Abstract

The aim of this project was to move a precision measuring device from single-core architecture to a multi-core architecture to improve performance. The application implemented in XC contributes to the research in the possibility of converting CSP into multithreaded software for XCore.

This project involves three aspects: Communication Sequential processes (CSP), XMOS architecture and XC programming language. The precision measuring device was called Servo Clock, and the specification of Servo Clock was given in CSP language by Renishaw. The project was divided into four steps. The first step was to analyze the behaviour of the *Servo Clock.* The second step was to check the definition equivalences between CSP and XC programming language. The third step was to refine the given specification into implementation level and do formal verification with FDR2, which was a formal verification tool on CSP. The last step was to do the converting work and evaluate the performance of sequential design against the parallel design of the Servo Clock.

In parallel design, the communication protocol was the most important and difficult part in the design flow. This project mainly focused on parallel communication protocol design and implementation. There were various ways to convert CSP models into XC models. The decision of which way to take depends on programming complexity in XC programming language, execution efficiency and resource efficiency in XMOS architecture. The *Conversion Rules* were applied in the manual conversion from CSP to XC.

The performance was evaluated by a set of experiments on sequential design and parallel design. The experimental results showed the parallel design greatly contributes to the performance of the Servo Clock, and applies the Amdahl's law [1].

Although this project achieved to some extent of success, there were still potential improvement in both parallel architecture design and module converting from CSP to XC to achieve even better performance. Moreover, there were still some issues need to be solved.

TABLE OF CONTENTS

Chapter 1 Introduction and Contexts

1.1 Introduction

Nowadays, with development of precision measuring industry, high-performance and high-accuracy devices are much in demand. For a part of business in Renishaw, higher accuracy means higher measurement quality. The measurement quality directly affects the quality of end products.

In general, the performance of embedded device is measured by frequency. Currently, with state-of-the-art industrial fabrication technology, to fabricate higher frequency device will significantly increase the cost. The multi-core design consists of several cores, and it could be fabricated at relevantly low cost with current fabrication technology. Thus, the multi-core architecture design is an alternative way to achieve high performance, and it is very popular in industry that pursues cost performance ratio. Compared to the sequential computing which executes instructions one after another, the parallel computing could execute several instructions simultaneously [2]. Obviously, the performance could be sped up in this way.

Designing a parallel system is more difficult than that of a traditional sequential system. Deadlock, livelock and determinism are commonly met in design phase. Figure 1-1 quoted from Pedersen J.B. in [3] shows the proportion of the kinds of errors occurred in parallel programming.

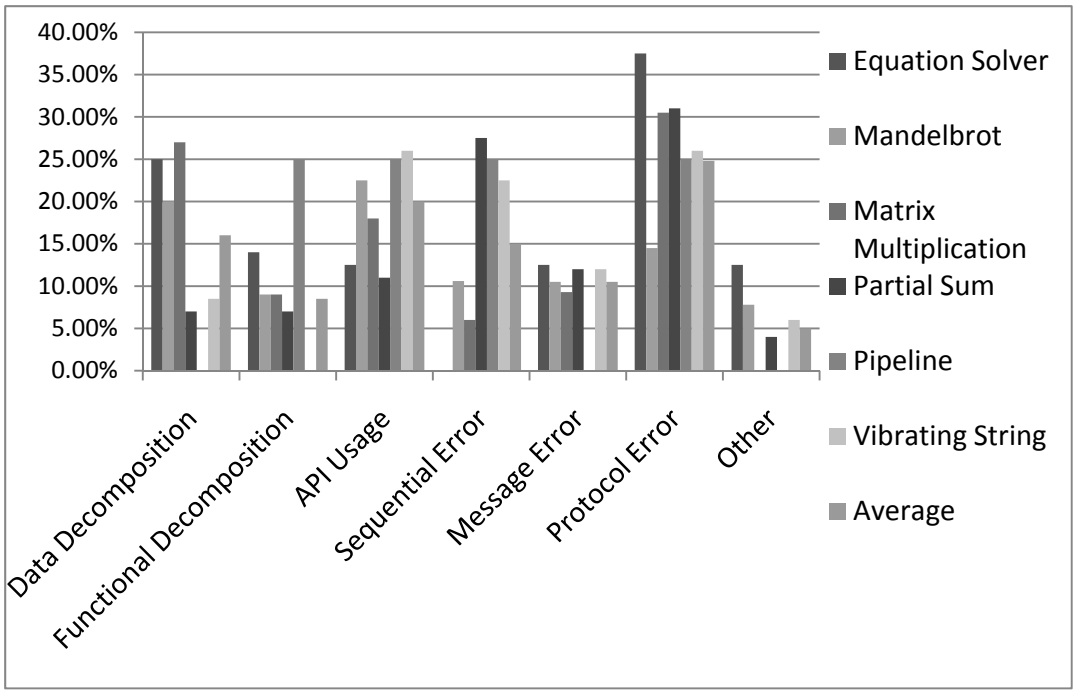


Figure 1-1 The results of the Error Reports (Quoted from Pedersen J.B. in [3])

It can be observed that *Protocol Problem* is the main problem of parallel programming. In this project, the idea of communicating sequential processes [4] was used to design the parallel system of *Servo Clock*. CSP is used to assist the design process of the parallel communication protocol of the *Servo Clock* in this project. CSP is a kind of process algebra and it can represent processes and analyze the in mathematical way. With the 30 years' development of CSP, it has been used in various areas, and it has an automatic tool FDR2 to assist analysis. XC is a programming language heavily influenced by CSP. They have similar programming idea, and it is a designated language for XCore.

This project has two aspects which are parallel communication protocol design and implementation of arithmetical algorithm. It introduced a design flow of a parallel system in *Chapter 3*. The design flow was divided into four steps. The first step was to analyze the behaviour of the *Servo Clock* which is performed at the beginning of *Section 3.2.* The second step was to check the definition equivalences between CSP and XC programming language, and derive the *Conversion Rules (Section 3.1)*. The third step was to refine the given specification into implementation level and do formal verification with FDR2 *(Section 3.2)*, which was a formal verification tool on CSP. The last step was to do the converting work *(Section 3.3)* and evaluate the performance of sequential design against the parallel design of the Servo Clock *(Chapter 4)*.

1.2 Aim and Objectives

The aim of this project is to move a precision measurement device from single-core architecture to a multi-core architecture to improve performance. It is achieved by designing parallel architecture in CSP language and then converting the CSP code into XC programming language. The XC code was evaluated on XC-1 Development Kit based on XS1-G4 XCore processor.

The objectives to achieve the aim are as follows:

1> Analyze the CSP specification of Servo Clock and investigate the behaviour of the original sequential design.
2> Refine the CSP specification into low-level implementation which could be converted into XC programming language
3> Check the equivalences between CSP and XC programming language, and do the converting work. If the CSP implementation could not be converted into XC due to the equivalences, repeat objective 2.
4> Integrate the modules written in XC to form a whole system.
5> Evaluate the performance of the XC application.

The aim and the main part objectives are the same as that in interim report. All the objectives were successfully completed.

Although some unexpected problems occurred when doing this project, most of them were successfully resolved in the end. The details will be shown in Chapter 3.

Chapter 2 Background

2.1 Communicating Sequential Processes (CSP)

In parallel system, compared to threads, processes have their own memory spaces, and run individually. They communicate with each other by communication channels instead of shared data, and there is no arbitrator to manipulate processes. In some cases, design a parallel system with processes is better than those using threads with arbitrator.

Communicating Sequential Processes (CSP) is a kind of process algebra, and it was first introduced by Hoare in 1978[4]. Like other process algebras, it can represent and calculate the interactions of processes in mathematical way. The determinism, freedom of livelock and deadlock are commonly met in concurrent system design. Deadlock happens when processes are waiting for the results of each other before they can make further progress. Similarly to deadlock, livelock is caused by process that falls into an infinite loop of internal events and runs without any progress.

There were several applications of CSP in the past. One of the CSP applications was to use CSP to write specification and do the verification of the complex superscalar pipeline of INMOS T9000 Transputer [5]. Another application of CSP was to formally detect whether an embedded system for International Space Station IIS was free of deadlock [6][7]. The major advantage of CSP is that the CSP is related by refinement, which means in CSP one process could be replaced by another process which is under the constraint of the original process and do more than the original process for the purpose of implementation[8]. Until now CSP has been development for more than 30 years, and with its automated tool FDR2, CSP has been widely used for the purpose of verification, specification and analysis of concurrent systems.

In this project, CSP approach was used to check refinement of a given specification and check the determinism, freedom of livelock and deadlock of implementation.

2.1.1 Specification Refinement

In CSP, events are declared as channels, and processes are constructed by sequences of events. A simple example which is written in machine readable CSP for FDR2 is shown in Code 2-1.

```
channel a,b,c
P = a-> b-> STOP
Q = a -> c -> Q
```

Code 2-1 A CSP sample

The P and Q are processes and a, b and c are events in the example. Processes could communicate with each other through channels. The advanced communication among processes is discussed in the next section. Refinement is the advantageous characteristic for CSP, and it is also an important part of this project.

In CSP, refinement means replacing a specification process by another low-level process [8]. In this project, the purpose of refinement is to replace the specification process into lower-level process which is appropriate to be converted into XC programming language.

2.1.1.1 Traces refinement

Traces refinement in CSP is usually used to check the safety property of the refining processes [9]. It's denoted as Equation 2-1.

$$\mathbf{P} \sqsubseteq_{\mathbf{T}} \mathbf{Q} \qquad \textbf{Equation 2-1}$$

It means P is refined by Q. All the traces of Q are a subset of the traces of P, that is $traces(Q) \subseteq traces(P)$. Thus, process P gives a boundary to Q defining what events Q can perform. In our project, we could replace P and Q in Equation 2-1 with *Specification* and *Implementation* shown in Equation 2-2

$$\textbf{Specification} \sqsubseteq_{\mathbf{T}} \textbf{Implementation} \qquad \text{Equation 2-2}$$

"Trace refinement only specifies safety. It doesn't require the refining process to do anything. [9]". Thus, it doesn't necessarily mean trace refinement equals to process equivalence. There are two processes *Spec* and *Impl* shown in the example as follows:

$$Spec = a \rightarrow b \rightarrow Spec$$

$$Impl = a \rightarrow b \rightarrow STOP$$

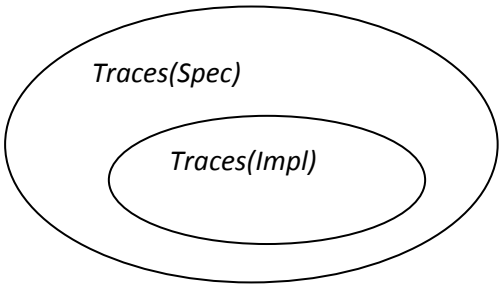


Figure 2-1 *Traces(Impl)* does not equal to *Traces(Spec)*

For process *Spec, traces(Spec)={<>,<a>,<a,b>,<a,b,a>…}.* For process *Impl, traces(Impl)={<>,<a>,<a,b>}.* From Figure2-1, it can be observed that$traces(Impl) \subseteq traces(Spec)$, thus,$Spec \sqsubseteq_{\mathrm{T}} Impl$, saying process *Spec* could be traces-refined by *Impl*.

