Scheduling in RTL Design Methodology

Dongwan Shin and Daniel Gajski

Technical Report CECS-02-11
April 12, 2002

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{dongwans,gajski}@cecs.uci.edu

Scheduling in RTL Design Methodology

Dongwan Shin and Daniel Gajski

Technical Report CECS-02-11
April 12, 2002

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{dongwans,gajski}@cecs.uci.edu

Abstract

In this report, we describe the novel RTL design methodology based on Accellera RTL semantics. We also propose the scheduling algorithm targeting bus-based architecture for the RTL design methodology. The proposed scheduling algorithm is based on resource constrained list scheduling, which considers the number of function units, storage units, buses and ports of storage units in each control step. It supports the pipelined/multicycle operations and storage units, such as pipelined register files and latched memory.

Contents

List of Figures

Scheduling in RTL Design Methodology

Dongwan Shin and Daniel Gajski
Center for Embedded Computer Systems
University of California, Irvine

Abstract

In this report, we describe the novel RTL design methodology based on Accellera RTL semantics. We also propose the scheduling algorithm targeting bus-based architecture for the RTL design methodology. The proposed scheduling algorithm is based on resource constrained list scheduling, which considers the number of function units, storage units, buses and ports of storage units in each control step. It supports the pipelined/multicycle operations and storage units, such as pipelined register files and latched memory.

1. Introduction

With the ever increasing complexity and time-to-market pressures in the design of embedded systems, the designers have moved the design to higher levels of abstraction in order to increase productivity. However, each design must be described, eventually, at the lower levels(e.g. layout masks) through various refinement processes. High-level synthesis has been recognized as one of the major design refinement processes.

The high-level synthesis involves the transformation of behavioral description of the design into a set of interconnected register transfer components which satisfy the behavior and some specified constraints, such as the number of resources, timing and so on. Three major synthesis tasks are applied during the transformation: allocation, scheduling, and binding. Allocation determines the number of the resources, such as storage units, buses, and function units, that will be used in the implemenation. Scheduling partitions the behavioral description into time intervals. Binding assigns variables to storage units(storage binding), assigns operations to function units(function binding), and interconnections to buses(connection binding).

Most works in high-level synthesis have been based on multiplexer-based architecture, in which all data transfers among RT components are achieved through dedicated connections with multiplexers. As the size of a design increases, the performance of the multiplexer-based architecture becomes slower than that of bus-based architecture. We propose new RTL design methodology, which is based on Accellera RTL semantics [Acc01], targeting bus-based architecture. In bus-based architecture, the connection binding is also one of major tasks during refinement processes. we also propose the scheduling algorithm which considers number of ports and number of buses.

The rest of the report is organized as follows: section 2 describes the motivation of our RTL design methodology and refinement tool. Section 3 describes our RTL design methodology and the program flow of the proposed RTL refinement tool. Section 4 takes a closer look at the scheduling algorithm. Section 6 shows the experimental results. Section 7 concludes the report with a brief summary and future work.

2. Motivation

Much research for High-level synthesis [GDLW92] has been going on since 1980s. Currently, many commercial and academical high-level synthesis tools exist in Electronic Design Automation market but the design community wouldn't integrate them into its design methodology and design flow due to the following reasons:

- they can support only several limited architectures like multiplexer-based architecture
- they lack interaction between tools and the designers
- the quality of the generated design is worse than that of manual design.

We must address these problems to have these tools accepted and assimulated in the design community. The proposed RTL design methodogy is based on Accellera RTL semantics, proposed by Accellera C/C++ Working Group [Acc01]. The target architecture for our RTL design methodology is bus-based architectural instead of mux-based architecture due to the reasons discussed above.

The Accellera RTL semantics is well-defined to represent each step of RTL refinement and to provide designers with more controllability to tools because designers can give tools to various constraints during refinement steps. To support interaction between designer and refinement tool, we use the finite state machine with data(FSMD) for RTL description and define 5 styles of RTL description according to the refinement steps.

As already mentioned, the target architecture of our RTL refinement tool is bus-based universal processor architecture [Acc01], in which all RTL components such as function units and storage units are connected through buses to transfer data. Also the function/storage units are pipelined or multi-cycled in our target architecture. The storage units can be composed of registers, register files and memories with different latency and pipeline scheme. In other word, target architecture is heterogenous in terms of storage units. The RT components are connected through the allocated buses and ports of function units and storage units. It makes the refinement problems hard to solve. The scheduling and binding should be extended to integrate pipelined or multi-cycle function/storage units in the target architecture.

3. The architecture of the proposed RTL refinement tool

This section describes the RTL semanatics [Acc01] and the architecture of the RTL refinement tool.

3.1. The Accellera RTL semantics

The RTL design can be modeled by Finite State Machine with Data(FSMD) [Acc01], which is FSM model with assignment statements added to each state. The FSMD can completely specify the behavior of an arbitrary RTL design. The variables and functions in FSMD may have different interpretations which in turn defines several different styles of RTL semantics.

