**Chapter 19**

**CCM Assembly**

**Marek Perkowski and Qihong Chen**

The programs written for the CCM are in the form of a list of binary bytes which are CCM instructions encoded by the programmer. This method of programming is very tedious and error prone. All instruction encoding formats had to be remembered in order to use them, and it is painful to look at and maintain this kind of programs. Therefore, a very simple assembly language, called CCM assembly was created. With the CCM assembly, the programmer needs only to remember the name of the CCM instructions. This programming methodology still requires the programmer to think in terms of registers and individual instructions. This chapter will describe the CCM assembly and will give some examples.

**19.1. CCM Assembly**

In the CCM assembly, one instruction has two or three fields and occupies one line; the fields are separated by blank spaces; and the comments can be added after the last field of the instruction. The names of the instructions, called mnemonics, occupy the first field in an instruction line. The subsequent fields are the operands of the instruction.

In the syntax of CCM assembly, keywords are represented in upper case, operands are represented in lower case, and they should be substituted by actual operands when they are used.

There are four instructions in the CCM assembly, and their corresponding instructions are discussed in detail in section 18.3. The syntax of them is described as follows:

• ENABLE/DISABLE: The ENABLE/DISABLE instructions are used to enable/disable tri-state buffers in the CCM. The corresponding CCM instruction is set tri-state buffers. The syntax is as follows:

ENABLE control\_signal\_name

and

 DISABLE control\_signal\_name

 where control\_signal\_name is one of ENADDRA, ENADDRB, ENIFIFOA, ENIFIFOD, ENMEMAWR, ENMEMBRW, ENILUA and ENILUB (see section 18.3.2).

• SET: The SET instructions are used to load the data into registers and address units from their inputs. The corresponding CCM instructions are set accumulator and set registers. The syntax is as follows:

SET register\_name, operand

where register\_name is one of ADDRA, ADDRB, ADDRR, WATER, RIGHT, INST, ACCU, CONF, PRPO; the operand in the syntax is a binary number that will show up on IBus, the width of this number depends on the register of the instruction (see section 18.3.3).

• EXEC: The EXEC instruction is used to execute only one cube operation. The corresponding CCM instruction is execute. The syntax is as follows:

EXEC operand

where operand is a 30-bit-wide binary data that will show up on IBus when the instruction is executed (see section 18.3.4).

• LOOP: The LOOP instruction is used to execute multiple cube operations continuously without fetching the input FIFO. The corresponding CCM instruction is loop. The syntax is as follows:

 LOOP operand

where operand is a 30-bit-wide binary number that will show up on IBus when the instruction is executed (see section 18.3.5).

Two more efforts are made to make the CCM assembly program easy to understand:

• The symbol slash ('-') can be inserted into binary number. The program that interprets the CCM assembly should ignore these slashes.

**Example 19.1** The functions of the following two instructions are identical.

set inst 000000000000100000000

set inst 00-0-0-0-0000-0001-0000-0000

It is obvious that the second instruction is easier to understand.

• The unused bits at the end of a binary number can be omitted. The program that interprets the CCM assembly should fill these bits with 0's or 1's (only the omitted bits of the binary number, that is the operand of “set water" instruction, will be filled with 1's, see the definition of water signal in section 16.4.1).

**Example 19.2**The functions of the following two instructions are identical.

set water 000011111111111

set water 0000

Both of these two instructions mean that only first 4 ITs are used; and it is obvious that the second instruction is easier to understand.

**Example 19.3** The functions of the following two instructions are identical.

set right 111100000000000

set right 1111

The CCM assembly is a very simple assembly language. It is easy to develop an interpreter for the CCM assembly, this CCM interpreter accepts CCM assembly programs as its input, then executes CCM instructions by calling proper VELOCE runtime library routines, and passes the result back to the host program.

For making the CCM easy to use, the CCM runtime library, a set of library calls which can be called from C/C++ programs, need to be developed by the next student in the CCM project group. For example, this library should have a routine (function) to carry out sharp operation on two arrays of cubes. The CCM runtime library can hide unnecessary details about the CCM hardware from the programmer, and enables programmers to think at a higher level and develop applications more efficiently.

**19.2 Examples of Using CCM Assembly**

This section presents several examples of solving some cube operation problems in CCM assembly. These examples serve as a tutorial about how to use the CCM to solve the problems. All these programs have an assumption that the CCM is reset, which means that all registers and control signals of tri-state buffers are zeroed.

