RTL Design and Synthesis of Sequential Matrix Multiplication

Pei Zhang
Daniel D. Gajski

CECS Technical Report 02-09
April 3rd, 2002

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

{pzhang, gajski} @ics.uci.edu

Abstract

This report describes the RTL design and synthesis of Sequential Matrix Multiplication using accellera RTL methodology. We first begin with the introduction of Sequential Matrix Multiplication. Then we give the FSMD of Sequential Matrix Multiplication and critical path analysis to decide the clock cycle. Based on different resource constraint, the synthesis results (scheduling, storage binding, functional unit binding and interconnection binding) are given. The source codes are also included in the Appendix.

Contents

Abstract 1
1. Introduction 1
2. FSMD (Behavioral Model) 1
  2.1 Get data (S0~S4) 1
  2.2 Execute matrix multiplication (S5~S12) 2
  2.3 Send output (S13~S17) 2
3. Implementation 3
4. Critical path analysis 3
5. RTL Synthesis and Results 4
  5.1 RTL synthesis 5
  5.2 Experiments Results 5
6. Conclusion 7
Reference 7
Appendix: SMM Behavioral Mode Source Code in SpecC 8

List of Figures

Figure 1: Block diagram of SMM 1
Figure 2: FSMD of SMM. 2
Figure 3: Possible architecture of SMM with critical path candidates 3
Figure 4: Architecture with RFs, Memories w/o dedicated registers 6
Figure 5: Architecture with RFs, Memories w/ dedicated registers 6

List of Tables

Tables 1: Resource delay 4
Table 2: Different binding results 7

RTL Design and Synthesis of Sequential Matrix Multiplication

Pei Zhang, Daniel D. Gajski
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

Abstract

This report describes the RTL design and synthesis of Sequential Matrix Multiplication using accellera RTL methodology. We first begin with the introduction of Sequential Matrix Multiplication. Then we give the FSMD of Sequential Matrix Multiplication and critical path analysis to decide the clock cycle. Based on different resource constraint, the synthesis results (scheduling, storage binding, functional unit binding and interconnection binding) are given. The source codes are also included in the Appendix.

1. Introduction

Matrix multiplication is the key part of Discrete Cosine Transform (DCT) which is widely used in image processing and compression. Matrix multiplication is used twice in the DCT. There are several ways to implement Matrix multiplication. Here we use sequential method which is called Sequential Matrix multiplication (SMM).

The block of SMM is shown in figure 1. It implement the function:

$$C_{8\times8} = A_{8\times8} \times B_{8\times8}$$

Here:
Inport1/Input2/Outport are for A, B and C, respectively. ***Start*** is the signal to let SMM begin to execute. ***Done*** is the signal to notify that SMM is finished.

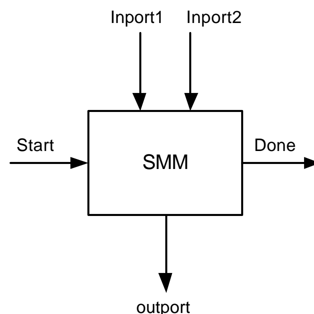

Figure 1: Block diagram of SMM

The rest of this report is organized as follows: first, the FSMD (behavioral model) of SMM is given. This FSMD describes the detailed functionality of SMM. Then the target implementation (structural model) and critical path analysis for SMM are discussed. Based on different sources, the RTL synthesis for SMM is made by running our RTL synthesis tools to estimate the performance. At last the summary and future works is included.

2. FSMD (Behavioral Model)

The finite-state machine with data (FSMD) is a very popular design model for RTL behavioral description. It describes the functionality of design with several states. The state transition is controlled by clock and control signal. We use Moore machine to implement SMM.

The FSMD of SMM is given in Figure 2.

The FSMD of SMM includes three parts:
(1) Get input;
(2) Calculate matrix multiplication;
(3) Send output.

Since every matrix has a size of 8x8, we use byte-serial wires to get/send data. For each matrix, it need 64 times to finish all data transfer.