The proposed RTL semantics by Accellera [Acc01] has 5 different styles of RTLs: unmapped RTL, storage mapped RTL, function mapped RTL, connection mapped RTL and exposed-control RTL. The unmapped RTL specifies the operations performed in each clock cycle with explicitly modelling the units in the component's datapath and is obtained by scheduling the operations into clock cycles. The exposed-control RTL explicity models the allocation of RTL components, the scheduling of data transfers into clock cycles, and the binding of operations, variables and assignments to functional units, storage units, buses and has explicitly exposed controllers. The storage mapped RTL models the binding of variables to storage units. The function mapped RTL specifies the binding of functions and variables to functional units and storage units respectiviely. The connection mapped RTL models the connections between functional units and storage units.

These models can also represent the refinement steps like scheduling, storage binding, function binding and connection binding in RTL refinement from unmapped RTL to exposed-control RTL. However, due to the interdependence of scheduling, allocation, and binding, the order of three steps should be able to interchanged to get better design.

3.2. The architecture of the RTL refinement tool

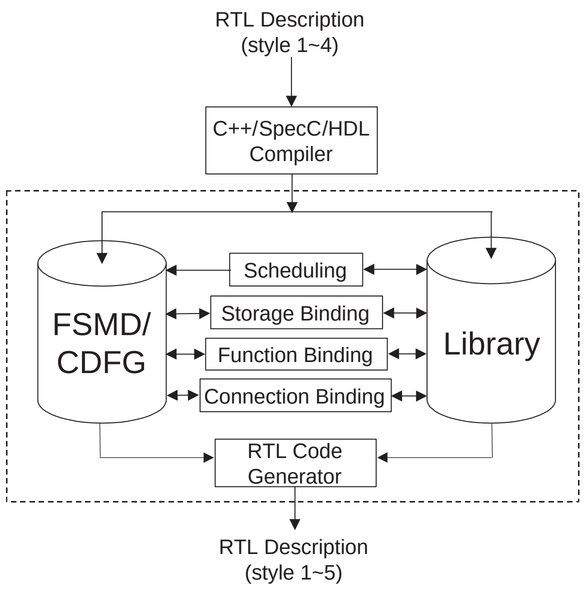


Figure 1. RTL design refinement flow

The Figure 1 describes the RTL refinement flow in RTL design environment. The RTL refinement tool uses the FSMD/CDFG as the internal data structure to read, write, and refine the design models. To get the FSMD/CDFG data structure, we uses the C++/SpecC/HDL as input [Acc01]. The RTL refinement tool also reads the RTL description in C++/SpecC/HDL and generates FSMD/CDFG as internal representation for refinement. Every refinement step is based on FSMD. The SpecC RTL generator makes each style of RTL description as the result of each refinement step. For example, RTL refinement tool reads the unmapped RTL SpecC code and generates FSMD/CDFG and performs storage binding and then generates the storaged mapped RTL description.

The FSMD/CDFG is FSMD represenation, in which each state has state transition information and its own Control/Data Flow Graph(CDFG). The task of scheduling is to divide one state into sub-states based on resource constraint. The storage binding, function binding and connection binding are performed considering every state transition.

The netlist mapper generates the exposed-control RTL in HDL or SpecC language from style 4 RTL. The style 5 exposed-control RTL in HDL can be used as input for gate-level synthesis like Synopsys Design Compiler.

The RTL component library has the information about datapath modules such as ALU, multiplier, register file, memory and bus. It is also written in SpecC language. When each synthesis step is performed, it refers the RTL component library to get the information about resource constraint. The RTL refinement tool reads and maintains the RTL component library.

3.3. Target Architecture

Our architecture is shown in Figure 2. It's composed of function units, register files, memories, multiplexers, buses and bus drivers. Function units performs operations such as multiplication, addition, and so on. Storage units stores data from function units or other storage units through the buses. All data transfers are achieved through buses. The function unit can have the registers at the input and output of it. The controller will determine the next state of the execution based on status signals and input signals, and generate the control signals for datapath, which will be implemented to FSM. The datapath, control signals and status signal can be pipelined by inserting registers.

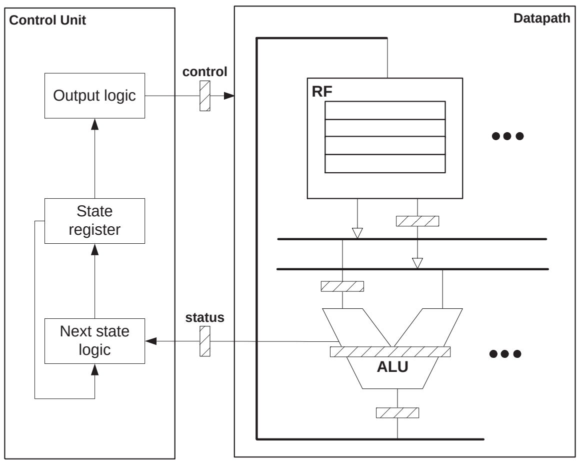


Figure 2. Bus based architecture

4. Internal representation for RTL design methodology

The RTL design is represented by FSMD, which has a set of states and transition among them. Each state has state transition information and its own CDFG.