**Example 19.4** Assuming two cubes A=ab and B=b$\overbar{c}$, where a, b and c are binary variables. Write a CCM assembly program to calculate the intersection of cubes A and B.

**Solution.** The intersection operation is a simple combinational cube operation, it does not have pre-relation/pre-operation (see section 16.5), and

rel= text{xxxx}, and\_or= x , bef=0001

Since we process cubes with 3 binary variables, we have

water= 000-1111-1111-1111, rightedge = 111-xxxx-xxxx-xxxx

Cubes A and B can be described in positional notation as:

 A=ab 01-01-11, B= bc 11-01-10

Therefore, the program is as follows:

1. enable enififod

2. set conf 100000100

3. set water 000

4. set right 111

5. set inst 00-0-0-0-0000-0001-0000-0000 ;intersection

6. set accu 01-01-11 ; cube A

7. exec 11-01-10 ; cube B

Line numbers are not part of the CCM assembly; they are used here to help identify specific lines of code in our discussion. Everything to the right of the semi-colon “;" are the comments.

Line 1 enables the tri-state buffer from the bus IBus to the DBusA, which means the data existing on the IBus exist also on the bus DBusA at the same time. Line 2 sets ASrc=OSrc=0, toOFifo=1 and enFinish=1, which means that the inputs of registers accu and data are connected to the bus DBusA (see Figure 18.3), and the CCM will write the result(s) to the output FIFO. This instruction means also that the CCM is enabled to generate finish word. In this example, we don't care about the other bits of the config register.

Lines 3 to 5 are very straightforward. Line 6 lets the CCM load the cube A into the accumulator. Line 7 lets the CCM load the cube B into the data register and execute the cube operation (defined by inst and prpo registers) on the operand cubes (stored in the Accu and the Data registers). Please note that the CCM does not know (or does not care) the configuration of the datapath, it is the programmer's responsibility to set config register and the control bits of tri-state buffers correctly before issuing the EXEC command. In this example, these signals are set in Lines 1 and 2 of above program.

This is an example of executing pattern (a) (see section 18.1), and the dataflow mode used by this example is shown in 18.2 (b) (see section 18.2.4).

**Example 19.5** Assuming four cubes A=ac, B=ad, C=bd and D=cd, where a, b, c and d are binary variables. Write a CCM assembly program to calculate the intersection of these four cubes: A. B. C.D (Try to use as few instructions as possible).

**Solution.** The program is as follows:

1. enable enififod

2. set water 0000

3. set right 1111

4. set inst 00-0-0-0-0000-0001-0000-0000 ; intersection operation

5. set accu 01-11-01-11 ; cube A

6. set conf 000010010 ; sent result back to ACCU

7. enable enIluB

8. exec 01-11-11-01 ; cube B

9. exec 11-01-11-01 ; cube C

10. set conf 100000100

11. disable enIluB

12. exec 11-11-01-01 ; cube D

As mentioned before, there is an assumption that the CCM is reset, which means the config register is set to 000000000. Lines 2 to 5 set the registers water, right\_edge, inst and accu. Lines 6 and 7 set a feedback path from the output of the ILU to the input of the Accu and let the CCM write the results back to the Accu.

Line 8 let the CCM calculate A.B and write the result back to Accu. Line 9 let the CCM calculate [Accu].C and write the result back to Accu, where [Accu] represents the content of the Accu. At this time, the content of Accu is the intersection of cubes A, B and C.

Line 10 let the CCM write the result to the output FIFO; and enables the CCM to generate “finish word". Line 11 breaks the feedback path created by Lines 6 and 7. Line 12 let the CCM calculate [Accu].D and write the result to the output FIFO.

This is an example of executing pattern (c) (see section 18.1), and the dataflow mode used by this example is shown in 18.2 (c) (see section 18.2.4).

**Example 19.6** Let us assume two cubes A= $\overbar{c}$ and B=bd, where a, b, c and d are binary variables. We present a a CCM assembly program to calculate the basic sharp of cubes A and B: A#basicB (Cube A is stored in the Accu and cube B is stored in the data register).

**Solution.** The program is as follows:

 1. enable enififod

2. set conf 100000100

3. set water 0000

4. set right 1111

5. set inst 00-1-0-0-0010-0011-0010-0011 ; basic sharp [Accu]#[Data]

6. set accu 11-11-10-11 ; cube A

7. exec 11-01-11-01 ; cube B

This example is the same as Example 19.4 except that it uses a different operation on different operand cubes. This is an example of executing pattern (b) (see section 18.1), and the dataflow mode used by this example is shown in 18.2 (b) (see section 18.2.4).

