then have to add a connection from REG_j to ALU_k to be able to execute the operation y . This change leads to extra hardware. For example, suppose we have the extra operation $sub3 = a - sub2$ in the data flow graph shown in Figure 5.8. Originally $sub2$ is stored in $r1$.

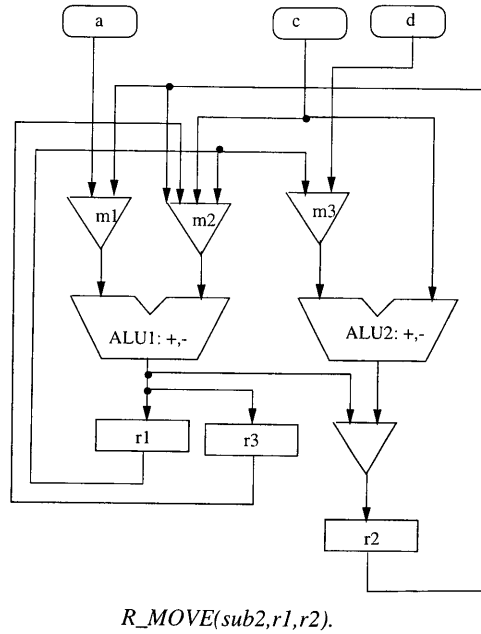


Figure 5.11: RRT move resulted in extra connection.

If we moved the result of *sub2* to be stored in *r2*, extra connection between *ALU1* right input MUX *m2* and *r2* must be added as shown in Figure 5.11 because *m2* is not connected previously to *r2* and this connection is needed to supply *sub2* for *sub3*. The original connection between MUX *m2* and *r1* is kept because it is needed by other operations.

The effects *R_SWAP* is the same as *R_MOVE*. It is equivalent to two *R_MOVE*s. Adding or deleting a register does not effect the reset of the datapath. The connection to an added register occur only when variables are moved to that register. Before a register is deleted, there is no connection to it because no variables are allocated to it.

Reallocation transformations affect the controller output bus width only. Adding, deleting or changing the size of a MUX, affect the controller output size. For example, if an output MUX is added new control signals are needed to control it. There is no effect on the controller when only reconnection is

required because no hardware is added or deleted. If a MUX is deleted, the controller output bus is reduced by one bit. If the size of an old MUX is reduced and the new size is 2^{k-1} where k is the original width of its select line, then a control line is removed from the controller output bus. When a new MUX is added, a new control line is added. If the size of the old MUX increase more than 2^k where k is the width of the MUX select lines originally, a new control line is added to the controller output bus.

5.4 ALU and Register Reallocation Transformation

The following example illustrates the practical usage of both ALU and register reallocation transformations. Suppose we have the datapath shown in Figure 5.12(b). We want to use a single ALU which will do addition and subtraction. The following transformations eliminate the need for a second ALU.

A_MOVE(p, ALU_2, ALU_1)

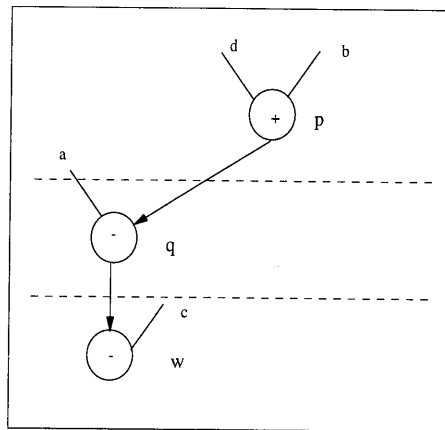
A_DELETE(ALU_2)

R_DELETE(REG_2)

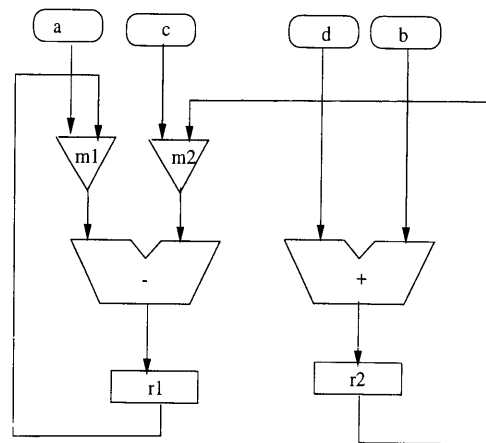
The resulting datapath is shown in Figure 5.12(c).

5.5 Rescheduling Transformations (RT)

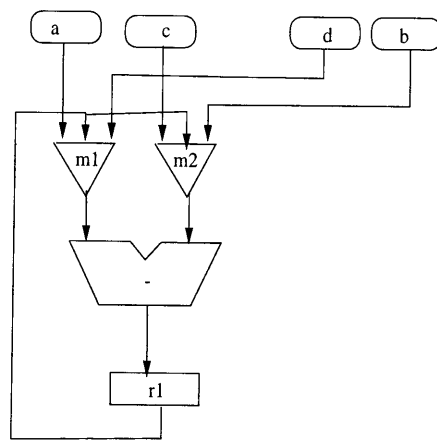
Rescheduling transformations deal with rescheduling the execution time of an operation. An operation can be rescheduled only within its slack time. Slack time is the time steps where an operation can be scheduled without any violation [5], which means an operation can be rescheduled as early as



(a): Data flow graph.



(b): Original datapath.



(c): Final datapath.

Figure 5.12: ART and RRT transformations.

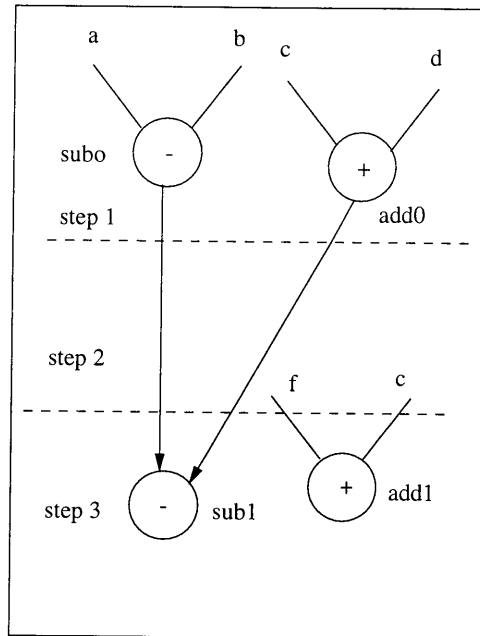


Figure 5.13: An operation can be rescheduled within its slack time.

its inputs are available or as late as one time step before its result is used as input. For example, in Figure 5.13 operation *add1* is scheduled to time step three. It can be rescheduled to any time from time step one to time step three. An operation can be moved across basic blocks as long as this movement is within the operation slack time. The basic blocks involved in the move must be specified.

A rescheduled operation must be executed in the same ALU to which it was allocated before rescheduling. Therefore, the ALU must be idle in the time step to which the operation is moved. Suppose we have one adder and one subtractor for the DFG in Figure 5.13. The operation *add1* can be rescheduled from time step one to the time step three. However, it cannot be rescheduled to time step one because the adder executes *add0*. Therefore, *add1* can be rescheduled only in two time steps two or three.

Rescheduling an operation changes the life span of its variable. For example, assume an operation is originally scheduled to time step t_i and it is moved to time step t_j . If $t_j > t_i$, then the life span of its variable is reduced by $t_j - t_i + 1$ and its new life span is contained within its original life span. Thus, it can be stored in the same register where it was stored before rescheduling. However, If $t_j < t_i$, then the life span of its variable is increased by $t_i - t_j + 1$ and its original life span is contained within its new life span. In this case, the register where an operation result is stored must be free during the entire new life span. The operation cannot be rescheduled to t_j if another variable is stored in that register between time step t_j and t_i . Register reallocation transformation (RRT) must be used to add a new register to store this variable before rescheduling can be done.

For example, suppose we have the following register allocation for the operations allocated to the subtractor and the adder in the DFG in figure 5.13: $r1 = \{sub0\}$, $r2 = \{sub1, add0\}$ and $r3 = \{add1\}$. If the operation $sub0$ is rescheduled from time step one to time step two, the life span of its variable changes from $\langle 1 - 3 \rangle$ to $\langle 2 - 3 \rangle$. The new life span is included in the old life span, so register reallocation is not needed. However, if the operation $sub1$ is rescheduled from time step three to two, the life span of its variable increases from $\langle 3 - 3 \rangle$ to $\langle 2 - 3 \rangle$. We cannot reschedule $sub1$ to time step two because register $r2$ is occupied by the result of operation $add0$. $sub1$ result must be reallocated to a different register using RRT before the operation $sub1$ can be rescheduled to time step two.

The following are the three rescheduling transformations:

1. $S_MOVE(p, t, B_i, B_j)$

Assume the time step t is within the slack time of operation p which is allocated to ALU_k . Move p from basic block B_i to basic block B_j and reschedule it to time step t . ALU_k must be free at time step t in basic block B_j .

2. $S_DELETE(t_x, B_i)$

Delete time step t_x from basic block B_i . No operations must be scheduled to time step t_x in basic block B_i . All time steps greater than t_x must be relabeled so that the new label is one less than the old label.

3. $S_ADD(t_x, B_i)$

Add a new time step t_x to basic block B_i . If t_x is not the last time step, then it is inserted at its location. All time steps that are equal to or greater than t_x must be relabeled so that the new label is one more than the old label.

Rescheduling is only done if the register where an operation result is stored is free during its life span. Therefore, a rescheduled operation is executed in the same ALU at a time step within its slack time and its result is stored in the same register and no changes are made to the datapath. However, changes are made to the controller FSM. If we view the whole FSM as made of connected basic block FSMs, rescheduling does not affect this overall FSM structure, instead, it affects the FSMs of the basic blocks involved. A basic block represent a data flow graph without any control structure, therefore its FSM is a sequencer with no inputs as described in chapter 2. In every clock cycle, an unconditional transition to the next state occurs. Every time step in a scheduled basic block is mapped to one state in its FSM.

For example, let state x map to time step t_x and state y map to time step t_y . Rescheduling an operation from time step t_x in basic block B_i to time step t_y in basic block B_j changes only the output of both basic block FSMs at time steps t_x and t_y . The moved operation enabling signals (MUXes select and register load) are moved from the B_i FSM state x to the B_j FSM state y . Adding a new time step t_z in B_i between t_x and t_y will lead to a new state z to be inserted between state x and state y in B_i FSM. Also, deleting a time step t_z in B_i between t_x and t_y will lead to a deleting the state z which is between state x and state y in B_i FSM.

For the example shown in Figure 5.14(a), let e_p be the enabling signal for operation p . Operation p is executed at time step two of basic block B_1 which maps to state three. After $S_MOVE(p, 2, B_1, B_2)$, e_p is moved to time step two of basic block B_2 which maps to state five as shown in Figure 5.14(b). Figure 5.15(a) and (b) shows the addition and deletion of a time step to the controller FSM in Figure 5.14(a). Part (a) shows the addition of time step four for basic block B_2 and part (b) shows the deletion of time step two of basic block B_2 .

