**Chapter 18**

**The Design of Cube Calculus Machine Co-processor**

**Marek Perkowski and Qihong Chen**

In our design, the cube calculus machine acts as a co-processor to the host computer, and it will be realized on the VELOCE system. Therefore, the unknown architecture of VELOCE is our only design constraint.

The input FIFO and the output FIFO on VELOCE allowed us to significantly simplify the communication between the host and the VELOCE. This feature lets the host and the VELOCE to work asynchronously. Therefore, we use these two FIFOs as the way of communicating between the host and the CCM.

By using the input and output FIFOs, the communication between the host and the CCM is as follows. The host just puts instructions into the input FIFO, and receives the results from the output FIFO. On the other side, the CCM takes an instruction from the input FIFO, executes this instruction and puts the results back into the output FIFO. This is shown in Figure 18.1.



*Figure 18.1. Communication between the host and the CCM.*

As described in Chapter 18, the width of FIFOs is 32 bits, which means the data transferred between the host and the CCM is 32-bit-wide. At this time, we want to keep the CCM as simple as possible, so we just use fixed-length instructions, and the width of all instructions will be 32 bits. This means that the op-code and the actual data of a CCM instruction are both included in one 32-bit-wide word.

**18.1 Executing Patterns**

Before we design the CCM instructions, we need to know what kind of execution patterns happen often in practical applications of the CCM, and how can our design be able to execute cube operations efficiently for these execution patterns.



*Figure 18.2 Cube Operation Patterns*

 Pattern (a) (Figure 18.2 (a)) is the general form of combinational cube operations. A combinational cube operation produces one resultant cube.

Pattern (b) (Figure 18.2 (b)) is the general form of sequential cube operations. A sequential cube operation produces as many as n resultant cubes, where n is the number of variables in the operand cubes.

Pattern (c) (Figure 18.2 (c)) is used in some combinational cube operations on an array of cubes, for example, the result of intersection operation on an array of cubes (A1. A2………An) is a single cube or an empty cube.

Pattern (d) (Figure 18.2 (d)) can be used both in combinational and sequential cube operations. A combinational operation example is a cofactor operation on an array of cubes:



A sequential operation example is a sharp operation on two arrays of cubes, and this is the most complicated case:



where  is the result of operation (A1 # B1 A2 # B1 ….. Am # B1). As we can see from the equation, the basic step for sharp operation on two arrays of cubes is the sharp operation on one array of cubes and one cube. This is what Pattern (d) describes. Therefore, the pattern of sharp operation on two arrays of cubes repeats pattern (d) as many times as the number of cubes in the array of cubes $\vec{B}.$

It can be seen from these execution patterns that the same cube operation is executed very many times before another kind of cube operation is executed in a practical application. Also, sometimes one operand cube does not change in subsequent operations or comes from the result of the previous cube operation. Thus, we have the following design considerations:

* We need an accumulator register for pattern (c), this accumulator can be set by the user or it receives the data being the result of a previous cube operation. As discussed in Chapter 16, most cube operations have two operand cubes. Thus, we need another general data register to store another operand cube.
* The CCM can execute cube operations by just accepting operand cube(s) without re-setting the instruction register.

**18.2 The Design of the CCM**

The block diagram of our design is shown in Figure 18.3. In this design, there are 5 data buses, 2 banks of memories, Global Control Unit (GCU), ILU and it's controller Operation Control Unit (OCU), two address units, registers, tri-state buffers and three multiplexers. The following section will discuss them in detail. The control signals are not shown in the figure, and they are all generated by GCU, which means that all components of the CCM work together under the control of the GCU.



*Figure 18.3: Block Diagram of Our Design*

**18.2.1 Data Bus**

Five data buses are used in the CCM. They are described as follows:

*IBus* is the short name of input FIFO data bus. The CCM receives the instruction from the input FIFO through this data bus. Only input FIFO can write this data bus.

*OBus* is the short form of output FIFO data bus. The CCM puts the results into output FIFO through this data bus. Only the ILU can write this data Bus.

*ABus* is the short name of Address data bus. The CCM sets the contents of two address units AddrA and AddrB and one address register AddrR through this data bus. The input FIFO and two address units can write this data bus, and they are controlled by three control signals: EnIFifoA, EnAddrA and EnAddrB, which control the corresponding tri-state buffers.