2.1 Get data (S0~S4)

After the SMM get the ***Start*** signal, it begins its operation. First it need to get $A_{8\times8}$ and $B_{8\times8}$ from its Inports. In this design, we have two Inports to let SMM get two inputs simultaneously. We add temporary variables between port and memories. We use ***count*** to count the times of iteration. After repeat 64 times, $A_{8\times8}$ and $B_{8\times8}$ get all data they need.

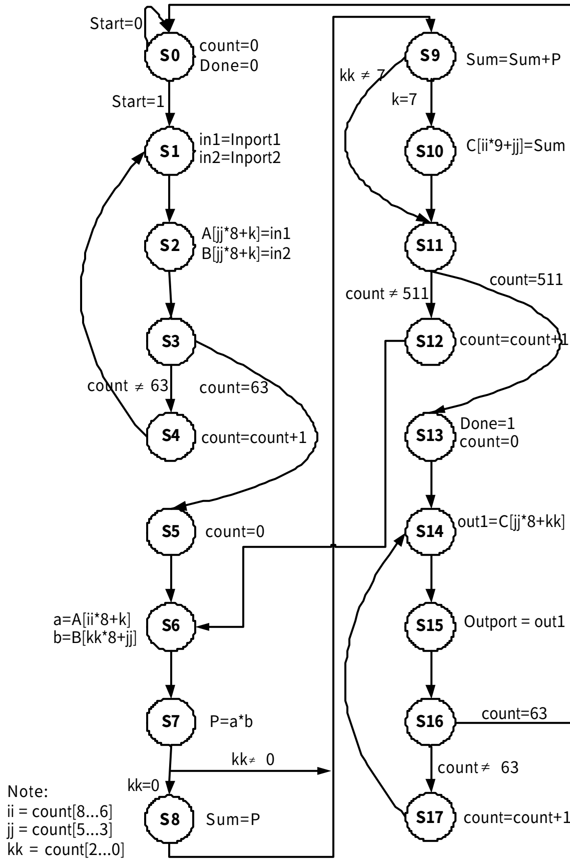

Figure 2: FSMD of SMM

- In State `S0`, SMM sets the ***count*** and ***Done*** signal to 0. It also checks the ***Start*** signal. If the ***Start*** is equal to 0, SMM remains in State `S0`. Otherwise it transits to state `S1`.
- In State `S1`, SMM get inputs for ***in1/in2*** from ***input1/inpout2***. Remember ***in1/in2*** and ***input1/inpout2*** are all 8-bit long. Then SMM transits to state `S2`.
- In State `S2`, the internal arrays (size 64) ***A, B*** are set to ***in1*** and ***in2*** respectively. The position of arrays is decided by ***count***. Then SMM transits to state `S3`.
- In State `S3`, SMM checks if ***count*** is equal to 63 which means SMM has got all 64 pair of inputs for ***A*** and ***B***. if so, SMM transits to state `S5`. Otherwise, SMM transits to state `S4`.
- In State `S4`, we let ***count*** get increment by 1, then let SMM transit to state `S1` to get another pair of inputs.

2.2 Execute matrix multiplication (S5~S12)

This part is the main part of SMM. It performs the $C_{ii \times ii} = \sum_{kk=0}^{7} A_{ii \times kk} \times B_{kk \times jj}$ operation. ***a*** and ***b*** are temporary variables to fetch data from $A_{8\times8}$ and $B_{8\times8}$. ***Sum*** and ***P*** are interim variable for $A_{8\times8}$. It uses the whole of part of ***count*** (9 bits) to control the state transition. ***ii, jj, kk*** are the most, middle and least significant 3 bits of ***count*** respectively.