Rescheduling transformations can be used to redesign a circuit. The following example shows one practical usage of rescheduling transformations. The CDFG for this example is shown in Figure 5.16(a). Figure 5.16(b) shows the datapath and Figure 5.16(c) shows the controller for this behavior.

This circuit has two multipliers and one adder, and it needs 5 clock cycles to execute its behavior. A modification is needed for the circuit when only one multiplier and one adder are available. Datapath transformations are used to eliminate one multiplier while preserving the circuit functionality. The multi-

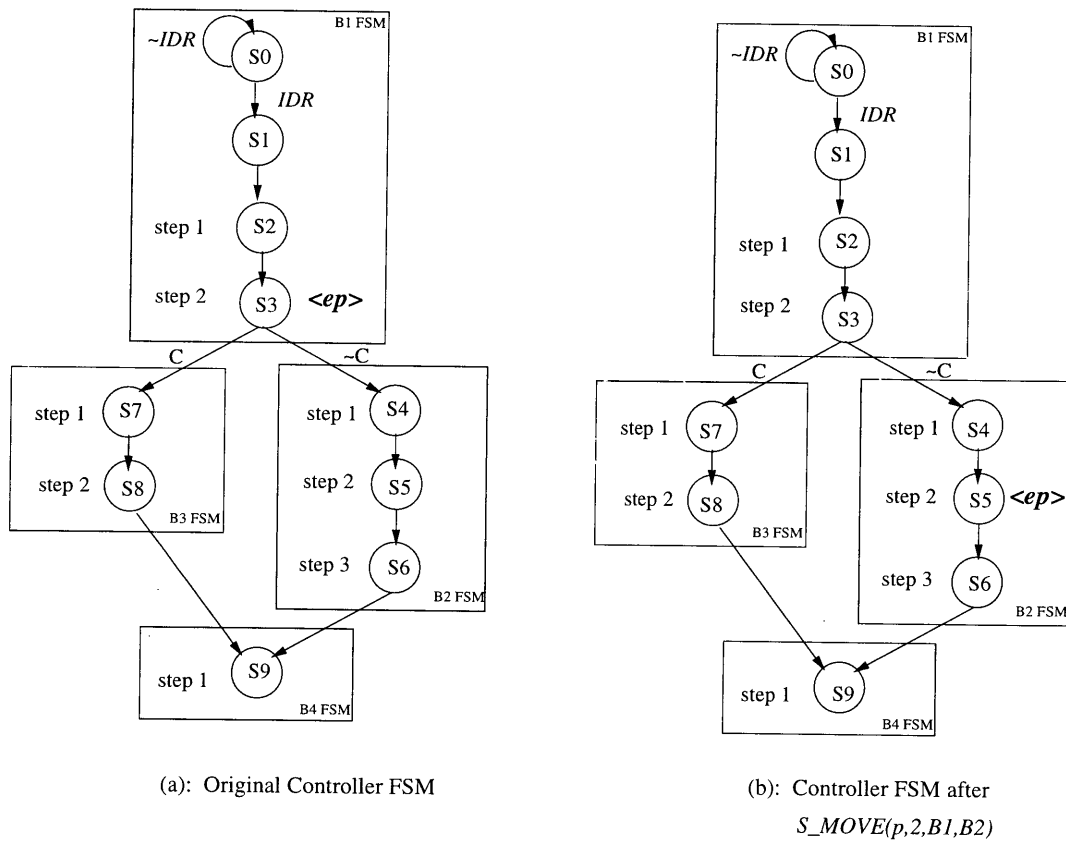


Figure 5.14: PART I: Effect of rescheduling transformation on the controller

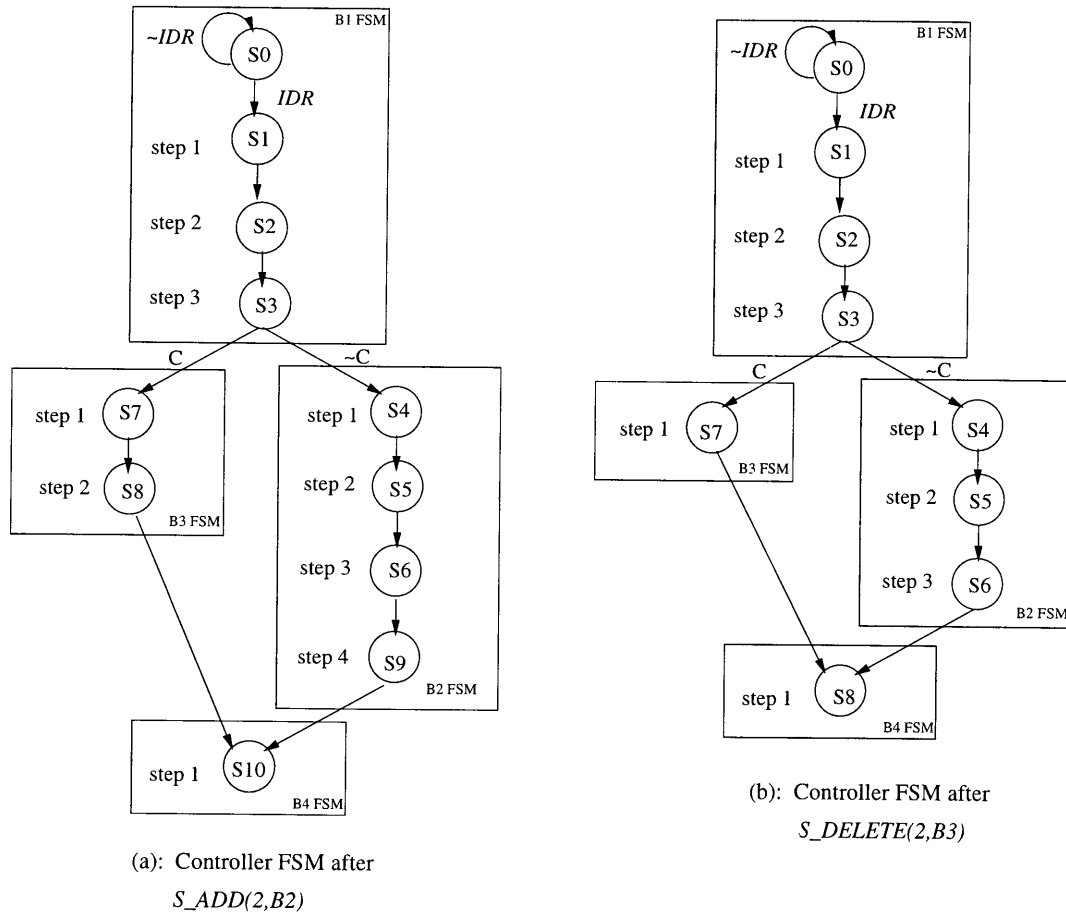


Figure 5.15: PART II: Effect of rescheduling transformation on the controller

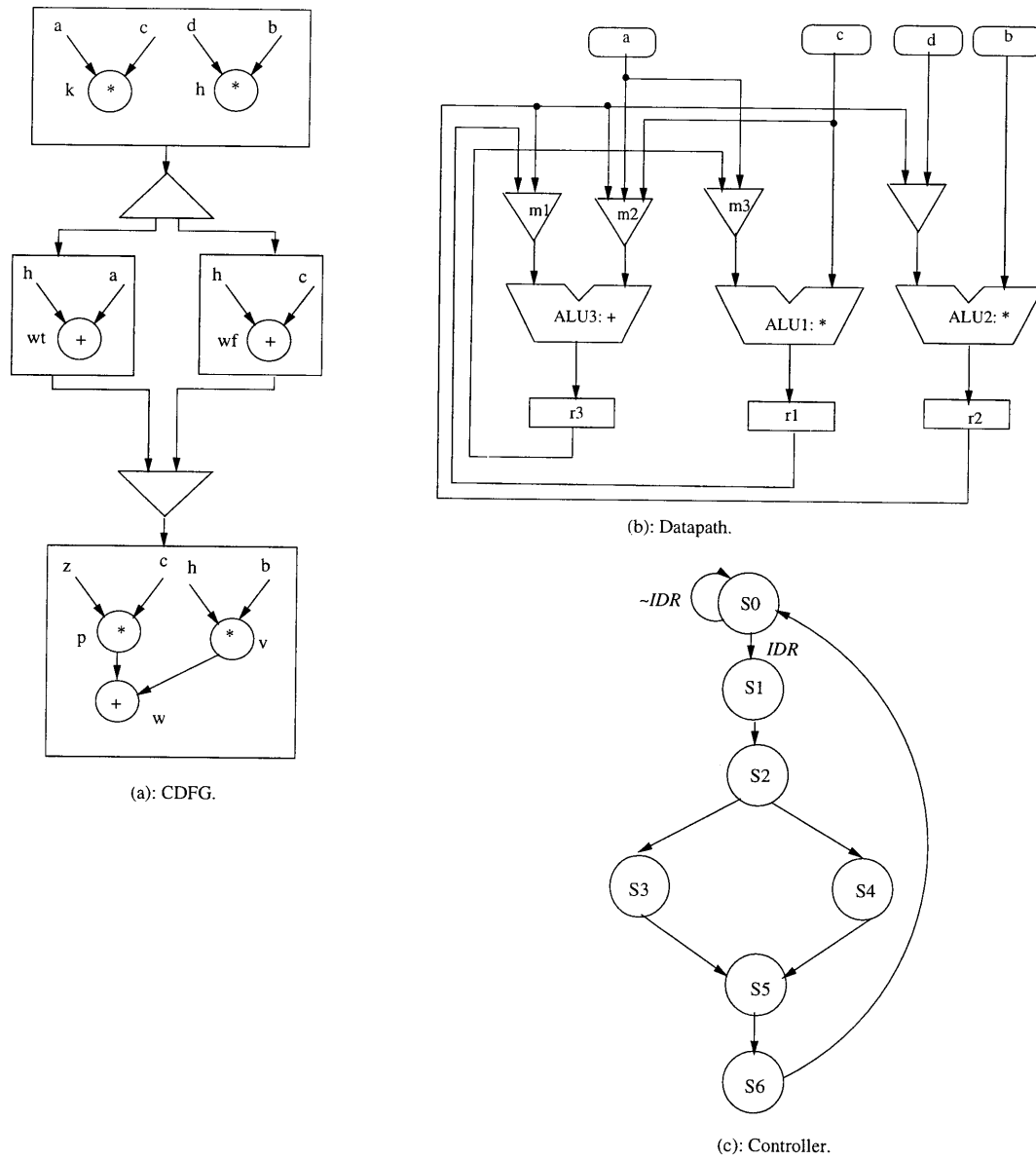


Figure 5.16: RT and ART transformations example.

plier that executes fewer operations is eliminated. *ALU1* has two operations *k* and *p* allocated to it and *ALU2* has also two operation allocated to it *h* and *v*. *ALU2* is chosen randomly and eliminated. However, we cannot use ALU reallocation transformation alone to move *h* and *v* from *ALU2* to *ALU1*. *ALU1* is busy executing *k* and *p* at time steps one and three, which are the same time steps needed for *h* and *v* execution. Rescheduling transformations are used to insert two new time steps and then *h* and *v* are moved to these new time steps. ART moves are then applied to move *h* and *v* to *ALU1*, which is idle at these new time steps. The following transformations are applied to eliminate the multiplier *ALU2*

S_ADD(2,B1)

S_MOVE(h,2,B1,B1)

S_ADD(3,B1)

S_MOVE(v,3,B4,B1)

A_MOVE(h,ALU2,ALU1)

A_MOVE(v,ALU2,ALU1)

A_DELETE(ALU2)

The resulting CDFG, datapath, and controller are shown in figure 5.17

5.6 Experimental results

Experimental results that show the effect of datapath transformations on the circuit the area and timing are presented in this section.

