[UNIVERSITY OF AIZU]

Architecture and Design of Shared Memory Multi-Queue Core Processor

Multi-Queue core System on a Chip (MQSoC)

Report

Shunichi Kato

2011/1/24

Contents

1 Introduction

Nowadays, processor performance cannot be achieved by simply increasing clock frequency. In addition, the single core chip architecture is scaled well due to various design challenges. Multicore systems with a large number of cores have been proposed to take advantage of micro-electronics development. Multicore systems are emerging as solutions for high performance embedded and general purpose computing.

However, although important works have been achieved in the design and implementation of such systems, the issue of synchronization mechanisms and memory arbitration has not been properly investigated yet.

Queue computing was earlier proposed in our laboratory. A queue processor has several promising advantages over register-based machines. First, queue programs have higher instruction level parallelism because they are constructed with a breadth-first algorithm. Second, queue based instructions are shorter because they don't need to specify operands explicitly. That is, data is implicitly taken from the head of operand queue and the result is implicitly written at the tail of the operand queue. This characteristic makes instruction lengths shorter and independent from the actual number of physical queue words. Finally, Queue based instructions are free from false dependencies. This characteristic eliminates the need for register renaming.

In this thesis, we propose architecture and design of a multicore system based on a simple Queue core and a new bus arbitration mechanism. All cores in the system are connected via a shared bus and communicate using shared memory.

In this thesis, we first discuss the communication and arbitration problems issue. In the second part, a new bus arbitration mechanism (BAM) is proposed, evaluated and discussed in detail. The remaining part of the thesis discusses the architecture and evaluation results of a multicore system, named Multi-Queue core System on a Chip (MQSoC).

2 Shared Bus Problem

Efficient communication between the cores is a key design issue in any systems. In previous studies, there has been a heavy focus on either hardware or software to provide facilities for this communication. In the multicore systems on Field Programmable Gate Array (FPGA), the focus on higher-level layer, or the communication arbitration, has given rise to more flexible systems that provide good speed-up and low cost.

2.1 Mechanism of Shared Bus Problem

The shared bus problem occurs in multi-core system. We show the mechanism of this problem in Figure 1. The accesses conflict to the shared bus if two or more processors try to access the bus. One processor core can only access the shared bus per one clock cycle when

shared memory has single port.

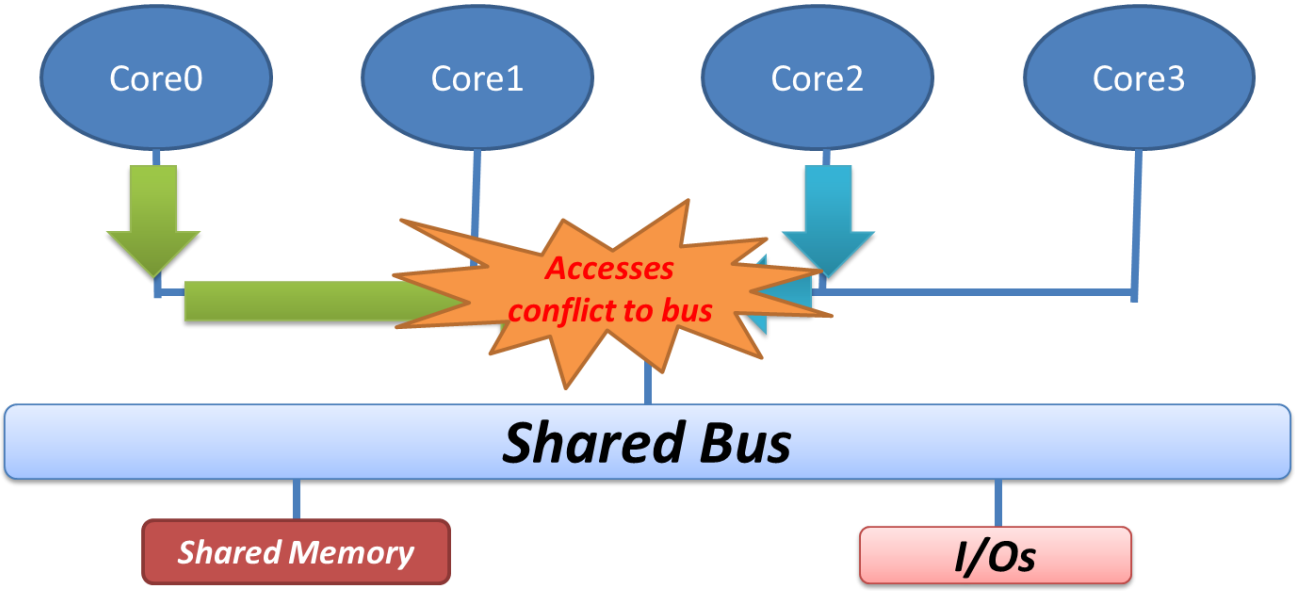

Figure 1: Mechanism of Shared Bus Problem

2.2 Solution of Shared Bus Problem

We add the bus arbitration to solute this problem. This bus arbitration is that the method to decide which a processor core is given access to the shared bus and this is essential to extend from a single-core system to a multi-core system. We show the action of bus arbitration in Figure2. When a core0 and a core2 try to access the shared bus, the core0 can access the bus if the core0 get the grant for an access from bus arbitration.

A programing model is similar to multi-threading on single-core system (but threads run on different processors). A number of processor cores looks like one processor core from shared memory and I/O peripherals to add this bus arbitration.

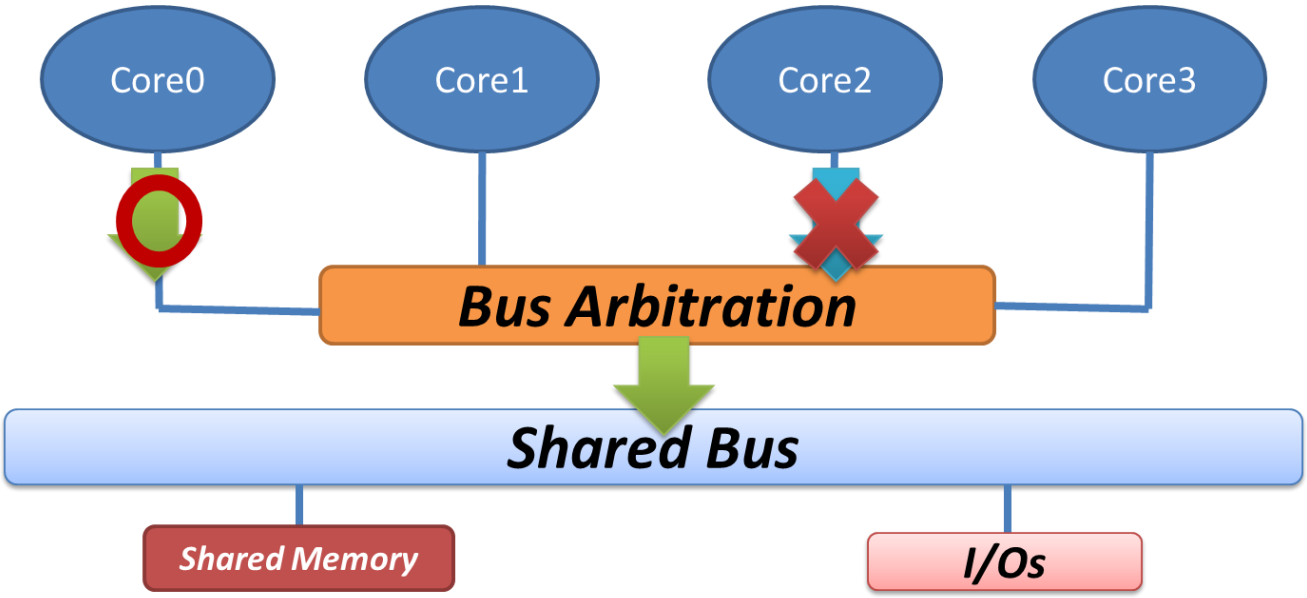

Figure 2: Solution of Shared Bus Problem

3 MQSoC System Architecture

The MQSoC System is a multi-core system that extend queue processor core from one processor core to four processor cores and add the bus arbitration. The bus arbitration includes the wait_core block. We call some processor cores, wait_core, and bus arbitration the Core System. This wait_core block and bus arbitration is important area in this system. Each core has each instruction memory. A divided program is written on each instruction memory. One processor core can only access this memory because the shared data memory has single port. The I/O peripherals are mapping by memory mapped I/O. We show block diagram of MQSoC system architecture in Figure3. We discuss the Core_System in next parts.