- In State `S5`, ***count*** is reset to 0. Then SMM transits to state `S6`.
- In State `S6`, temporary variables ***a*** and ***b*** get values for ***A*** and ***B***. Similarly, the position of arrays is decided by ***count.*** Then SMM transit to state `S7`.
- In State `S7`, ***P*** stores the product of ***a*** and ***b***. Then SMM checks the value of ***kk*** (the least significant 3 bits of ***count***). If ***kk*** is equal to 0, the next state is `S8`. Otherwise, SMM transits to `S9`.
- In State `S8`, we let ***Sum*** be equal to ***P***. Then SMM transit to state `S9`.
- In State `S9`, Sum is set the sum of ***Sum*** and ***P***. Then if ***kk*** is equal to 7, the next state is `S10`. Otherwise, SMM transits to `S11`.
- In State `S10`, one position of array ***C*** is set to ***Sum***. The position of arrays is also decided by ***count.*** Then SMM transit to state `S11`.
- In State `S11`, if ***count*** is equal to 511, the next state is `S13`. Otherwise, SMM transits to `S12`.
- In State `S12`, ***count*** increases by 1. Then SMM transit to state `S6` to start next iteration to compute ***C***.

2.3 Send output (S13~S17)

After the computation finishes, SMM generates ***Done*** signal and send output to the ***Outport*** in byte-serially.

- In State `S13`, SMM reset the ***count*** to 0 and set ***Done*** signal to 1.Then SMM transits to state `S14` to start to send outputs.
- In State `S14`, variable ***out1*** get a value from array ***C***. The position of arrays is also

decided by ***count.*** Then SMM transits to state `S15`.

- In State `S15`, ***out1*** is set to the ***Outport*** as to let other part fetch the value. Then SMM transits to state `S16`.
- In State `S16`, if ***count*** is equal to 63 which means SMM has finished all jobs, SMM transits to state `S0` to wait for ***Start*** signal to start another computation. Otherwise, SMM transits to state `S17`.
- In State `S17`, we let ***count*** get increment by 1, then let SMM transit to state `S14` to send another output.

3. Implementation

In this part, we give the possible target implementation (structural model) for SMM.

(1) Datapath
The datapath part is a bus-based architecture which means it uses busses and multiplexers instead of multiplexers only as the interconnection.

Figure 3 gives one possible structure of SMM based on the given resource (1 register file, 3 memories, several dedicated registers, 1 ALU, 1 Multiplier and 3 busses). Here we put three matrixes used in the SMM into separate memory. If the memories have bigger size and more input/output ports, some of these matrixes maybe mapped into the same memory. When the SMM goes through RTL synthesis tools, it will have different result.

All component and tri-state drivers are controlled by the control signals generated by control unit. Also, the comparator generates 4 comparison outputs (0/7/63/511) to the control unit for state transition decision.

Also, there maybe have different architectures in Figure 3. Then the synthesis results should be different. Part 5 will give other synthesis results based on different resources.

(2) Control Unit
The control unit implements the state transition in its next-state logic and state register. It also generates the control signals in its output logic for the datapath.