The following example will be used to study the effects of datapath transformations. It is written in SYNTTEST format:

```
c1 := y  0
```

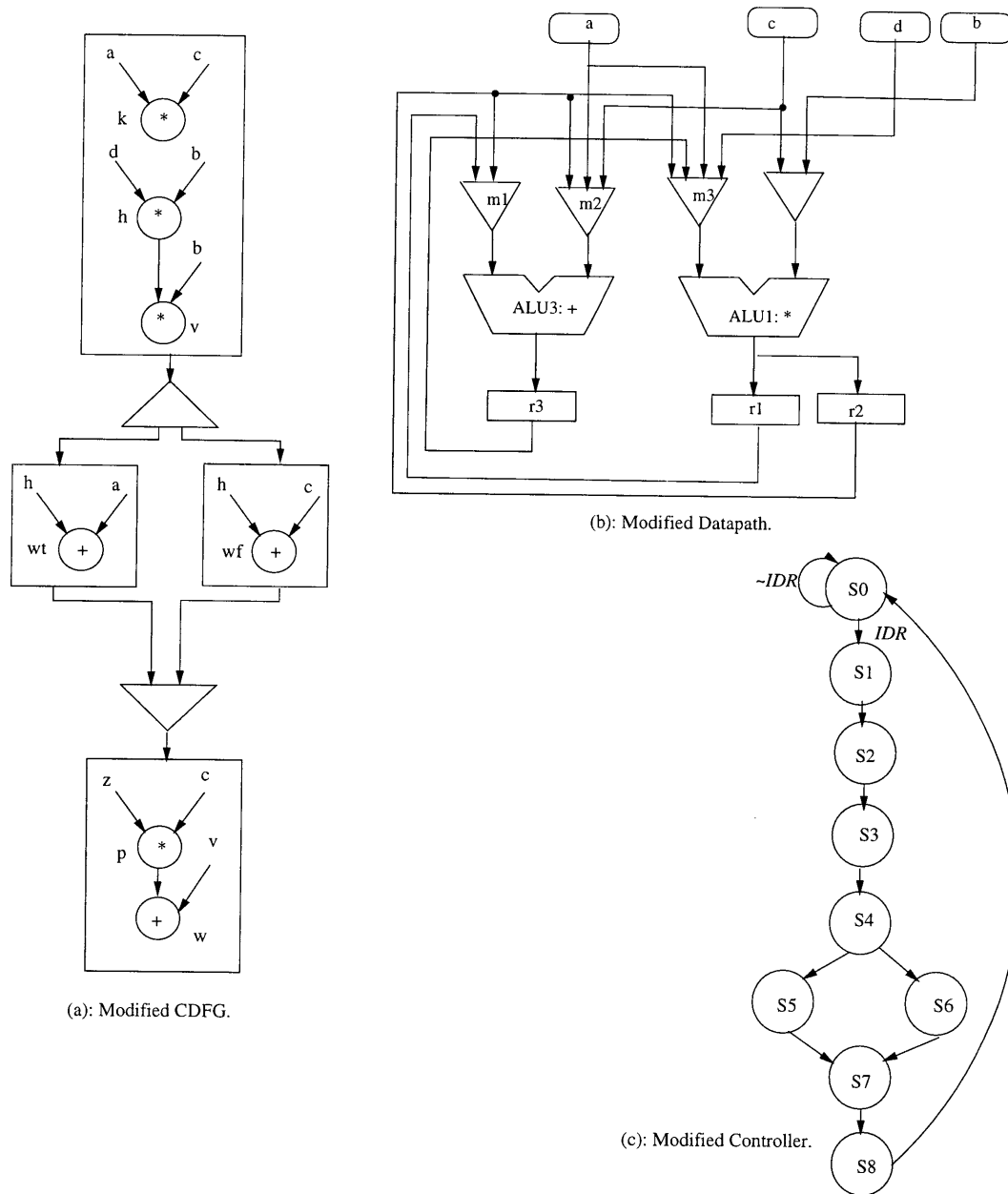


Figure 5.17: Circuit after RT and ART transformations.

```
k := a + b
CONDITION c1
DIVERGE 2
CASE True
t := a + k
ENDCASE
CASE False
c2 := y = 0
CONDITION c2
DIVERGE 2
CASE True
ft := k + k
ENDCASE
CASE False
ff := k + b
ENDCASE
MERGE 1
JOIN ft tf INTO z2
ENDCASE
MERGE 1
JOIN t z2 INTO z1
ENDCONDITION
q1 := a + c
p1 := z1 + q1
q2 := q1 + b
```

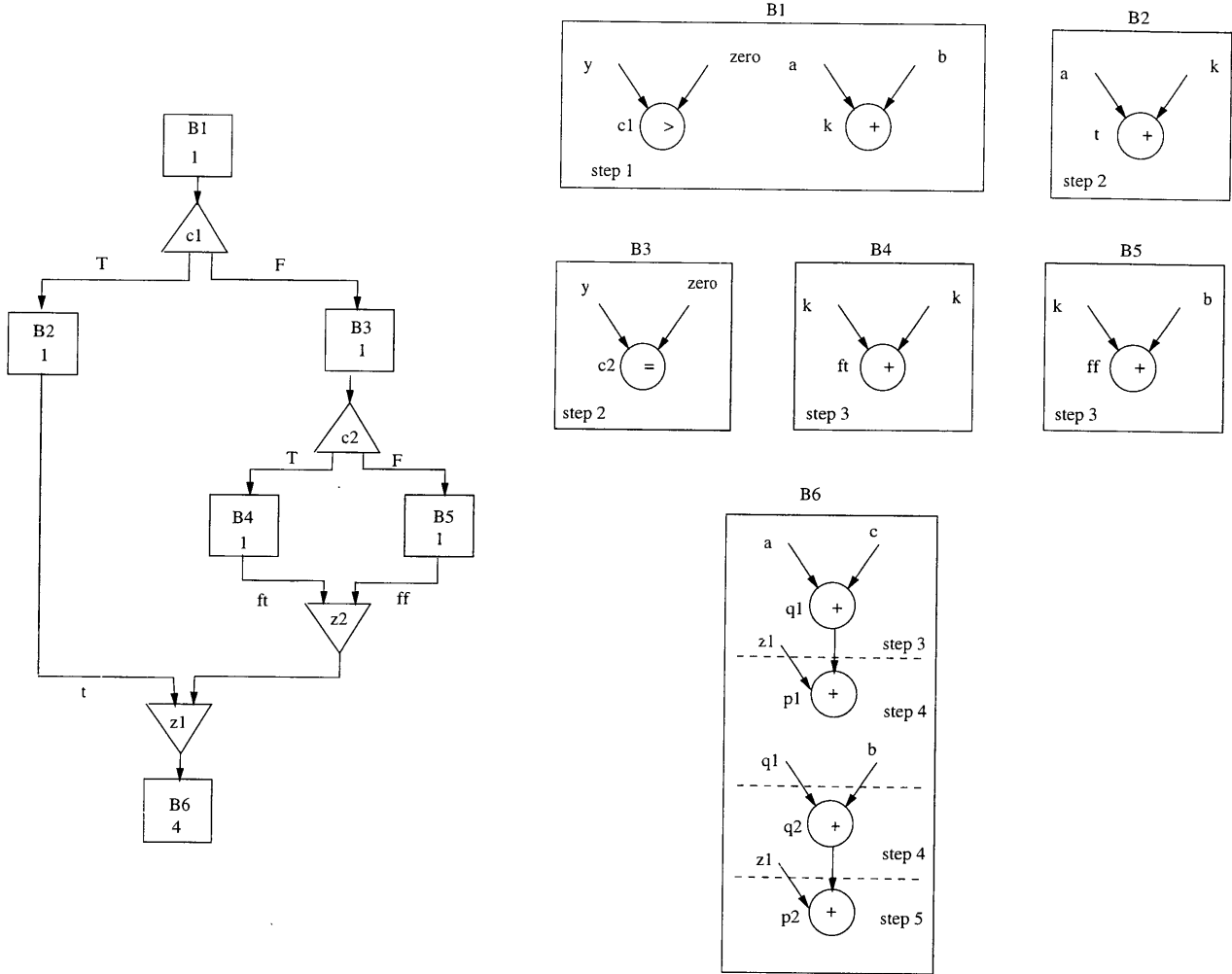


Figure 5.18: A CDFG for datapath transformation.