*DBusA* is the short name of Data Bus A. This data bus connects to the input FIFO, memory bank A (MEM\_A) and the input and output of the ILU. The input FIFO, MEM\_A and the ILU can write this data bus, and they are controlled by three control signals: EnIFifoD, MemARW, and EnIluA, which control the corresponding tri-state buffers.

*DBusB* is the short name of Data Bus B. This data bus connects to the memory bank B (MEM\_B), the input and output of the ILU. The MEM\_B and the ILU can write this data bus, and they are controlled by two control signals: MemBRW and EnIluB, which control the corresponding tri-state buffers.

The examples that show how to use these buses will be given in section 19.2.??

**18.2.2 Memory and Address Units**

In this design, we use two banks of memory, MEM\_A and MEM\_B, to store intermediate results. Each bank of memory connects to one data bus: MEM\_A connects to the DBusA and MEM\_B connects to the DBusB.

The address signals of MEM\_A come from Address Unit A (AddrA for short). The address signals of MEM\_B come from Address Unit B (AddrB for short). The contents of these two address units can be set or incremented under the control of GCU. These two address units are realized by 18-bit-wide loadable-up-counters. A Address Register (AddrR for short) is used to store an address data shown on ABus. As we mentioned before, ABus can be written by IBus, AddrA or AddrB, so the address data could be one of these three sources. The example of using AddrR is shown in section 19.2.

The control signal MemARW controls the MEM\_A in read mode or write mode, which means the MemARW can read from or write to the data bus DBusA. When the MemARW is set, the MEM\_A is in read mode, otherwise, the MEM\_A is in write mode. The control signal MemBRW controls the MEM\_B in the same manner.

**18.2.3 Registers**

There are six registers used by ILU in our design (AddrR register is mentioned in the above section). They are described as follows:

*Accu* is an accumulator register used to store one operand cube for the cube operation. It is 30 bits wide.

*Data* is a general data register used to store one operand cube for the cube operation. It is 30 bits wide.

*Water* is a 15 bits wide register used to store water signals.

*Rightedge* is a 15 bits wide register used to store right\_edge signals.

*Inst* is a 21 bit wide register used to store cube operation instruction. The content of the inst register is shown in Figure 18.4. The meaning of these nine fields is the following:

p1 field represents whether the first pre-relation/pre-operation is valid or not.

p2 field represents whether the second pre-relation/pre-operation is valid or not when p1 = 1.



*Figure 18.4: The content of instruction register*

*sc* is sequential/combinational bit. When it is 1, the operation is a sequential operation, otherwise, the operation is combinational operation.

*pm* is prime bit. When it is 1, the operation is a complex combinational operation, otherwise, the operation is a simple combinational operation.

*ao* is *and\_or* bit. When it is 1, the relation type of the operation is “AND", otherwise, the relation type is “OR".

*rel, bef, act* and *aft* are the four bitwise functions used to describe the operation (see Chapter 15 and 16).

*PRPO* is a 24 bits wide register used to store two pairs of pre-relation/pre-operation. The content of the prpo register is shown in Figure [18.5]. The meaning of these eight fields are as follows:



Figure 18.5: The content of prop register

pand\_or1, prel1, pcmp1, pval1 and poper1 are the partial pre-relation type, partial pre-relation, pre-relation compare type, pre-relation compare value and pre-operation for the first pair of pre-relation/pre-operation, respectively.

pand\_or2, prel2, pcmp2, pval2 and poper2 are the partial pre-relation type, partial pre-relation, pre-relation compare type, pre-relation compare value and pre-operation for the second pair of pre-relation/pre-operation, respectively.

For the more information about pre-relation/pre-operation, please see section 16.7.

As shown in Figure 18.3, the input of these six registers is connected to either DBusA or DBusB. The signal ASrc controls to which bus the Accu is connected. The signal OSrc controls to which bus the data register is connected. Every register has a load signal used to load data from its inputs, and all load signals are generated by GCU (they are not shown in Figure 18.3).

There is one more register called config register (not shown in Figure 18.3), and it will be discussed in section 18.3.