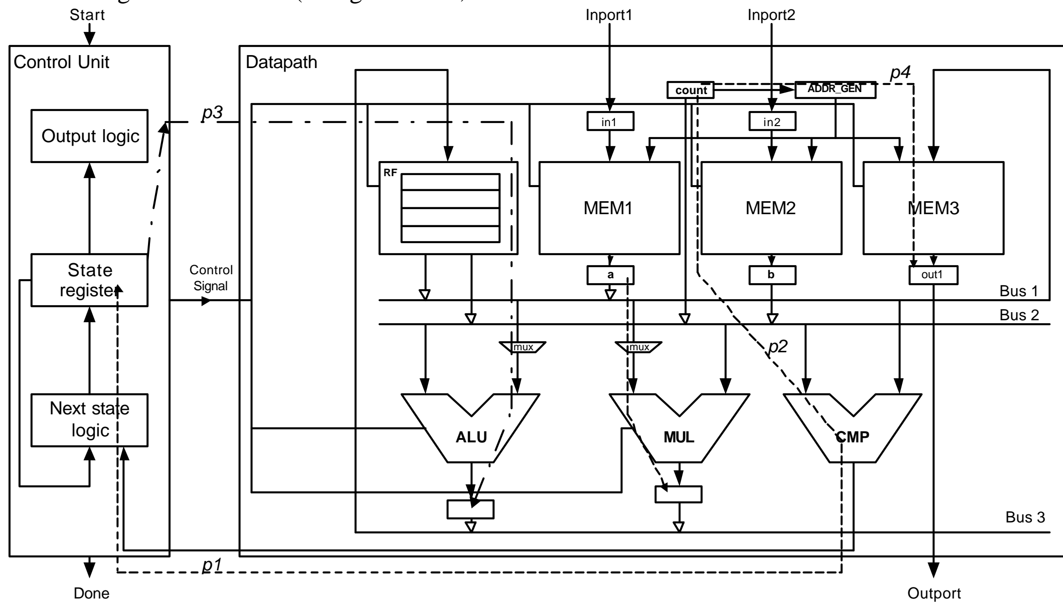

Figure 3: Possible architecture of SMM with critical path candidates

4. Critical path analysis

During the RTL synthesis, we need to know the clock cycles to calculate the execution time for the design performance purpose. The execution time can be computed as:

$$execution_time = num_cycles \times clock_cycle$$

Different resource allocation can cause different clock cycle. Given the SMM design shown in Figure 3 (1 ALU, 1MUL, 3 Bus and 1 RF with special registers and memories), the minimum reasonable clock cycle can be determined as the maximum of the critical path candidates

(including the control unit delay) as follows based on the Table 1:

Component	Operations	Delay (ns)
ALU	+, -	3.02
Multiplier	*	4.09
Reg32	Count, in1, in2, out1, a, b, State register	.75
Reg32(setup)		.59
MUX	Choose data	.66
ADDR_GEN	Memory address generation	1.94
MEM	Memory access	4.20
Register file	P, Sum	1.46
CMP	Comparator	1.22
Output Logic	Output logic	3.85
Next State Logic	Next state logic	3.85

Tables 1: Resource delay

There are four critical path candidates that also showed in the Figure 3. For each candidate, the time delay are calculated in the following:

$$\begin{aligned}T_{p1} &= \text{delay(REG)} + \text{delay(CMP)} \\ &+ \text{delay(NSL)} + \text{setup(SR)} \\ &= 6.41\text{ns}\end{aligned}$$

$$\begin{aligned}T_{p2} &= \text{delay(REG)} + \text{delay(MUX)} \\ &+ \text{delay(MUL)} + \text{setup(REG)} \\ &= 6.09\text{ns}\end{aligned}$$

$$\begin{aligned}T_{p3} &= \text{delay(SR)} + \text{delay(OL)} + \text{delay(RF)} \\ &+ \text{delay(MUX)} + \text{delay(ALU)} \\ &+ \text{setup(REG)} \\ &= 10.31\text{ns}\end{aligned}$$

$$\begin{aligned}T_{p4} &= \text{delay(REG)} + \text{delay(AGEN)} \\ &+ \text{delay(MRD)} + \text{setup(REG)} \\ &= 7.48\text{ns}\end{aligned}$$

Here:

- delay(SR) is the delay of reading state register
- delay(OL) is the delay of output logic
- delay(NSL) is the delay of next state logic
- delay(REG) is the reading register
- delay(AGEN) is the delay of memory address generation
- delay(MRD) is the delay of reading memory
- delay(RF) is the delay of reading RF
- delay(CMP) is the delay of comparator
- delay(MUX) is the delay of multiplexer
- delay(ALU) is the delay of ALU
- delay(MUL) is the delay of multiplier
- setup(REG) is the setup time of register
- setup(SR) is the setup time of state register

Hence, the minimum clock cycle is :

$$\text{clock_cycle}=\max(T_{p1}, T_{p2}, T_{p3})=10.31\text{ns}$$

and the execution time of SMM with 20 state is:

$$\text{execution_time} = 10.31\times 20=206.2\text{ns}$$

One thing to remember here is this calculation is only for ideal case. In out implementation, something maybe be changed, such as we do not have memory address generation component. So, in other resource allocation, we can do the similar tasks to get clock cycle and execution time, but will give different critical path candidates.