$p2 := z1 + q2$

The variable $z1$ is used as an alias for three variables t , ft and ff . Since only one of these three variables will be valid, $z1$ represents the valid variable. This behavior CDFG is shown in Figure 5.18 and the datapath is shown in Figure 5.19. The variables t , ft and ff are stored in registers $r1$, $r2$ and $r3$ respectively. So $z1$ is stored physically in three different registers and only one value will be valid according to the values of both conditions $c1$ and $c2$. These two conditions are the result of two operations in the CDFG. If y greater than

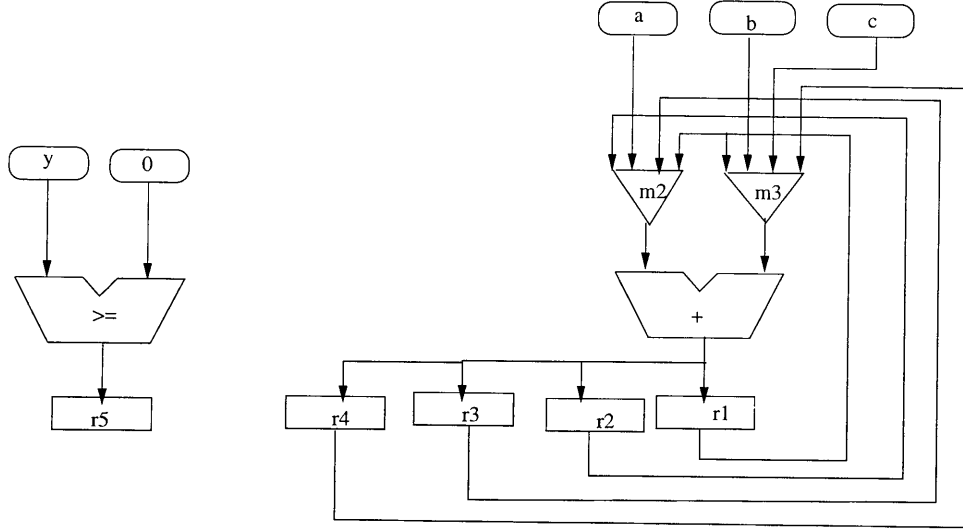


Figure 5.19: The original datapath for the datapath transformation.

zero then $c1$ will be true and the first true branch will be taken, so t will be valid and $z1$ will take the value of register $r1$. If $c1$ is false, the condition $c2$ needs to be resolved. If y is equal to zero then ft will be valid and $z1$ will take the value of register $r2$. If y is less than zero, then ff will be valid and $z1$ will take the value of register $r3$. The two operations $p1 := z1 + q1$ and $p2 := z1 + q2$ use $z1$ as operand. The valid value of $z1$ must be used.

Even though, the adder input MUX $m1$ is connected to all three registers that represent $z1$, the controller must generated the correct MUX select signal so only the valid $z1$ value is used. The controller for this behavior is shown in Figure 5.20. The operation $p1 := z1 + q1$ is executed at state $S8$ if the $c1$ true, at state $S9$ if $c1$ is false and $c2$ is true, and at state $S10$ if both $c1$ and $c2$ are false. The second operation $p2 := z1 + q2$ is executed at state $S12$ if $c1$ true, at state $S13$ if $c1$ is false and $c2$ is true, and state $S14$ if both $c1$ and $c2$ are false.

After implementing this circuit in COMPASS, the number of transistors

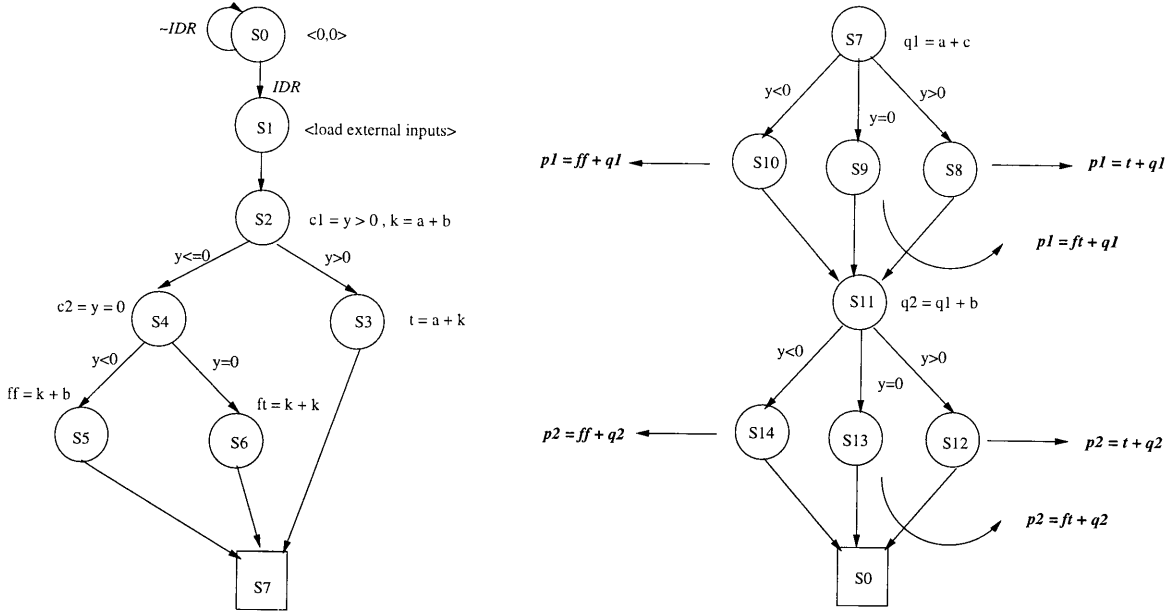


Figure 5.20: controller for the previous CDFG.

of the datapath are 1743 and the controller has 490 transistors. The critical path delay is 21.79 *ns*. These number are shown as the first group in chart 4.1.

5.7 Rescheduling Transformations

The other solution is to use rescheduling transformations to move the previous two operations $p1 := z1 + q1$ and $p2 := z1 + q2$ inside each branch and eliminate the alias variable $z1$. Each operation will be replaced with three operations where the $z1$ operand is replaced with the valid variable. For example $p1 := z1 + q1$ will be replaced with $p1 := t + q1$ after the operation $t := a + k$, with $p1 := ft + q1$ after the operation $ft := k + k$, and $p1 := ff + q1$ after the operation $ft := k + b$. All other operand for the rescheduled operations must be available before their execution, So $q1 := a + c$ and $q2 := a + b$ must be

rescheduled before the condition structure. The following are the rescheduling transformations:

```

S_ADD(2,B1)
S_MOVE(q1,2,B6,B1)
S_ADD(3,B1)
S_MOVE(q2,3,B6,B1)
S_ADD(2,B2)
S_MOVE(p1,2,B6,B2)
S_ADD(2,B4)
S_MOVE(p1,2,B6,B4)
S_ADD(2,B5)
S_MOVE(p1,2,B6,B5)
S_ADD(3,B2)
S_MOVE(p2,3,B6,B2)
S_ADD(3,B4)
S_MOVE(p2,3,B6,B4)
S_ADD(3,B5)
S_MOVE(p2,3,B6,B5)
S_DELETE(1,B6)
S_DELETE(2,B6)
S_DELETE(3,B6)
S_DELETE(4,B6)

```

The new CDFG after rescheduling is shown in figure 5.21

The modified datapath is shown in Figure 5.22 and the modified controller is shown in Figure 5.23.

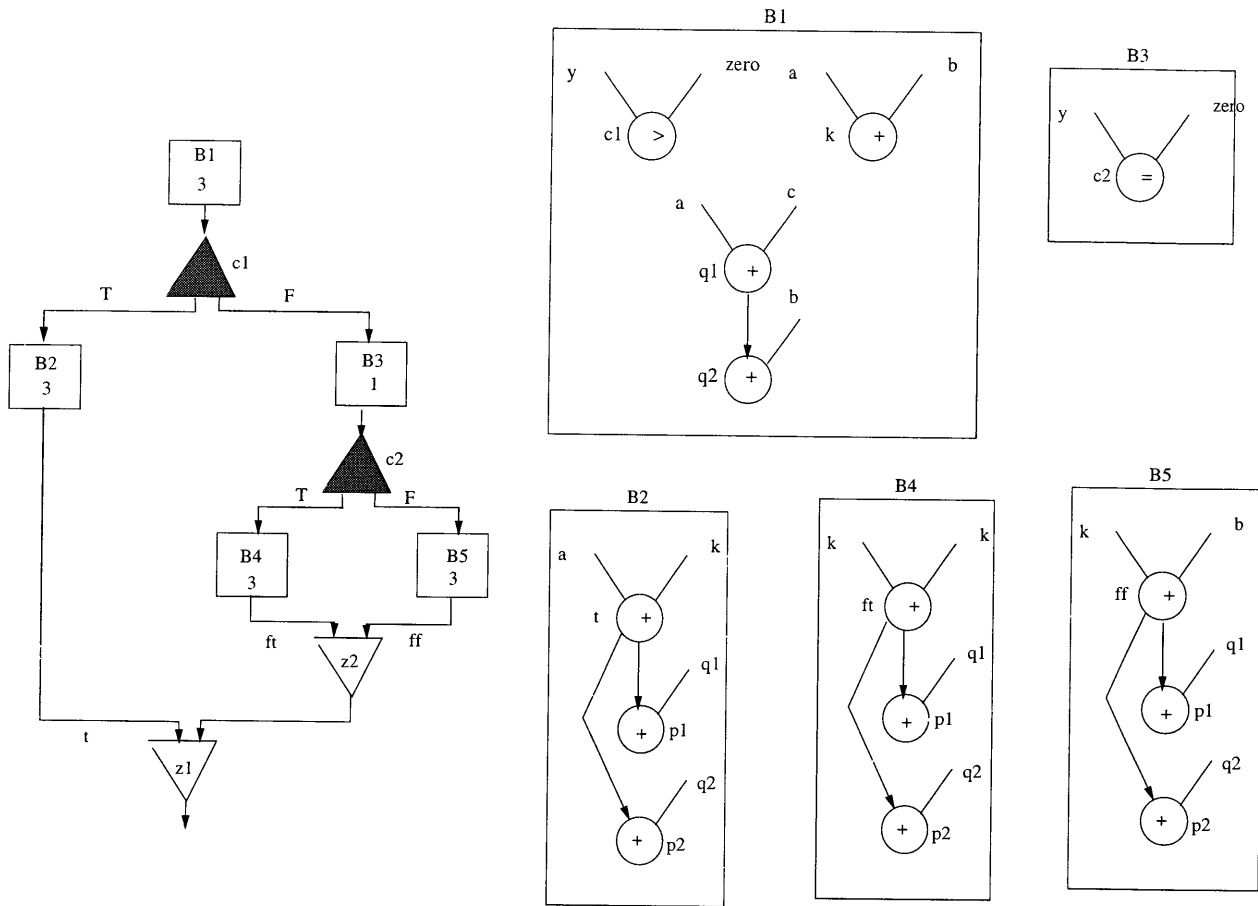


Figure 5.21: the CDFG after rescheduling transformations

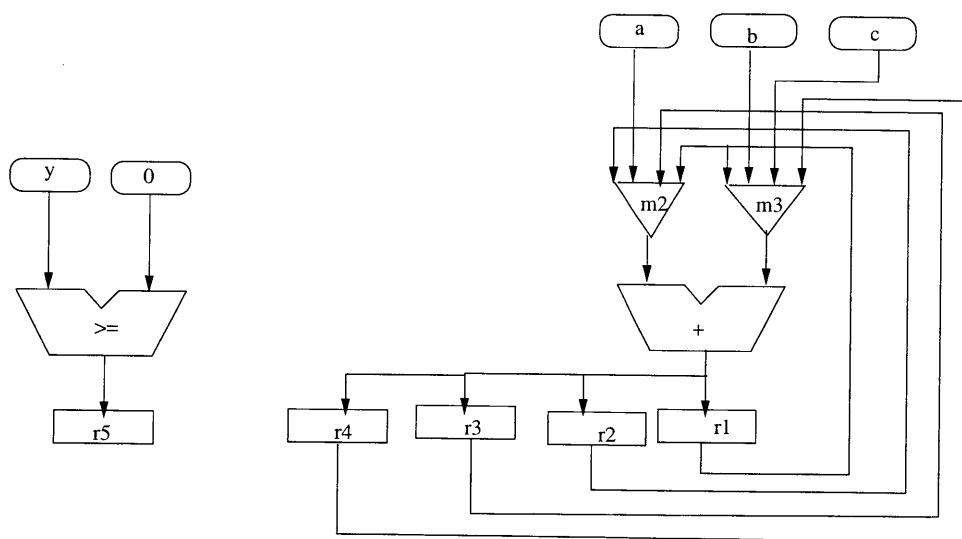


Figure 5.22: Datapath after rescheduling transformations.