But actually, process *Spec* is a recursive process, it can infinitely keep running while process *Impl* could run only once and then leads to deadlock (*STOP*). Hence, *traces(Spec)≠traces(Impl).* Therefore, as indicated before, traces refinement just specifies what events a refining process can do, but it doesn't require the refining process to do anything [10]. The traces model is suitable for analyzing safety, because the traces related to processes give the information to verify safe properties [10].

In conclusion, traces refinement is not enough to say process equivalence and it only specifies safety, where Steve Schneider said: "safety properties are generally of the form 'something bad will not happen' " [10].

2.1.1.2 Failures refinement

Similarly to traces refinement, the failures refinement specifies the events that a process can refuse in a state, denoted as Equation 2-3.

$$P \sqsubseteq_F Q \qquad \text{Equation 2-3}$$

It's an addition a step after a traces refinement check. Although it doesn't require the refining process to do anything, it could indicate what events a state refuses. Denote *refusals (Process/tr)* as the traces that *Process* can refuse after performing trace *tr.* Taking the previous example again:

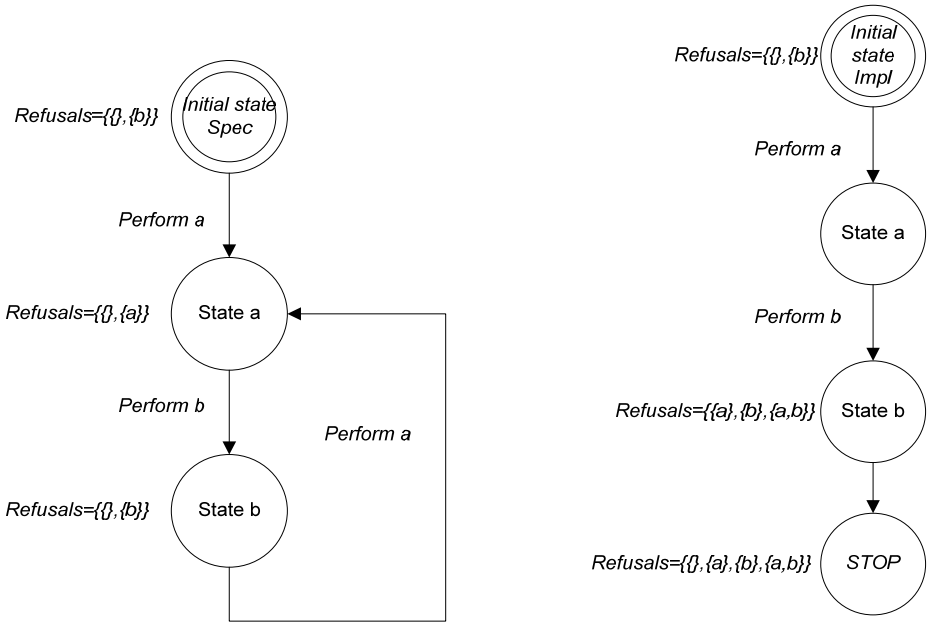


Figure 2-2 *Refusals* of process *Spec* and *Impl*

From Figure 2-2 it can be observed that after traces <a,b> the refusals of process *Spec* and *Impl* are different from each other. That is *refusals (Spec)⊈ refusals(Spec).* With this example, Equation 2-3 could not be satisfied.

2.1.1.3 Failures-divergence refinement

Divergence or livelock in CSP means an infinite loop of internal events which causes the process non-progress [10]. Denote *failures-divergence* refinement as Equation 2-4

$$P \sqsubseteq_{FD} Q \qquad \text{Equation 2-4}$$

When a process reaches the divergent state, there is no way to predict what events it will perform afterwards. With only the failures model and the traces model, it is not possible to detect divergence in a process where a much more complicated scenario is presented. Denote *divergences(Process/tr)* as the divergent trace after which *Process* goes into divergent state as shown in Figure 2-3. (The operator | means OR)

P = b -> STOP | a -> P

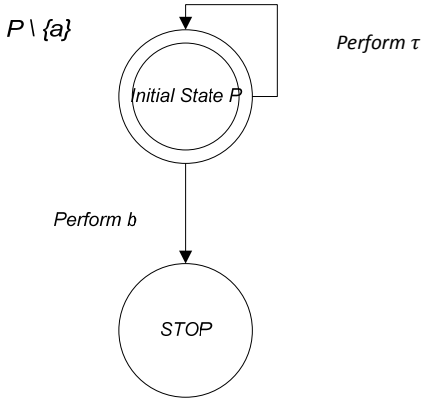


Figure 2-3 Process *P* with hidden event *a*

The operator \ allows hiding a set of events in a process and making them performs as the invisible internal event τ. In this example, the event '*a*' is hidden, and it is performed as an internal event. Thus, there is a possibility that the process *P\{a}* could fall into a infinite sequence, and there is no guarantee whether it could jump out of that sequence [10]. The divergent trace of this example is *divergences(P/tr)=<>.* In a refinement, usually the refining processes are doing more events than the specification processes, and the extra events are invisible internal events when seen from the specification. As mentioned above, this will lead to divergence. So, a more comprehensive model is needed, which is the failures-

divergence model. **Together with *traces* mode, they are able to prove the refining processes are valid refinement to specifications with the failures-divergence model.**

See the example below:

$$SPEC = a \rightarrow b \rightarrow SPEC$$

$$IMPL = a \rightarrow b \rightarrow DIVERGENCE \setminus \{|c|\}$$

$$DIVERGENCE = c \rightarrow DIVERGENCE$$

traces(SPEC)={<>,<a>,<a,b>,<a,b,a>,<a,b,a,b>...}. As event *c* is hidden when seen from the environment, *traces(IMPL)={<>,<a>,<a,b>},* the equation *SPEC [T= IMPL* is satisfied. Using the principle in section 2.1.1.2, the equation *SPEC [F= IMPL* is satisfied. After performing event *b,* the process *IMPL* will jump into an infinite sequence {c,<c,c>, <c,c,c>,…}, which is an internal sequence of *IMPL* seen from the environment. From *divergences(SPEC/tr)={}* and *divergences(IMPL/tr)={<a,b>},* it can be observed that:

$$divergences(SPEC/tr) \neq divergences(IMPL/tr)$$

Thus the equation $SPEC \sqsubseteq_{\mathrm{FD}} IMPL$ for *failure-divergence* refinement is not satisfied.

2.1.1.4 Synchronization and parallel

Parallelism is the most important property in concurrent system design, and also, in some cases, the source of deadlock and livelock. Fortunately, CSP gives us an algebraic way to analyze parallel systems. In CSP, there are three forms of parallel, *alphabetised parallel, interface parallel* and *interleaving*. In CSP, the way parallel processes interact with each other is to synchronize with events via channels [10].

a> Alphabetised parallel

The alphabetised parallel form offered a way for processes to interact with each other through synchronization. The form of alphabetised parallel is denoted as Equation 2-5

$$\mathrm{P}_1 {}_{\mathrm{A}}||_{\mathrm{B}} \mathrm{P}_2 \qquad \textbf{Equation 2-5}$$

Where A and B are alphabets of P_1 and P_2 respectively, and each process could only perform the events in their own alphabet. The two processes will synchronize on the common events if $\mathrm{A} \cup \mathrm{B} \neq \{\}$. If one of the processes failed to synchronize on the common event, the whole system will lead to deadlock. It's

important to define correct alphabets for each process, or it will have unexpected results.

b> Interleaving

Processes executing independently in parallel without synchronization is called interleaving [10], which is denoted as Equation 2-6

$$\mathbf{P_1|||P_2} \qquad \textbf{Equation 2-6}$$

In this combination, processes do not interact with each other except for termination [10], even if those two processes have common events in their alphabet. In this case, the environment does not have the control on the combination. In Equation 2-4, if $\mathrm{A}\cup\mathrm{B} = \{\}$, the behaviour of the combination in Equation 2-4 is the same as that in Equation 2-5.

c> Interface parallel (or *Sharing*)

Interface parallel is a mixed combination form of alphabetized parallel and interleaving [10]. It's denoted as Equation 2-7.

$$\boldsymbol{P_1|_A|P_2} \qquad \textbf{Equation 2-7}$$

The processes will synchronized on events which are treated as barriers in alphabet '*A*', and then perform the following events individually. See example III below.

$$P_1 = a \rightarrow b \rightarrow c \rightarrow d \rightarrow STOP$$

$$P_2 = c \rightarrow e \rightarrow f \rightarrow STOP$$

$$P_1|_{\{c\}}|P_2$$

Process P_1 and P_2 will execute independently and wait each other to synchronize on the common event {c}, then execute independently afterwards until termination. It's different from alphabetized parallel, because there is only one common event allowed to synchronize at one time.

Furthermore, the interface parallel could be indexed to allow two more processes to share a same interface, as denoted in Equation 2-8

$$\boldsymbol{|_{A_{i\in I}}|P_i} \qquad \textbf{Equation 2-8}$$

2.1.1.5 External Choice

External Choice (also known as ALT [11]) denoted in Equation 2-9 is one of the most useful and important properties in CSP. It can accept several events from

environment and engage in only one of them [10]. With this property, CSP is able to interact with environment.

$$P_1 \square P_2 \qquad \text{Equation 2-9}$$

Where P1 and P2 are processes and which processes would be engaged in depends on the events from environment.

2.1.2 FDR2-Formal Systems (Europe)

With the development of CSP, there are a lot of CSP tools for analyzing systems which are described in CSP, such as *Adelaide Refinement Checker* (*ARC*)[12], *Process Analysis Toolkit* (PAT) [13] and FDR2[14].

