Scheduling in RTL Design Methodology

Dongwan Shin and Daniel Gajski

Technical Report ICS-01-65
July 1, 2001

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

{dongwans,gajski}@ics.uci.edu

Scheduling in RTL Design Methodology

Dongwan Shin and Daniel Gajski

Technical Report ICS-01-65
July 1, 2001

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

{dongwans,gajski}@ics.uci.edu

Abstract

This report describes the scheduling algorithm in RTL design methodology. The proposed scheduling algorithm is based on resource constrained list scheduling, which considers the number of function units, storage units, busses and ports of storage units in each control step, and 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
Information and Computer Science
University of California, Irvine

Abstract

This report describes the scheduling algorithm in RTL design methodology. The proposed scheduling algorithm is based on resource constrained list scheduling, which considers the number of function units, storage units, busses and ports of storage units in each control step, and 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, 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 level(e.g. layout masks) through various refinement processes. Register transfer level(RTL) refinement has been recognized as one of the major design methodology.

The high-level synthesis involves the transformation of behavioral description of the design into a set of interconnected RT 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, busses, 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 busses(connection binding).

This report focuses on the scheduling in our RTL design methodology. We developed the scheduling algorithm based on list scheduling heuristic to consider the number of function units, storage units, busses and ports of storage units in each control step, and to support the pipelined/multicycle operations and storage units, such as pipelined register files and latched memory.

The rest of this report is organized as follows: section 2 describes the motivation of RTL design methodology and refinement tool. In section 3 describes the 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 result. Section 7 concludes this report with a brief summary and future work.

2. Motivation

Many researches for High-level synthesis [GDLW92] have been done 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, because 1) they can support only several limited architectures, 2) they lack interaction between tools and the designers, and 3) the quality of the generated design is worse than that of mannual design. To make them popularly used in design community, we should tackle these problems. The proposed RTL design methodogy, which is proposed by Accellera C/C++ Working Group [Acc01] supports the more advanced architecture like bus-based architecture instead of mux-based architecture, whose performance is not good in large design and provides the interaction between designer and refinement tool to tackle these problems. 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 high-level synthesis is bus-based universal processor architecture [Acc01], in which the function/storage units are pipelined or multi-cycled. The storage units can be composed of registers, register files and memories with different latency and pipeline scheme. In other word, target architecture can have heterogenous in terms of storage units. It makes the problems for the refinement hard to solve and scheduling and binding of the refinement should be ex-

tended 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 which generates the exposed-control RTL from behavioral RTL.

3.1. The Accellera RTL semantics

The RTL design is 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, such as unmapped RTL(style 1), storage mapped RTL(style 2), function mapped RTL(style 3), connection mapped RTL(style 4) and exposed-control RTL(style 5). 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, busses 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(style 1) to exposed-control RTL(style 5). However, due to the interdependence of scheduling, allocation, and binding, the order of three steps should be interchangeable to get the exposed-control RTL.

3.2. The architecture of the RTL refinement tool

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 CDFG data structure, we uses the C++/SpecC/HDL as input [Acc01]. The RTL refinement tool also reads the RTL description

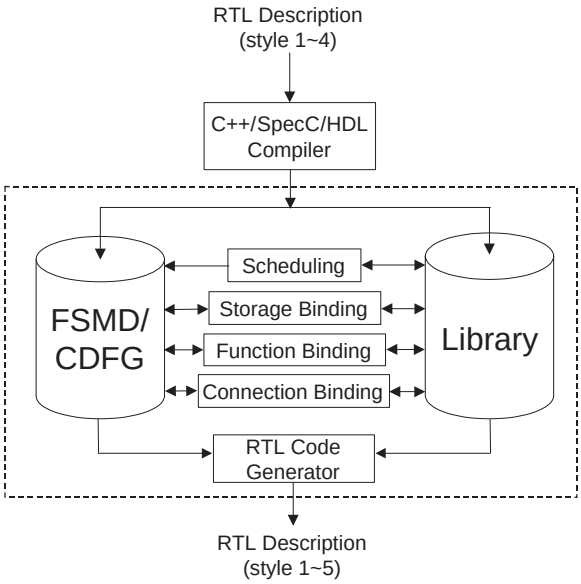


Figure 1. RTL design refinement flow

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 the each style RTL description as the result of each refinement step. For example, RTL refinement tool reads the style 1 RTL SpecC code and generates FSMD/CDFG and performs storage binding and then generates the style 2 RTL description.