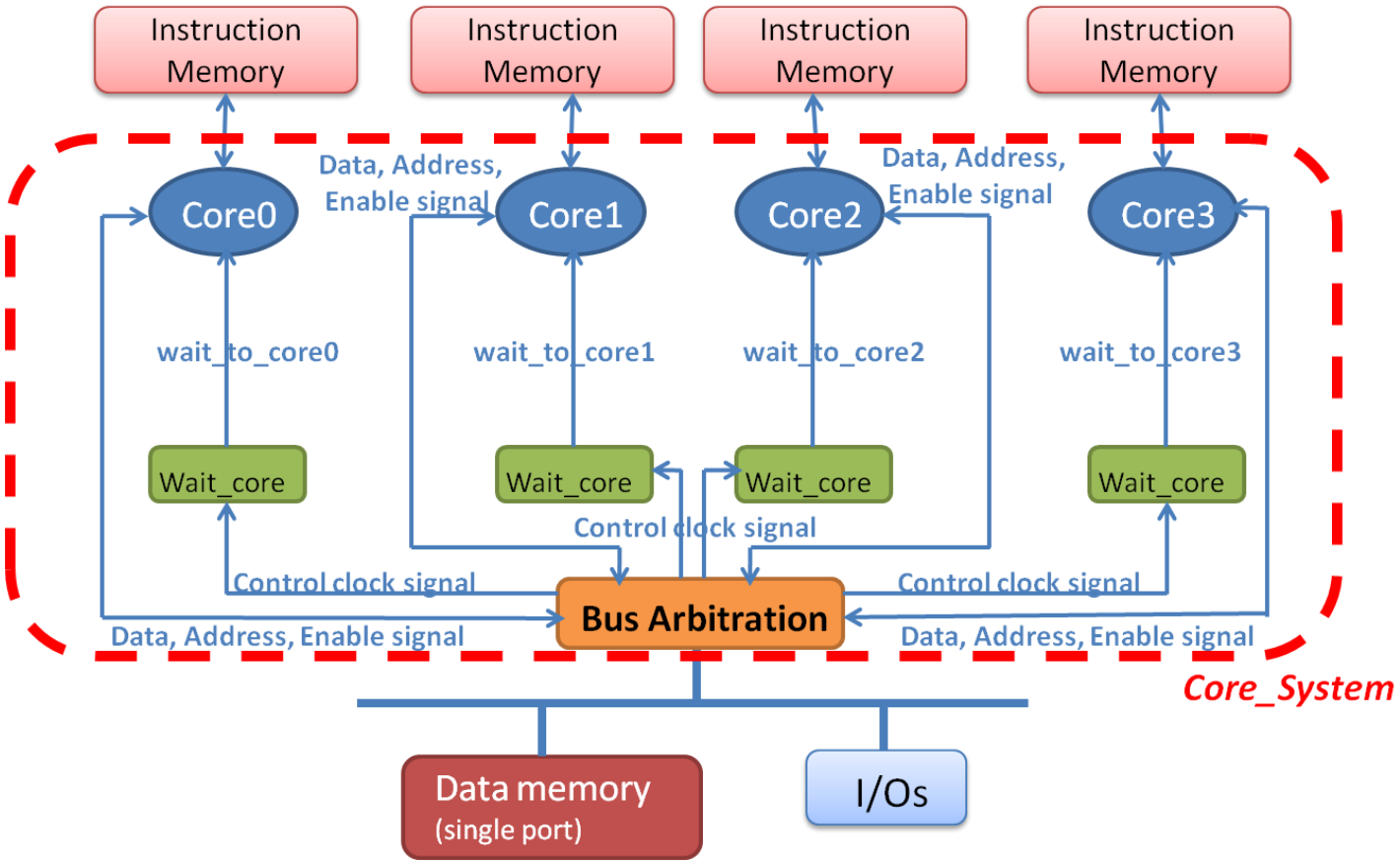

Figure 3: MQSoC system architecture

3.1 Bus Arbitration Mechanism (BAM)

The bus arbitration and wait_core block control accesses for the shared bus from many processor cores. Many processor cores looks like one processor core from shared memory and I/O peripherals because one processor core can only access the bus per one cycle by adding these two blocks. We don't need to change other architecture.

3.1.1 Organization

We show creating the bus arbitration module (Core System) that in Figure 4. The wait_core block exists at each processor core. The bus arbitration module has 3 inputs by each processor core. These inputs are Address, Data, Control (Enable signal). The main outputs of this module are Address, Data, Control (Enable signal). These outputs are similar to single-queue core system. Nothing is changed about how memory is accessed and connected the bus arbitration to processor cores is straightforward. The additional outputs are used to make decisions used in wait_core block. We discuss details of bus arbitration mechanism.

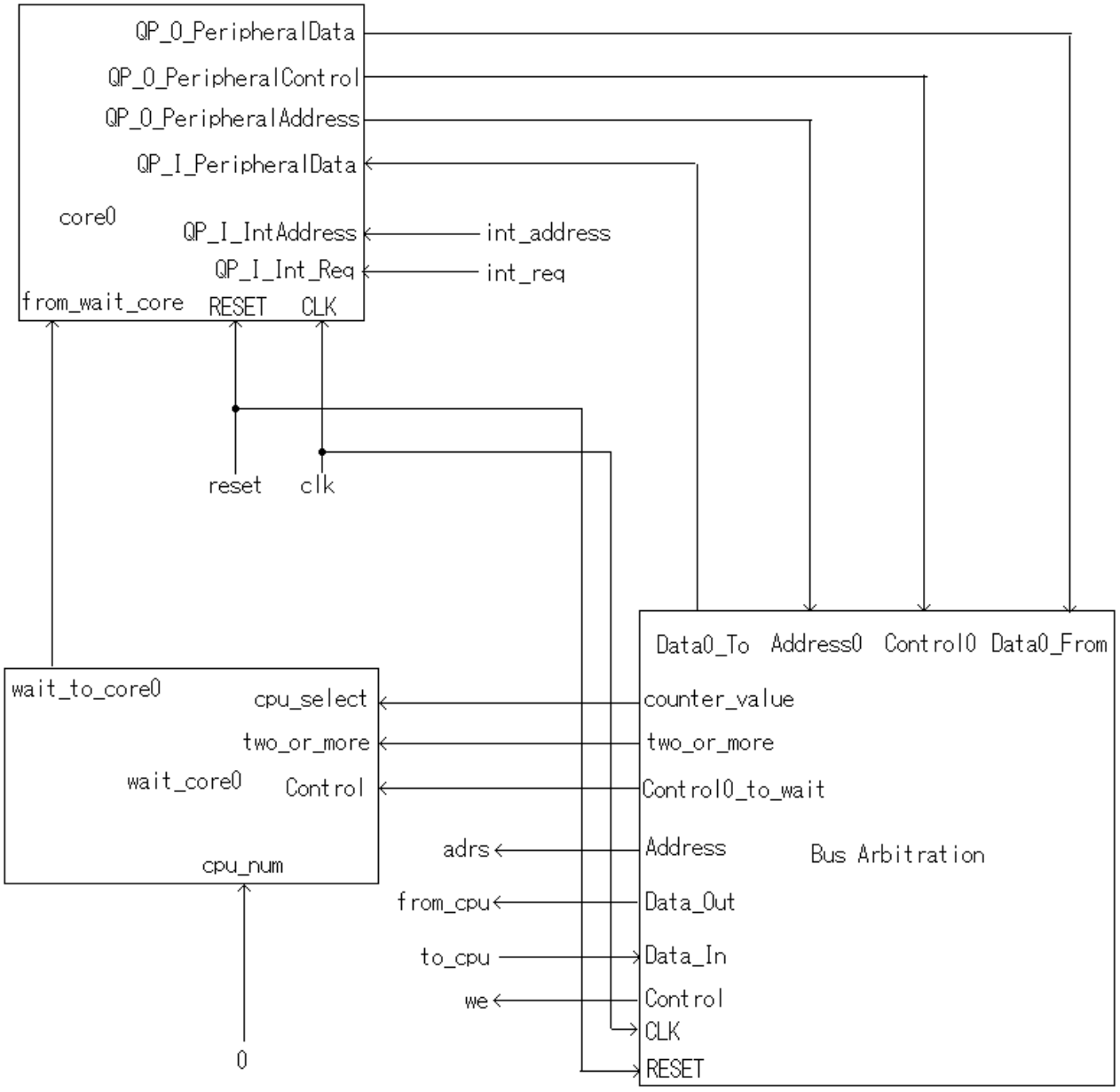

Figure 4: Block diagram of core system

3.1.2 Algorithm of Bus Arbitration

The bus arbitration can takes "Control" (read and write enable) as accessing signal from each processor core if a processor core try to access the shared memory. These are two important cases to handle. First case is that either zero or one processor core tries to access the shared memory. Second case is that two or more processor cores try to access the shared memory. In first case, the bus arbitration correctly sends the address, data, and control signal to the shared bus. In second case, the bus arbitration sends "two_or_more=1" signal to wait_core block. The bus arbitration decide accessible processor core by "two_or_more", "counter_value", and "wr[3:0]" that write and read enable signal for all processor cores. The information of selected processor core is sent to wait_core block. The address, data, and control signal are sent to the shared bus in last state of this bus arbitration. We show the block diagram of bus arbitration in Figure5.