Failures-divergence Refinement (FDR) is a product of Formal Systems (Europe) Ltd. and it's free for academic use. FDR2 is the later version of FDR. It is a CSP refinement tool which is now widely used in refinement checking, determinism checking and freedom of deadlock and livelock checking. The main functionality of FDR2 is to check refinement processes to see whether the refining processes refine the specification processes.

In this project, FDR2 is the only formal analysis tool for CSP code, which gives the fundamental proof of correctness. FDR2 can do refinement checking by applying the models of *traces*, *failures* and *failures-divergence*. As mentioned in section 2.1.1, trace model is used to check safety properties which are about what a process can do. Failures model is used to check processes which are free of divergence for the purpose of liveness and safety property. In the FDR2 user manual [14], it said: "Failures-divergence refinement is used for proving safety, liveness and combination properties, and also for establishing refinement and equality relations between systems."

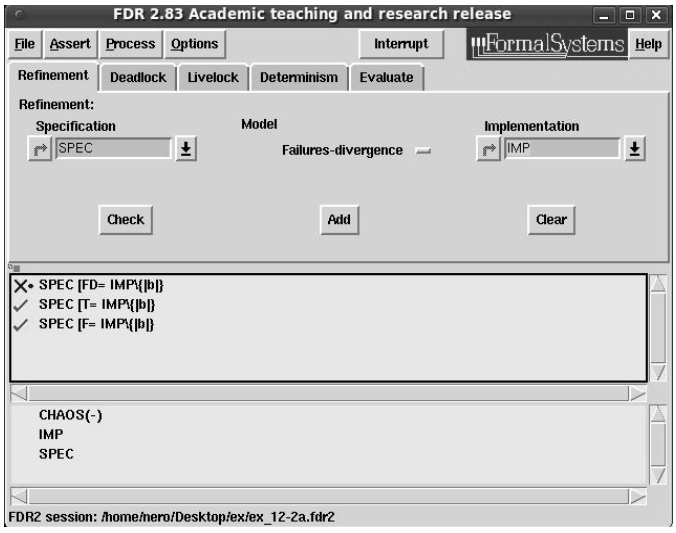


Figure 2-4 Checking refinement with failures-divergence, traces, failures models in FDR2

If the tests were failed, the FDR2 will give feedback indicating where the problem is and show how the problem happened through debug window. See Figure2-5 below:

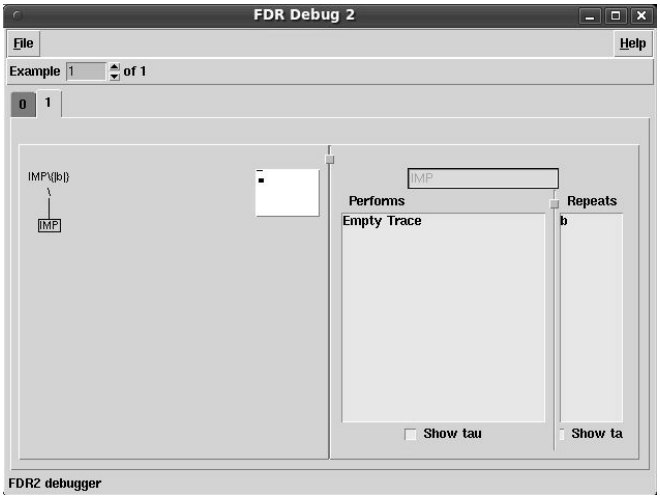


Figure 2-5 FDR debugger

With those properties of FDR2, the refinement could be checked automatically, which greatly reduced the workload.

FDR2 also have accompany software Process Behaviour Explorer (ProBe) shown in Figure 2-6, which is very useful for the user to track the processes by performing events step by step.

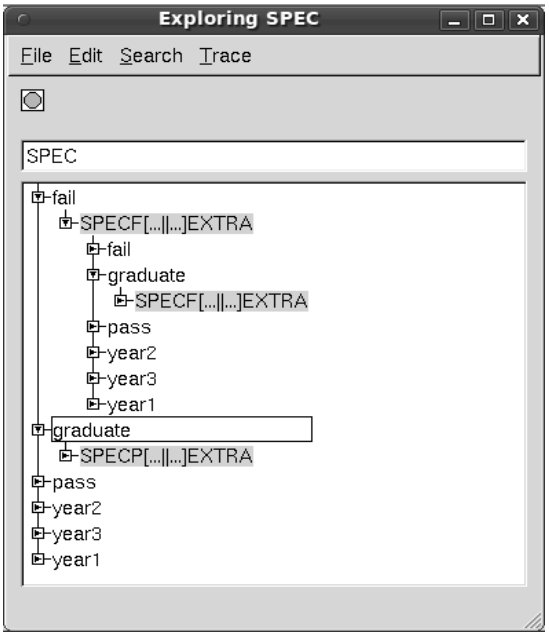


Figure 2-6 Process Behaviour Explorer (ProBe)

With ProBe, we could intuitively observe the behaviour of a process, and even easily locate the problem from the feedback of FDR2.

Another usage of ProBe is to track the traces of the XC program which is running on target platform. It enables developer to check the equivalences between XC program and CSP implementation at a certain step.

2.2 XMOS Architecture and XC Programming Language

The target platform of this project was XC-1 Development Kit from XMOS Industry. It is based on XS1-G4 device.

2.2.1 XMOS Architecture

As shown in Figure 2-7, the XS1-G4 has 4 XCore processors, which is the basic building component of XMOS devices. The XCore is both a general purpose processor and a multi-threaded processor. If programmed as general processor, it can execute traditionally C programs. If programmed as multi-threaded processor, it enables the ability to support concurrent programming. Each XCore processor has its own clock input, timers, 64 I/O ports and memory, all of which could be programmed by software. In XS1-G4, XCore processors communicate with each other through *XMOS Link Interface,* which is based on a mechanism of high-performance *switch* [15]. The messages exchange among XCore processors could be efficiently routed by *XMOS Link.* With the interconnect mechanism, each XCore processor can access the memory of other XCore processors efficiently.

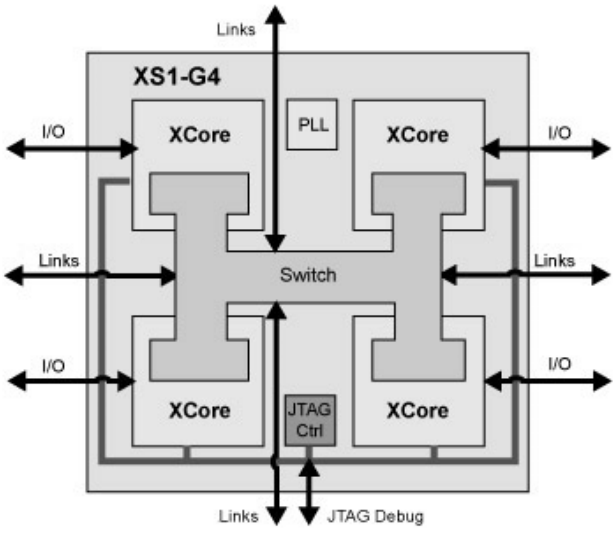


Figure 2-7 *XS1_G4 architecture* from XMOS official website

(Source : XMOS Ltd.2010[11])

Each XCore processor, is has 8 threads and 32 channels, and each thread has its own memory space and registers. Thus, each thread could be seen as a logic core. The thread scheduler in the XCore processor schedules the group of threads with round-

robin manner [15]. So the execution time for each thread is predictable and measureable. With the advantage above, threads could perform hardware level I/O tasks which require high-accuracy timing.

Threads could access data in shared memory through channels and I/O pins. There is a 64kb shared memory inside each XCore processor, to which threads could access without conflict. The I/O pins are more sophisticated than general-purpose I/O pins. If data is sampled from I/O pins, there is a hardware mechanism which allows streams of data to be serialized or de-serialized without consuming any processing resource [15]. The XCore processor can be programmed as an event-driven processor, and the synchronization between different threads is performed by hardware. Therefore the synchronization doesn't consume any software processing effort. There is a significant difference between the event-driven processor and the general processor. For the event-driven processor the context of event has already been initialised before waiting for the event while for general processor with interrupt the context should be restored before jumping into interrupt handler after the receiving the interrupt signal [16]. In addition, the thread speed could reach 400 MIPS [15].

With those advantages mentioned above, XCore processor could respond to complicated signals rapidly and simultaneously, and could be used to implement some of the functions implemented by hardware.

In this project, the XMOS architecture could potentially benefit the performance of the *Servo Clock* which requires a concurrent framework and rapid and accurate responding to the environment.

2.2.2 XC Programming Language

As the XCore processor has different architecture from general processors, the programming method is also different to some extent. "The XC programming language, developed by XMOS Company, is an extension of general C programming language."[17] It comprises functionality of general C, I/O definition, timer definition and parallel statements, etc.

XC is a designated programming language for the XCore architecture, and it is highly optimised for the XCore instruction set [17]. XC is heavily influenced by CSP as it supports the parallel programming in a similar way as CSP. One of the greatest differences between XC and the other high level programming languages for embedded system is that the XC has integrated the timers and ports efficiently and safely, which means user can declare and use timer and port type variables, instead of manipulating relevant control registers directly.

2.2.2.1 Concurrent Threads and Synchronization

Parallel programming is intrinsic nature of XC programming language which has dedicated statement *par* for creating concurrent threads. The example is shown as follows:

```
int main(void)
 {
int i ,j,k;
      par {
            on stdcore[0]:i=k+1;
            on stdcore[1]:j=k-1;
      }
}
```

The par statement creates two threads distributed on 2 different cores, namely XCore [0] and XCore [1]. In XS1-G4, there are 4 XCore processors on chip, and each processor could accommodate 8 threads.

"*par* statements may be used anywhere in a program. Each XS1 device has a limit of eight threads available on each of its processors, and a program that attempts to exceed this limit is invalid."[17] So at most 4*8=32 concurrent threads could be created with par statement. The concurrent threads could only be created by applying *Thread Disjointness rules* [17].

"The rules for disjointness on a set of threads T0 . . . Ti and a set of variables V0 . . . Vj are as follows:

a> If thread Tx contains any modification to variable Vy then none of the other threads (Tt ; t 6. x) are allowed to use Vy.

b> If thread Tx contains a reference to variable Vy then none of the other threads (Tt ; t 6. x) are allowed to modify Vy."[17]

So the program in the example above is legal. k is read by two threads, but should never been modified by more than one thread.