5. RTL Synthesis and Results

In the [ACCE01], it defines five styles in the RTL synthesis that includes behavioral RTL, storage-mapped RTL, function-mapped RTL, connection-mapped RTL and exposed-control RTL. The exposed-control RTL is the architecture description just like the Figure 5 that includes the datapath and control unit.

The key points in [ACCE01] are:

(1) Five well-defined style to separate the synthesis tasks;
(2) User-defined resource constraint on which synthesis tasks are based
(3) Bus-based architecture;

There are three major synthesis tasks during the RTL synthesis: allocation, scheduling, and binding. Allocation determines the number of the resources, such as storage units, busses, and function units, which will be used in the implementation. 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). After these three tasks, we get the RTL style 4 description that does not include control unit part.

In our implementation, the resource allocation is given by user.

In our experiments, we use the following cases:

1. Bind all variables into register files and arrays to memories ;
2. Assign some variables (in/out buffers etc.) to special registers and bind others into register files;

we use SpecC language [GZDG00] as our test input/output language. We have developed the synthesis tools, which include scheduling, storage binding, function unit binding and interconnection binding, based on [ACCE01].

We can also use the following cases:
1. Using multi-cycle register files and memories (Not support now in scheduling and storagind binding)
2. Using pipelined functional units
3. Use datapath pipelining
4. Control pipelining
5. Status pipelining

Unfortunately, till now, these features are not supported by both SpecC and our tools. We will test all supported cases to find the performance difference of different architectures through our tools. For other cases, we will test them in future when they are available in SpecC and our tools.

5.1 RTL synthesis

5.1.1 Scheduling

In [SHGA01], there is a detail description and implementation of scheduling based on the [ACCE01].

Let SMM go through the scheduling tools, we can see scheduler splits some states into several sub-states. The number of sub-states and how and where to split the state are depended on different given resources.

Table 2 gives the total number of states after scheduling task for different resources. There are 18 states before scheduling. Also, clock cycle and execution time are given for different resource.

Here we have to mention the address generation of memories. To keep our architecture simple, we use SpecC-only bit-vector concatenation operation ***jj* @ *kk*** in the code as the index to access array. As a result, we use a functional unit ***CONCAT*** in our library that should be finish the operation in one state.

In our implementation, we can not access the two 3-bit slices of the same port (***count***) at the same time. So during the generation of the index of memory, we have to use two read ports for the ***count***. This is also caused by RTL constraint. In gate level, we can use two slices of the same port simultaneously.

5.1.2 Binding

Binding task includes storage binding, functional unit binding and interconnection busing.

For the storage binding, the arrays ***A***, ***B*** and ***C*** are assigned to the dedicated memories. For other variables, in one case we assign all of them into register files. In another case we assign some dedicated registers for some special variable. Some are connecting memory and port, like ***in1***, ***in2***, ***out2***. Some are for memory output to the functional unit, like ***a*** and ***b***. These registers can not used by other variables. Also, For other variables, we use register file as storage unit.

Besides the original variables in the design, we should also consider the temporary variables generated during the scheduling. Some temporary variables become wires during synthesis if they are generated and used in one state. For other temporary variables, they are assigned to register files.

During the storage binding, the lifetime overlapping and weight will be considered [ZHGA01].

For functional unit binding and interconnection binding, [XIGA01] and [YUGA01] give the procedure and methods in detail.

5.2 Experiments Results

5.2.1 All in RF, Memory

We use 2 RFs, 3 memories, 1 ALU, 1 Multiplier, 1 Comparator, 2 Concatenations and 3 busses this case. The Figure 4 gives the binding results for this case.