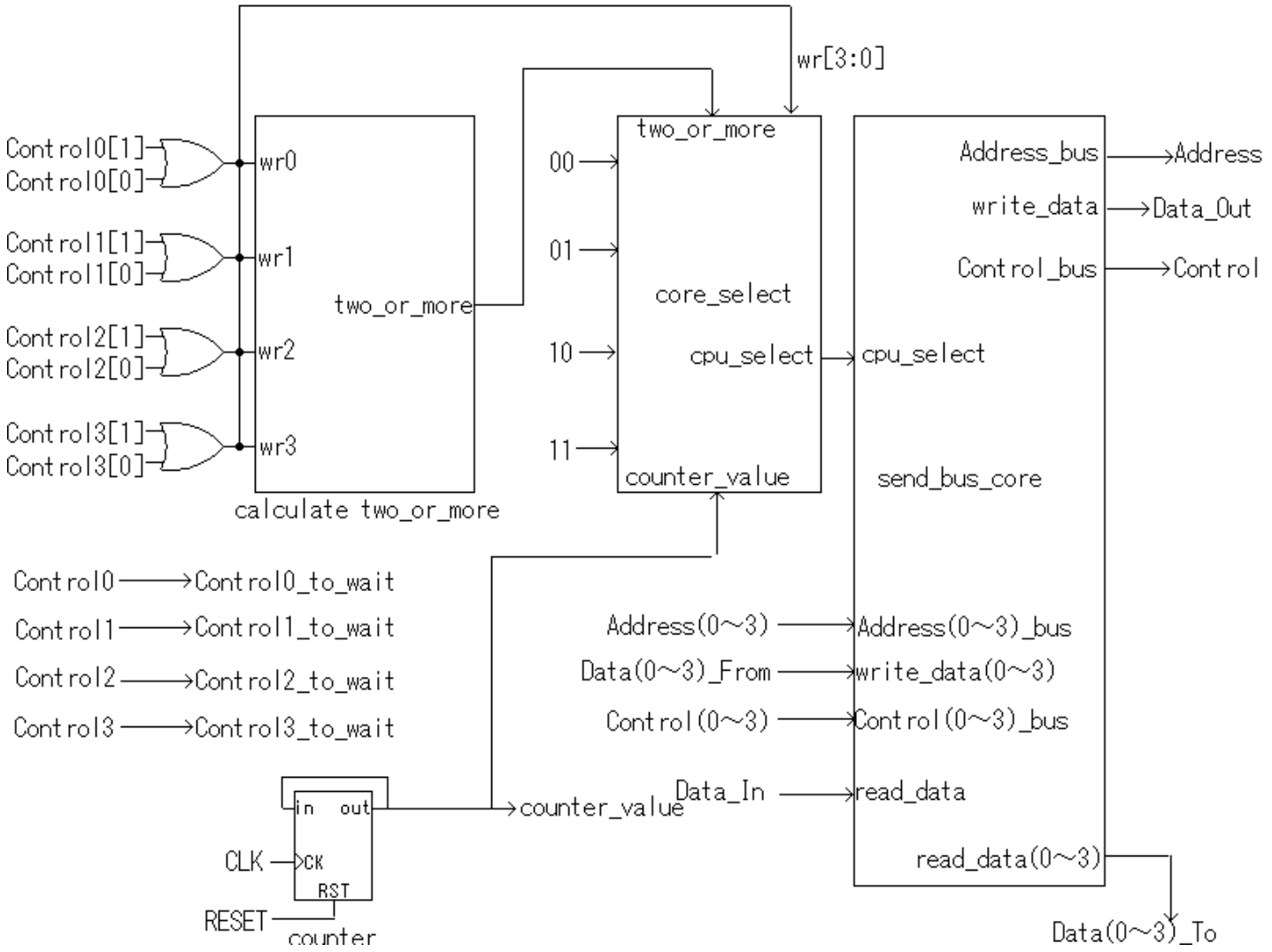

Figure 5: Block Diagram of Bus Arbitration

3.1.3 Algorithm of wait_core

The wait_core block receives three information signals from bus arbitration. This block sends "wait_to_core=1" signal to the processor core when these three signal condition is true. The PC of processor core that receives the "wait_to_core=1" from wait_core block doesn't update. Condition is that "cpu_select = cpu_num", "two_or_more=1", and "Control=1". We show the block diagram of wait_core in Figure 6.

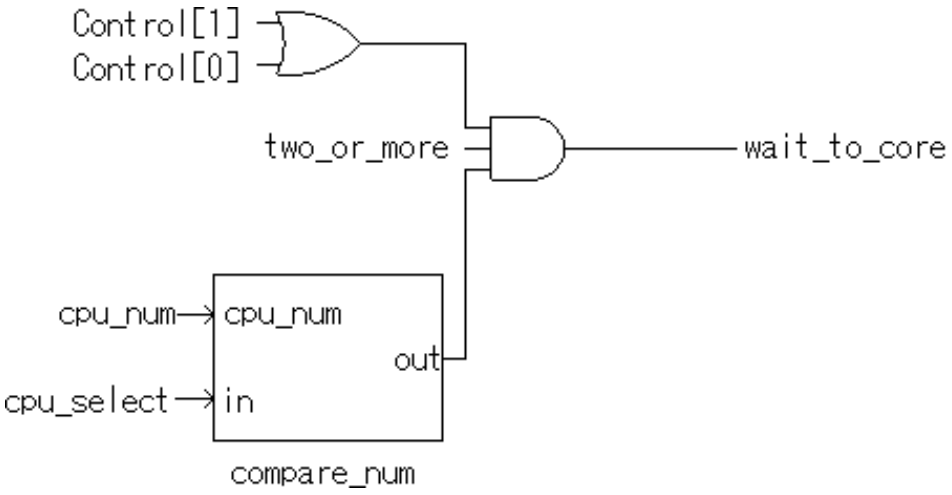

Figure 6: Block Diagram of wait_core

3.1.4 Scheduling and Fairness

We select the Round Robin for scheduling because a system needs to be implemented such that all processor cores have the same priority. The Bus arbitration rotates through processor cores on every clock tick by counter. We show the example of scheduling in Figure 7.

Counter	Accessible core
00	Core0
01	Core1
10	Core2
11	Core3

Figure 7: Example of Scheduling

3.1.5 Timing chart of Bus Arbitration

We show a timing chart of Bus Arbitration in Figure 8. In a case of this timing chart, processor core3 is selected in first. The processor core that receives the "wait_to_core=1" from wait_core block doesn't update PC of it.

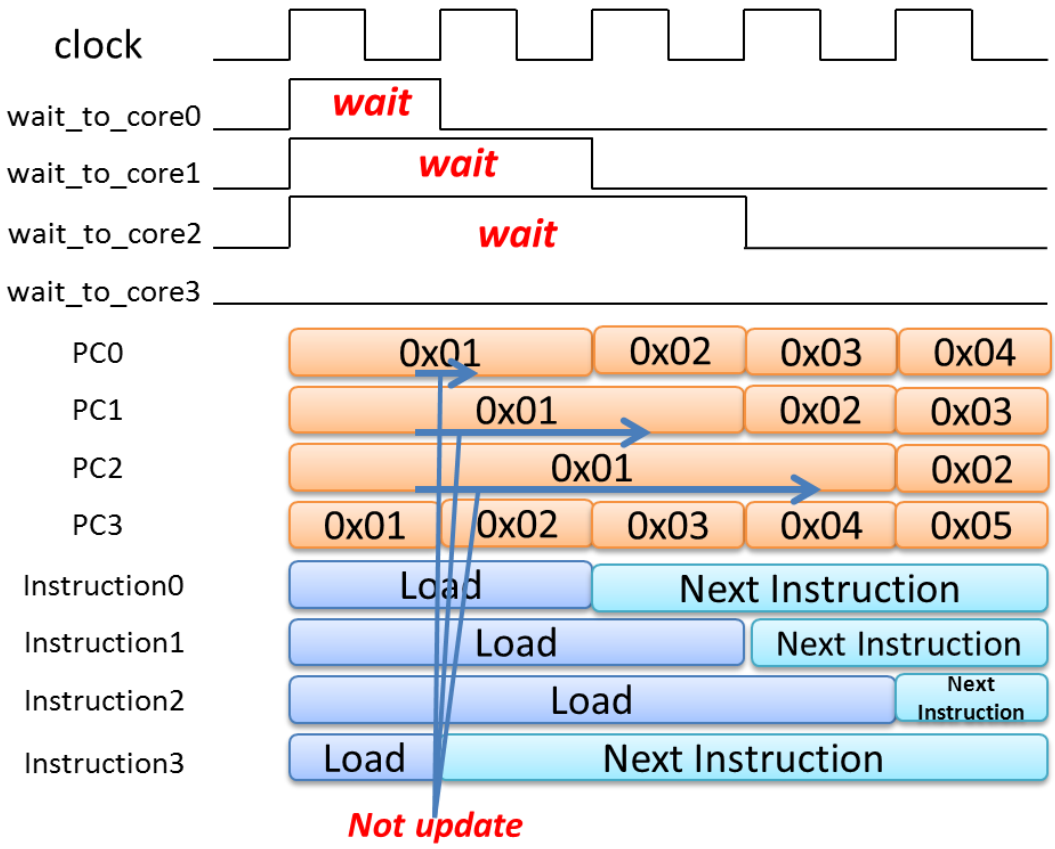

Figure 8: Timing Chart of Bus Arbitration

3.1.6 Correctness

We show the waveform of scheduling in Figure 9. The core3 is selected by bus arbitration when the "cpu_select" signal is 3. Then, the "wait_to_core" signals of other processor core are high (1) and the processor core3 can access the shared bus. The core0 is selected by bus arbitration in next clock edge because the "cpu_select" signal is 0. The "wait_to_core" signals of core1 and core2 are high (1) and the processor core0 can access the shared bus in same way. Nothing processor core are selected when the "cpu_select" signal is 7 (3'b111).