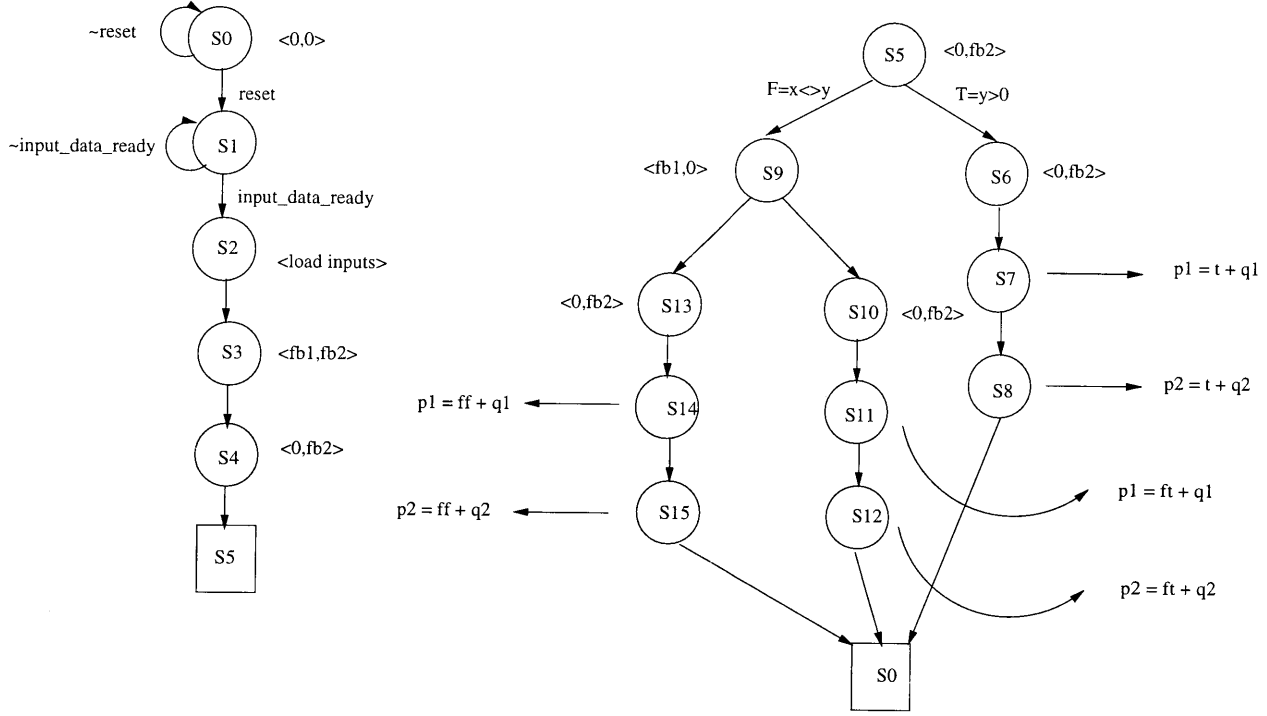


Figure 5.23: Controller after rescheduling transformations.

The modified circuit is implemented in COMPASS. the modified datapath has 2037 transistors and the modified controller has 484 transistors. The critical path delay is 22.51 *ns*. These numbers are shown as the third group in chart 4.1.

5.7.1 Register Reallocation Transformation Solution

Register reallocation transformation can be used to store all variables t , ft , and ff in a single register. As a result, the valid value of $z1$ will be store in one register, so the controller does not have to select between different registers.

The following are the register reallocation transformations:

R_MOVE(ft , REG2, REG1)

R_DELETE(REG2)

R_MOVE(ff , REG3, REG1)

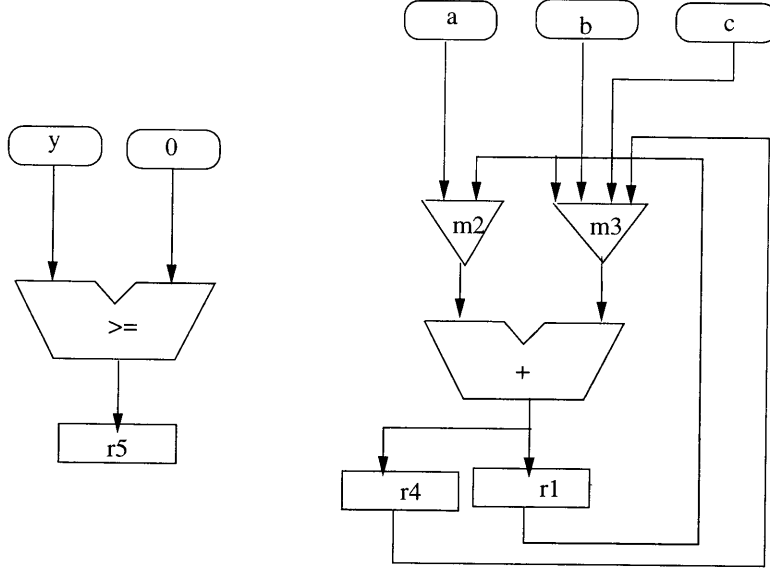


Figure 5.24: Datapath after RRT transformations.

R_DELETE(REG3)

The resulted datapath and controller are shown in Figure 5.24 and in Figure 5.25. The input MUX $m1$ size is reduced from a 4-to-1 MUX to a 2-to-1 MUX. Both register $r2$ and $r3$ are eliminated. The controller is simpler because all $z1$ values are in a single register. Thus, only one state is needed to execute the operation $p1 := z1 + q1$ and one state to execute the operation $p2 := z1 + q2$. Six states are used to execute these two operations in the original controller.

The modified circuit is implemented in COMPASS. the modified datapath has 1407 transistors and the controller has 438 transistors. The critical path delay is 20.52 ns. These numbers are shown in chart 4.1 as second group.

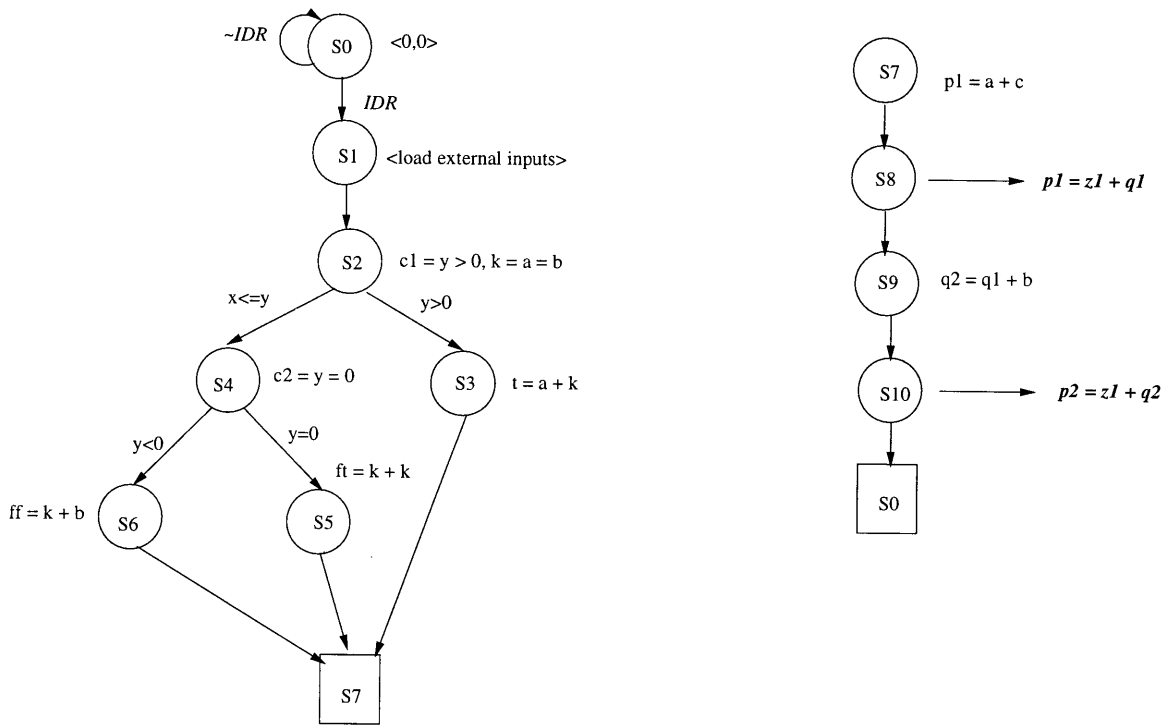


Figure 5.25: Controller after RRT transformations.