5.2.2 RF with Dedicated Registers

In this case, we assign some dedicated registers for some special variable. Some are connecting memory and port, like ***in1***, ***in2***, ***out2***. Some are for memory output to the functional unit, like ***a***

and ***b***. These registers can not used by other variables. Also, For other variables, we use 1 RF to store them. For functional unit, we still use 1 ALU, 1 Multiplier, 1 Comparator and 1 Concatenations. Also, 3 busses are allocated.

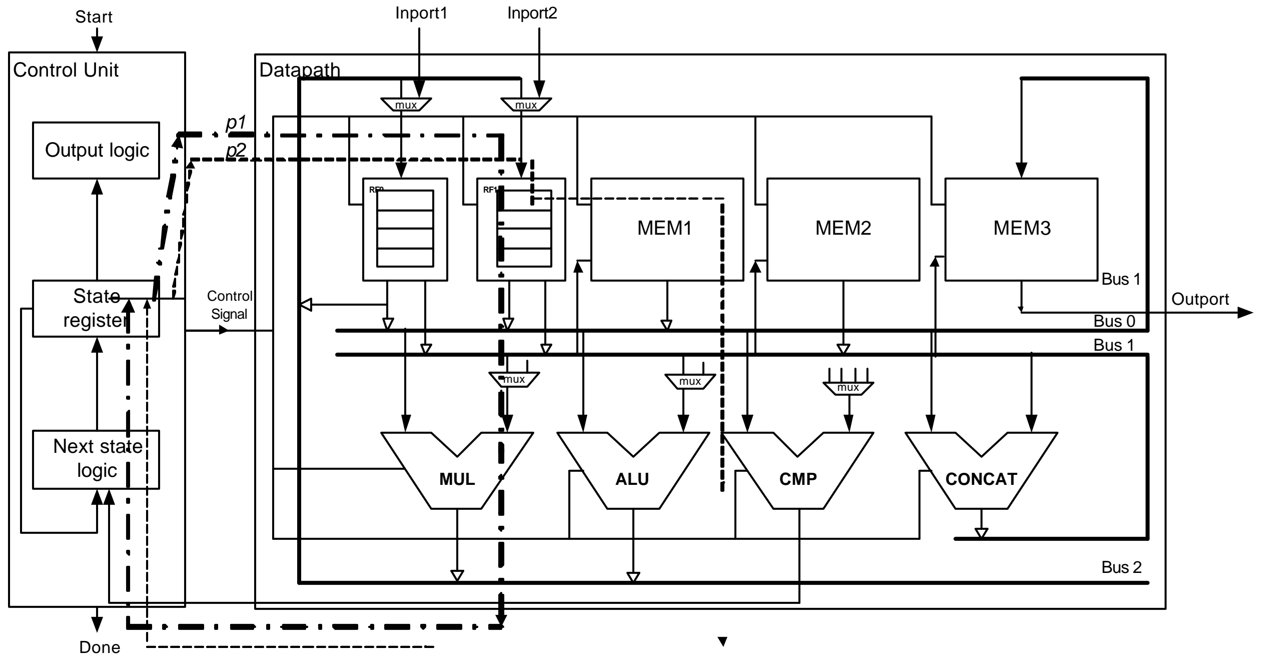

Figure 4: Architecture with RFs, Memories w/o dedicated registers

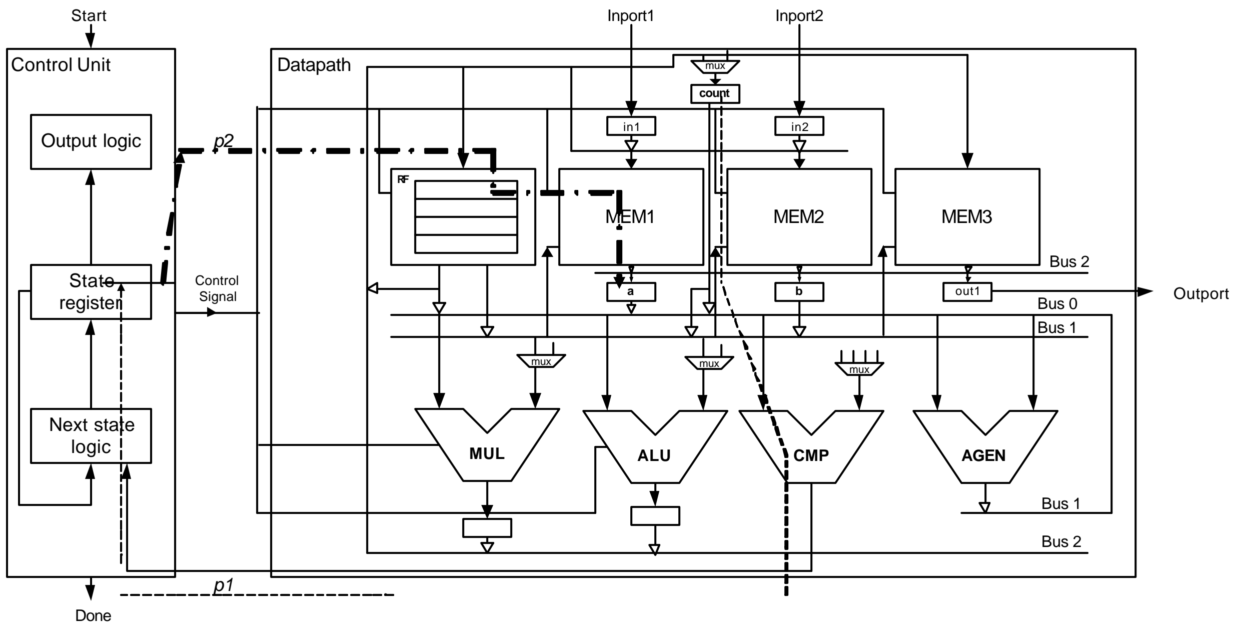

Figure 5: Architecture with RFs, Memories w/ dedicated registers

Resource	T_{p1} (ns)	T_{p2} (ns)	Clock cycle(ns)	Total states	Execution time(ns)
RF and MEM only	***11.38***	10.82	11.38	28	318.64
RF, MEM and Reg.	6.41	***10.85***	10.85	29	314.65

Table 2: Different binding results

The Figure 5 gives the binding results for this case.

The results are given in the Table 2. Due to the different architectures, the performances are also different. In the case 2, RF with Dedicated Registers is the best architecture for SMM in our testing.