We also see the waveform of bus arbitration in Figure 10. The instruction that 5000 is add instruction and 4400 is load instruction. The "wait_to_core" signals of not selected processor are high (1) when four processor cores try to access the shared bus by load instruction. In this time, the "wait_to_core0", "wait_to_core1", and "wait_to_core2" signals are high (1) and the not selected processor core don't update the PC of it. This bus arbitration is correctness because this waveform in simulation is same that the timing charts of Figure 8.

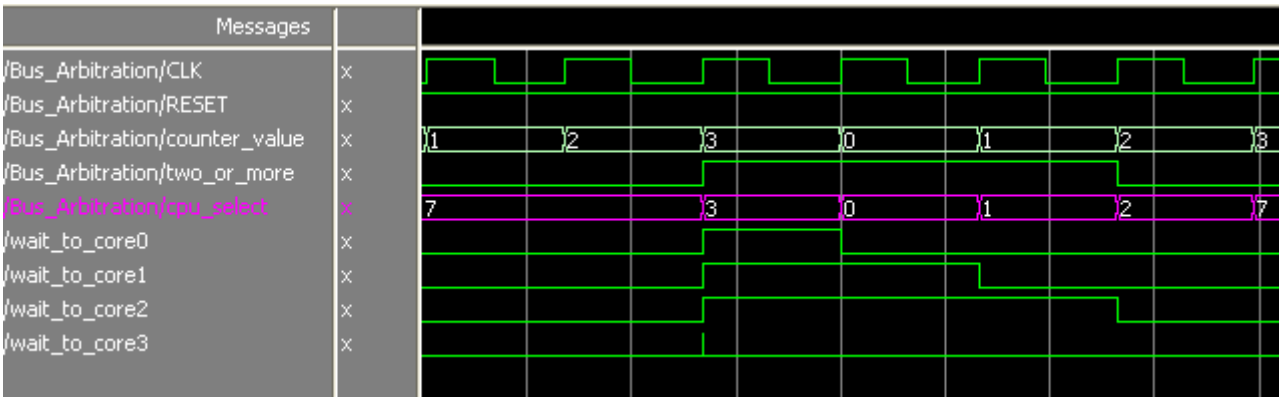

Figure 9: Waveform of Scheduling

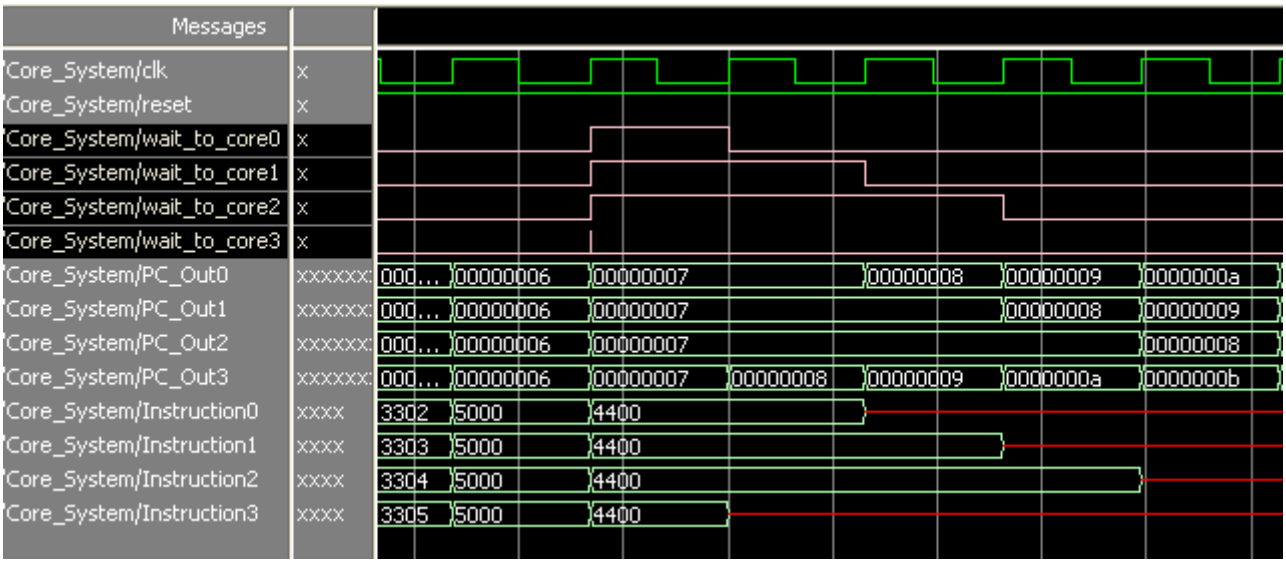

Figure 10: Waveform of Bus Arbitration

3.2 Memory Space

This MQSoC System has next memory space.

3.2.1 Data Memory

- Capacity: 32bits * 2048words = 8KB
- Address space: 0x00000000~0x000007FF

3.2.2 7-segment LED

- Address: 0x80000000

3.2.3 Instruction Memory

- Capacity: 16bits * 1024words = 4KB
- Address space: 0x00000000~0x000003FF

4 Evaluation of MQSoC System

4.1 Synthesis Results

We show the hardware results in Figure 11. The complexity of each system is given as the number of the Logic Elements (LEs). The speed indicates the maximum frequency that each system correct runs. The power indicates the thermal power dissipation. Number of the LE in dual core system is 1.94 times it in single core system. This result indicates that the LE in dual core system is not twice it in single core system if we double the number of processor and add the bus arbitration. Each multi-core system correct runs when the frequency is less than 24MHz. The thermal power dissipation grows by being proportional to the number of LEs.

	Single core	Dual core	Quad core
LEs	4,879	9,460	18,554
Speed (MHz)	29.17	24.05	27.25
Power (mW)	127.66	130.75	139.83

Figure 11: Hardware Results

4.2 Benchmark Programs

We evaluate the MQSoC System by three benchmark programs. We divide the one benchmark program into four programs for each processor core in assembly language level. We use a level-order traversal because this system uses queue processor. We show the example of assembly program in queue computing in Figure12.