**Example 19.7.** Write a program in the CCM assembly to calculate the basic sharp operation of two cubes A and B: B#basic; A, where cubes A and B are the same as in the previous example (Again, cube A is stored in the Accu, and cube B is stored in the data register).

**Solution**. It can be seen from the definition of sharp that (A#basicB)  (B#basicA), and BbasicA is not listed in Table 16.1. This is a “new" cube operation. This operation is very useful to execute the sharp operation on an array of cubes and a cube.

The functions rel, bef, act and aft are 2 inputs Boolean function f(ai,bi). The 4 output values of each function are corresponding to minterms , respectively, where ai comes from operand cube stored in Accu, and bi comes from operand cube stored in data register.

Now, we want to perform sharp operation [Data] #basic[Accu] (where [Data], [Accu] represent the contents of data and Accu registers, respectively), therefore, we have to substitute ai with bi and bi with ai in function f(ai , bi) in order to obtain function f(bi,ai). Its minterms are (bi is the most significant bit now):  and  respectively. We have to use the format of f(ai , bi) to represent f(bi,ai) in the instruction, and we can obtain it by swapping minterms . Therefore, we just swap second and third output values of the functions rel, bef, act and aft.

Therefore, only Line 5 of previous example needs to be changed. The whole program is as follows:

 1. enable enififod

2. set conf 100000100

3. set water 0000

4. set right 1111

5. set inst 00-1-0-0-0100-0101-0100-0101 ; basic sharp [Data]#[Accu]

6. set accu 11-11-10-11 ; cube A

7. exec 11-01-11-01 ; cube B

**Example 19.8** Write a program in the CCM assembly to calculate basic sharp operation: , where B is an array of 3 cubes, A is a cube, and a, b, c and d are binary variables (Try to use as few instructions as possible).

**Solution.** The program is as follows:

 1. enable enififod

 2. set conf 000000100

 3. set water 0000

 4. set right 1111

 5. set inst 00-1-0-0-0100-0101-0100-0101 ; basic sharp [Data]#[Accu]

 6. set accu 11-11-10-11 ; cube A

 7. exec 01-11-11-11 ; cube B1

 8. exec 11-01-11-11 ; cube B2

 9. set conf 100000100 : generate finish word

 10. exec 11-11-01-11 ; cube B3

This is an example of executing pattern (d) (see section 18.1).

**Example 19.9** Write a program in the CCM assembly to calculate disjoint sharp operation: where A is a cube,  is a array of cubes, and a, b and c are three binary variables. Please note that this example shows how to use the loop instruction.

**Solution.** The program is as follows:

1. set conf 000000000

2. enable enififoa

3. set addrb 0

4. disable enififoa

5. enable enififod

6. set water 000

7. set right 111

8. set prpo 1-1110-10-0-0101--0-0100-01-0-0000 ; disjoint sharp [D]#d[A]

9. set inst 11-1-0-0-0100-0101-0100-0001 ; disjoint sharp [D]#d[A]

10. set accu 01-01-11 ; cube B1

11. set conf 001000001 ; write result to MEMB

12. enable enIluB

13. disable MemBRW

14. exec 11-11-11 ; cube A1

15. disable enIluB

16. enable enaddrb ; [AddrB] => [AddrR]

17. set addrr 0

18. disable enaddrb

19. enable enIFifoA

20. set addrb 0

21. disable enIFifoA

22. set accu 10-11-01 ; cube B2

23. disable enIFifoD

24. set conf 011101001 ; memB=>ILU=>memA

25. enable MemBRW

26. disable MemARW

27. enable enIluA

28. loop 0

29. disable MemBRW

30. disable enIluA

31. enable enaddra ; [AddrA] => [AddrR]

32. set addrr 0

33. disable enaddra

34. enable enIFifoA

35. set addra 0

36. disable enIFifoA

37. set conf 000000000

38. enable enIFifoD

39. set accu 11-10-10 ; cube B3

40. disable enIFifoD

41. set conf 110000100 ; memA=>ILU=>OFifo

42. enable MemARW

43. loop 0

This is a little bit more complex example. Lines 2 to 4 set AddrB to 0. Lines 5 to 9 set water right edge, inst and prpo registers. Please note that this example shows how to carry out prerelation/preoperation.

Lines 10 to 15 calculate A # B1 and write result (called $\vec{I1}$ here) to MEM\_B (please note that the result cannot write to MEM A at this time because the execute instruction needs DBusA to load cubes A). The data\_ow mode used in this step is shown in Figure 18.2(d) (see section 18.2.4).