The illegal example is shown on next page.

```
int main(void)
 {
      int i ,j,k;
      par {
            on stdcore[0]:i=k+1; //Thread A
            on stdcore[1]:j=i+1;  //Thread B
           }
}
```

The statement j=i+1 in *Thread B* depends on the result of *Thread A,* therefore the *Thread Disjointness rules* could not be satisfied. Statements in created threads should not have data dependency on other threads.

The most important feature of XC for our project is synchronization. Synchronization gives a way to exchange data and messages between threads.
Threads could communicate and synchronize with each other through channels.

```
# include <platform .h>
on stdcore [1] : in port   port_input = XS1_PORT_8B ;
void thread_a ( chanend dataIn) {
        char data ;
        while (1) {
                dataIn :> data ;
                func_processing (data);
                 }
}

void thread_b ( chanend c, port port_input) {
        char data ;
        while (1) {
                data = waitForInputData (port_input );
                c <: data ;
        }
}

int main ( void ) {
        chan c;
        par {
                on stdcore [0] : thread_a (c);  // Thread A
                on stdcore [1] : thread_b (c, port_input );   // Thread B
        }
}
```

The statement *chan c* declared a channel *c*, and it has two channel-ends distributed on *Thread A* and *Thread B.* Thus *Thread A* and *Thread B* are connected by the *channel c*, and they can therefore communicate and exchange data via *channel c*.

The functionality of this program is that *Thread A* gets an input data from port *port_input*, and transfer the data via channel to *Thread B*. As the channel defined in XC is synchronous, it'll wait for *Thread B* to synchronize on *channel c* before *Thread B* could continue executing,

It's very useful to implement CSP specifications which are in *alphabetized* or *interface parallel*.

Consider the XC code above, the CSP specification could be written as follows:

$$THREAD_B = port\_input.data \rightarrow c.data \rightarrow THREAD_B$$

$$THREAD_A = c.data \rightarrow func\_processing.data \rightarrow THREAD_A$$

$$SPEC = THREAD_A|_c|THREAD_B$$

In this example, the process *SPEC* is the CSP implementation of the XC code, and the XC code is the executable code of this CSP implementation. The $THREAD_A$ and $THREAD_B$ are synchronized using interface parallel. They synchronized on the interface of event *c*. In concurrent system design, there is a lot of synchronization among processes. With *channel* provided by XC, it's quite convenient and easy to convert the synchronized processes into executable XC code due to the inherent characteristics of XC programming language. As the synchronization through channels is done by hardware, it has potential benefits to the performance.

2.2.2.2 I/O Ports

The I/O ports in XC have three modes: direct I/O mode, clocked mode and conditional mode [17]. For direct I/O mode the I/O pins operate as general-purpose I/O pins. For clocked mode, the I/O pins are synchronized with a reference clock, which could be programmed by software. For conditional mode, the input and output could be delayed by hardware when the condition is met. This mode is quite useful when implementing certain kinds of hardware functions such as UART.

2.2.3 Development Environment

The design tool from XMOS consists of *XMOS Development Environment (XDE), XMOS Compiler, XMOS GNU Debugger, XMOS Timing Analyzer, XMOS Simulator* and *XMOS Board and Manufacturing Utilities.*

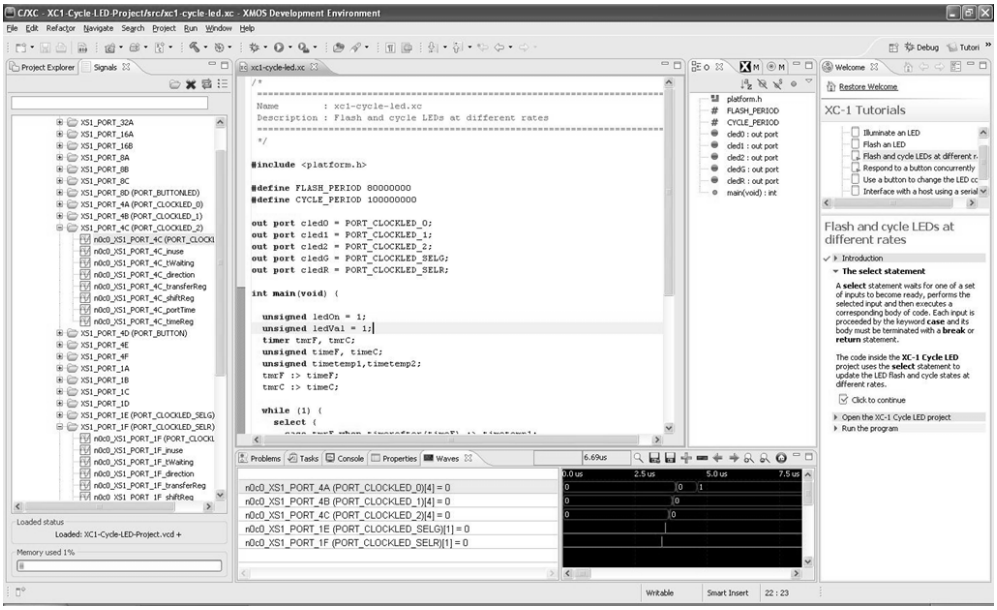


Figure 2-8 XMOS IDE GUI with *C/XC perspective*

In this project, Eclipse based IDE, *XMOS Compiler* and *XMOS Simulator* are mainly used. *XMOS Compiler* and *XMOS Simulator* have already been integrated in the IDE (XDE). The *XDE* has two perspectives, the *debug perspective* and *C/XC perspective (shown in Figure 2-8).*

The *C/XC perspective* provides the *Outline window,* the *editor window,* and the *resource windows,* which are helpful when writing and debugging code in a project.

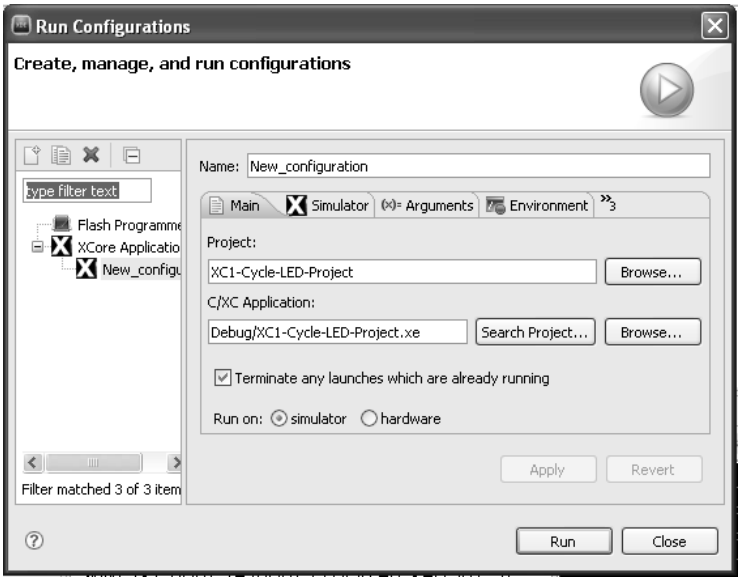


Figure 2-9 *Run Configuration Window*

The design tool provides two running targets. The program could be run either on the *Simulator* or on a hardware device. In the *Run Configuration window* as shown in Figure 2-9, the running target should be chosen before running. It's better to run on

simulator than on hardware device when debug is needed, because the *XMOS Simulator* provides Waveform *tracing* [18]. With this function, it's possible to analyze the behaviour of the signals in each thread of XCore visually as shown in Figure 2-10.

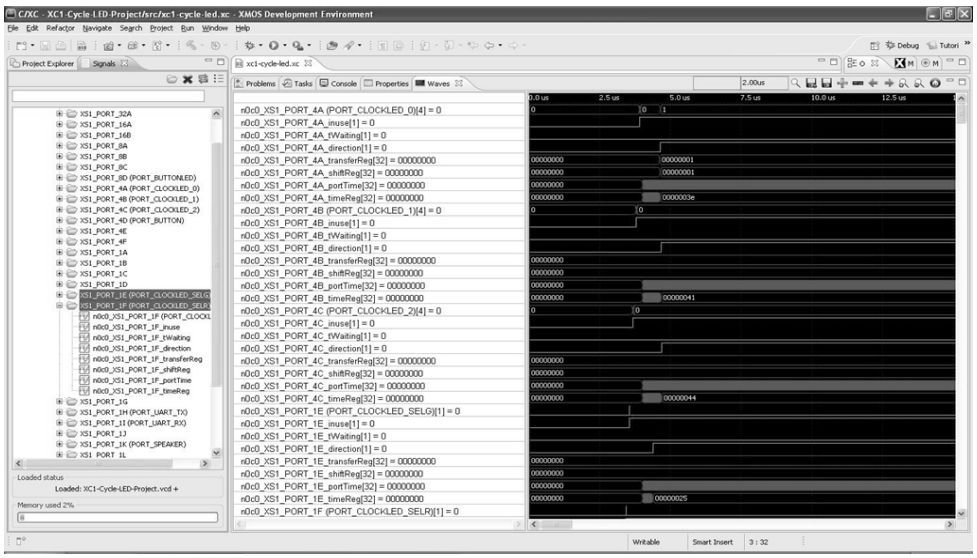


Figure 2-10 Signal waveforms generated by *XMOS Simulator*

In the *Debug Perspective,* the running state of XCore processors can be monitored and the instruction on threads can also be tracked.