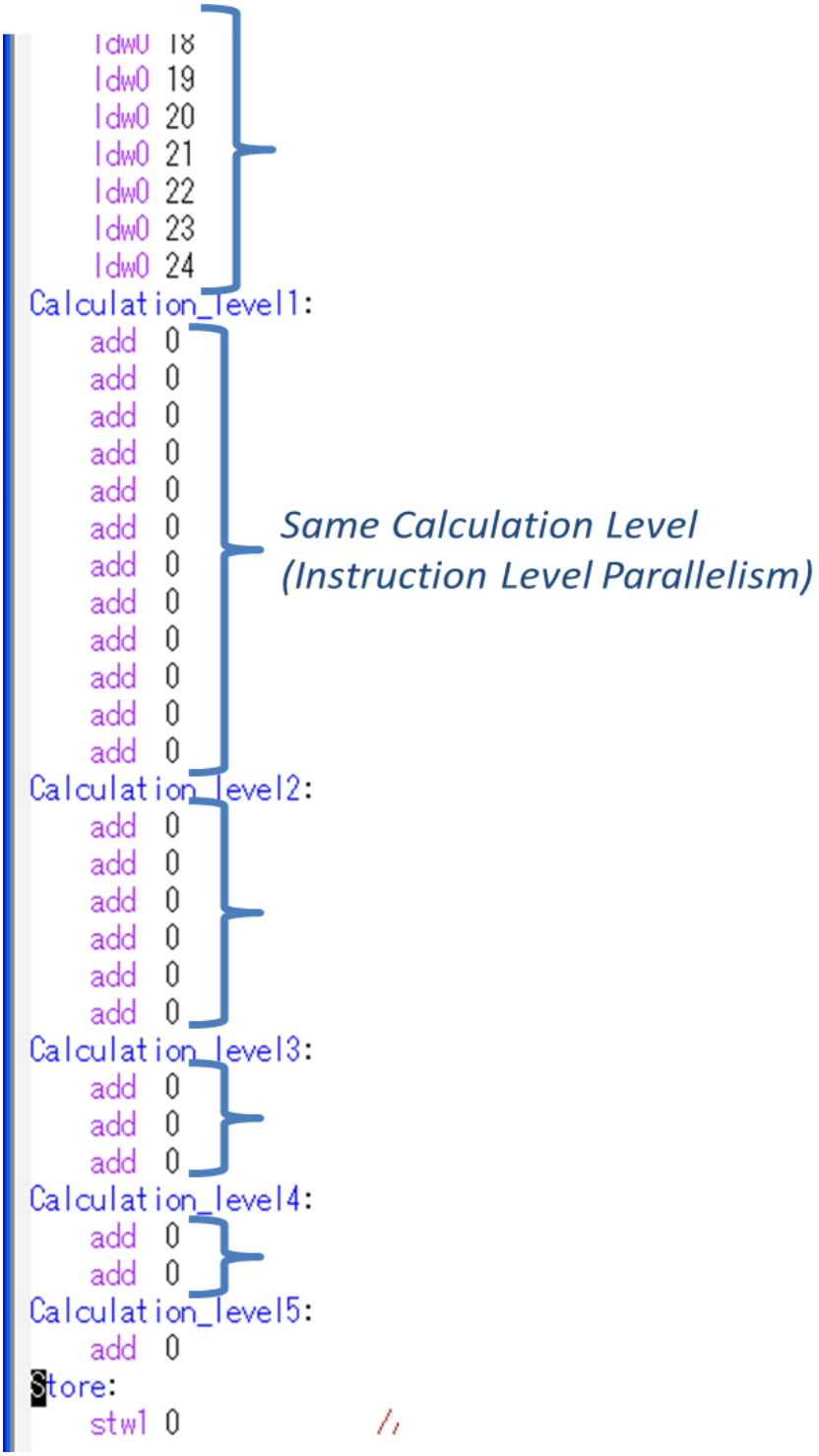

Figure 12: Assembly program

4.2.1 Benchmark1 ($\sum_{i=0}^{100} \sum_{j=0}^{100} i * j$)

This benchmark calculate product of from 0 to 100 and from 0 to 100. This program is divided for each processor core. First, sum of from 0 to 25 is calculated in the core0. Sum

of from 26 to 50 is calculated in the core1 and also other value is calculated in same way. Each processor core stores the flag to the shared memory after each calculation. Second, product of from 0 to 100 and previous results (core0: 0+1+…+24+25, core1: 26+27+…+49+50, core2: 51+52+…+74+75, core3: 76+77+…+99+100) is calculated in each processor core. Finally, the core0 calculate last result from each result on the shared memory after that the core0 check flags of other processor core on the shared memory.

4.2.2 Benchmark2-1 ($\sum_{i=0}^{100} i$)

This benchmark calculate sum of from 0 to100. We show the data flow graph in Figure 13. This program is divided for each processor core. Sum of from 0 to 25 is calculated in the core0. Sum of from 26 to 50 is calculated in the core1 and also other value is calculated in same way. Each processor core stores the flag to the shared memory after each calculation. Finally, the core0 calculate last result from each result on the shared memory after that the core0 check flags of other processor core on the shared memory.

4.2.3 Benchmark2-2 ($\sum_{i=0}^{100} i$)

This benchmark calculate sum of from 0 to100. But, this benchmark program does not use level-order traversal to decrease the number of the memory accesses in the same cycle. We show the data flow graph in Figure 14. This program is divided for each processor core. Sum of from 0 to 25 is calculated to use SPR (Special Purpose Register) in the core0. The SPR is used for index register of a loop. Sum of from 26 to 50 is calculated to use it in the core1 and also other value is calculated in same way. Each processor core stores the flag to the shared memory after each calculation. Finally, the core0 calculate last result from each result on the shared memory after that the core0 check flags of other processor core on the shared memory. In this benchmark2-2, the number of the memory accesses is less than it in benchmark2-1.

4.2.4 Benchmark3 (Factorial Calculation: N! (N=3))

This benchmark calculates the factorial calculation and four times same calculation on a program. In single core system, four times calculation are calculated on one processor core. In dual core system, twice calculation is calculated on each processor core. In quad core system, one calculation is calculated on each processor core. Finally, all processing will finish after that the core0 check flags of other processor core on the shared memory. In this benchmark3, the number of memory accesses is fewer.

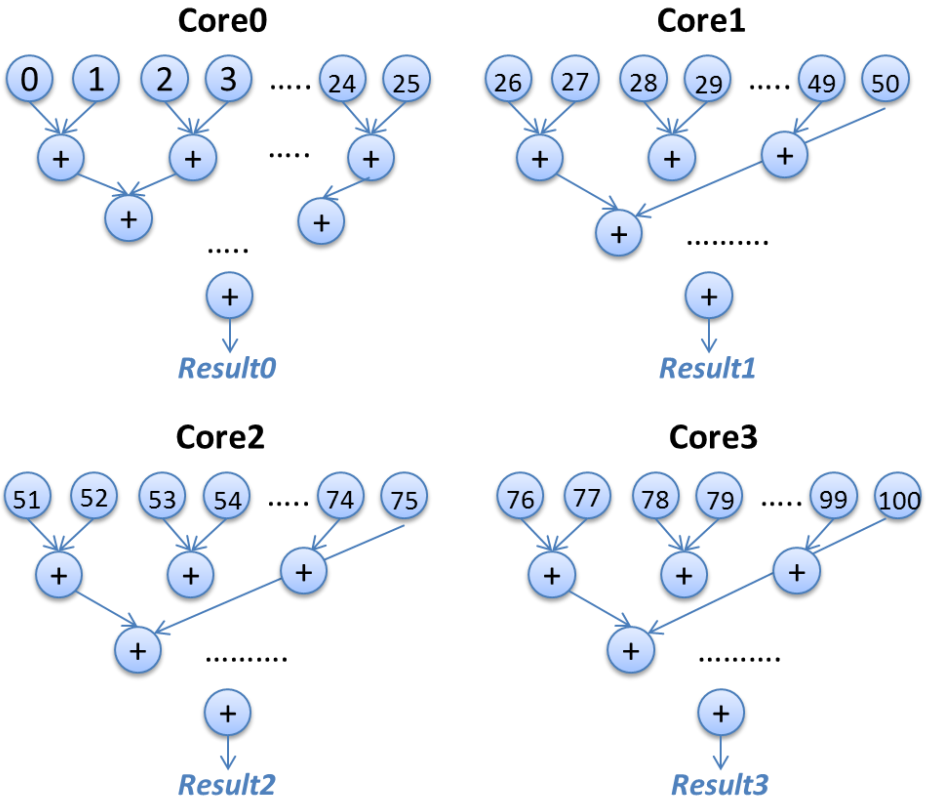

Figure 13: Data flow of Benchmark2-1

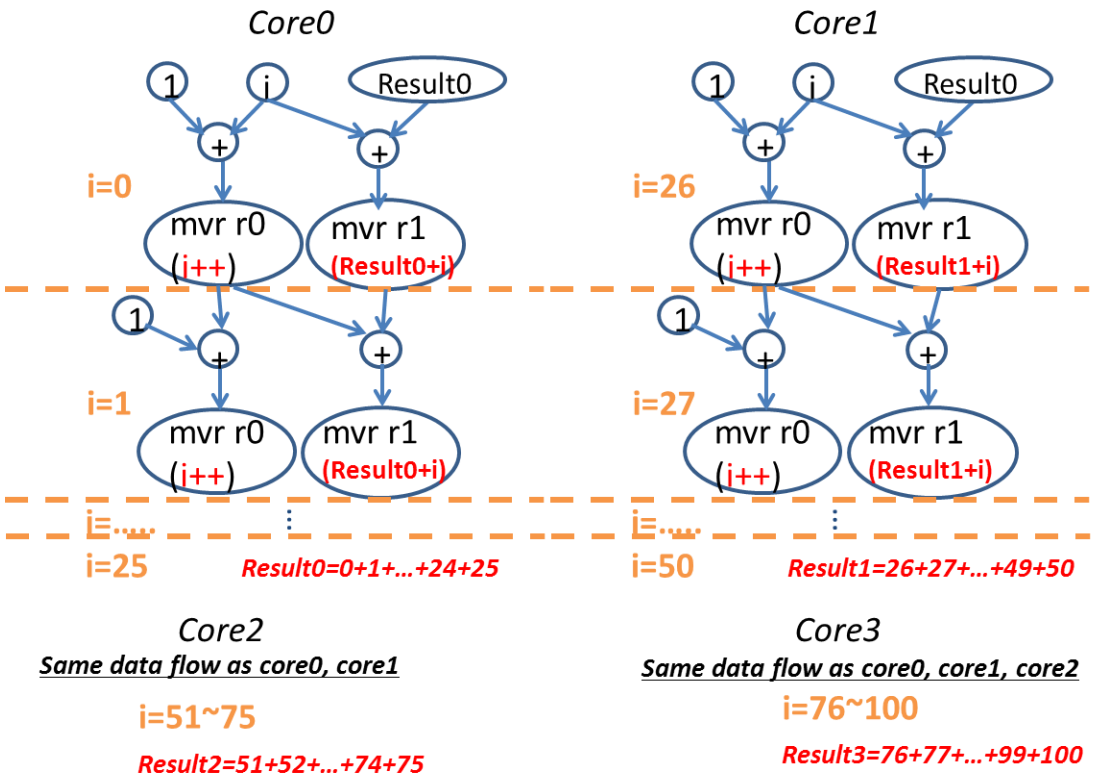

Figure 14: Data flow of Benchmark2-2

4.3 Assumption of each benchmark programs

We show the assumption of each benchmark programs in Figure 15. All instructions are processed by two cycles on a FPGA. We indicate the code size and the number of memory accesses by the assumption. The percentage of memory accesses in the Benchmark1 program is the most all of it benchmark programs. On the other hand, the percentage of it in the Benchmark3 program is the fewest all of it in benchmark programs. The number of memory accesses is very important in shared memory multi-core system.