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

The netlist mapper generates the style 5 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 the 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, busses 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

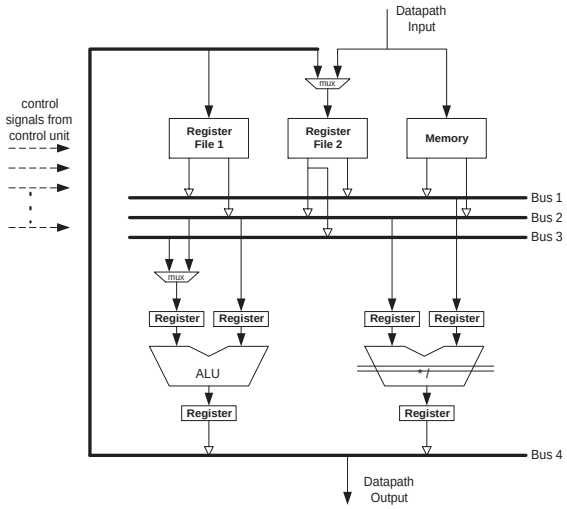


Figure 2. Bus based architecture

```
// 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;
```

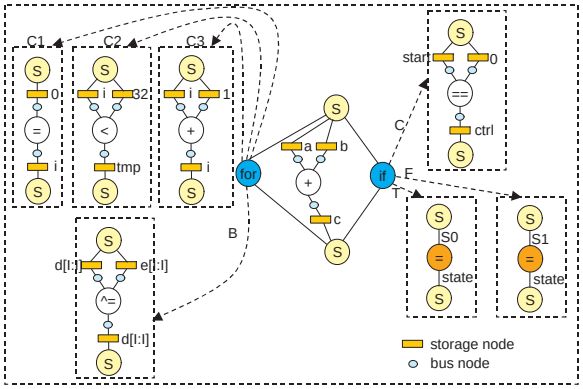


Figure 3. CDFG for unmapped(style 1) RTL description)

busses. All data transfers are achieved through busses. 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.

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 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 edge has the dependency information between nodes such as control dependency and data dependency. The node has all information 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 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 is 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 hierarchical control/data flow graph CDFG(V, E) for each state, where V is a set of vertices representing operations, storages, busses, and hierarchical nodes such as branch and loop.

2. A component library containing functional units, storage units and busses characterized by type, area, delay, and pipeline states. In addition, storage units have the number of read/write ports.

3. clock period and resource allocation for the behavior, such as number of function units, storage units, busses 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 constrol steps is minimized.

2. the resource allocation constraint is satisfied.

5.2. Proposed Scheduling Algorithm

We extend resource-constrained list scheduling algorithm to schedule the CDFG with pipelined operation and multi-cycle operation with different types. The proposed scheduling algorithm is shown in Algorithm 1.

The propsed scheduling algorithm gets CDFG(G), initial control step l_i and resource utilization table $R_{l,k}$ and returns the last control step of the CDFG. In scheduling algorithm, we use several lists and table as following:

- resource utilization table $R_{l,k}$: has the number of resources k, which are used at control step l. It will be explained in detail in Section 5.2.1.

- 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 nodes list. It will be shown in 2

- 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 nodes 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 nodes 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 nodes 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 busses 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 busses 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 node which can meet resource constraint in ready nodes 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, storage unit type of the storage node, which is output of the operation node to write value, 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 busses 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 busses by *num* at control step l.