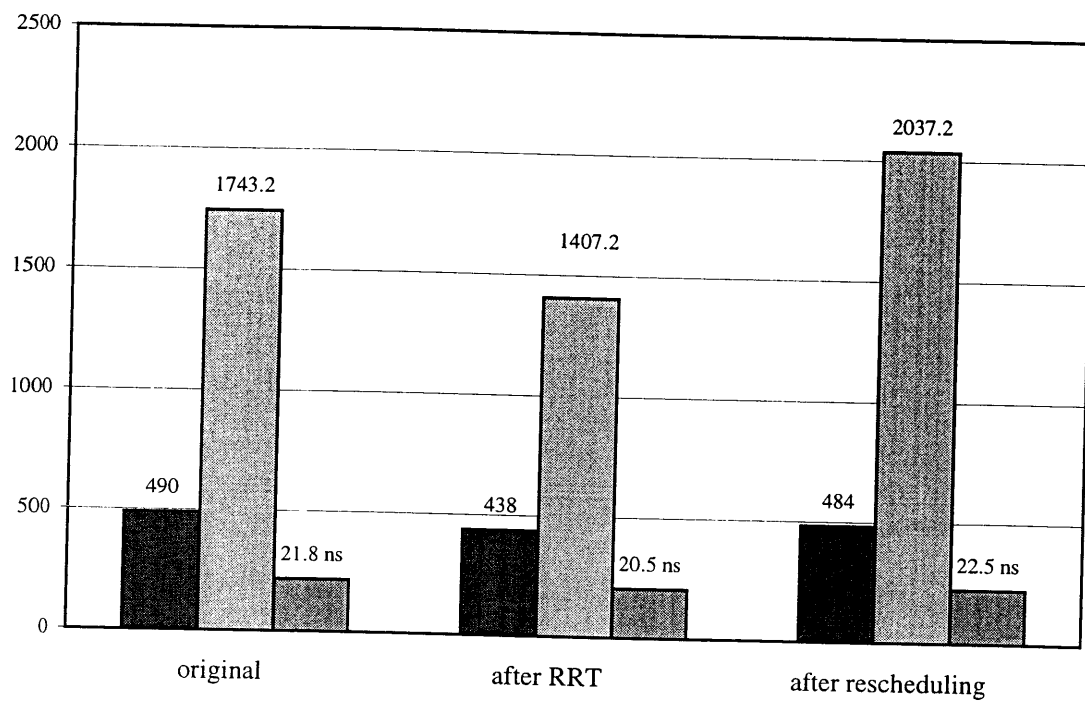
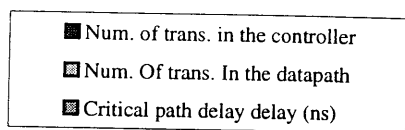


Chart 4.1: Effect of datapath transformation on delay and transistor count



Summary

In this dissertation we studied in detail transformation techniques that apply to redesign of RTL circuits. These transformations apply to RTL description of controller and datapath. The transformations preserve the original circuit functionality, while altering the cost of area, delay and power.

The main objective of controller transformations is to obtain better delay and power consumption, specially at sub-micron level technology. Given an RTL description for a datapath and its centralized controller FSM description, it is transformed to a circuit with a distributed controller. Distributed controller is made up of multiple local controllers performing the same task as the original centralized controller. Distributed controller is generated by partitioning a datapath into functional blocks, and then decomposing the centralized controller into multiple local controllers, each controller for each functional block. Hierarchical layout is used to place each local controller close to its functional block, in order to reduce the overall control wire length.

Two types of distributed controller designs are presented. In the first one, NCLC, where all local controllers do not communicate and synchronization is achieved by letting every local controller having the same states and transition function as the centralized controller. Thus, at any clock cycle all local controllers are in the same state, and at the next cycle all make transitions to the same state. The second design is CLC distributed controller where the local

controllers could communicate. However, local controller which is associated with the most active ALU is chosen to be the master local controller. The master controller has the same FSM structure as an NCLC local controller. In addition, it has extra output: the next state, which is broadcasted to all sleeping local controllers. All idle states in all other local controllers are removed and the remaining states are reconnected using new transition function. A sleeping controller will loop in a sleeping state until it is waken up by the master controller.

Our experimental results show that the critical path delay has been reduced about 8-22% in the distributed controller design compared to the original centralized controller design. This improvement resulted from short control wires after placing each local controller closer to its functional blocks during the hierarchical layout. The improvement on the delay would be better when the design is fabricated using sub-micron technology ($\lambda < 0.5$). As the sub-micron level, gate delay become negligible and the interconnect delay become dominant. Thus, for the circuit to operate at very high frequencies, the distributed controller design is very promising.

The circuit power consumption was not sacrificed to increase the circuit performance. Indeed, our experimental results show that the power consumption is reduced in the distributed controller design. This reduction in power occurs because the distributed controller design has less transistor activities than the centralized controller design. Partitioning the control signals, localize the activities of the datapath to specific functional block. Therefore, functional block activities will occur only at the time steps when operations are executed, while in conventional design the whole datapath is active at all

the time steps.

Decomposing the centralized controller output into fields, simplify each local controller output logic, so local controllers have fewer transistors. This reduction on the number of transistors contributes to reducing both the critical path delay and average power consumption.

In chapter 5, general datapath transformations are discussed. ALU reallocation, register reallocation and rescheduling transformations affect both the datapath and the controller. The reallocation transformations modify the location where an operation will be execute and where its result will be stored. Rescheduling modify when an operation is executed. These transformations alter the area, delay and power of a circuit. The objective of these transformation is to allow the designer to explore other feasible designs which may meet new technology demands.

Future Work

In our future work, we will investigate in more detail the performance and power of the distributed controller at sub-micron technology. How our distributed controller design effects the whole interconnect area. How data transfer wires get effected by our transformations. Also, the effects of number of conditions, number of inputs, number of ALUs, types of ALU, and number of time steps on the delay and power will be investigated. Besides NCLC and CLC new distributed controller designs will be explored. Experiments will be performed to find which distributed controller is better and in which situation. Different partitioning schemes will, also, be investigated. Clustering more than one functional blocks during partitioning will be investigated and clustering criteria will be developed. Moreover, a guidelines for datapath

transformations usage before or after distributed controller transformations to increase performance and reduce power will be investigated.

APPENDIX A

Algorithms

Datapath Partitioning Algorithm

The following is the datapath partitioning algorithm: Input: A datapath RTL description. Output: n functional blocks; n is the number of functional blocks. Let M, C be a datapath where M is the component set and C is the connection set. Let FB_i represents the set of components for functional block i .

1. DO PART I
2. Create n FB s empty partitions, where n is the number of ALUs in the set M .
3. Place each ALU in a unique FB partition.
4. For every ALU in M do
5. Do PART II
6. Let ALU_i be the ALU in the set FB_i . Put all MUXes that make a connection to ALU_i ($MUX_k \rightarrow ALU_i \in C$) in new set called $Q1$.
7. If $Q1$ is empty goto step 13

8. Let MUX_k be a MUX in the set $Q1$. Check if MUX_k is also connected to other ALUs $\forall j MUX_k \rightarrow ALU_j \in C$ and $j \neq i$). Place all of these ALUs in the set $R1$.
9. Check if ALU_i has the largest number of instructions compared to each ALU in $R1$.
10. If ALU_i has the largest number of instructions then add MUX_k to the partition FB_i .
11. $Q1 = Q1 - MUX_k$ (remove MUX_k from the set $Q1$).
12. Goto step 7
13. Do PART III
14. Put all MUXs that have a connection from ALU_i ($ALU_i \rightarrow MUX_k \in C$) in new set called $Q2$
15. If $Q2$ is empty goto step 22.
16. Let MUX_k be in set $Q2$. Check if other ALUs are connected to MUX_k ($\forall j ALU_j \rightarrow MUX_k \in C$ and $j \neq i$). Place all of these ALUs in the set $R2$.
17. Check if ALU_i has the largest number of instructions compared to each ALU in $R2$.
18. If ALU_i has the largest number of instructions then add MUX_k to the partition FB_i .
19. Add all of the registers that have are connected to MUX_k ($\forall j MUX_k \rightarrow REG_j \in C$) to the partition FB_i
20. $Q2 = Q2 - MUX_k$ (remove MUX_k from the set $Q2$).
21. Goto step 15
22. Do PART IV

23. add all registers that ALU_i have a direct connection to them
 $(ALU_i \rightarrow REG_k \in C)$ to the FB_i partition.
24. Repeat step 5-23 with next FB partition.
25. Do PART V
26. Put every MUX which does not belong to any FB partition into
the set $Q3$
27. If $Q3$ is empty STOP
28. Let MUX_k be in set $Q3$. Create a new FB partition and put
 MUX_k in it.
29. Add all of the register that are connected to MUX_k :
 $(\forall j MUX_k \rightarrow REG_j \in C)$ to the newly created FB partition.
30. $Q3 = Q3 - MUX_k$ (remove MUX_k from the set $Q3$).
31. Goto step 27.

APPENDIX B

Wire Delay Calculation

Wire Resistance estimation:

a conduction wire has a resistance $R = (p/t)(l/w)$ ohms [3], where p is the conducting material resistivity, t is the wire thickness, l is the wire length, and w is the wire width (see Figure .1 (a)). The previous formula can be written as $R = R_s(l/w)$ ohms, where R_s is the sheet resistance have units of ohms/square.

Wire Capacitance estimation:

COMPASS “interconnect” utility [49] computes three types of capacitance to get the wire parasitic capacitance. It computes overlap, lateral and fringing capacitances (see Figure .1 (b)). Overlap capacitance is the capacitance between a wire and the substrate or between two overlapping wires (C13o). Fringing Capacitance is the capacitance between the vertical side of one wire and the plane above or below is. lateral capacitance is between to vertical sides of two wires running side by side. The lateral capacitance is computed only if the distance between the wires is less than a defined threshold.

The Interconnect extractor uses the empirical capacitance model described in [9]. The overlap capacitance per unit length is $C_o/E_{ox} = 1.15(w/h)$, where E_{ox} is the dielectric constant of silicon dioxide, w is the wire width, and h is the wire height from the substrate. The fringing capacitance per unit length is $C_f/E_{ox} = 2.80(t/h)^{0.222}$, where t is the wire thickness. the lateral capacitance

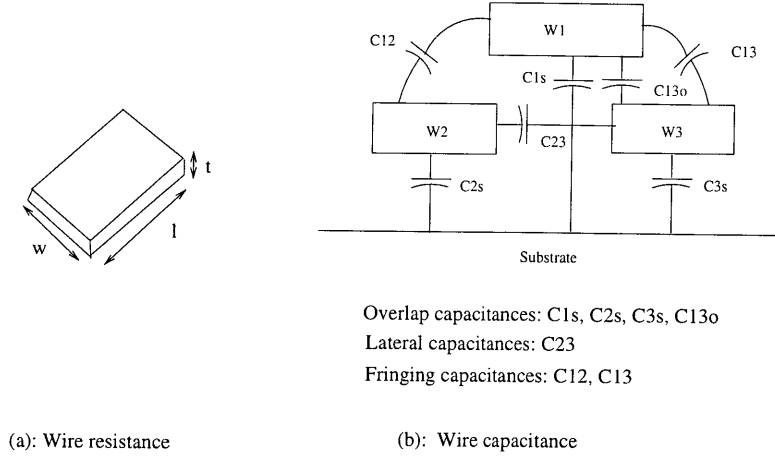


Figure .1: Wire resistance and capacitance

is $C_l/E_{ox} = 0.03(w/h) + 0.83(t/h) + 0.07(t/h)^{0.222}(s/h)^{-1.34}$, where s is the spacing between the wires.

Wire Delay:

The wire delay is given in [3] by $t = 0.7rc l^2/2$, where r is the resistance per unit length, c is the total capacitance per unit length and l is the wire length. The wire length dominate the wire delay.

As the fabrication of digital circuits enters the sub-micron era, more devices are placed on the same area of silicon. The topology of the sub-micron chip is different, the gate area is smaller, and the routing area is larger. Routing density, high frequency, smaller gate, and longer wires makes the wires delay the dominant factor in circuit performance. As chip density increases the controller may be placed further from some control points, leading to longer wires and high delay. Placing the controller closer to each control point reduces wire length, which in turn reduces the wire delay drastically.