	Code Size (Lines)	Number of memory accesses		Percentage of memory access in a program (Code size / memory accesses)
		Load	Store	
Benchmark1	2391	940	39	41.4%
Benchmark2-1 (level-order traversal)	459	108	11	26.0%
Benchmark2-2 (use index register)	196	15	8	11.7%
Benchmark3	182	7	7	7.7%

Figure 15: Assumption of Benchmark programs

4.4 Evaluation Results

We show two evaluation results in Figure 16 and Figure 17. One evaluation results in Figure 16 indicate the standard evaluation results by processing cycles and waiting cycles. Another evaluation results in Figure 17 indicate the evaluation results on number of memory accesses by processing cycles and waiting cycles because the number of memory accesses is very important in the shared memory multi-core system. The waiting cycles indicate the cycles that other processor cores wait the memory access when two or more processor cores try to access the shared memory.

	Single core		Dual core		Quad core	
	Processing cycles	Waiting cycles	Processing cycles	Waiting cycles	Processing cycles	Waiting cycles
Benchmark1	1289	0	1615	940	2451	1,890
Benchmark2-1	477	0	353	166	361	204
Benchmark3	473	0	263	4	171	12

Figure 16: Evaluation Results

	Percentage of memory accesses	Single core		Dual core		Quad core	
		Processing cycles	Waiting cycles	Processing cycles	Waiting cycles	Processing cycles	Waiting cycles
Benchmark2-1 (level-order traversal)	26.0%	477	0	353	166	361	204
Benchmark2-2 (use index register)	11.7%	2477	0	1323	2	779	8

Figure 17: Evaluation Results on number of memory accesses

4.5 Discussion of Evaluation Results

In the Benchmark1, the more the number of processor core increase, the more the processing cycle increase because the waiting processor cores increase by increasing the memory accesses. The waiting cycles indicate the basis for this discussion. Actually, the waiting cycle makes up 58.2% in the dual core system and 77.1% in the quad core system. In the Benchmark2-1, the number of the processing cycle in the dual core system is the fewest all of it. This result also indicate same basis of the discussion in the Benchmark1. The number of processing cycle in the quad core system is more than it in dual core system because the memory accesses make up 26% in this program. In the Benchmark3, the number of processing cycle in the quad core system is the fewest all of it. The number of processing cycle in the quad core system is 36.1% it in the single core system and 65% it in the dual core system because the number of the memory access is few. From these results, trying to decrease the number of the memory access (waiting cycle) affects improving performance in the shared memory multi-core system.

We also compared the Benchmark2-2 with Benchmark2-1 in Figure 17 because to discuss the change of the number of the processing cycle by the increasing and decreasing of the number of the memory access. In the Benchmark2-2, the number of the waiting cycle drastically decreases by decreasing the number of the memory access. And the number of the processing cycle in the quad core system is the fewest all of it. However, all of the number of the processing cycle in the Benchmark2-2 is more than it in the Benchmark2-1. As a result, the data flow of queue computing (level-order traversal) is better than trying to decrease the memory accesses.

5 Conclusion

In order to control the memory accesses from some processor cores, which is important in the shared memory multi-core system, the BAM was implemented in this study. It was found that the memory accesses needed to be optimized in the software program (Benchmark program). The performance in multi-core system worsens when the memory accesses from some processor cores are too much in the same cycle. On the other hand, the performance in multi-core system improves as the number of the processor core increases when the memory accesses are optimized in the software program (Benchmark program). In summary, to control the memory accesses that the BAM in the hardware and to optimize the memory accesses in the software are the two main issues in a shared memory multi-core system.

6 Future work

To optimize the memory accesses was one of the main issues in our system. Adding a cache is one solution to this issue. It will be able to decrease the memory accesses by reading the data from the cache. Also a compiler and a parallel programing are very important issues to improve the performance in the multi-core system. To optimize the software (application) for the multi-core system also improves the performance.

7 Structure of Modules

MQSoC_System_Quad (MQSoC_System_Quad.v)

- Core_System (Core_System.v)
  - Bus_Arbitration (Bus_Arbitration.v)
    - calculate_two_or_more (calculate_two_or_more.v)
    - core_select (core_select.v)
    - counter (counter.v)
    - send_bus_core (send_bus_core.v)
  - core0 (QP_CPU.v)
    - fu0 (QP_FU.v)
    - du0 (QP_DU.v)
    - qcu0 (QCU.v)
    - iu0 (QP_IU.v)
    - eu0 (QP_EU.v)
    - mu0 (QP_MU.v)
    - wbu0 (QP_WBU.v)
    - qp_c0 (QP_CONTROLLER.v)
  - wait_core0 (wait_core.v)
    - compare_num (compare_num.v)
  - core1 (QP_CPU.v)
  - wait_core1 (wait_core.v)
  - core2 (QP_CPU.v)
  - wait_core2 (wait_core.v)
  - core3 (QP_CPU.v)
  - wait_core3 (wait_core.v)
- Instruction_Memory_core0 (imem0.v)
- Instruction_Memory_core1 (imem1.v)
- Instruction_Memory_core2 (imem2.v)
- Instruction_Memory_core3 (imem3.v)
- LED_controller (LED_controller.v)
  - LED_chip_selector0 (LED_chip_selector.v)
  - LED_interface0 (LED_interface.v)
    - LED_dec0 (LED_decoder.v)
    - LED_dec1 (LED_decoder.v)
    - LED_dec2 (LED_decoder.v)
    - LED_dec3 (LED_decoder.v)
    - LED_dec4 (LED_decoder.v)

        - ✧ LED_dec5 (LED_decoder.v)
        - ✧ LED_dec6 (LED_decoder.v)
        - ✧ LED_dec7 (LED_decoder.v)
- PERI_KEY (PERI_KEY.v)
- PERI_SW (PERI_SW.v)
- PERI_MEM (PERI_MEM.v)
    - ➢ data_memory (dmem.v)
- chattering (chattering.v)
- frq (frq.v)
- stop_clock_SW (stop_clock_SW.v)

References

[1] B. A. Abderazek et al., 並列キュープロセッサの基本設計 Heiretsu Queue Processor no Kihon Sekkei [Fundamental Design of a Parallel Queue Processor] *the institute electronics. Information and communication engineers*, 2002 (in Japanese).

[2] M. Levy, "Multi-core technology: trends and design challenges," *The Embedded Microprocessor Benchmark Consortium A Non-profit Association (EEMBC),* Embedded Control Europe, 2006.

[3] M. Peter, and K. Plamena, "Shared Memory Design for Multicore Systems," International Scientific Conference Computer Science, 2008.

[4] P. Clancy, "Concurrency in Multi-Core Processor Design," thesis, Haverford College 2007

[5] H. Hoshino, "Implementation of a Simple Queue Processor on a FPGA," Technical Report, Queue Group, 2009.

[6] H. Hoshino, "Advanced Hardware Oprimization Algorithms for High Performance Queue Processor Architecture," graduation thesis, School of Computer Science and Eng., Univ. of Aizu, Fukushima, 2009.

[7] Y. Omoto, "Development Environment for Single Chip Computer intended for Queue Computing Development and Education," graduation thesis, School of Computer Science and Eng., Univ. of Aizu, Fukushima, 2010.

[8] K. Kimura, 今さら聞けないマルチプロセッサの基礎教えます Imasara Kikenai Multi-processor no Kiso Oshiemasu [Teach the Basic of Multi-processor], 18 Feb. 2005; http://www.kumikomi.net/archives/2005/02/02multi.php.

[9] A. Asahara, 並列処理を体感してみよう Heiretsu Syori wo Taikan Sitemiyou [Feel the Parallel Processing], 8 July. 2009; http://www.atmarkit.co.jp/fcoding/articles/parallel/01/para01a.html.

[10] S. Kato, "Implementation of a Bus Arbitration –Functional specification-," Technical Report, Queue Group, 2010.

[11] S. Kato, "Implementation of a Bus Arbitration –Designing specification-," Technical Report, 2010.

Appendix

8.1 Assembler (qasm_for_MQ.pl)

We must write divided programs for four processor cores in assembly language by ourselves. We must write ".data" in assembly file when we want to insert data to data memory. We must write ".core0" or ".core1" in assembly file when we want to write instruction in instruction memory of core0 or core1. We show the example of assembly language program in Figure 12.