**18.2.4 Dataflow mode**

A simplified block diagram of the CCM is shown in Figure 18.2.3 (a). It can be seen that the CCM has two data buses that connect the input/output FIFOs, memory and data path (ILU in the CCM) together.

This two-bus structure has better performance than the single bus structure. Suppose we use single bus structure, which means that the input/output FIFOs, memory and ILU are connected together by one data bus. For a sequence of cube operations that read data from memory and write the results back to the memory, the algorithm would be:

1**. for (i=0; i++; i<n)**

**2. { set MEM be the writer and ILU be the reader of the data bus**

**3. read data from MEM to ILU**

**4. execute cube operation**

 **5. set MEM be the reader and ILU be the writer of the data bus**

**6. write data from ILU to MEM**

**7. }**

It is easy to observe that the lines 2 through 6 are executed n times. With our two data buses structure, the algorithm is now changed to:

**1. set** **MEM be the writer and ILU be the reader of the data bus A**

**2. set MEM be the reader and ILU be the writer of the data bus B**

**3. for (i=0; i++; i<n)**

**4. { read data from MEM to ILU through data bus A**

**5. execute cube operation**

**6. write data from ILU to MEM through data bus B**

**7. }**

This time, the lines 1 and 2 are outside of loop and are only executed one time in our architecture. Therefore, we improve the performance by using two data buses. Some of useful dataflow modes are shown in Figure 18.2.3 (b) to (f). The examples of using these modes are given in Chapter 19.



*Figure 18.6: The dataflow modes of the CCM*

**18.3 Instructions and Their Encoding**

The CCM has two categories of instructions called “CCM instructions", config instructions and execute instructions. The config instructions set the CCM to be ready to execute a specified cube operation. The execute instructions let the CCM executes cube operation(s) currently set in the instruction register.

There are three config instructions: Set Accumulator, Set Tri-state Buffers and Set Registers. And there are two execute instructions: Execute and Loop. This section will discuss these instructions and their encoding in detail (the examples are given in section 19.2).

**18.3.1 Set Accumulator**

The “set accumulator" instruction loads the data into the accumulator (Accu) from its input. The encoding of the instruction is as follows:



The first two bits “01" is the opcode of this CCM instruction. The 30-bit data in the instruction will be shown on the bus IBus. For loading the correct data into Accu, the control bits EnIFifoD and Asrc (see Figure 18.3) must be set properly (by issuing set tri-state buffers and set register instructions) before issuing this instruction. For example, when EnIFifoD is 1 and Asrc is 0, this instruction will load the data shown on bits 29 to 0 of the instruction word into the Accumulator.

**18.3.2 Set Tri-state Buffers**

The “set tri-state buffers” instruction is used to set the control bits of the tri-state buffers that control the data flow. Some useful dataflow modes are discussed in section 18.2.4. There are 8 bits of this kind in our design.

These 8 bits are registered by an 8 bits register in the CCM, and this register can be set by one CCM instruction. As we discussed in section 18.2.1, the three data buses (ABus, DBusA and DBusB) have more than one possible driver, but at any given time, there is only one driver for each data bus. If there were more than one driver for a given bus at a given time, then the FPGA chip would be destroyed permanently.

For protecting the hardware from destroying by a “bad program", in our design, the set tri-state buffers instruction can only set one control bit a time, and a special circuit is used to check potential contention (multiple drivers). The idea of this special circuit is shown in Figure 18.7.

*Figure 18.7: Avoiding contention which would result from multiple drivers*



*Figure 18.8: Timing diagram of special circuit for avoiding bus contention*

As shown in Figure 18.7 two control bits (cntrbit1 and cntrbit2) control two tri-state buffers that drive one data bus (the tri-state buffer and data bus are not shown in the figure). At any given time, at most one control bit can be set to 1.

For understanding how this circuit works, let's see a timing diagram shown in Figure 18.8. At time point 0ns, all signals are 0. For setting cntrbit1 to 1, the signal databit is set to 1 first at 25ns, then there is a raising edge on the signal ld\_bit1 at 50ns. As shown in the figure, after a little delay, the signal cntrbit1 is set to 1.