APPENDIX C

Studied Behaviors

The following is listing of the eight behaviors that are used to analyze the distributed controller design. These behaviors are written in SYNTTEST format.

Hough Filter Behavior:

DFG Description of : hough

Input : SINE COSINE X Y

Output : IRO

InOut :

Preset Variables :

Preset Values :

Constants : XL YL ROC DRO

Constant Values :

Bit Width : 4

--

$R1 = X - XL$

$R2 = R1 * SINE$

$R3 = YL - Y$

$R4 = R3 * COSINE$

$R5 = R2 + R4$

$R6 = R5 + ROC$

$IRO = R6 * DRO$

AR Filter Behavior:

DFG Description of : arfilter

Input : A B C D E F G H J K L M N O P

Output : ADD4 ADD5 ADD10 ADD11

InOut :

Preset Variables :

Preset Values :

Constants :

Constant Values :

Bit Width : 4

--

MUL0 = A * B

MUL1 = C * D

MUL2 = E * F

MUL3 = G * H

MUL4 = E * J

MUL5 = K * L

MUL6 = M * N

MUL7 = O * P

ADD0 = MUL0 + MUL1

ADD1 = MUL2 + MUL3

ADD2 = MUL4 + MUL5

ADD3 = MUL6 + MUL7

ADD4 = ADD2 + A

ADD5 = ADD3 + B

MUL8 = ADD5 * C

MUL9 = ADD4 * D

MUL10 = ADD4 * E

MUL11 = ADD5 * F

ADD6 = MUL8 + MUL9
ADD7 = MUL10 + MUL11
MUL12 = ADD7 * G
MUL13 = ADD6 * H
MUL14 = ADD6 * K
MUL15 = ADD7 * L
ADD8 = MUL12 + MUL13
ADD9 = MUL14 + MUL15
ADD10 = ADD8 + ADD0
ADD11 = ADD9 + ADD1

Bandpass Filter Behavior:

DFG Description of : bandpass

Input : x0 x1 x2 x3 x4 x5 x6

Output : y20 y21 s6

InOut :

Preset Variables :

Preset Values :

Constants : c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12

Constant Values : 1 2 2 3 2 2 3 4 1 5 1 6

Bit Width : 4

--

$y1 = x4 + x2$

$y2 = x0 - x1$

$y3 = y2 * c7$

$v1 = y1 - y3$

$y4 = c12 * v1$

$y5 = v1 * c11$

$y6 = x5 * c5$

$y7 = x6 - x4$

$v2 = y6 + y7$

$y8 = c10 * v2$

$y9 = x5 + y8$

$s5 = y5 + y9$

$y10 = v2 * c9$

$y11 = y10 + x1$

$s1 = y4 + y11$

$y12 = c4 * x5$

$y13 = x5 * c2$

$y14 = y12 - y13$

```
s2 = x2 + y14
y15 = x4 * c8
s3 = x3 + y15
y16 = x1 - x3
y17 = y16 - x5
y18 = y17 * c6
s4 = y18 + x4
y19 = x1 * c3
y20 = x5 * c1
y21 = y20 + x6
s6 = y19 - y21
```

Wave Filter Behavior:

DFG Description of : wave

Input : inpi rega regb regc regd rege regf regg regh sv2 sv13
sv18 sv26 sv33 sv38 sv39

Output : outp

InOut :

Preset Variables :

Preset Values :

Constants :

Constant Values :

Bit Width : 4

--

```

op3 := inpi + sv2
op32 := sv33 + sv39
op12 := op3 + sv13
op20 := op12 + sv26
op25 := op20 + op32
op21 := op25 * rega
op24 := op25 * regb
op19 := op12 + op21
op27 := op24 + op32
op11 := op12 + op19
op22 := op19 + op25
op29 := op27 + op32
sv26i := op22 + op27
op9 := op11 * regc
op30 := op29 * regd
op8 := op3 + op9
op31 := op30 + sv39

```

```
op7 := op3 + op8
op10 := op8 + op19
op28 := op27 + op31
op41 := op31 + sv39
op6 := op7 * rege
op15 := op10 + sv18
op16 := regg * op15
op35 := sv38 + op28
op36 := op35 * regh
outp := op41 * regf
op4 := inpi + op6
sv39i := op31 + outp
sv2i := op4 + op8
sv18i := op16 + sv18
sv38i := sv38 + op36
sv13i := op15 + sv18i
sv33i := sv38i + op35
```

Discrete Cosine Transform (DCT) Behavior:

DFG Description of : cosine

Input : d0 d1 d2 d3 d4 d5 d6 d7

Output : p1 p2 p0 p6 p5 p3 p7 p4 p2

InOut :

Preset Variables :

Preset Values :

Constants : two

Constant Values :

Bit Width : 4

--

$u_{12} = d_2 + d_5$

$u_{11} = d_1 + d_6$

$u_{13} = d_3 + d_4$

$u_{10} = d_0 + d_7$

$v_{12} = d_2 - d_5$

$v_{11} = d_1 - d_6$

$v_{13} = d_3 - d_4$

$v_{10} = d_0 - d_7$

$v_{21} = u_{12} - u_{11}$

$v_{20} = u_{13} - u_{10}$

$u_{21} = u_{11} + u_{12}$

$u_{20} = u_{13} + u_{10}$

$w_{210} = v_{21} + v_{20}$

$w_{111} = v_{11} + v_{12}$

$w_{110} = v_{10} + v_{13}$

$w_{120} = w_{111} + w_{110}$

$z_3 = two * v_{21}$

$z_2 = two * w_{210}$

```
z4 = two * v20
z1 = two * u21
z0 = two * u20
z5 = two * v12
z6 = two * v12
z7 = two * w111
z8 = two * v11
z9 = two * v11
z10 = two * w120
z11 = two * v13
z12 = two * v13
z13 = two * w110
z14 = two * v10
z15 = two * v10
p6 = z2 + z3
p2 = z2 + z4
p0 = z1 + z0
p4 = z0 - z1
p5l = z5 + z7
p5r = z10 + z14
p5 = p5l + p5r
p3l = z7 + z8
p3r = z10 + z11
p3 = p3l + p3r
p7l = z9 + z10
p7r = z12 + z13
p7 = p7l + p7r
p1l = z6 + z10
p1r = z13 + z15
p1 = p1l + p1r
```

Fast Fourior Transform (FFT) Behavior:

DFG Description of : fft2

Input : fftr_in1 ffti_in1 fftr_in2 ffti_in2

Output : fftr_out1 ffti_out1 fftr_out2 ffti_out2

InOut :

Preset Variables :

Preset Values :

Constants : c1 c2 c3 zero one

Constant Values : 57 29 128 0 1

Bit Width : 8

--

o0 := fftr_in1 < zero

CONDITION o0

DIVERGE 2

CASE True

o1 := ffr_in1 >> one

o2 := o1 | c3

ENDCASE

CASE False

o3 := fftr_in1 >> one

ENDCASE

MERGE 1

JOIN o2 o3 INTO rr

ENDCONDITION

z := rr | rr

o4 := ffti_in1 < zero

CONDITION o4

DIVERGE 2

CASE True

```

o5 := ffti_in1 >> one
o6 := o5 | c3
ENDCASE
CASE False
o7 := ffti_in1 >> one
ENDCASE
MERGE 1
JOIN o6 o7 INTO ri
ENDCONDITION
r0 := c2 * fftr_in1 r1 := c2 * ffti_in1
r2 := c1 * fftr_in2
r3 := c1 * ffti_in2
r4 := r0 - r3
r5 := r4 << one
r6 := r1 + r2
r7 := r6 << one
fftr_out1 := rr - r5
ffti_out1 := ri - r7
fftr_out2 := rr + r5
ffti_out2 := ri - r7

```

Postage Stamp Calculator (POST) Behavior:

DFG Description of : Post

Input : fzone dzone r w w0 f q1 q2 q3 q4 q5

Output : price

InOut :

Preset Variables :

Preset Values :

Constants : zero one two three four six

Constant Values : 0 1 2 3 4 6

Bit Width : 4

--

$\text{delta} = \text{fzone} - \text{dzone}$

$\text{c1} = \text{delta} > \text{zero}$

CONDITION c1

DIVERGE 2

CASE True

$\text{dzt} = \text{delta} - \text{zero}$

ENDCASE

CASE False

$\text{dzf} = \text{zero} - \text{delta}$

ENDCASE

MERGE 1

JOIN dzt dzf INTO dz

ENDCONDITION

$\text{c2} = \text{dz} > \text{zero}$

CONDITION c2

DIVERGE 2

CASE True

$\text{c3} = \text{dz} > \text{three}$

```

CONDITION c3
DIVERGE 2
CASE True
c4 = dz > six
CONDITION c4
DIVERGE 2
CASE True
r1ttt = one + three
ENDCASE
CASE False
r1ttf = zero + two
ENDCASE
MERGE 1
JOIN r1ttt r1ttf INTO r1tt
ENDCONDITION
ENDCASE
CASE False
r1tf = zero + one
ENDCASE
MERGE 1
JOIN r1tt r1tf INTO r1t
ENDCONDITION
ENDCASE
CASE False
r1f = zero + zero
ENDCASE
MERGE 1
JOIN r1f r1t INTO r1
ENDCONDITION
pz = r1 * r
div1 = w - w0
c5 = div1 > f

```

```
CONDITION c5
DIVERGE 2
CASE True
add5 = q1 + q2
add6 = add5 + q3
add7 = add6 + q4
mul1 = add7 * f
mul2 = four * f
sub3 = div1 - mul2
mul3 = sub3 * q5
pwt = mul1 + mul3
ENDCASE
CASE False
pwfz = q1 * div1
pwf = pwfz + zero
ENDCASE
MERGE 1
JOIN pwt pwf INTO pw
ENDCONDITION
price = pz + pw
```

Floating-Point Addition (FPADDS) Behavior:

DFG Description of : fpadds.dfg

Input : a b p q

Output : d s expov expun

InOut :

Preset Variables :

Preset Values :

Constants : ZERO ONE MIN

Constant Values : 0 1 8

Bit Width : 8

--

cx0 := p < q

CONDITION cx0

DIVERGE 2

CASE T

rt := q

tt := q - p

cx1 := q - p

cx2 := a =0 a (compare a with zero)

cx3 := cx1 | cx2 (cx1 or cx2)