We can convert the assembly language program to machine language program by the assembler (qasm_for_MQ.pl).

```
// Benchmark3
// Sigma Calculation
// Author: Shunichi Kato
// Date: 2010/12/09
// For Quad core

.data
base_data0:
    .word 0
    .word 1
    .word 2
    .word 3
    .word 0

// instruction of core0
.core0
Init:
    seta0 exit       // infinity loop
    setd0 base_data0 // load address(0x000)
    setd1 Result0    // load address for result of core0

// First
Load_level0:
    ldw0 0
    ldw0 1
    ldw0 2
    ldw0 3
--(Unix)-- Benchmark3_quad.s   23% (121,0)    (Assembler)----[
    jump0

// instruction of core1
.core1
Init:
    seta0 exit       // infinity loop
    setd0 base_data1 // load address
    setd1 Result1    // load address for result of core1
    setd2 Flag1      // load address for flag of core1
```

Figure 18: Example of Assembly Language

8.2 Converting file (bin_to_hex.c)

This file is to convert the machine language program to ".hex" format program. We can load this ".hex" format program in In System Memory Content Editor.

8.3 In System Memory Content Editor

We can use In System Memory Content Editor. We can reload the data and instruction in the real time when we use this tool. The memory of Altera Megafunction is needed to use this tool. The data of this memory is outputted after one cycle because this memory has a register. We must modify top module (MQSoC_System_Quad.v) in this system to solute this

problem. We must assert the signal (Use_In_System_Memory) in top module if we want to use In System Memory Content Editor. We show the block diagram of the memory in Figure 13.

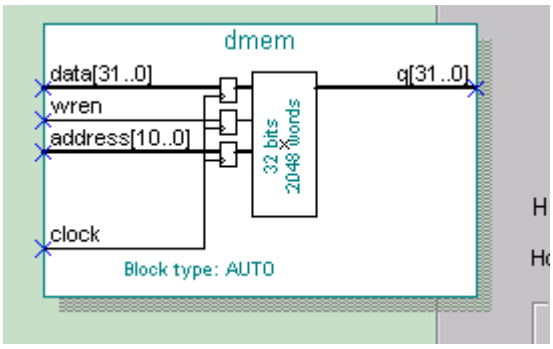

Figure 19: Block Diagram of Altera Mega-function Memory

8.4 Source Files (Benchmark Programs)

8.4.1 Benchmark1 ($\sum_{i=0}^{100} i * \sum_{j=0}^{100} j$)

```
.data
base_data0:
    .word 0
    .word 1
    .word 2

~~~Skip~~~
.core0
init:
    seta0 exit                  // infinity loop
    setd0 base_data0            // load address of base_data0
    setd1 j_sigma_Result0       // load address of j_sigma_Result
    setd2 Result0               // load address of Result0
// sigma j calculation (0+1+....+24+25)
// First
Load_level0:
    ldw0 0
    ldw0 1
    ldw0 2
    ldw0 3
```

```
    ldw0 4
    ldw0 5
    ldw0 6
    ldw0 7
    ldw0 8
    ldw0 9
    ldw0 10
    ldw0 11
    ldw0 12
    ldw0 13
    ldw0 14
    ldw0 15
    ldw0 16
    ldw0 17
    ldw0 18
    ldw0 19
    ldw0 20
    ldw0 21
    ldw0 22
    ldw0 23
    ldw0 24
    ldw0 25
Calculation_level1:
    add  0
    add  0
    add  0
    add  0
    add  0
    add  0
    add  0
    add  0
    add  0
    add  0
    add  0
    add  0
    add  0
    add  0
Calculation_level2:
    add  0
```

```
        add   0
        add   0
        add   0
        add   0
        add   0
Calculation_level3:
        add   0
        add   0
        add   0
Calculation_level4:
        add   0
        add   0
Calculation_level5:
        add   0
Store:
        stw1 0                  // store the result   to j_sigma_Result0

// i calculation (0*j_sigma_Result0 + 1*j_sigma_Result0 +...+ 99*j_sigma_Result0 +
100*j_sigma_Result0)
// First
Load_level0:
        ldw0 0
        ldw1 0
        ldw0 1
        ldw1 0
        ldw0 2
        ldw1 0
        ldw0 3
        ldw1 0
        ldw0 4
        ldw1 0
        ldw0 5
        ldw1 0
        ldw0 6
        ldw1 0
        ldw0 7
        ldw1 0
        ldw0 8
```

ldw1 0

ldw0 9

ldw1 0

ldw0 10

ldw1 0

ldw0 11

ldw1 0

ldw0 12

ldw1 0

ldw0 13

ldw1 0

ldw0 14

ldw1 0

ldw0 15

ldw1 0

Calculation_level1:

mult 0

mult 0

mult 0

mult 0

mult 0

mult 0

mult 0

mult 0

mult 0

mult 0

mult 0

mult 0

mult 0

mult 0

mult 0

mult 0

Calculation_level2:

add 0

add 0

add 0

add 0

add 0

```
    add   0
    add   0
    add   0
Calculation_level3:
    add   0
    add   0
    add   0
    add   0
Calculation_level4:
    add   0
    add   0
Calculation_level5:
    add   0
Store:
    stw2                // store result to Result0
```
~~~Continue to other processor cores calculation~~~
```
Calculation_level1:
    mult 0
    mult 0
    mult 0
    mult 0
    mult 0
Calculation_level2:
    add   0
    add   0
Calculation_level3:
    add   0
Calculation_level4:
    add   0
Store:
    ldw2 0        // load result from Result3
    add   0
    stw2 0        // store result to Result3

set_flag:
    ldil 1
    stw3 0            // store 1 to Flag3
exit:
```
~~~

```
    jump0 0
```

8.4.2 Benchmark2-1 ($\sum_{i=0}^{100} i$: level-order traversal)

```
.data
base_data0:
    .word 0
    .word 1
    .word 2
    .word 3
    .word 4
    .word 5
    .word 6
    .word 7
    .word 8
    .word 9
    .word 10
    .word 11
    .word 12
    .word 13
    .word 14
    .word 15
    .word 16
    .word 17
    .word 18
    .word 19
    .word 20
    .word 21
    .word 22
    .word 23
    .word 24
    .word 25
base_data1:
    .word 26
    .word 27
    .word 28
    .word 29
```

```
        .word 30
~~~Skip~~~

// instruction of core0
.core0
Init:
        seta0 exit              // infinity loop
        setd0 base_data0 // load address(0x000)
        setd1 Result0       // load address for result of core0

// First
Load_level0:
        ldw0 0
        ldw0 1
        ldw0 2
        ldw0 3
        ldw0 4
        ldw0 5
        ldw0 6
        ldw0 7
        ldw0 8
        ldw0 9
        ldw0 10
        ldw0 11
        ldw0 12
        ldw0 13
        ldw0 14
        ldw0 15
        ldw0 16
        ldw0 17
        ldw0 18
        ldw0 19
        ldw0 20
        ldw0 21
        ldw0 22
        ldw0 23
        ldw0 24
        ldw0 25
```

```
Calculation_level1:
    add   0
    add   0
    add   0
    add   0
    add   0
    add   0
    add   0
    add   0
    add   0
    add   0
    add   0
    add   0
    add   0
Calculation_level2:
    add   0
    add   0
    add   0
    add   0
    add   0
    add   0
Calculation_level3:
    add   0
    add   0
    add   0
Calculation_level4:
    add   0
    add   0
Calculation_level5:
    add   0
Store:
    stw1 0                  // store the result to Result0

// Finally
    setd1 Flag1             // load address for flag of core1
    setd2 Flag2         // load address for flag of core2
    setd3 Flag3         // load address for flag of core3
Check_flag1:
```

```
    ldw1   0                  // load Flag1 of core1
    compi 1                   // condition code = Flag1 - 1
    bne    Check_flag1   // if Flag1 != 1, branch Check_flag1
Check_flag2:
    ldw2   0          // load Flag2 of core2
    compi 1           // condition code = Flag2 - 1
    bne    Check_flag2   // if Flag2 != 1, branch Check_flag2
Check_flag3:
    ldw3   0                  // load Flag3 of core3
    compi 1                   // condition code = Flag3 - 1
    bne    Check_flag3   // if Flag3 != 1, branch Check_flag3

Load_finally:
    setd0 Result0
    setd1 Result1
    setd2 Result2
    setd3 Result3
    ldw0   0
    ldw1   0
    ldw2   0
    ldw3   0
Calculation_finally:
    add    0
    add    0
    add    0

Store:
    setd0 Last_Result
    stw0 0
    halt

    setd1 Flag1
    setd2 Flag2
    setd3 Flag3
    ldil   0
    ldil   0
    ldil   0
    stw1   0        // store 0 to Flag1
```

```
    stw2    0       // store 0 to Flag2
    stw3    0           // store 0 to Flag3
exit:
    jump0

// instruction of core1
.core1
```
~~~Continue to other processor cores calculation~~~
```
Calculation_level4:
    add    0
    add    0
Calculation_level5:
    add    0
Store:
    stw1 0                  // store the result to Result3

set_flag:
    ldil 1
    stw2 0          // store 1 to Flag3
exit:
    jump0 0
```

8.4.3 Benchmark2-2 ($\sum_{i=0}^{100} i$: use index register)