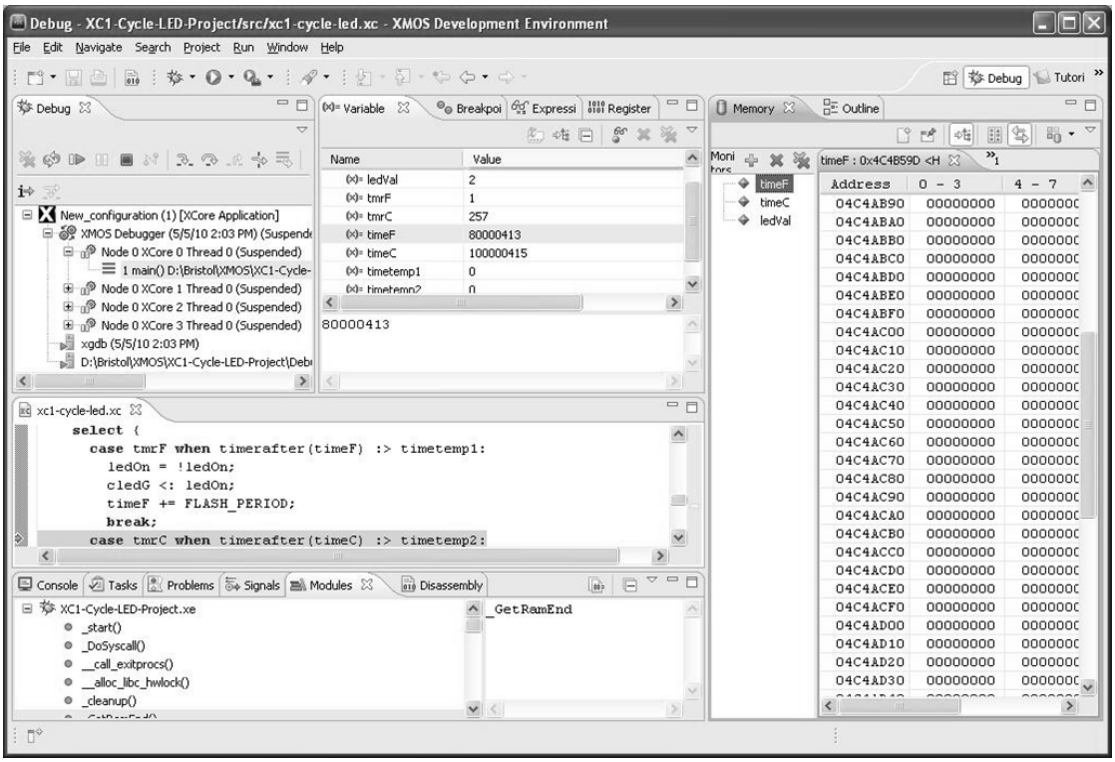


Figure 2-11 XMOS IDE GUI with *Debug Perspective*

In Figure 2-11 developers can observe the real-time state of the program running on hardware. With the debugger, processor can be suspended at any time to monitor the current state, including the contexts of timers, I/O ports, current instructions etc.

2.2.4 Timing Analysis

In embedded system design, timing determines the performance of a system, as the critical path acts as a performance bottleneck of the system. There are two kinds of timing analysis. One is static timing analysis and the other is dynamic timing analysis. The static timing analysis could analyse whether a piece of code constantly meet the time requirement before running in real-time.

If a system running in real-time, the timing becomes more complicated than that of static. In this case, dynamic timing analysis is required.

2.2.4.1 Static Timing Analysis

XMOS Timing Analyzer (XTA) is kind of static timing analyzer. It is useful and powerful tool. As mentioned in section 2.2.1, the execution time is predictable in XMOS architecture. The tool can give out the shortest and longest path through a piece of code as shown in Figure 2-12.

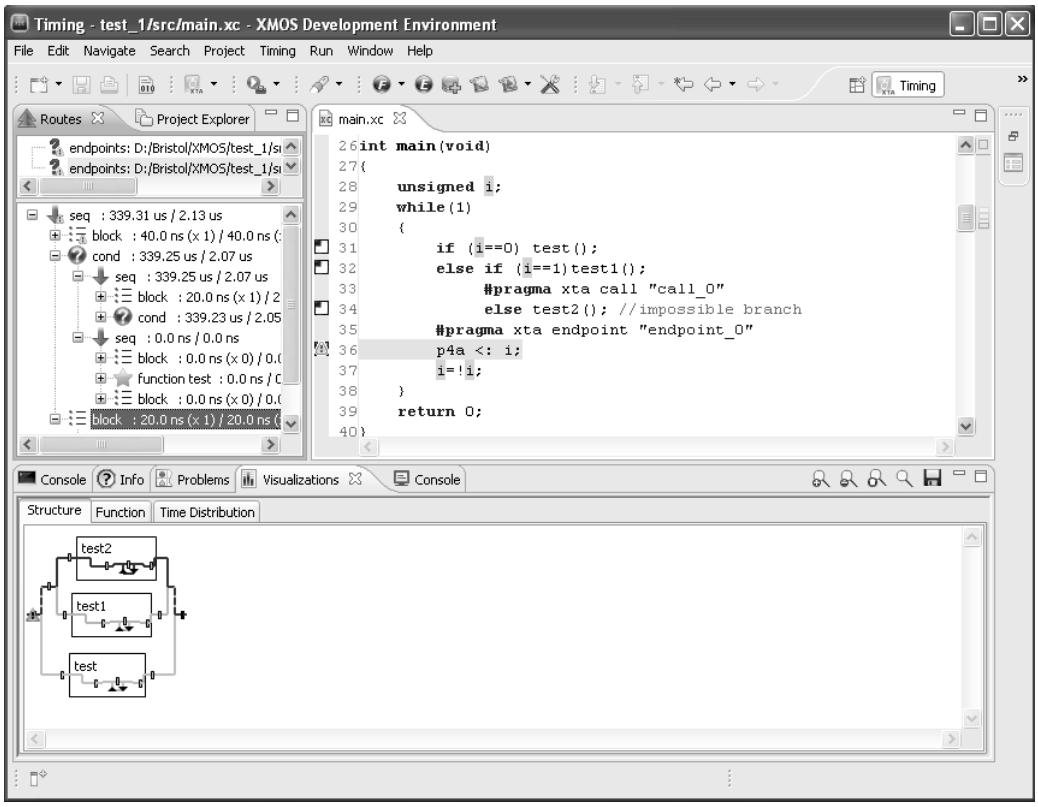


Figure 2-12 Visual results of XTA

The visualization view of XTA which is at the bottom of the window (Figure 2-12) provides graphical representation of the route. It can clearly represent the structure of the code and the results of the analysis. The red line indicates the longest path while the green line indicates the shortest path. The *XTA* also gives out the detailed timing information which is shown on the left side of the window.

With the given specified time requirement, the tool can also detect timing failures and give feedback[19] which is shown in Figure 2-13

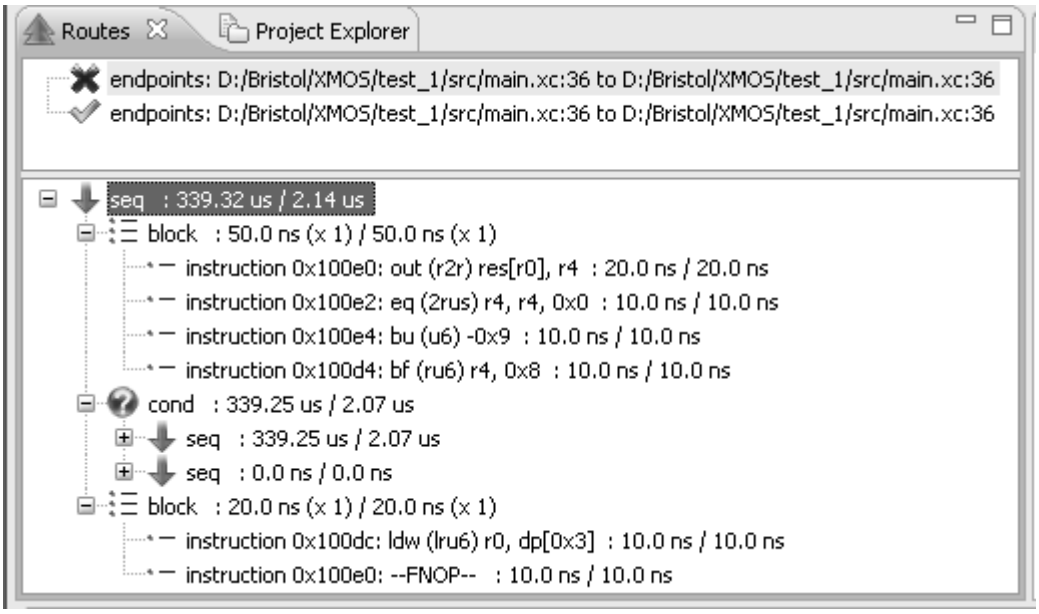


Figure 2-13 Feedback from XTA

The *red cross* indicates the analyzed part of the code doesn't meet time requirement while the *green tick* indicates the analyzed part of the code meets time requirement.

2.2.4.2 Dynamic Timing Analysis

The dynamic timing analysis could be achieved by using timers in *XCore* and the *print()* function in XC. The example is shown in Code 2-2.

```
timer tmr;
unsigned start_tmr,stop_tmr;
tmr:>start_tmr;
// functions to execute.
chk_tmr:>stop_tmr;
printstr("time:");
printuint((stop_tmr-start_tmr)/100);
printstrln("us");
```

Code 2-2 Dynamic timing analysis sample

The output would be like: "time: 10us". The result is given in the unit of us.

2.3 Servo Clock and the Control System

The Servo Clock is the core of the precision measuring device. As shown in Figure 2-12

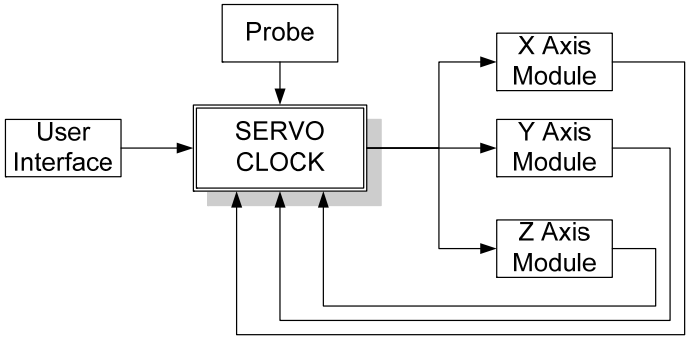


Figure 2-14 System topology

The Servo Clock receives the data from Device Interface and User Interface. Then it sends back the control signal to Device Interface after performing the algorithm. The peripheral device consists of 2 main parts, which are the *Probe* and three *Axes.* The device can move in a 3-dimentional space.

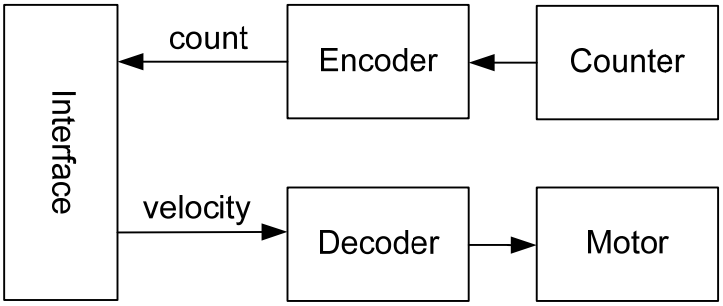


Figure 2-15 Structure of each axis

As shown in Figure 2-13, each of the three *Axes* is responsible for x, y, z axis respectively, and each of them is an integrated device which is formed by an encoder, a motor, and a decoder. The encoder encodes the signal from the motor as '*Count*', and sends the data to the *Servo Clock*. The '*Count*' indicates the distance the objective has gone in a running cycle. The decoder receives the *velocity demand* from the *Servo Clock*, and sends the control signal to the motor after the *velocity demand* data decoded.