Maybe with pipeline mechanism will have better performance, we will test them later after we improve our synthesis tools.

6. Summary

This report describes the procedure of RTL design and synthesis based on the accellera RTL semantics using the Sequential Matrix multiplication. The test results show that we can use this semantics as a main part during the RTL synthesis.

In the future, we can improve both SMM design or synthesis tools to get better performance.

Reference

[GZDG00] D. Gajski er al. *SpecC: Specification Language and Design Methodology*, Kluwer Academic Publishers, 2000

[ACCE01] Accellera C/C++ Working Group. *RTL Semantics: Draft Specification*. Feburary 2001.

[GERS00] A. Gerstlauer *SpecC Modeling Guidelines*, University of California, Irvine, Technical Report ICS-00-xx, September 2000

[GAJS97] D. Gajski *Principles of Digital Design*, Prentice-Hall, Inc, 1997

[GDPD01] A. Gerstlauer, R. Dö mer, J. Peng, D. D. Gajski, *System: Design: A Practical Guide with SpecC*, Kluwer Academic Publishers, Boston, MA, ISBN 0-7923-7387-1, June 2001

[SHGA01] Dongwan Shin, Daniel D. Gajski, *Scheduling in RTL Design Methodology,* University of California Irvine, Technical Report ICS-TR-01-51, July 2001.