Now let us try to set cntrbit2 to 1 to create a bus contention. The signal databit is set to 1 at 125ns, then there is a raising edge on the signal ld\_bit2 at 150ns. Since one of two control bits (cntrbit1) is 1, then both inputs of gate 2 (NAND gate) are 1's, thus, the output of gate 2 is 0. Then the raising edge on the signal ld\_bit2 cannot go through the gate 4 (AND gate), which means that the raising edge cannot reach the clock input of DFF2 (D flip-flop), thus Q output of the DFF2 does not change. Therefore, this circuit ensures that at most one control bit can be set to 1.

For setting cntrbit2 to 1 at this time, two steps are needed. First step is to set cntrbit1 to 0 at 250ns (please note that the signal databit is set to 0 before the raising edge on the signal *ld\_bit1*), then second step is to set cntrbit2 to 1 at 350ns. By using this kind of circuit, nothing happens when the CCM encounters a “bad instruction" that tries to create multiple drivers. This circuit can be described by the following VHDL code:

...

signal databit, ldbit1, ldbit2, cntrbit1, cntrbit2 : stdlogic;

signal dff1clk, dff2clk, gate2output: stdlogic;

...

 begin

 ... DFF1: dff port map ( d=>databit, clk=>dff1clk, q=>cntrbit1);

 DFF2: dff port map ( d=>databit, clk=>dff2clk, q=>cntrbit2);

gate2output <= not ((cntrbit1 or cntrbit2) and databit);

 dff1clk <= ldbit1 and gate2output;

dff2clk <= ldbit2 and gate2output;

...

 end;

The encoding of the “set tri-state buffers" instruction is as follows:



The first three bits “000" is the opcode of this CCM instruction. The bit 25 is the databit signal in Figure 18.7. The bits 28 to 26 (mmm in the encoding format) is the “address" of these eight control bits of tristate buffers. The address of these control bits is as follows: 000 is EnAddrA, 001 is EnAddrB, 010 is EnIFifoA, 011 is MemARW, 100 is EnIluA, 101 is EnIFifoD, 110 is MemBRW, and 111 is EnIluB (all these eight control bits are shown in Figure 18.3).

**18.3.3 Set Registers**

The “set registers” instruction loads the data into registers (except Accu and Data) from their inputs. For loading the correct data into registers, the tri-state buffers must be set properly before issuing this instruction. The encoding of the instruction is as follows:



The “set registers” instruction loads the data into registers (except Accu and Data) from their inputs. For loading the correct data into registers, the tri-state buffers must be set properly before issuing this instruction. The encoding of the instruction is as follows:

The first three bits “001" is the opcode of this CCM instruction. The bits 28 to 26 (mmm in the encoding format) is the “address" of the target register. The addresses of these registers are as follows: 000 is AddrA, 001 is AddrB, 010 is AddrR, 011 is WATER, 100 is RightEdge, 101 is INST, 110 is CONF, and 111 is PRPO.

AddrA and AddrB are the Address Units (see section 18.2.2). Since they can be set in the same way we set registers, the same instruction is used to set Address Units and registers. AddrA, AddrB and AddrR are 18-bit wide, so bits 17 to 0 are used when the target register is one of them. WATER and RightEdge registers are 15-bit wide, so bits 14 to 0 are used when the target register is one of them.

When the target register is the instruction register, 21-bit data is needed. The bits 25 to 23 represent the highest 3 bits of the instruction register. The bits 17 to 0 represents the lowest 18 bits of the instruction register (see the format of instruction register discussed in section 18.2.3). When the target register is the PRPO register, the bits 23 to 0 are used (see the format of prpo register discussed in section 18.2.3).



Figure 18.9: The format of *config* register

When the target register is config register (CONF), the bits 8 to 0 of the instruction are used. The config register is the collection of eight configuration bits of the CCM. The content of config register is shown in Figure 18.9. The meaning of these eight bits is as follows:

* enFinish determine whether the instructions Execute and Loop will generate “finish word" or not. The “finish word" will be discussed in section 18.3.4.
* enMemA and enMemB determine whether the memory banks are used in cube operation or not. Do not confuse with tri-state buffer control signals memARW and memBRW; the signals memARW and memBRW determine the operation mode (read or write) of memory banks (see section 18.2.2). If enMemA is set to 1, then the MEM\_A is used in the following cube operation, otherwise MEM\_A is not used. The bit enMemB controls MEM\_B in the same manner. Only Loop instruction will use memory banks, and it will be discussed later in this section.
* CmpSrc, ASrc and Osrc are three “select" signals of the multiplexers (see Figure 18.3).
* toOFifo,toAccu and toMem are three output control signals. These three signals tell the GCU just whether or not to generate the corresponding control signals to load data into the output FIFO, and/or Accu, and/or the memory from their inputs after a resultant cube is generated. The GCU doesn't care about where these inputs come from. For using proper dataflow mode, the tri-state buffers must be set properly before executing the operation. It is possible to write the resultant cube to all these three targets at the same time.

A cube operation is completely described by functions relation, before, active, after and pre-relation/pre-operation. As it can be seen from this instruction, all these functions can be programmed by users through setting registers inst and prpo instead of “hard-code" values. This is similar to microprogramming and makes it easy to execute a “new” cube operation that is not discussed in this book and can be classified into one of three classes of cube operations without re-designing of the entire CCM. For example, the cofactor operation is a “new" operation to the CCM. A student was asked by me to implement the cofactor operation on the CCM, which did not exist there yet. Therefore, he described the cofactor operation using Equation 16.6, and I derived before, active and relation functions. The final equation used to describe cofactor operation is shown in Equation 15.11. Now we can perform cofactor operation without changing the design of the CCM. This is a powerful feature of the presented CCM design and it is hoped that it will find many applications in CCM assembly programs. **You will be also asked to design “new” CCM operations.**

**18.3.4 Execute**

The “execute" instruction is used to execute only one cube operation. It is the realization of the executing patterns (a) and (b) (see section 18.1). When the CCM receives this instruction, the CCM loads data into Data register from its input, then executes cube operation on two operand cubes currently stored in Accu and Data registers. The resultant cubes being written to Accu, and/or memory, and/or the output FIFO depending on three output control bits (The address of memory will be automatically increased by one after every memory write operation). The encoding of the instruction is as follows:



The first two bits “10" is the opcode of this CCM instruction. The 30-bit data in the instruction will be shown on the bus IBus after the instruction is read from the input FIFO. For loading the correct data into Data register, the control bits EnIFifoD and Osrc must be set properly before issuing this instruction.

The last step of executing execute instruction is that a “finish word” is pushed into the output FIFO if the bit enFinish is 1. The finish word is a special 32-bit word whose highest two bits are set to 10. On the other hand, the highest two bits of the general data word (represents cubes) are always 00. The finish word is used to separate two arrays of (resultant) cubes of two set of cube operations.

The finish word is necessary for our CCM co-processor. For example, we calculate a sharp operations on two arrays of cubes. The sharp operation is carried out by a set of CCM instructions, and produces an array of cubes. Without the finish word, the host computer would never know where is the end of the resultant cubes even if the host computer fetches all words in the output FIFO. The question is how the host computer could determine whether a operation (or a set of operations) is completed or not. Therefore, we introduced here the concept of finish word to solve this problem. We can let the CCM generate a finish word after the operation is completed. With the finish word, the host computer can tell whether a operation is already completed or not. Another example of use of the finish word is when we calculate two sharp operations, each of them produces an array of cubes as the result. Without the finish word, the two resultant cubes would be concatenated together, so the host computer would be not not able to separate these two arrays of cubes. With the finish word, the host computer has a way to separate these two arrays of cubes.

**18.3.5 Loop**

The “loop” instruction is used to execute multiple cube operations continuously without fetching the input FIFO. It is the realization of the executing patterns (c) and (d) (see section 18.1). When the CCM receives LOOP instruction, the CCM loads the data from memory into Data register (The control signals MemARW, MemBRW, EnIluA, EnIluB, OSrc and ASrc must be set properly before issuing LOOP instruction).

Then the CCM executes cube operation on two operand cubes currently stored in Accu and Data registers. The resultant cubes are written to Accu, and/or memory, and/or the output FIFO determined by three output control signals. After that, the CCM loads the next data from memory into Data register and executes the same cube operation again (The address of memory will be automatically increased by one after every memory read/write operation).