The *Probe* is a touch sensor, it is used to detect whether it is clear in current position. The basic close-loop control algorithm of the system is shown in Figure 2-14.

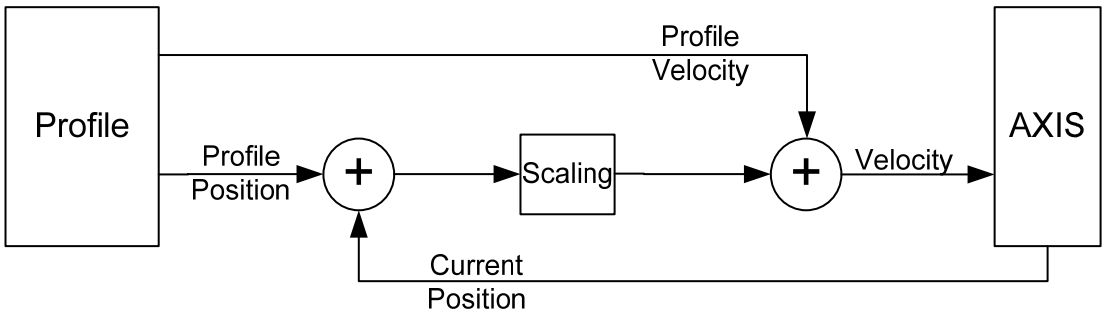


Figure 2-16 Close-loop control algorithm

The *profile* is generated with position demand and current position. The *Servo Clock* compares the latest position from *Axes* with *profile* and amplifies the difference with a certain scaling factor. Then add *Velocity Compensation* to the *amplified difference* and transmit the result to *Axes.* In reality, the control algorithm is much more complex than the basic control algorithm, but it is out of the scope of this project. The simple algorithm used in this project is valid to demonstrated the full functionality of the *Servo Clock,* although it is not as accurate as complex algorithms.

2.4 Previous Work

As the titles suggests, this project involved parallel system design using CSP approach and converting CSP into executable code. There were several relevant works of implementing CSP in executable target language and using CSP to design and verify parallel systems.

For the conversion work, besides XC, there are several target languages that CSP could be converted into. One of these languages is *Java* with the project of *CSP for Java* which was proposed in 1997. “Java is a sequential programming language that features Object-Orientation and multithreading. However, multiple threads run in parallel processes are created in a sequential fashion”[20].The concept of the project was based on early version of CSP introduced in [21]. G. Hilderink, J. Borenink, W. Veroort and A. Bakkers implemented the channels of CSP, which is used to synchronize processes, in Java with the concept of *monitors* in[21]. They extended *Java* by creating *Channel Class* and additional constructs which are compliant to CSP definitions.

The other work is of C++CSP, which is an extension for C++ enabling C++ to adopt CSP implementation. It was implemented using CSP-derived approach instead of locks-and-threads approach [22]. “The *CSP-derived approach* is to eliminate sharing of data between concurrent processes. Processes are separate pieces of code that communicate explicitly via channels. When programming a process, you only need to consider the channel communications (and other synchronisations, such as barriers),

and do not need to worry about race hazards involving shared data or scheduling corner-cases"[22].

V.Raju, L.Rong and G.S.Stiles introduced two approaches of conversion of CSP in[23]. One of the approaches was to use *Mathematica*, a very expensive tool package, to provide a quick path to build a translator with the given *pattern matching* and *processing features*[23]. The other approach was to implement translators in C++ to convert CSP into JCSPP and CCSP, which was helpful to doing my project.

A. Freitas and A.Cavalcanti did the work of converting *Circus* into *Java.* "*Circus* is a combination of *Z* and *CSP* that supports the development of state-rich reactive systems based on refinement."[24] In their paper, they introduced a translation strategy. The translation strategy was divided into two phases, which were collecting information and converting Circus into Java.

Although there were relevant previous work of conversion of CSP executable code, XC has different definitions to the definitions of CCSP, JCSP and CTJ, etc. However, their work gave ideas and strategies to this project.

For the parallel system design, there were plenty of works have been done before.

Karen Seidel used CSP to design the PI-Bus in [25]. He wrote a high-level CSP specification corresponding to the PI-Bus draft standard, and refined the CSP specification into two implementations to indicate the draft standard was prejudicial. His design procedure was helpful to my project.

2.5 Summary

It is known from section 2.1 that "processes in CSP are related by refinement"[8]. Fortunately, there are several assistant tools for analyzing systems written in CSP, especially FDR2 which may reduce workload during parallel system design phase.

This chapter reviewed the background knowledge of the project and showed basic examples of converting CSP into XC as well. There is some similar work of converting CSP into executable target languages, and even the automatic tools have been produced previously. As XC programming language has different definitions to CCSP, JCSP, etc., the techniques and some relevant method from previous works could not be used directly in this project.

The next chapter demonstrates the detailed design flow of the parallel *Servo Clock*.

Chapter 3 Design Flow

This chapter consist of two main parts which are communication protocol design with CSP and implementation in XC. The first two sections talks about the design procedure of the parallel communication protocol. The last section talks about the implementation in XC which includes the conversion of CSP and numerical algorithm design.

3.1 Equivalence Models of XC and CSP

CSP and XC are different languages. Although XC was heavily influenced by CSP and came after CSP, it has different definitions to CSP. Hence, system described in CSP could not always be converted into XC to some extent. Thus, before converting CSP implementation into XC, the equivalence between XC and CSP should be checked in advance. Also, in system design phase with CSP, the equivalence should be taken into account for conversion purpose.

This section did some experiments and made some equivalence models of XC and CSP which would be applied in the converting work of the next section. The CSP code in this chapter is written as machine readable CSP for FDR2. The notations of the CSP could be found in *Appendix A*.

3.1.1 Channel Communication

The type of channel in XC is “a synchronous, point-to-point connection between two threads over which data may be communicated [17]”, which is different from the channel in JCSP where the channel has been proved equivalent to the channel in CSP by Jeremy Martin [26].

In JCSP channels are implemented in the classes of “One2OneChannel, One2AnyChannel, Any2OneChannel and Any2AnyChannel [24]”, which means a channel may have one or multiple ends for input and output respectively in CSP and JCSP. So, CSP structures involve *channels* could not always be directly converted into XC. Code 3-1 shows an invalid example.

In Code 3-1, the CSP code described that the three processes were synchronized on event e. *PE* output the value 1 through the channel e while *PB* and *PA* received the data from channel e.

It fails when compiling XC code in Code 3-1, which means the XC code in Code 3-1 is invalid. The XC program requires three channel-ends on a channel, but the channel in XC just has two channel-ends. Thus the CSP script could not converted into XC in this case.

```
int main(void) //XC code
{
    chan e;
    par
    {
        { //process PE
            while(1) e<:1;
        }
        { // process PB
            int b;
            while(1) e:>b;
        }
        {// process PA
            int a;
            while(1) e:>a;
        }
    }
}
```

```
channel e : {0..1}    -- CSP code

PE = e!1 -> PE

PB = e?b -> PB

PA = e?a -> PA

MAIN = (PE [|{e}|] PB) [|{|e|}|] PA
```

Code 3-1 Invalid conversion from CSP to XC

In XC "channels are lossless, which means that data output in one thread is guaranteed to be delivered for input by another thread. Each output in one thread must therefore be matched by an input in another, and the amount of data output must equal the amount input or else the program is invalid.[17]" Thus, if input and output could not be matched, the program will lead to deadlock. The example in Code 3-2 is a valid conversion.

```
int main(void) //XC code
{
    chan e;
    par
    {
        { //process PE
            while(1) e<:1;
        }

        {// process PA
            int a;
            while(1) e:>a;
        }
    }
}
```

```
channel e : {0..1}    -- CSP code

PE = e!1 -> PE

PA = e?a -> PA

MAIN = PE [|{|e|}|] PA
```

Code 3-2 Valid conversion of channels

The XC code is tested by monitoring the behaviour of the two threads with step-by-step execution mode in XDE. The CSP code is tested by monitoring the behaviour of

the two processes with ProBe which is mentioned in Section 2.1.2. The test in XC was with 50 steps execution while that in ProBe was also 50 steps. The behaviours of the code for both languages are the same.

As discussed above, we can derive *Conversion Rule One* and *Two.*

Conversion Rule One: in CSP, the channels could be converted into XC only when the channel has two channel-ends and input and output are matched.

Conversion Rule Two: the event synchronization could be converted into XC using channel only when no more than to processes synchronize on one event.

3.1.2 Barrier Synchronization

In machine readable CSP, events and communication channels are all declared as channels. The difference between them is the events do not carry any messages while the channels are used to pass messages and data. Thus events in CSP could used to implement synchronization for processes.

As the channel in XC is synchronous, it could be used to synchronize two processes. The example could be seen in section 2.2.2.1

Barrier synchronization commonly occurs in parallel system design, and it is used to block processes until after all the processes reach the barrier. The operation of a barrier is shown in Figure 3-1.