4.1. Control/Data Flow Graph

This section describes the CDFG representation, which is selected for internal data representation for RTL design. The CDFG is the hierarchical graph which has the data flow information to describe the operations and their depepencies and has the control flow information which is related to branching and iteration constructs. The CDFG has been used for the internal representation of high-level synthesis tool since mid-1980s and has many variations. It can be hierachical or non-hierarchical, polar or non-polar, and cyclic or acyclic.

We made the novel CDFG structure to represent the RTL description and to perform the RTL refinement steps. Our CDFG is hierarchical, acyclic polar graph, which is shown in Figure 3. The acyclic graph makes it easy to implement the graph algorithm, because it has no loop. The polar graph has the single-entry and single exit property using no-operation(source node/sink node in our graph) and makes it easy to build hierachical graph. The node *S* in Figure 3 represents the no-operation node. The top *S* is the source node and the bottom *S* is the sink node. In this graph, the

```
// Behavioral RTL(style1)
a=b+c;
for(I = 0; I < 32; I=I+1)
   d[I:I] ^= e[I:I];
if (start == 0)
   state = S0;
else
   state = S1;
```

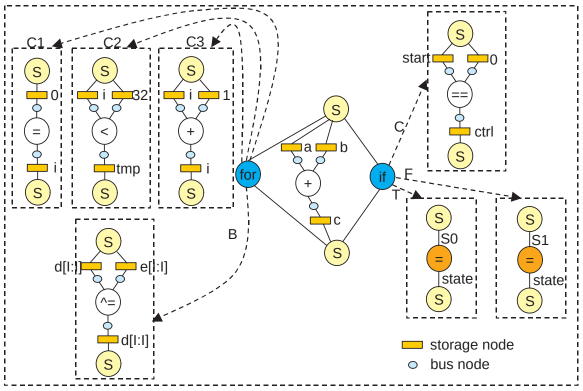


Figure 3. CDFG for unmapped(style 1) RTL description)

edge has the dependency information between nodes such as control dependency and data dependency. The node has all informations except the flow information. The node is decomposed of the non-hierachical node and the hierarchical node. The non-hierachical node has the datapath operation information such as operation node to perform arithmetic/logic operation, storage node to store the data, bus node to transfer the data between functional unit and storage unit, control node to generate the status information of datapath, and state transition node to store state transition information in finite state machine. In Figure 3 shows the operation node which is the white circle node, storage node which is the shaded rectangular node, bus node which is the small shaded circle node between operation node and storage node. The hierachical node is divided to the module node to represent the structural hierarchy in the RTL description, branch node to represent branching information and loop node to represent the iteration information. The branch node(`if` node) and loop node(`for` node) are shown in Figure 3.

5. Scheduling Algorithm

In the proposed RTL design methodology, the scheduling plays a major role in refining from behavioral RTL to exposed-control RTL by re-scheduling each state in FSMD in the behavioral RTL description. The scheduling algorithm is divided into two layers: one is state scheduling and the other is CDFG scheduling.

The state scheduling determines the order of each state in FSMD and reflects resource utilization tables of the already scheduled states to next states which are affected by scheduling result of predecessor states. For example, if the states have multicyle operations and pipeline operations with more than 1 cycles delay, the next states will be affected by these previous states.

The CDFG scheduling is to schedule the operations in each state based on resource utilization table which is determined by resource allocation and scheduling result of the previous states. The CDFG scheduling will be done in the corresponding CDFG in each state. Because resource allocation like number of FUs, the ports of storage units and buses, is given by the designer, the resource-constrained scheduling should be done.

In addition, our scheduling algorithm performs component type selection. The aim of this task is reduce the number of states at minimal hardware cost. Our scheduling algorithm allows for resources to be shared amongst multiple operations, while component selection allows a mixture of fast and slow components to be used in the design. The components are selected such that the fast and expensive components are used for critical operations, and the slower ones are used for non-critical operations.

5.1. Problem Definition

Given:

1. A behavior represented by state transition graph, *STG(S, T)*, where *S* is state in FSMD and *T* is state transition among states.
2. Each state *S* contains hierarchical control/data flow graph, *CDFG(V, E)*, where *V* is a set of vertices representing operations, storages, buses, and hierarchical nodes such as branch and loop, and *E* is dependency between nodes.
3. A component library containing functional units, storage units and buses characterized by type, area, delay, pipeline states and so on. In addition, storage units have the number of read/write ports.
4. clock period and resource allocation, such as number of function units, storage units, buses and read/write ports of storage units.

Determine:

1. control step of each node in a behavior
2. type selection for each node but hierarchical node

Such that:

1. the number of control steps is minimized.
2. the resource allocation constraint is satisfied.

5.2. Proposed Scheduling Algorithm

As already mentioned, the proposed scheduling algorithm is divided into two layers: state scheduling and CDFG scheduling which are shown in Algorithm 1 and Algorithm 2 respectively. The state scheduling algorithm uses breadth-first search to find next state to be scheduled in FSMD. During state scheduling, resource utilization table for each state is updated by considering the resource utilization table of scheduled predecessor states. Each state calls the LIST_RC scheduling algorithm to schedule nodes in CDFG. In state scheduling algorithm, we use candidate list and resource utilization table as following.