Algorithm 1 LIST_RC(G, l_i, $R_{l,k}$): List scheduling algorithm

```
    Initialize the unfinished operations U_{l,k} = {};
    Initialize the step l = l_i;
    SetDelay(v_0, 0);
    ScheduleNode(v_0, l);
 5: UpdateReadyNodes(S_{l,k}, v_0, l);
    while (S_{l,k} or U_{l,k} is not empty) do
      v_k = GetSchedulableNode(G, S_{l,k}, R_{l,k});
      if (v_k == NULL) then
        l = l + 1;
10:     UpdateReadyNodes(l);
        UpdateUnfinishedNodes(l);
        UpdateScheduledNodes(l);
        continue;
      else if (v_k is a bus/port/ctrl node) then
15:     SetDelay(v_0, 0);
        ScheduleNode(v_0, l);
      else if (v_k is a branch node) then
        d_c = LIST_RC(conditional subgraph of v_k, l, R_{l,k})
        - l;
        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);
        UpdateResUtilTable(v_k, R_{l,k});
      else if (v_k is a function call node) then
        d_b = LIST_RC(function body subgraph of v_k, l,
        R_{l,k}) - l;
        SetDelay(v_k, d_b);
25:     UpdateResUtilTable(v_k, R_{l,k});
      else if (v_k is a storage node) then
        ScheduleNode(v_k, l);
      end if
      RemoveNode(S_{l,k}, v_k);
30:   if (the delay of v_k is more than 1) then
        AppendNode(U_{l,k}, v_k);
      else if (the delay of v_k is 0) then
        UpdateReadyNodes(v_k);
      end if
35: end while
```

5.2.1 Resource Utilization table

In order to consider the multicycled or pipelined operations(operation nodes and storage nodes) in the proposed scheduling, we utilize the resource utilization table, which has the number of resources which are not used in specified control step for function units, storage units, busses and read/write ports of each storage unit.

Algorithm 2 GetSchedulableNode(G, $S_{l,k}$, $R_{l,k}$): find schedulable node in ready nodes list

```
    for (all nodes(v_k in ready nodes list S_{l,k}) do
      w_k = GetWriteStroageNode(v_k)
      if (v_k) is assignment operation node then
        r_k = GetReadStorageNode(v_k, 0)
 5:     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);
10:       UpdateBus(l, 1);
        end if
      else
        r0_k = GetReadStorageNode(v_k, 0);
        r1_k = GetReadStorageNode(v_k, 1);
15:     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);
20:         UpdateReadPorts(l, r, 1);
            UpdateWritePorts(l, r, 1);
            UpdateBus(l, 3);
            ScheduleOp(v_k, l);
            ScheduleOp(w_k, l + 1);
25:       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);
30:         UpdateBus(l + d_k − 1, 1);
            ScheduleOp(v_k, l);
            ScheduleOp(w_k, l + d_k);
          end if
        end if
35:   end if
    end for
```

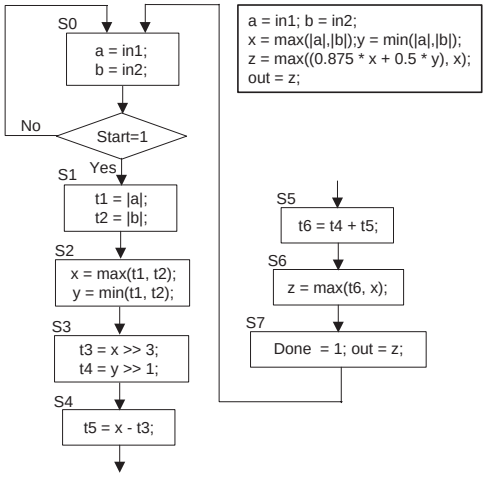


Figure 4. Algorithmic State Machine for Square Root Approximation

5.2.2 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 nodes list. The other is for resource type selection from library.

1. node selection: the priority list is sorted by urgency, mobility, number of successors in decreasing order to select the node among the ready nodes 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 nodes 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

To explain the proposed scheduling algorithm, the square root approximation is shown in Figure 4, which calculates the square root of two values approximately. It takes two input variables and generates one output result. The Figure 5 shows the target datapath organization for the square root approximation, which consists of an input latched ALU(ALU0), an output latched ALU(ALU1), a register file and 3 busses. The function unit ALU0 can perform addition/subtraction/right shift operations in 2 cycles, and ALU1 can calculate absolute value and min/max value in 2