Lines 16 to 30 calculate $\vec{ I1} \#$ B2 and write result (called $\vec{I2}$ to MEM A. This is an example of using loop instruction. Now what we want to do is to calculate the following operation:



Because the array of cube $\vec{ I1} $ is stored in memory bank MEM\_B, we can use the “loop" instruction to carry this out.

The loop instruction loads one cube from one of two memory banks (determined by signal enMemA, enMemB, MemARW and MemBRW, see section 18.2) to the Data register. Then the memory address pointer will be increased by 1, and the operation currently set in the inst and the prpo registers is executed on two cubes stored in Accu and Data registers. After the operation is done, the GCU checks if the loop operation is done by comparing the content of AddrR with AddrA or AddrB. If their contents are not the same, the GCU will load one cube from the memory to the Data register again, and will repeat the whole process until the contents of AddrR and AddrA (or AddrB) become the same.

The array of cubes $\vec{ I1} $ is stored in MEM\_B. For using loop instruction, we need to set the content of AddrR to the number of cubes of $\vec{ I1}$, which is currently stored in AddrB. Lines 16 to 18 copy the content of AddrB to AddrR. After that, the content of AddrB is set to 0 (Lines 19 to 21) to point the beginning of $\vec{ I1}$. Line 22 loads cube B2 into Accu.

The operation $\vec{ I1} $#B2 is not the last operation of this example, and the result will be used in the subsequent operation, therefore, the result array of cubes of the operation will be stored in MEM\_A. The dataflow mode used in this step is shown in Figure 18.2 (f) (see section 18.2.4). Lines 23 to 27 set the data flow mode.

The loop instruction is issued in Line 28. Lines 29 and 30 remove the drivers of buses DBusA and DBusB for the subsequent operation since the subsequent operation will use different dataflow mode, which means that the driver of DBusA and DBusB will be changed (remember for setting new bus driver, we have to remove the previous driver first, see section 18.3.2).

Lines 31 to 43 calculate $\vec{ I2} $#B3 in the similar way with Lines 16 to 30. The difference is that the GCU loads the array of cubes $\vec{ I2} $ from MEM\_A, and write the results to the output FIFO this time. By comparing to Line 16 to 30, Line 31 to 43 are not hard to understand. The dataflow mode used in this step is shown in Figure 18.2 (e) (see section 18.2.4).

The operation. This example is used as a test program to test the entire CCM (see section 20.2.6).

**Example 19.10** Write a program in the CCM assembly to calculate the following operation:





where A, B and C are three arrays of cubes, and a, b, c and d are four binary variables.

**Solution** The program is as follows:

1. set conf 000000000

2. enable enififoa

3. set addrb 0

4. disable enififoa

5. enable enififod

6. set water 0000

7. set right 1111

8. set inst 00-0-0-0-0000-0001-0000-0000 ; intersection

9. set accu 01-01-11-11 ; cube A1

10. set conf 001000001 ; write result to MEMB

11. enable enIluB

12. disable MemBRW

13. exec 11-01-01-11 ; cube B1

14. exec 11-11-01-01 ; cube B2

15. exec 01-11-11-01 ; cube B3

16. set accu 11-01-01-11 ; cube A2

17. exec 11-01-01-11 ; cube B1

18. exec 11-11-01-01 ; cube B2

19. exec 01-11-11-01 ; cube B3

20. set accu 11-11-01-01 ; cube A3

21. exec 11-01-01-11 ; cube B1

22. exec 11-11-01-01 ; cube B2

23. exec 01-11-11-01 ; cube B3

24. disable enIluB

25. enable enaddrb ; [AddrB] => [AddrR]

26. set addrr 0

27. disable enaddrb

28. enable enIFifoA

29. set addrb 0

30. set conf 001101100 ; memB=>ILU=>OFifo

31. enable MemBRW

32. set accu 10-11-11-11 ; cube C1

33. loop 0

34. set addrb 0

35. set accu 11-10-11-11 ; cube C2

36. loop 0

37. set addrb 0

38. set accu 11-11-10-11 ; cube C3

39. set conf 101101100 ; generate finish word

40. loop 0

This program is very straightforward. Lines 1 to 23 calculate  and write the results (called $\vec{ I}$) to MEM\_B. Lines 24 to 38 calculate $\vec{ I}.\vec{ C} $and write the result to the output FIFO. This example is used as a test program to test the entire CCM (see section 20.2.6), the result is also shown there.