Algorithm 1 State scheduling(STG, R_o): state scheduling algorithm

```
1:
2:  $S_0$ = GetResetNode($STG$);
3:  AppendState($S_s$, $S_0$);
4:  while ($S_s$ is not empty) do
5:      $s$ = RemoveFrontState($S_s$);
6:      $s_o$ = $s$;
7:      $R_s$ = GetResUtilTable($R_o$, GetPredStates($s$));
8:      LIST_RC($G_s$, $R_s$);
9:      if ($s$ != $s_o$) then
10:         AppendState($S_s$, GetSuccStates($s$));
11:     end if
12: end while
```

Algorithm 2 LIST_RC(*CDFGG*, $R_{l,k}$): List scheduling algorithm

```
1:  Initialize the unfinished operations $U_{l,k} = \{\}$;
2:  Initialize the step $l = 0$;
3:  SetDelay($v_0$, 0);
4:  ScheduleNode($v_0$, $l$);
5:  UpdateReadyNodes($S_{l,k}$, $v_0$, $l$);
6:  **while** ($S_{l,k}$ or $U_{l,k}$ is not empty) **do**
7:     $v_k = GetSchedulableNode(G, S_{l,k}, R_{l,k})$;
8:     **if** ($v_k$ == NULL) **then**
9:        $l = l + 1$;
10:       UpdateReadyNodes($l$);
11:       UpdateUnfinishedNodes($l$);
12:       UpdateScheduledNodes($l$);
13:       continue;
14:    **else if** ($v_k$ is a bus/port/ctrl node) **then**
15:       SetDelay($v_0$, 0);
16:       ScheduleNode($v_0$, $l$);
17:    **else if** ($v_k$ is a branch node) **then**
18:       $d_c$ = LIST_RC(conditional subgraph of $v_k$, $l$, $R_{l,k}$) - $l$;
19:       $d_b$ = LIST_RC(false/true subgraph of $v_k$, $l + d_c$, $R_{l,k}$) - $l - d_c$;
20:       SetDelay($v_k$, $d_c + d_b$);
21:       UpdateResUtilTable($v_k$, $R_{l,k}$);
22:    **else if** ($v_k$ is a storage node) **then**
23:       ScheduleNode($v_k$, $l$);
24:    **end if**
25:    RemoveNode($S_{l,k}$, $v_k$);
26:    **if** (the delay of $v_k$ is more than 1) **then**
27:       AppendNode($U_{l,k}$, $v_k$);
28:    **else if** (the delay of $v_k$ is 0) **then**
29:       UpdateReadyNodes($v_k$);
30:    **end if**
31: **end while**
```

- resource utilization table R_o, R_s: are original resource utilization table based on resource allocation and resource utilization table for state s, respectively.

- candidate states S_s: are those states which need to be re-scheduled because scheduling result of their predecessors is changed.

In this scheduling algorithm, S_0 is reset node which is first executed after reset is deasserted. In the state scheduling algorithm, there are several functions as follows:

- `GetResUtilTable(`R`,` *states*`)`: returns resource utilization table which the scheduling result of *states* is reflected into.

- `LIST_RC(`*CDFG*`,` R`)`: calls CDFG scheduling algorithm for each state.

- `GetResetNode(`*STG*`)`: returns reset state which is first executed after reset is deasserted.

- `AppendState(`S`,` s`)`: appends state s to the end of the list S.

- `RemoveFrontState(`S: removes and returns the front state in the list S

- `GetPredStates(`s`)`: returns the predecessor states of the state s.

- `GetSuccStates(`s`)`: returns the successor states of the state s.

We extend resource-constrained list scheduling algorithm to schedule the CDFG with pipelined operation and multi-cycle operation with different types. The propsed scheduling algorithm gets CDFG(G), and resource utilization table $R_{l,k}$ and returns the last control step of the CDFG. In scheduling algorithm, we use candidate and unfinished operation list as following:

- resource utilization table $R_{l,k}$: has the number of resources k(function units, storage units, busses, and read/write ports of storage units), which are used at control step l.

- candidate operations $S_{l,k}$: are those opearations of type k whose prodecessors have already scheduled early enough, so that the corresponding opreations are completed at control step l.

- unfinished operation $U_{l,k}$: are those operations of type k that started at earlier cycles and whose execution is not finished at control step l. If the execution delay of an operation is 1 or less, the operation should not be included in the set of unifinished operations.

The node v_0 is the source node and v_{-1} is the sink node.