This procedure is repeated until the signal AddrEQ (see Figure 18.3) becomes 1, which means that the memory address (the signal Cmpsrc determines which memory address is used, see Figure 18.3) is equal to the content of the AddrR register. The encoding of the instruction is as follows:



The first two bits “11" is the opcode of this CCM instruction. The 30-bit data in the instruction will be shown on the bus IBus, but typically it is not used.

Similarly as in the execute instruction, the last step of executing the loop instruction is also that a finish word is pushed into the output FIFO if the bit enFinish is 1.

**18.4 Global Control Unit**

The Global Control Unit (GCU) handles the communication between the host computer and the CCM, and it is also the controller of the whole CCM. As mentioned in section 16.5, another controller OCU is used to control the datapath of the CCM. Certainly, we can design a single controller to control all of them. The reason why we design two controllers in our CCM is that it is easier to design and test two simple controllers than one complex controller.

The algorithm of the CCM is very simple: under the control of GCU, the CCM fetches an instruction from the input FIFO; then the CCM executes the instruction: set the contents of registers, or tri-state buffers, or executes cube operation(s). After that, the CCM is ready to process next instruction from the input FIFO. The GCU will remove an empty cube by using signal empty (see section 16.4.2). The state diagram of the GCU is shown in Figure 18.10. The signals opc in the state diagram is the opcode the CCM instruction, and it occupies the highest 3 bits of the CCM instruction. The other signals will be discussed when we will discuss the related states. Now let us take a look at these states.

• **State S0:** This is the initial state of the GCU. In this state, the GCU checks if there are CCM instruction(s) in the input FIFO by asserting signal FifoInEmpty (see section 17.2.6). If there are CCM instruction(s) in the input FIFO, then the GCU goes to state S1, otherwise, the GCU stays in state S0.

• **State S1:** In this state, the GCU generates signal FifoInRead (see section 17.2.6) to fetch a CCM instruction from the input FIFO. This state has three exits, states S2, S3 and S4. If the instruction is one of instructions “set accumulator”, “set tri-state buffers” or “set registers” (the highest bit of opcodes of these three instructions all are 0's), then the GCU goes to state S2; if the instruction is the instruction “exec” (the corresponding opcode is 10x), then the GCU goes to state S4; if the instruction is the “loop” instruction (the corresponding opcode is 11x), the GCU goes to state S3.

• **State S2:** In this state, the GCU generates the load signals to load data into the corresponding register, or the accumulator, or the 1-bit register that store the control bit of the corresponding tri-state buffer. For example, if the instruction is “set instruction register”, then the GCU generates signal ld\_reg, and the signal ld\_inst that is used to load data into the instruction register is generated as follows: 

(please note that the instruction register is encoded as 101, see section 18.3.3), where b28, b27 and b26 are the 28th to 26th bit of the CCM instruction. Load signals for other registers and 8 1-bits registers for 8 tri-state buffers' control bits are generated similarly. When the instruction is “set accumulator", the GCU generates signal ld\_accu.

After that, the GCU always goes back to state S0 and becomes ready to process the next CCM instruction.

• **State S3:** In this state, the GCU checks if the loop operation is completed by asserting signal AddrEQ (see Figure 18. 3 and section 18.3.5). If the loop operation is completed, then the GCU goes back to state S0 and becomes ready to process the next CCM instruction; otherwise, the GCU goes to state S4.



*Figure 18.10: The state diagram of the GCU*

• **State S4:** In this state, the GCU generates signal ld\_data to loads the data into the data register from its input. If the CCM is executing “exec” instruction, the input of the data register should come from the IBus, otherwise (the CCM is executing “loop” instruction), the input of the data register should come from one of the two memory banks. This state has two exits, states P2 and P1. If the first pre-relation/pre-operation is used in the operation (represented by p1 field of the instruction register, see section 18.2.3), then the GCU goes to state P2, otherwise, the GCU goes to state P1.

• **State P2:** In this state, the GCU sets the signal prel\_sel to 00. After that, the pre-relation/pre-operation circuitry begins to evaluate the first pre-relation (see section 16.7). The GCU always goes to state P3 from state P2.