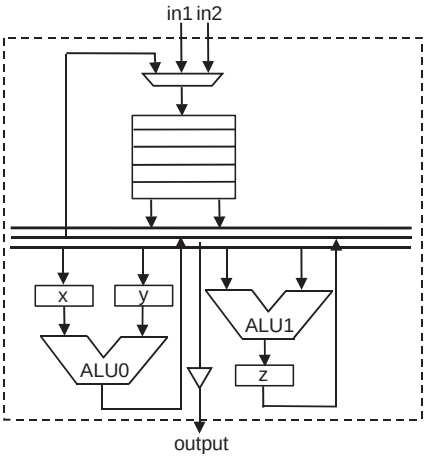
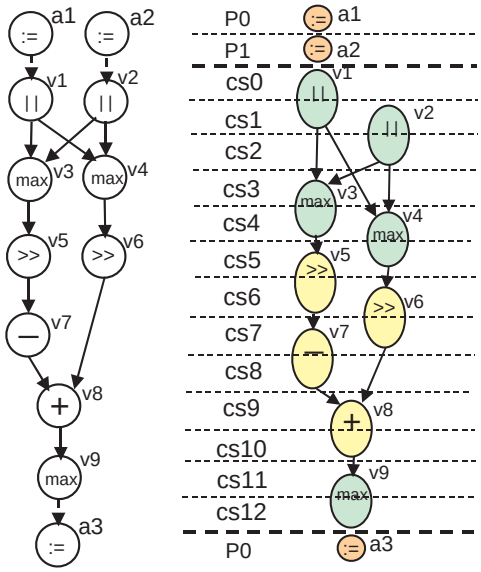


Figure 5. Target datapath organization for Square Root Approximation



(a) Unscheduled CDFG (b) Scheduled CDFG

Figure 6. CDFG for Square root approximation

cycles. The regiser file with two read ports and one write port is neither pipelined nor latched.

Figure 6(a) shows the CDFG which is generated from the ASM in Figure 4. The CDFG has 3 assignment nodes for input/output operations and 9 ALU operations. The storage nodes and bus nodes are omitted in Figure 6.

Figure 7 shows the scheduling step according to the proposed scheduling algorithm. The 1st column represents the control steps. The next 3 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 the this table. The latency of the scheduled CDFG is 16 control steps. The scheduled CDFG is shown in Figure 6(b).

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.

We applied the scheduling alogrithm to one's counter, square root approximation, matrix mulipication of DCT design and Chen DCT in jpeg encoder which are shown in Appendix **??**. Table 1 shows the resource allocation such as number of ALU, shifter, multiplier, storage units and busses, 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 2 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, busses, 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 1. 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

Table 2. number of states and resource utilization

example	states	rport	wports	busses
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

	Ready Operations			Resource Utilization Table				Scheduled	Unfinished
	Assign	ALU0	ALU1	ALU0 (1)	ALU1 (1)	BUS (3)	RF (1/2)	Operations	Operations
P0	A1,A2						1/0	A1	
P1	A2						1/0	A2	
CS0			V1,v2	0 0	0 1	0/0 0/2	0/0 1/2	V1	
CS1			V2	0 0	0 1	0/1 0/3	1/0 1/2	V2	V1
CS2				0 0	0 0	0/1 0/1	1/0 1/0		V2
CS3			V3,v4	0 0	0 1	0/0 0/2	0/0 1/2	V3	
CS4			V4	0 0	0 1	0/1 0/3	1/0 1/2	V4	V3
CS5		V5		0 1	0 0	0/1 2/1	1/0 1/2	V5	V4
CS6		V6		0 1	0 0	1/0 3/0	1/0 1/2	V6	V5
CS7		V7		0 1	0 0	1/0 3/0	1/0 1/2	V7	V6
CS8				0 0	0 0	1/0 1/0	1/0 1/0		V7
CS9		V8		0 1	0 0	0/0 2/0	0/0 1/2	V8	
CS10				0 0	0 0	1/0 1/0	1/0 1/0		V8
CS11			V9	0 0	0 1	0/0 0/2	0/0 1/2	V9	
CS12				0 0	0 0	0/1 0/1	1/0 1/0		V9
P0	A3						1/1	A3	

Figure 7. Scheduling process for Square Root Approximation

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.