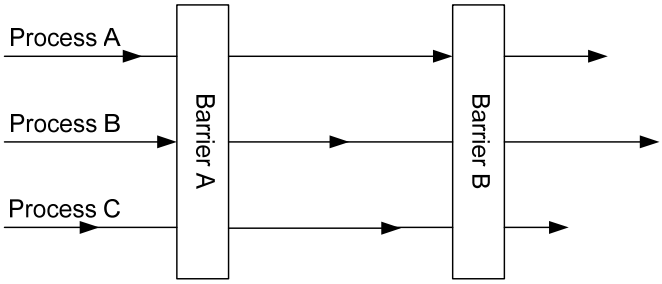


Figure 3-1 The operation of barrier synchronization

As shown in Figure 3-1, the arrows indicate the progress of the processes. Processes will be blocked by a barrier until after all processes reach the barrier. Thus it requires a mechanism to synchronize more than two processes on an event. One of the most efficient barrier synchronization implementation is to use *Hypercube Topology* [27], and it has been implemented in XC by Jamie Hanlon from XMOS Industry [28]. For 8-node barrier synchronization, it will require $8*[\log_2^8]$ = 24 channels. It's quite resource consuming, as each XCore has just 32 channels and 8 threads. If synchronize 8 threads with hCUBE barrier synchronization, the rest of channels may not sufficient to perform other tasks. At each barrier, it requires each node to

communicate with neighbour-nodes which is quite time consuming and processing effort consuming.

```
//XC code
int main(void)
{
 //Time Point A
  par
  {
        Process_A0();
        Process_B0();
        Process_C0();
  }
  //virtual barrier A
  //Time Point B
  par
  {
        Process_A1();
        Process_B1();
        Process_C1();
  }
  //virtual barrier B
  //Time Point C
}
```

```
--CSP code
channel a0,a1,b0,b1,c0,c1
channel barrierA,barrierB

PA0 = a0 -> barrierA -> PA1

PB0 = b0 -> barrierA -> PB1

PC0 = c0 -> barrierA -> PC1

PA1 = a1 -> barrierB -> STOP

PB1 = b1 -> barrierB -> STOP

PC1 = c1 -> barrierB -> STOP

MAIN = PA0 [|{|barrierA,barrierB|}|] PB0
```

Code 3-3 Barrier synchronization in XC and CSP

Due to the intrinsic nature of XCore, XC provided a way to perform barrier synchronization in hardware. The *par* statement is used to create concurrent threads in XC, and could be used to implement barrier synchronization within an XCore due to the nature of it. An example is shown in Code 3-3.

The *par* statement creates 3 threads which are running concurrently. Three processes are running in parallel using *fork-join parallelism* [17] at the beginning of the *par*, and the *par* would not be closed until all the three processes reach the end.

The synchronization is performed by hardware. Compared against the software implemented barrier synchronization, the hardware synchronization does not consume any channel resources and does not require software effort. This can potentially benefit the performance of XC program.

The behaviour of XC code has been checked by the timing analysis methods mentioned in *Section 2.2.4.* The functions of the processes are implemented by simple *while loops* with different cycles of iterations. Firstly, use *XTA* to identify the time taken by each process, and then use *dynamic timing analysis* to measure the time taken by the two *par* blocks via *Time Points* commented in the XC code.

The result is that time taken by both *par* blocks are the same as the time taken by the processes which have the most cycles of iterations. To some extent, this means the barrier synchronization could be implemented by *par* blocks.

From the discussion above we can derive *Conversion Rule Three.*

Conversion Rule Three: In CSP, more than two processes synchronize on one event could be implemented by *par* statement in XC within an XCore.

3.1.3 *Choices* in CSP and XC

The *choice* in CSP, which is also known as ALT [11], is a useful feature with which processes could wait for multiple signals and engage only one of them. Like the *channel*, the definition in CSP is not the same as that in XC. In CSP, "The ALT construct consists of guards that in turn each guard a process. There are several types of guards: input guard, output guard, skip guard and time-out guard, which can either be conditional or unconditional.[11]"

For XC, it only accepts conditional or unconditional input guards, and other kinds of guards are not supported so far. The guards in XC are organized by *select* statement, and have the similar behaviour as that in CSP. But the difference of *select* statement in XC and *external choice* in CSP is that the *case* statement in a *select* could not do any output operation [17]. Thus the *external choice* in CSP could not directly be converted into XC.

```
//XC code
select
{
   case sig_input[0]:>a :
            FUNCTION0(a);
            break;
   case sig_input[1]:>b :
            FUNCTION1(b);
            break;
   case sig_input[2]:>c :
            FUNCTION2(c);
            break;
}
```

```
--CSP code

PA = sig_input0?a -> FUNCTION0(a)
   [] sig_input1?b -> FUNCTION1(b)
   [] sig_input2?c -> FUNCTION2(c)
```

Code 3-4 Guards in CSP and XC with inputs

The example shown in Code 3-4 are valid conversions proved by XC compiler. The equivalence has been check with a test bench in XC and ProBe respectively. The functions in each guard of the *select construction* are end up with infinite *while loops (e.g. while(1);)*, and the FUNCTIONs in CSP code is end up with *STOP.* Thus in each test the *choice* only executes one time. The test stimulus is shown in Code 3-5.

```
par{
 sig_input[0]<:1;
 sig_input[1]<:1;
 sig_input[2]<:1;
}
```

Code 3-5 Test stimulus for *select construction*

With the results from *XMOS Simulator*, in each test, it always engage in the case of sig_input[0]. From the test results of *XMOS Simulator,* the priority of the *case* statements is fixed to the first *case* initially. Thus, this equivalence model should be used with caution. **Avoid using it with more than one signals arrive at the same time for the *select construction*.**

If the *guard* in CSP is an event instead of an input from a channel, the model is different as the *guard* in XC could only accept input signals. In this case, the guard of the event in CSP could be implemented in XC using timers. The example is shown in Code 3-6.

```
//XC code
select
{
   case sig_input:>a :
      FUNCTION0(a);
      break;
   case tmr when timerafter(time):>void :
      sig_output<:1;
      break;
}
```

```
--CSP code

PA = sig_input?a -> FUNCTION0(a)
      [] t -> sig_out!1
```

Code 3-6 Guards in XC and CSP with output operations

The test for Code 3-6 is similar to that of Code 3-4. So far, as discussed above, *Conversion Rule Four* could be derived.

Conversion Rule Four: the case statement of guards in XC must not contain any output operations. It could be any combination of conditional or unconditional inputs (e.g. Timers, channels and ports).

3.2 Specification Refinement

The original *Servo Clock* is a sequential design, shown in Figure 3-2.

It receives the current coordinates and relevant data from the peripheral device and *User Interface* respectively, and performs the algorithms before sending back the control data to the peripheral device. The reception and delivery of data could be in any order in design phase. The flow chart shown in Figure 3-2 is only a specific case.

The original *Servo Clock* is running under the control of a clock at the frequency of 1 KHz, which means a whole running cycle would be finished within 1ms. Obviously, higher is the frequency, the better. Because if the *Servo Clock* frequency increases, the sampling rates also increases this would benefit the accuracy of the control system according to *Shannon Theorem*[29].

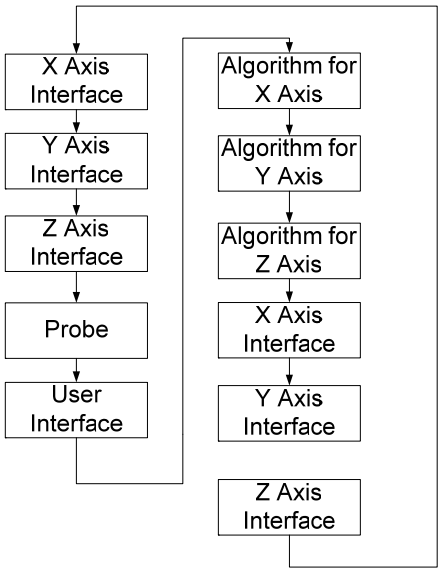


Figure 3-2 Flow chart of original *Servo Clock*

The *Servo Clock* specification described in CSP was derived from the original *Servo Clock.* The specification is shown Code 3-7. Unlike the specification in [25], which required the specification to be definite in order to ensure the components from different vendors were compatible. The *Servo Clock* specification is more flexible than that in [25], as the abstraction-level CSP specification just constrains the behaviour of the inputs and outputs of the *Servo Clock* and it does not care about the details of how it would be refined in implementation level. Thus, with this specification, the *Servo Clock* could be refined in various ways. As CSP is not good at calculating, it is only used to design the communication protocol for this project for the simplicity.

Considering the resource limitation of XC-1 Development Kit which has 4 XCore, 32 threads (8 threads per XCore) and 128 channels (32 channels per XCore), the resource consumption should be taken into account when refining the specification.

The specification consists of four modules, *SPEC_AXES, SPEC_USR_DMDS, SPEC_DATA and SPEC_PROBE.* Actually, each of the modules represents the interface of the peripheral device. The *SPEC_DEVICES* and *SPEC_DEMANDS* organises these modules and constrains the order of events.

Additional materials to *Servo Clock* specification:

1. The *usr_dmd* is a block of data from user, and it contains the target coordinates for x, y, z respectively. The x, y, z will be used to generate a profile which is an array of position information for *Axes.*
2. The *data_req* is data requirement signal from user, and *data_trn* is the required data package to user.
3. The *probe_rec_defns* is received from *Probe*, and it indicates whether the device contacts an object at current position.
4. The *axis_rec_count* is encoded data '*Count*' from each *Axis,* and *axis_trn_vel* is *velocity demand* to peripheral device.
5. In *SPEC_DEMANDS,* the requirement signal of *data_req* and *usr_dmd* may not arrive in each running cycle. So the servo clock should not be blocked by these two signals. If the there is no user demand for the target position, the *Servo Clock* should be able to hear the state of the peripheral device and maintain the device at current coordinates.
6. The *Probe* and *Axes* are synchronized on event '*start*' at the beginning of each cycle.

```
dummy(val) = val
spec_data_dummy(a)=0
SPEC_AXIS(inst) = start -> axis_rec_count.inst?c_val -> sync ->
                        axis_trn_vel.inst!dummy(c_val) -> SPEC_AXIS(inst)

SPEC_AXES = [| SpecAllAlpha |] inst:Axes @ SPEC_AXIS(inst)

SPEC_PROBE = start -> probe_rec_defns?a.b.c -> sync -> SPEC_PROBE

SPEC_USR_DMDS = usr_dmd?x.y.z -> SPEC_PROC_USR_DMDS(CALC_TIME)
                              [] sync -> SPEC_USR_DMDS

SPEC_PROC_USR_DMDS(0) = SPEC_USR_DMDS

SPEC_PROC_USR_DMDS(n) = sync -> SPEC_PROC_USR_DMDS(n-1)

SPEC_DATA = data_req?req -> sync -> data_trn!spec_data_dummy(req) -> SPEC_DATA
                    [] sync -> SPEC_DATA

SPEC_DEVICES = SPEC_AXES [| SpecAllAlpha |] SPEC_PROBE

SPEC_DEMANDS = SPEC_USR_DMDS [| SpecSyncAlpha |] SPEC_DATA

SERVO_CLOCK_TMP = SPEC_DEVICES [| {|sync|}|] SPEC_DEMANDS

SERVO_CLOCK_SPEC = SERVO_CLOCK_TMP \ SpecSyncAlpha
```

Code 3-7 *Servo Clock* specification

The topology of the *Servo Clock* is shown in Figure 2-14.