In the proposed scheduling algorithm, there are several functions as follows:

- GetSchedulableNode(G, $S_{l,k}$, $R_{l,k}$): find the node which can meet resource constraint in ready node list. It will be shown in 3
- UpdateResUtilTab(v_k, $R_{l,k}$): updates the resource utilization table $R_{l,k}$ using scheduling information of hierarchical node v_k. If v_k is branch node, the largest number of the used resources of the branch will be selected to update resource utilization information.
- function `SetDelay(`v, t_k`)`: assigns the delay of the node v to the delay of the type t_k of the function/storage unit.
- function `ScheduleNode(`v, l`)`: specifies the start time of node v at the control step l, and the the end time of the node v will be the addition of l and the delay of node v.
- function `UpdateReadyNode(`$S_{l,k}$, v, l`)`: updates ready node list $S_{l,k}$ with the effect of scheduing of v at the control step l.
- function `UpdateScheduledNodes(`v`)`: will append scheduled nodes v to the scheduled node list $U_{l,k}$.
- function `UpdateUnfinishedNodes(`$U_{l,k}$, v, l`)`: will updates the unfinished nodes by using the scheduled node v and the specified control step l;
- function `AppendNode(`$List$, v`)`, `RemoveNode(`$List$, v`)`: appends/removes v to(from) a set of nodes $List$.

In the proposed scheduling algorithm, types of the storage unit will be assigned to the storage node(type selection of storage node) if the storage node is selected as candidate node among ready node list according to the cost function. If the storage node belongs to the critical path of the CDFG, it will be assigned to the fastest cost storage unit with least cost. The number of ports of storage units and buses which are used in the specified control step, will be determined when the node is scheduled, because the function unit will use the ports and the buses in order to read data at the start time and to write data at the end time of the node. In other word, the data transfer will occur at the start and the end of execution of the node. The read time of the storage node will be changed according to the start time of execution of the node nodes, which will read data from the storage node. The write time of the storage node is the same as the end of the execution of the node, which will write data to the storage node. The function `GetSchedulableNode(`G, $S_{l,k}$, $R_{l,k}$`)` utilizes the resource utilization table to find the

Algorithm 3 GetSchedulableNode(G, $S_{l,k}$, $R_{l,k}$): find schedulable node in ready node list

```
for (all nodes($v_k$ in ready node list $S_{l,k}$) do
    $w_k$ = GetWriteStroageNode($v_k$)
    if ($v_k$) is assignment operation node then
        $r_k$ = GetReadStorageNode($v_k$, 0)
        if (HasReadPorts($l$, $r_k$, 1) and HasBus($l$, 1)) then
            SetDelay($v_k$, 1);
            ScheduleOp($v_k$, $l$);
            UpdateReadPorts($l$, $r_k$, 1);
            UpdateWritePorts($l$, $w_k$, 1);
            UpdateBus($l$, 1);
        end if
    else
        $r0_k$ = GetReadStorageNode($v_k$, 0);
        $r1_k$ = GetReadStorageNode($v_k$, 1);
        if HasReadPorts($l$, $r0_k$, 1) and HasReadPorts($l$, $r1_k$, 1) and HasBus($l$, 2)) then
            $k$ = SelectType($v_k$, $l$);
            $r$ = SelectType($w_k$, $l + d_k - 1$);
            if ($d_k$ is 1 and HasBus($l$, 3)) then
                UpdateFU($l$, $k$);
                UpdateReadPorts($l$, $r$, 1);
                UpdateWritePorts($l$, $r$, 1);
                UpdateBus($l$, 3);
                ScheduleOp($v_k$, $l$);
                ScheduleOp($w_k$, $l + 1$);
            else if ($d_k$ is more than 1 and HasBus($l + d_k - 1$, 1)) then
                UpdateFU($l$, $k$);
                UpdateReadPorts($l$, $r$, 1);
                UpdateWritePorts($l + d_k - 1$, $r$, 1);
                UpdateBus($l$, 2);
                UpdateBus($l + d_k - 1$, 1);
                ScheduleOp($v_k$, $l$);
                ScheduleOp($w_k$, $l + d_k$);
            end if
        end if
    end if
end for
```

node which can meet resource constraint in ready node list. It have to look ahead control step to check available resources for nodes because operation node and storage node can be multicycled or pipelined. When the function unit type of an operation node is selected, the storage unit type of the storage node, which is output of the operation node to write value, is also determined. It is composed of the following functions:

- `GetWriteStroageNode(`v_k`)`: returns the storage node where v_k will write a variable.
- `GetReadStorageNode(`v_k`, `lr`)`: returns the storage node where v_k will read a variable. If the lr is 0(1), it return left(right) side of input variables of operation node v_k.
- `HasReadPorts(`l`, `r_k`, `num`)`: checks if storage unit type r_k has num number of available read ports in control step l.
- `HasWrite(`l`, `r_k`, `num`)`: checks if storage unit type r_k has num number of available write ports in control step l.
- `HasBus(`l`, `num`)`: checks if there is num number of available buses in control step l.
- `SelectType(`v`, `l`)`: returns the availabe resource type for node v_k at control step l.
- UpdateFU(l, k): updates the number of available function unit of type k by num at control step l.
- UpdateReadPorts(l, r_k, num): updates the number of available read ports of storage unit r_k by num at control step l.
- UpdateWritePorts(l, w_k, num): updates the number of available write ports of storage unit w_k by num at control step l.
- UpdateBus(l, num): updates the number of available buses by num at control step l.