CONDITION cx3

DIVERGE 2

CASE T

ctt := b

ENDCASE

CASE F

et := a >> tt (shift right a tt positions)

ctf := b + et

ENDCASE

```

MERGE 1
JOIN ctt ctf INTO ct
ENDCONDITION
ENDCASE
CASE F
rf := p
tf := p - q
cx4 := p - q
cx5 := b =0 b (compare b with zero)
cx6 := cx4 | cx5
CONDITION cx6
DIVERGE 2
CASE T
cft := a
ENDCASE
CASE F
ef := b >> tf (shift right a tf positions)
cff := a + ef
ENDCASE
MERGE 1
JOIN cft cff INTO cf
ENDCONDITION
ENDCASE
MERGE 2
JOIN rt rf INTO r
JOIN ct cf INTO c
ENDCONDITION
cx7 := c @+ c (c xor c)
CONDITION cx7
DIVERGE 2
CASE T
u := c ++ c (count number of leading zeros)

```

```

v := r - MIN
ovt := ZERO
cx8 := v < u
CONDITION cx8
DIVERGE 2
CASE T
dtemp_tt := c
stemp_tt := r
untt := ONE
ENDCASE
CASE F
dtemp_tf := c << u (shift left c u positions)
stemp_tf := r - u
untf := ZERO
ENDCASE
MERGE 3
JOIN dtemp_tt dtemp_tf INTO dtemp_t
JOIN stemp_tt stemp_tf INTO stemp_t
JOIN untt untf INTO unt
ENDCONDITION
ENDCASE
CASE F
unf := ZERO = ZERO
cx9 := r =7 r (compare r with 7)
CONDITION cx9
DIVERGE 2
CASE T
dtemp_ft := c
stemp_ft := r
ovft := ONE
ENDCASE
CASE F

```

```
dtemp_ff := c
stemp_ff := r +1 ONE (increment r by one)
ovff := ZERO
ENDCASE
MERGE 3
JOIN dtemp_ft dtemp_ff INTO dtemp_f
JOIN stemp_ft stemp_ff INTO stemp_f
JOIN ovft ovff INTO ovf
ENDCONDITION
ENDCASE
MERGE 4
JOIN dtemp_t dtemp_f INTO d
JOIN stemp_t stemp_f INTO s
JOIN ovt ovf INTO expov
JOIN unt unf INTO expun
ENDCONDITION
```

Bibliography

- [1] C. Papachristou, H. Harmanani and M. Nourani, "An Approach for Redesign in Data Path Synthesis," in *30th DAC*, 1993.
- [2] J Eppling and D. Thomas, "Multiple Controller Synthesis for Reducing Control Path Delay", in *SRC TECHCON*, 93.
- [3] N. Weste and K. Eshraghian, *PRINCIPLE of CMOS VLSI DESIGN*, Addison-Weseley Publishing company 1993.
- [4] A. Goel, *High-Speed VLSI Interconnections*, John Willey & Sons, Inc. 1994.
- [5] G. DeMicheli, *Synthesis and Optimization of digital circuit*, McGraw-Hill, Inc., 1994.
- [6] C. Gebotys and M. Elmasry, *Optimal VLSI Architectural Synthesis*, Kluwer Academic Publishers, 1992.
- [7] B. Wiederhold, "Delay Calculation Methods Need to Change for Deep-submicron ICs", in *Electronic Design*, 1995, pp. 60-64.
- [8] R. Vivek, "Submicron Delay Calculation for Accurate Timing Analysis", in *Wescon*, 1995, pp. 124-128.
- [9] Sakuria and Tamaru, "Simple Formulas for Two and Three-Dimensional Capacitances", in *IEEE Transactions on Electron Devices*, Vol. 30, No. 2, 1983, pp. 183-185.
- [10] M. Fujita, T. Kakuda and Y. Matsunaga, "Redesign and automatic error correction of combinational circuits," in *IFIP (Logic and Architecture Synthesis)*, 1991, pp. 253-262
- [11] D. Knapp, "Fasolt: A Program for Feedback-Driven Data-Path Optimization," in *IEEE trans. on CAD*, 1992, vol. 11,no.6, pp. 677-695
- [12] S. Prasad, P. Anirudhan and P. Bosshart, "A System for Incremental Synthesis to Gate-Level and Reoptimization following RTL design changes," in *31st CAD*, 1994, pp. 441-446.
- [13] F. Tsai and Y. Hsu, "Data Path Construction and Refinement," in *IC-CAD*, 1990, pp. 308-311.

- [14] V. Chaiyakul, D. Gajski and L. Ramachadran, "High-Level Transformation for Minimizing Syntactic Variances," in *30th CAD, 1993*, pp. 413-418.
- [15] C. Chu, M. Potykonjak, M. Thaler and J. Rabaey, "HYPER: An Interactive Synthesis Environment for High Performance Real Time Applications," in *ICCAD*, 1989, pp. 432-435.
- [16] R. Ang and N. Dutt, "Equivalent Design Representations and Transformation for Interactive Scheduling," in *ICCAD*, 1992, pp. 332-335.
- [17] M. Potkonjak, "Optimizing Resource Utilization Using Transformations," in *IEEE trans. on CAD*, vol. 13, no. 3, pp. 277-291.
- [18] S. Amellal and B. Kaminska, "Functional Synthesis of digital Systems with TASS," in *IEEE Trans. on CAD*, 1994, vol. 13, no.5, pp. 537-552.
- [19] A. Sharma and R. Jain, "Estimating Architectural Resources and Performance for High-Level Synthesis Applications," in *30th DAC*, 1993, pp. 355-360.
- [20] B. Gregory, D. MacMillen and D. Fogg, "ISIS: A System for Performance Driven Resource Sharing," in *29th DAC*, 1992, pp. 285-290.
- [21] F. Tsai and Y. Chin, "STAR: An Automatic Data Path Allocator," in *IEEE Trans. on CAD*, 1992, vol. 11, no.9, pp. 1053-1064.
- [22] M. Rim, R. Jain and R. De Leone, "Optimal Allocation and Binding in High-Level Synthesis," in *29th DAC*, 1992, pp. 120-123.
- [23] N. Wehn, M. Held and M. Glesner, "A Novel Scheduling/Allocation Approach for Datapath Synthesis Based on Genetic Paradigms," in *IFIP (Logic and Architecture Synthesis)*, 1991, pp. 47-56.
- [24] S. Note, W. Geurts, F. Gattthoor and H. De man, "Cathedral-III: Architecture-Driven High-level Synthesis for High Throughput DSP Applications," in *28th DAC*, 1991, pp. 597-602.
- [25] A. Parker, P. Gupta and A. Hussain, "The Effects of Physical Design Characteristics in the Area-Performance Tradeoff Curve," in *28th DAC*, 1991, pp. 530-534.
- [26] R. Camposano, "From Behavior To Structure: High-Level Synthesis," in *IEEE Design of & Test of Computer*, 1990, vol. 0, no.0, pp. 8-19.
- [27] J. Nestor and G. Krishnamoorthy, "SALSA: A New Approach to Scheduling with Timing Constraints," *27th DAC*, 1990, pp. 262-265.

- [28] K. Kucukcakar and A. Parker, "Data Path tradeoffs using MABAL," in *27th DAC*, 1990, pp. 511-516.
- [29] G. De Micheli, D. Ku, F. MailHot and T. Truong, "The Olympus Synthesis System," in *IEEE Design & test of Computer*, 1990, vol. 13, no.5, pp. 37-53.
- [30] J. Lee, Y. Hsu and Y. Lin, "A New Integer Linear Programming Formulation For The Scheduling Problem in Data Path Synthesis," in *26th DAC, 1989*, pp. 20-23
- [31] P. Paulin and J. Knight, "Force-Directed Scheduling in Automatic Data Path Synthesis," in *24th DAC, 1987*, pp. 195-202.
- [32] A. Parker, J. Pizarro and M. Mlinar, "MAHA: A Program For DataPath Synthesis," in *23rd DAC, 1986*, pp. 461-466.
- [33] S. Devadas and R. Newton, "Algorithms for Hardware Allocation in Data Path Synthesis," in *IEEE Trans. on CAD*, 1989, vol. 8, no.7, pp. 768-781.
- [34] S. Ali, S. Sait and M. Benten, "GSA: Scheduling and Allocation Using Genetic Algorithm," in *31st DAC*, 1994, pp. 84-89.
- [35] S. Bhattacharya, S. dey and F. Brglez, "Clock Period Optimization During Resource Sharing and Assignment," in *31st DAC, 1994*, pp. 195-200.
- [36] R. Martin and J. Knight, "Operation Research In The High-Level Synthesis of Integrated Circuits," in *Computers Operation Research*, 1993, vol. 20, no. 8, pp. 845-856.
- [37] P. Marwedel, "Tree-Based Mapping of Algorithms to Predefined Structures," in *30th DAC*, 1993, pp. 586-593.
- [38] R. Dutta, J. Roy and R. Vemuri, "Distributed Design-Space Exploration for High-Level Synthesis Systems," in *29th DAC*, 1992, pp. 644-650.
- [39] I. Park and C. Kyung, "Fast and Near Optimal Scheduling in Automatic data Path Synthesis," in *28th DAC*, 1991, pp. 680-685.
- [40] R. Jain, A. Mujumdar, A. Sharma and H. Wang, "Empirical Evaluation of Some High-level Synthesis Scheduling Heuristics," in *28th DAC*, 1991, pp. 686-689.
- [41] R. Cloutier and D. Thomas, "The Combination of Scheduling, Allocation and Mapping in a Single Algorithm," in *27th DAC*, 1990, pp. 71-76.

- [42] C. Papchristou and H. Konuk, "A Linear Program Driven Scheduling and Allocation Method Followed by an Interconnection Optimization Algorithm," in *27th DAC*, 1990, pp. 77-83.
- [43] C. Huang, Y. Chen, Y. Lin and Y. Hsu, "Data Path Allocation Based on Bipartite Weighted Matching," in *27th DAC*, 1990, pp. 499-504.
- [44] C. Hwang, Y. Hsu, and Y. Lin, "Optimum and Heuristic Data Path Scheduling Under Resource Constraints," in *27th DAC*, 1990, pp. 65-70.
- [45] D. Springer and D. Thomas, "Exploiting the Special Structure of Conflict and Compatibility Graphs in High-Level Synthesis," in *ICCAD*, 1994, pp. 254-257.
- [46] D. Rabaey, C. Chu, P. Hoang and M. Potkonkjak, "Fast Prototyping of Datapath-Intensive Architectures", *IEEE Design and Test of Computer*, in 1991, June, pp.40-51.
- [47] A. Nagle, R. Cloutier and A. Parker, "Synthesis of hardware for the control of digital systems", in *IEEE Trans. on CAD*, 1982, Vol. 1, No. 4, pp.201-212.
- [48] H. Harmanani, C. Papachristou, S. Chiu and M. Nourani, "SYNTEST: An Environment for System-Level Design for Test," in *European Design Automation Conference (EURO-DAC)*, (Hamburg, Germany), Sept. 1992.
- [49] Compass Design Automation, "User Manuals for COMPASS VLSI V8R4.7," Compass Design Automation, Inc., 1995.