[YUGA01] Haobo Yu, Daniel D. Gajski, *Interconnection Binding in RTL Design Methodology,* University of California Irvine, Technical Report ICS-TR-01-38, June 2001.

[ZHGA01] Pei Zhang, Daniel D. Gajski, *Storage Binding in RTL Synthesis,* University of California Irvine, Technical Report ICS-TR-01-37, August 2001.

[XIGA01] Qiang Xie, Daniel D. Gajski, *Function Binding in RTL Design Methodology,* University of California Irvine, Technical Report ICS-TR-01-37, June 2001.

Appendix: SMM Behavioral Mode Source Code in SpecC

```
//////////////////////////////////////////////////////////////////////
// Design:  Sequential Matrix Multiplication
// Author:  Pei Zhang, University of California, Irvine
// Purpose: RTL C++ semantics experiment
// File:    smm.sc
// Date:    Apr. 1, 2001
//////////////////////////////////////////////////////////////////////

import "lib";

#define ii count[8:6]
#define jj count[5:3]
#define kk count[2:0]

behavior smm(     in event    clk,
            in bit [0:0]      rst,
            in bit[31:0]      Inport1,
            in bit[31:0]      Inport2,
            out bit[31:0]     Outport,
            in bit[0:0]       Start,
            out bit[0:0]      Done)
{
    note smm.scheduled = "0";
    note smm.fubind = "0";
    note smm.regbind = "0";
    note smm.busbind = "0";

    note smm.clk = "clk";
    note smm.rst = "rst";
    note smm.Inport1 = "data";
    note smm.Inport2 = "data";
    note smm.Outport = "data";
    note smm.Start = "ctrl";
    note smm.Done = "ctrl";

      void main(void) {
            bit[31:0] A[64], B[64], C[64];
            bit[31:0] P, a, b;
            bit[31:0] Sum;
            bit[31:0] count;
            bit[31:0] in1, in2, out1;

            enum state { S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10,
            S11, S12, S13, S14, S15, S16, S17 } state;

            state = S0;

            while (1) {
                  wait(clk);
                  if (rst) {
                        state = S0;
                  }
                  switch (state) {
```

```
case S0:
      Done = 0b;
      Outport =0 ;
      count = 0b;
      if (Start == 1b)
            state = S1;
      else
            state = S0;
      break;
case S1:
      in1 = Inport1;
      in2 = Inport2;
      state = S2;
      break;
case S2:
      A[jj @ kk] = in1;
      B[jj @ kk] = in2;
      state = S3;
      break;
case S3:
      if (count == 63)
            state = S5;
      else
            state = S4;
      break;
case S4:
      count = count + 1;
      state = S1;
      break;
case S5:
      count = 0;
      state = S6;
      break;
case S6:
      a = A[ii @ kk];
      b = B[kk @ jj];
      state = S7;
      break;
case S7:
      P = a * b;
      if (kk == 0)
            state = S8;
      else
            state = S9;
      break;
case S8:
      Sum = P;
      state = S9;
      break;
case S9:
      Sum = Sum + P;
      if (kk == 7)
            state = S10;
      else
            state = S11;
      break;
case S10:
```

```
                                        C[ii @ jj] = Sum;
                                        state = S11;
                                        break;
                                case S11:
                                        if (count != 511)
                                                state = S12;
                                        else
                                                state = S13;
                                        break;
                                case S12:
                                        count = count + 1;
                                        state = S6;
                                        break;
                                case S13:
                                        Done = 1b;
                                        count = 0;
                                        state = S16;
                                        break;
                                case S14:
                                        out1 = C[jj @ kk];
                                        state = S15;
                                        break;
                                case S15:
                                        Outport = out1;
                                        state = S16;
                                        break;
                                case S16:
                                        if (count == 63)
                                                state = S0;
                                        else
                                                state = S17;
                                        break;
                                case S17:
                                        count = count + 1;
                                        state = S14;
                                        break;
                        }
                }
        }
};
```