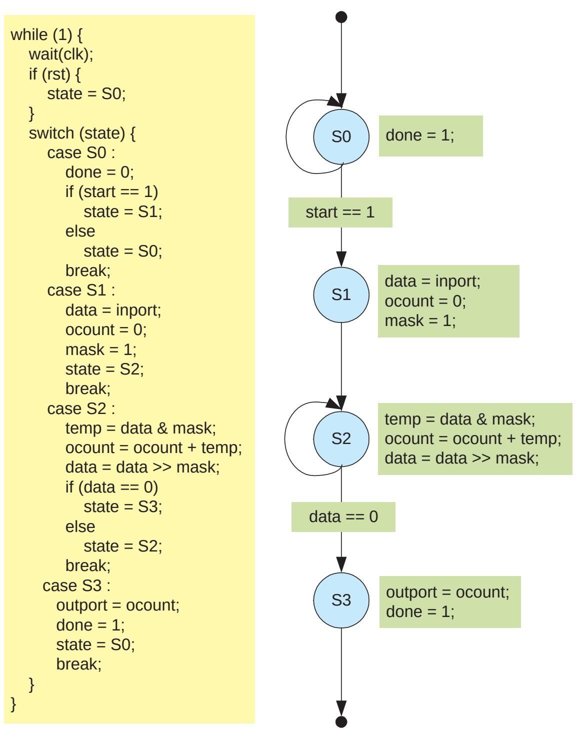
5.2.1 Priority function

The list scheduling algorithms are classified according to the selection step. A priority list of the operations is used in choosing among the operations, based on some heuristic measure. Our proposed algorithm has two priority functions. One is for node selection among ready node list. The other is for resource type selection from library.

1. node selection: the ready node list is sorted by the priority: urgency, mobility, number of successors in decreasing order, to select the node among the ready node list.
2. type selection: to select type of operation/storage nodes, cost function in library is utilized. The designer selects cost function according to the latency of unit, size of unit, whether or not it's pipelined.

The ready node list is sorted by the priority list for node selection and the resource utilization table is sorted by the priority list for type selection.

5.3. Scheduling process by example



```
while (1) {
   wait(clk);
   if (rst) {
      state = S0;
   }
   switch (state) {
      case S0 :
         done = 0;
         if (start == 1)
            state = S1;
         else
            state = S0;
         break;
      case S1 :
         data = inport;
         ocount = 0;
         mask = 1;
         state = S2;
         break;
      case S2 :
         temp = data & mask;
         ocount = ocount + temp;
         data = data >> mask;
         if (data == 0)
            state = S3;
         else
            state = S2;
         break;
     case S3 :
        outport = ocount;
        done = 1;
        state = S0;
        break;
   }
}
```

Figure 4. FSMD for one's counter

To explain the proposed scheduling algorithm, we will use one's counter as an example, shown in Figure 4, which calculates the number of one's in given number. It takes one input variables and generates one output result. The left side of this figure shows SpecC code for one's counter example and the corresponding FSMD is shown in the left side of this figure. Before scheduling, this FSMD consists of 4 states. State `S0` is reset state, and if reset is asserted, FSMD will enter this state first. State S2 has self loop to calculate number of one's in `data` variable until `data` is equal to 0. The Figure 5 shows the target datapath organization for the one's counter, which consists of two 2-stage pipelined ALUs(ALU0 and ALU1) and one register file and 3 buses. The function unit ALU0 can perform bitwise and operation and addition operation in 2 cycles, and ALU1 can

Table 1. Scheduling process for one's counter

	ready		resource utilization table				scheduled	unfinished
	ALU0	ALU1	ALU0(1)	ALU1(1)	RF(1/2)	bus(1/2)		
cs1	&&	>>	0	0	0/0	0/0		
		>>	1	0	0/2	0/2	&&	
cs2		>>	0	0	1/0	1/0		&&
			0	1	1/2	1/2	>>	
cs3	+		0	0	1/0	1/0		>>
			1	0	1/2	1/2	+	
cs4		==	0	0	1/0	1/0		+
			0	1	1/2	1/2	==	
cs5			0	0	0/0	0/0		==
			0	0	0/0	0/0		

Table 2. Scheduling process for one's counter(II)

	ready		resource utilization table				scheduled	unfinished
	ALU0	ALU1	ALU0(1)	ALU1(1)	RF(1/2)	bus(1/2)		
cs1	&&	>>	0	0	0/0	0/0		
			1	1	0/2	0/2	&& >>	
cs2			0	0	1/0	1/0		&& >>
			0	0	1/0	1/0		
cs3		==	0	0	1/0	1/0		&&
			0	1	1/2	1/2	==	
cs4	+		1	0	0/0	0/0		==
			0	1	0/2	0/2	+	
cs5			0	0	0/0	0/0		+
			0	0	0/0	0/0		
cs6			0	0	1/0	1/0		+
			0	0	1/0	1/0		

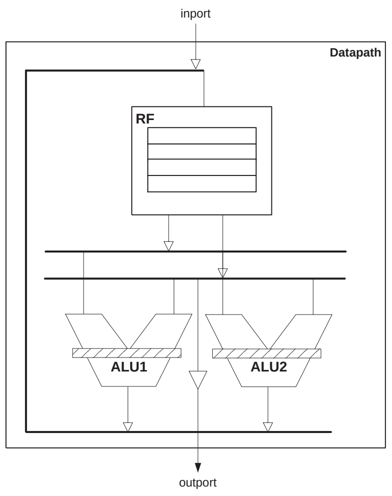


Figure 5. Target datapath organization for ones's counter

perform left/right shift and comparison operation in 2 cycles. The regiser file with two read ports and one write port is neither pipelined nor latched. Figure 6 shows the CDFG