• **State P3:** In this state, the GCU still keeps the signal prel\_sel to be 00, and checks if the first pre-relation is satisfied by asserting signal prel\_res (see section 16.7). If the first pre-relation is satisfied (the signal prel\_res is 1), then the GCU goes to state P4; otherwise, the GCU will check if the second pre-relation/pre-operation is used in the operation (represented by p2 field of the instruction register, see section 18.2.3). If the second pre-relation/pre-operation is used, then the GCU goes to state P5; otherwise, the GCU goes to state P1.

• **State P4:** Achieving this state means that the first pre-relation is satisfied. The GCU calculates the cube operation by using the first pre-operation function (the signal pre\_sel keeps 00), and the GCU will generate signal write\_output. By using signal write\_output, the proper write signals can be generated to write the resultant cube to Accu, and/or memory, and/or the output FIFO depending on three output control bits if the empty signal is 0 in this state. For example, the signal write\_fifo that writes the result to the output FIFO is generated as follows:



where the signal toOfifo is discussed in section 18.3.3, and the signal empty is discussed in section 16.4. The GCU always goes to state S7 from state P4.

• **State P5:** This state is similar to state P2, and the difference is that the GCU will check the second pre-relation rather than check the first pre-relation. The signal prel\_sel is set to 01 in this state. The GCU always goes to state P6 from this state.• **State P6:** In this state, the GCU still keeps the signal prel\_sel to be 01, and checks if the second pre-relation is satisfied by asserting signal prel\_res (see section 16.5). If the second pre-relation is satisfied, then the GCU goes to state P7; otherwise, the GCU will go to state P1.

• **State P7:** Achieving this state means that the second pre-relation has been satisfied. The GCU calculates the cube operation by using the second pre-operation function (the signal pre\_sel keeps 01), and the GCU will generate proper write signals (in the same way with state P4) to write the resultant cube to Accu, and/or memory, and/or the output FIFO depending on three output control bits if the empty signal is 0 in this state. The GCU always goes to state S7 from state P7.

• **State P1:** This state means that the cube operation that is executed on the CCM does not have the pre-relation/pre-operation or these pre-relations are not satisfied. In the state P1, state S5, and state S6, the signal prel\_sel is set to 10, which means the cube operation will be carried out by using relation/operation specified by the instruction register. State P1 has two exits, states S5 and S6. If the cube operation is a sequential operation (The sc field of the instruction register represents whether a operation is sequential or combinational, see section 18.2.3), then the GCU goes to state S5; otherwise (combinational operation), the GCU goes to state S6.

• **State S5:** This state means that the operation is a sequential cube operation. In this state, the GCU will generate signal ilu\_enable which enables the ILU to execute sequential operation under the control of OCU (the control unit of the ILU). After the operation is done, the OCU will generate signal ilu\_done to tell the GCU that the cube operation is done. The GCU will keep checking the signal ilu\_done to see if the cube operation is done. If not, the GCU will remain in state S5; otherwise (the operation is done), the GCU will check if this is a loop operation by asserting the opcode of the current CCM instruction on the IBus. If this is a loop operation (opc=11x), then the GCU will go to state S3, otherwise, the GCU goes back to state S0 and becomes ready to process the next CCM instruction.

• **State S6:** This state means that the operation is a combinational cube operation. In this state, the signal ilu\_enable keeps 0, which means that all ITs in the ILU will remain in before states, or goes to active states if the current cube operation is a complex combinational cube operation (the pm field of the instruction register represents whether the operation is complex or not, see section 18.2.3) and the given IT is a special variable (or part of special variable). The ILU executes this (complex) combinational cube operation by using before and active functions (see section 15.3.1 and 15.3.2).

The GCU will generate proper write signals (in the same way with state P4) to write the resultant cube to Accu, and/or memory, and/or the output FIFO depending on three output control bits if the empty signal is 0 in this state. The GCU always goes to state S7 from state S6.

• **State S7:** In this state, the GCU will check if this is a loop operation by asserting the opcode of the current CCM instruction on the IBus. If this is a loop operation (opc=11x), then the GCU will go to state S3, otherwise, the GCU goes to state S0 and becomes ready to process the next CCM instruction.

In this state, the GCU also adjusts the address of the memory if the CCM read the data from and/or write the data to the memory bank(s).

**18.5 Questions and problems to solve by students**