```
.data
N0:
    .word 25
N1:
    .word 50
N2:
    .word 75
N3:
    .word 100
Result0:
    .word 0
Result1:
    .word 0
Result2:
```
~~~

```
        .word 0
Result3:
        .word 0
Last_Result:
        .word 0
Flag1:
        .word 0
Flag2:
        .word 0
Flag3:
        .word 0

// instruction of core0
.core0
Init:
        setd0 N0
        setd1 Result0
        seta0 exit
        seta1 loop
        ldil   0
        ldw1   0
        mvr    r0 // i = 0
        mvr    r1 // Result0 = 0

loop:
        mvq    r0 // QT <= i
        ldw0   0 // load N0
        comp   0 // condition code = i - N0
        bgt    store
Calculation:
        mvq    r0
        mvq    r1
        add    0
        mvr    r1 // Result0 = Result0 + i
        mvq    r0 // QT = i
        addi   1 // i++
        mvr    r0 // r0 = i++
        jump1 0 // branch to loop
```

```
store:
    mvq   r1 // QT = Result0
    stw1   0 // memory <= Result0
Check_Flag1:
    setd1 Flag1
    ldw1   0 // load Flag1
    compi 1 // condition code = Flag1 - 1
    bne    Check_Flag1
Check_Flag2:
    setd2 Flag2
    ldw2   0 // load Flag2
    compi 1 // condition code = Flag2 - 1
    bne    Check_Flag2
Check_Flag3:
    setd3 Flag3
    ldw3   0 // load Flag3
    compi 1 // condition code = Flag3 - 1
    bne    Check_Flag3
Last_calculation:
    setd0 Result0
    setd1 Result1
    setd2 Result2
    setd3 Result3
    ldw0   0
    ldw1   0
    ldw2   0
    ldw3   0
    add    0
    add    0
    add    0
Last_store:
    setd0 Last_Result
    stw0   0
    halt      // inform end of instruction to processor
exit:
    jump0 0

// instruction of core1
```

```
.core1
Init:
    setd0 N1
    setd1 Result1
    setd2 Flag1
    seta0 exit
    seta1 loop
    ldil   26
    ldw1   0
    mvr    r0 // i = 26
    mvr    r1 // Result1 = 0

loop:
    mvq    r0 // QT <= i
    ldw0   0 // load N1
    comp   0 // condition code = i - N1
    bgt    store
Calculation:
    mvq    r0
    mvq    r1
    add    0
    mvr    r1 // Result1 = Result1 + i
    mvq    r0 // QT = i
    addi   1 // i++
    mvr    r0 // r0 = i++
    jump1 0 // branch to loop
store:
    mvq    r1 // QT = Result1
    stw1   0 // memory <= Result1
set_flag:
    ldil   1
    stw2   0 // Flag1 = 1
exit:
    jump0 0

// instruction of core2
.core2
Init:
```

```
        setd0 N2
        setd1 Result2
        setd2 Flag2
        seta0 exit
        seta1 loop
        ldil    51
        ldw1    0
        mvr     r0 // i = 51
        mvr     r1 // Result1 = 0

loop:
        mvq     r0 // QT <= i
        ldw0    0 // load N2
        comp    0 // condition code = i - N2
        bgt     store
Calculation:
        mvq     r0
        mvq     r1
        add     0
        mvr     r1 // Result2 = Result2 + i
        mvq     r0 // QT = i
        addi    1 // i++
        mvr     r0 // r0 = i++
        jump1 0 // branch to loop
store:
        mvq     r1 // QT = Result2
        stw1    0 // memory <= Result2
set_flag:
        ldil    1
        stw2    0 // Flag2 = 1
exit:
        jump0 0

// instruction of core3
.core3
Init:
        setd0 N3
        setd1 Result3
```

```
    setd2 Flag3
    seta0 exit
    seta1 loop
    ldil   76
    ldw1   0
    mvr    r0 // i = 76
    mvr    r1 // Result3 = 0

loop:
    mvq    r0 // QT <= i
    ldw0   0 // load N3
    comp   0 // condition code = i - N3
    bgt    store
Calculation:
    mvq    r0
    mvq    r1
    add    0
    mvr    r1 // Result3 = Result3 + i
    mvq    r0 // QT = i
    addi   1 // i++
    mvr    r0 // r0 = i++
    jump1 0 // branch to loop
store:
    mvq    r1 // QT = Result3
    stw1   0 // memory <= Result3
set_flag:
    ldil   1
    stw2   0 // Flag3 = 1
exit:
    jump0 0
```

8.4.4 Benchmark3 (Factorial Calculation: N! (N=3))

```
.data
N:
    .word 3
a:
    .word 0
b:
    .word 0
```

```
c:
    .word 0
d:
    .word 0
Flag1:
    .word 0
Flag2:
    .word 0
Flag3:
    .word 0

.core0
//First
init:
    seta0 loop
    seta1 Last_exit
    setd0 N
    setd1 a
    ldil   1          // index i = 1
    ldil   1          // a = 1
    mvr    r0
    mvr    r1

loop:
    mvq    r0         // QT <= i
    ldw0   0          // QT <= N
    comp   0          // condition code = i-N
    bgt    exit       // if i>N, branch exit

calculate:
    mvq    r0
    mvq    r1
    mult   0          // a*i
    mvr    r1         // a = a*i
    mvq    r0         // QT <= i
    addi   1          // i++
    mvr    r0         // r0 <= i++
    jump0 0           // branch to loop
```

```
exit:
    mvq     r1          // QT <= a*i
    stw1    0           // memory <= a*i
Load_flag:
    setd1 Flag1
    setd2 Flag2
    setd3 Flag3
Check_flag1:
    ldw1    0       // load Flag1 of core1
    compi 1         // condition code = Flag1 - 1
    bne     Check_flag1     // if Flag1 != 1, branch Check_flag1
Check_flag2:
    ldw2    0       // load Flag2 of core2
    compi 1         // condition code = Flag2 - 1
    bne     Check_flag2     // if Flag2 != 1, branch Check_flag2
Check_flag3:
    ldw3    0       // load Flag3 of core3
    compi 1         // condition code = Flag3 - 1
    bne     Check_flag3     // if Flag3 != 1, branch Check_flag3
    halt
Last_exit:
    jump1 0

.core1
//First
init:
    seta0 loop
    seta1 Last_exit
    setd0 N
    setd1 b
    setd2 Flag1
    ldil    1           // index i = 1
    ldil    1           // b = 1
    mvr     r0
    mvr     r1

loop:
```

```
        mvq     r0          // QT <= i
        ldw0    0           // QT <= N
        comp    0           // condition code = i-N
        bgt     exit        // if i>N, branch exit

calculate:
        mvq     r0
        mvq     r1
        mult    0           // b*i
        mvr     r1          // b = b*i
        mvq     r0          // QT <= i
        addi    1           // i++
        mvr     r0          // r0 <= i++
        jump0 0             // branch to loop

exit:
        mvq     r1          // QT <= b*i
        stw1    0           // memory <= b*i
set_flag:
        ldil    1
        stw2    0       // Flag1 = 1
Last_exit:
        jump1 0

.core2
//First
init:
        seta0 loop
        seta1 Last_exit
        setd0 N
        setd1 c
        setd2 Flag2
        ldil    1           // index i = 1
        ldil    1           // c = 1
        mvr     r0
        mvr     r1

loop:
```

```
    mvq     r0          // QT <= i
    ldw0    0           // QT <= N
    comp    0           // condition code = i-N
    bgt     exit        // if i>N, branch exit

calculate:
    mvq     r0
    mvq     r1
    mult    0           // c*i
    mvr     r1          // c = c*i
    mvq     r0          // QT <= i
    addi    1           // i++
    mvr     r0          // r0 <= i++
    jump0 0             // branch to loop

exit:
    mvq     r1          // QT <= c*i
    stw1    0           // memory <= c*i
set_flag:
    ldil    1
    stw2    0       // Flag2 = 1
Last_exit:
    jump1 0

.core3
//First
init:
    seta0 loop
    seta1 Last_exit
    setd0 N
    setd1 d
    setd2 Flag3
    ldil    1           // index i = 1
    ldil    1           // d = 1
    mvr     r0
    mvr     r1

loop:
```

```
        mvq     r0          // QT <= i
        ldw0    0           // QT <= N
        comp    0           // condition code = i-N
        bgt     exit        // if i>N, branch exit

calculate:
        mvq     r0
        mvq     r1
        mult    0           // d*i
        mvr     r1          // d = d*i
        mvq     r0          // QT <= i
        addi    1           // i++
        mvr     r0          // r0 <= i++
        jump0 0             // branch to loop

exit:
        mvq     r1          // QT <= d*i
        stw1    0           // memory <= d*i
set_flag:
        ldil    1
        stw2    0       // Flag3 = 1
Last_exit:
        jump1 0
```

Updated information

date	Contents	details	version
12/14	New	Create	1.0
1/13	Modify	Created	1.0
1/24	Modify	Add source code in Appendix	1.1