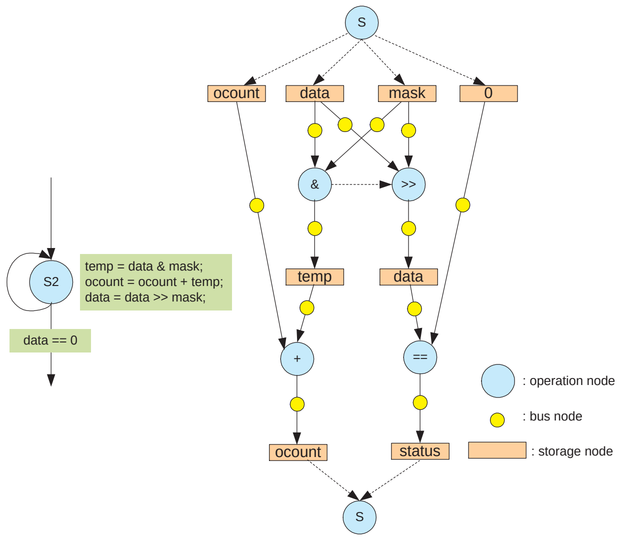


Figure 6. CDFG for state S2 in one's counter

of state S2 which is generated from FSMD in Figure 4. The CDFG has 4 ALU operations and 8 storage nodes and 12 bus nodes. Figure 1 shows the scheduling step according to the proposed scheduling algorithm. The 1st column represents the control steps. The next 2 columns represent the ready operations for each type of function unit. The next 4 columns represent the resource utilization table, which has the number of resources used in each control step. In BUS column, the left value shows the number of buses which is used to transfer the result of function units to storage units, the right value shows the number of buses which is used to transfer the input data for function units from storage units. In RF column, the 1st value represents the number of the used write ports, and the other value shows the number of the used read ports in register file. The last two columns represents the scheduled operations and unfinished operation in current control step. According to the proposed scheduling algorithm, all nodes in CDFG are scheduled using this table. The latency of the scheduled CDFG is 5 control steps. The scheduled CDFG is shown in Figure 7. If we change 2-stage pipelined ALU0 to 3-stage pipelined ALU0, the scheduling result will be changed as shown in Figure 8 and in Figure 2. In Figure 8, operation + should be executed in 3 cycles but the control step of the state S2 should be finished in cs4. The S2 has self loop, then the remaining two cycles of the operation + will be performed in cs1 and cs2 in the state S2.

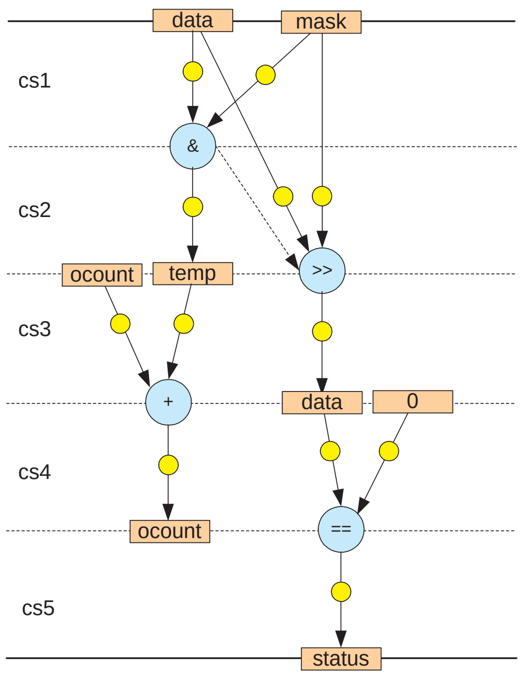


Figure 7. Scheduled CDFG for state S2 in one's counter

6. Experimental Results

We implemented the internal representation for the RTL description on 9000 lines of C++ code and our scheduling algorithm on 1000 lines of C++ code. Our scheduling algorithm is integrated in the RTL refinement system, which can perform the scheduling, storage binding, function binding and connection binding in arbitrary order.

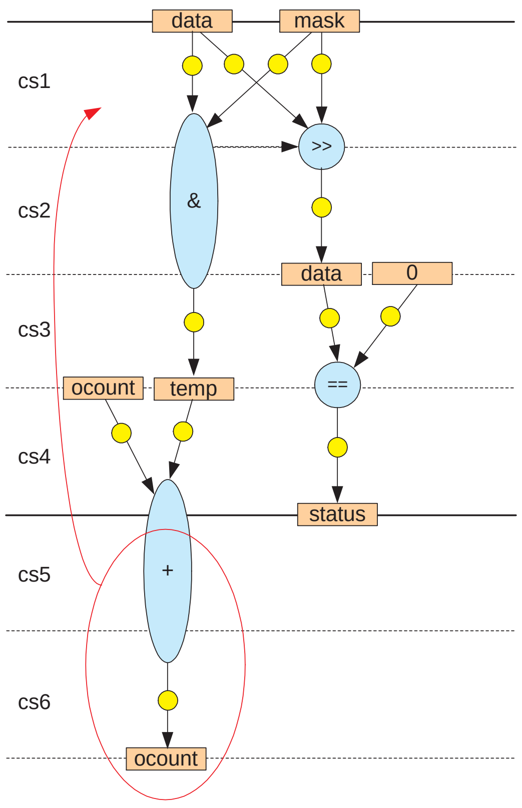


Figure 8. Scheduling process for one's counter(II)

Table 3. resource allocation for examples

example	ALU num (d/p)	shift num (d/p)	mult num (d/p)	RF num (d/p/r/w)	bus num
ones(4)	1(1/n)	1(1/n)	-	1(0/n/2/1)	3
	2(1/n)	1(1/n)	-	3(0/n/2/1)	6
	2(1/n)	1(1/n)	-	1(1/p/2/1)	3
	2(2/p)	1(2/p)	-	2(0/n/2/1)	4
	2(2/p)	1(2/p)	-	3(0/n/2/1)	4
SRA(6)	1(1/n)	1(1/n)	-	1(0/n/2/2)	3
	2(1/n)	1(1/n)	-	2(0/n/2/1)	6
	1(2/p)	1(2/p)	-	1(0/n/2/1)	3
	1(2/p)	1(2/p)	-	2(0/n/2/1)	3
	1(2/n)	1(2/n)	-	2(0/n/2/1)	4
MAT(13)	1(1/n)	1(1/n)	1(2/p)	1(0/n/2/1)	3
	2(1/n)	1(1/n)	1(2/p)	3(0/n/2/1)	6
	1(2/p)	1(2/p)	1(4/p)	2(0/n/2/1)	3
	1(2/p)	1(2/p)	1(4/p)	2(1/p/2/1)	3
	2(2/p)	2(2/p)	1(4/p)	3(1/p/2/1)	4
DCT(16)	1(1/n)	1(1/n)	1(2/p)	3(0/n/2/1)	8
	2(1/n)	2(1/n)	2(2/p)	4(0/n/2/1)	10
	3(1/n)	2(1/n)	2(2/p)	4(0/n/2/1)	14
	3(1/n)	2(1/n)	2(2/p)	4(1/p/2/1)	10
	1(2/p)	1(2/p)	1(4/p)	3(1/p/2/1)	7
	2(2/p)	2(2/p)	2(2/p)	4(1/p/2/1)	10
	3(2/p)	2(2/p)	2(2/p)	4(1/p/2/1)	10

We applied the scheduling alogrithm to one's counter, square root approximation, matrix mulipication of DCT design and Chen DCT in jpeg encoder. Table 3 shows the resource allocation such as number of ALU, shifter, multiplier, storage units and buses, and number of read/write ports of storage units. In this table, the number and character in parenthesis represents the delay of the unit and whether or not the unit is pipelined(p) or not(n). The storage unit column has number of read ports(r) and write ports(w) of storage units. Table 4 show the scheduling results for each example. It has number of states and ports/bus utilization in state.

7. Conclusion

This report has shown the scheduling algorithm in the RTL design methodology, which is based on the resource constrained list scheduling, which considers the number of function units, storage units, buses, and ports of storage units in each control step. The scheduling algorithm supports the pipelined/multicycle operations and storage units, such as pipelined register files and latched memory. The scheduling algorithm is integrated in the RTL refinement system. Experimental results for several examples show

Table 4. number of states and resource utilization

example	states	rport	wports	buses
ones	9	0.9/3	0.7/1	1.7/3
	5	1.6/6	1.2/3	3/6
	10	0.8/2	0.6/1	1.5/3
	8	1.0/4	0.8/2	1.9/4
	7	1.1/6	0.9/3	2.1/4
SRA	18	0.9/2	0.6/2	2.2/3
	17	1.0/4	0.6/2	1.8/6
	29	0.6/2	0.4/1	1.0/3
	26	0.7/4	0.5/2	1.2/3
	26	0.7/4	0.5/2	1.2/4
MAT	44	0.6/2	0.7/1	2.1/4
	38	0.9/6	0.7/3	1.7/6
	63	0.5/4	0.4/2	1.0/3
	88	0.4/4	0.3/2	0.7/3
	86	0.4/6	0.3/3	0.8/4
DCT	135	1.4/6	1.1/3	3.3/8
	88	2.2/8	1.7/4	5.1/10
	77	2.5/8	2.0/4	5.9/14
	103	1.9/8	1.5/4	4.4/10
	198	1.0/6	0.8/3	2.3/7
	154	1.2/8	1.0/4	2.9/10
	146	1.0/8	1.0/4	3.1/10

that proposed scheduling algorithm generates efficient results under resource constraints.

References

[Acc01] Accellera C/C++ Working Group. RTL Semantics:Draft Specification. Feburary 2001.

[GDLW92] D. Gajski, N. Dutt, S. Lin, and A. Wu. *High Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.