3.2.1 Core Implementation

Before refining the specification, in order to cooperate with the modules in the specification, the control flow of the *Servo Clock* should be designed. The control flow is implemented by a module called '*Core*', of which the flow chart is shown in Figure 3-4.

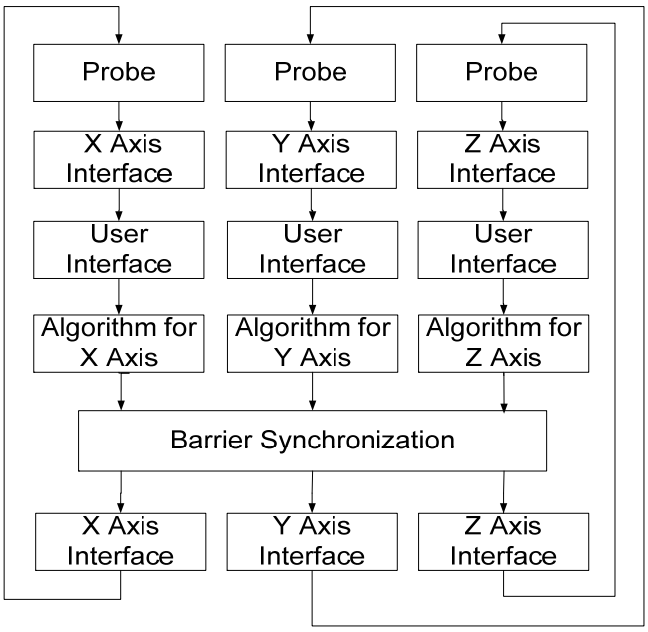


Figure 3-3 Flow chart for the *Core*

In the specification, all the modules synchronize on the event *'sync'* which means all the modules will be blocked at *'sync'* and will be released only after all the modules synchronize on it. Thus, it requires the '*Core*' with a mechanism to block the modules. In Figure 3-3, the '*Core*' collects the relevant data from user and peripherals and performs the algorithms individually on each axis before the barrier synchronization. Processes with *barrier synchronization* will be blocked and will not be released until after all the processes with barrier synchronization reach the barrier.

The simplified CSP code for '*Core*' is shown in Code 3-8. The channel *to_core_probe_defns*, *to_core_posi*, *to_core_data* represent the *Probe, Axis Interface* and a part of user interface in Figure 3-3 respectively. Considering *Conversion Rule Four,* in order to meet the requirement in the fifth bullet point of the additional materials of the *Servo Clock*, the 'Core' is designed with an *external choice* of *core_sync* and *core_sync2* which could release the *Core* when there was no *usr_dmd* or *data_req* from user. The decision of which branch would be selected

depends on which signal arrives first. The branch of *core*_sync2 means the *profile*[1] is ready while that of *core_sync* means the *profile* is empty. Thus, if there is no user demand and the profile is empty, the *'Core'* still keeps running and hears the state of the peripheral device by choosing *core_sync* branch.

```
CORE_AXIS(inst) = to_core_probe_defns.inst?defns -> to_core_posi.inst?curr_posi ->
                    core_to_data.inst!curr_posi ->
          (
                core_sync2 -> to_core_prof_vel.inst?prof_vel ->
                to_core_prof_acc.inst?prof_acc  ->  to_core_prof_posi.inst?prof_posi ->
                core_sync ->
                core_trn_vel.inst!calc_velo () -> CORE_AXIS(inst)
           []
                core_sync ->
                core_trn_vel.inst!calc_velo() -> CORE_AXIS(inst)
          )
CORE_AXES = [| {| core_sync, core_sync2|} |]    inst: Axes @CORE_AXIS(inst)

CORE_ALGS = CORE_AXES
```

Code 3-8 CSP implementation for the *Core*

The *CORE_AXIS ()* for each axis is synchronized on event *core_sync* and *core_sync2*, and it means in one cycle there are at most two barrier synchronization.

The resource requirements of this module are 23 channels and 4 threads.

3.2.2 Modules Implementation

"An implementation is a valid refinement of the global specification if hiding all the additional events present in the implementation gives a process which exhibits only (a subset of the) behaviours allowed by the specification" [25]. **The refinements of the modules in the specification apply this principle.**

In the specification, the data from the peripheral devices and user could be received in any order, so it requires each module to handle the potential data at any time before the event '*sync*'. Hence, each module should have a buffer to save the data when the *Core* could not able to handle it immediately. Since '*sync'* in the specification is hidden, there is no requirement for it in the implementation and it could be implemented in an alternative way.

Refinement for SPEC_AXES:

For the implementation of SPEC_AXES, it consists of three *IMP_AXIS ()* and *GET_NEW_POSI()*. The Code 3-9 shows the simplified CSP code for the

[1] The profile is generated by IMP_USR_DMDS, and it is an array of control data used to direct the peripheral device moving in a 3-D space.

implementation of *SPEC_AXES*. The function of *get_posn()* is a dummy function for position calculation. The process *GET_NEW_POSI()* is used to import the result from *get_posn(), as CSP* is not appropriate for shared data processing. The same method would be used to design the rest of modules. Considering the Conversion *Rule Four,* in *GET_NEW_POSI (),* there is an *external choice* for *core_trn_vel* and *prof_ask_posi* are designed as input signals and for which branch will be chosen depends on which signal comes firstly.

```
IMP_AXIS(inst, pre_posi, pre_count) = start -> axis_rec_count.inst?curr_count ->
                                        GET_NEW_POSI (inst, get_posn(),curr_count)

GET_NEW_POSI (inst, curr_posi, curr_count) = to_core_posi.inst!curr_posi ->
                    (
                        core_trn_vel.inst?curr_vel -> axis_trn_vel.inst!dummy(curr_vel) ->
                        IMP_AXIS(inst,curr_posi,curr_count)
                     []
                        prof_ask_posi.inst?abc -> to_prof_posi.inst!curr_posi ->
                        core_trn_vel.inst?curr_vel ->   axis_trn_vel.inst!dummy(curr_vel) ->
                       IMP_AXIS(inst,curr_posi,curr_count)
                    )
IMP_AXES = [| {|start|} |] inst: Axes @ IMP_AXIS(inst,0,0)
```

Code 3-9 Refinement of IMP_AXES for SPEC_AXES

The IMP_AXES organises three of IMP_AXIS() which are running concurrently using *replicated sharing parallel*[14]. In the branch of *core_trn_vel*, each IMP_AXIS() receives the *Count* from peripheral device, and waits for the velocity demand from the *Core* after sending the current position to the *Core.* In the branch of *prof_ask_posi*, the procedure is similar to the branch of *core_trn_vel*: it sends the information to profiler before sending it to the *Core.* The channel *to_core_posi* acts as the buffer mentioned at the beginning of this section.

The implementation does not have an event like 'sync' which is in the specification. As the *Core* already has the barrier synchronization which is on event '*core_sync*', when the *IMP_AXES* interacts with *CORE_AXES*, the three *IMP_AXIS ()* will be virtually synchronized on *core_sync* due to the effects of the synchronized channels *core_trn_vel* and *axis_trn_vel*.

The resource requirements for *IMP_AXES* are 19 channels and 4 threads.

Refinement for SPEC_PROBE:

```
IMP_PROBE = start -> probe_rec_defns?a.b.c -> DISTR_PROBE(<a,b,c>) ; IMP_PROBE

DISTR_PROBE (<>) = SKIP

DISTR_PROBE (<x>^xs) = (to_core_probe_defns.(#xs)!x -> SKIP ||| DISTR_PROBE(xs))
```

Code 3-10 Refinement of IMP_PROBE for SPEC_PROBE

For the implementation of *SPEC_PROBE*, the simplified CSP code is shown in Code 3-10. The *IMP_PROBE* and *SPEC_AXES* are synchronized on event *start*, which actually is barrier synchronization. The *IMP_PROBE* distributes the data of deflections using *interleaving parallel* [14], after receiving the data through the channel *probe_rec_defns* from the peripheral device. Like *IMP_AXES*, in *IMP_*PROBE there is also no such event like '*sync*'. The event *start* on which *IMP_PROBE* synchronizes with IMP_AXES will guarantee that the IMP_PROBE could run only once in each cycle. The channel *to_core_probe* synchronized with *Core* will make sure the event *probe_rec_defns* could only be triggered before *core_sync.* Principally, *IMP_PROBE* is a valid refinement for *SPEC_PROBE*.

The resource requirements for *IMP_PROBE* are 4 channels and 4 threads,

Refinement for SPEC_USR_DMDS:

For the refinement of *SPEC_USR_DMDS*, the flow chart is shown in Figure 3-4.

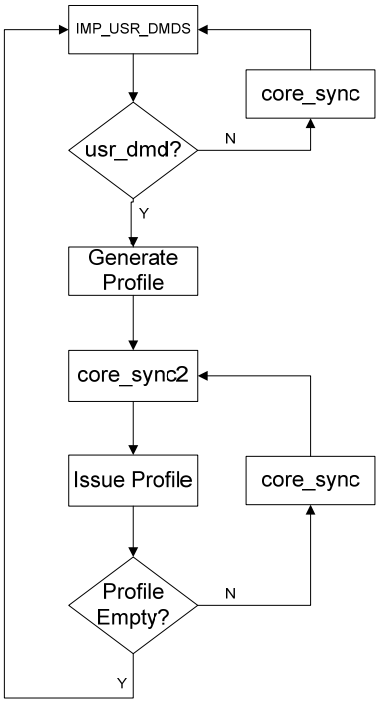


Figure 3-4 Flow chart of IMP_USR_DMDS

The refinement consists of the following modules:

1. *DMDS_SPLIT* is used to distribute the position demands from user to 3 axes.
2. *DMDS_COLLECT_AXES* is used to collect current coordinates from *IMP_AXES* module and send the data to PROF_GEN (). It sends the requirement signal *prof_ask_posi* to *IMP_AXE,* and saves the returned data through *to_prof_posi.* Finally it sends profile to *DISTR ()* after the profile was generated by prof_dummy ().
3. *PROF_GEN()* is used to generate the profile with the data collected from user demands and *IMP_AXES*
4. DISTR () is used to distribute the profile to the *Core* of three axes. The *core_sync* and *core_sync2* are toggled with the *Core*. The module sends the profile items of velocity, acceleration, demanded position to the *Core* sequentially.
5. IMP_USR_DMDS is the master module for the modules above.

The simplified CSP code is shown in Code 3-11.

```
prof_dummy(a,b) = PROF_GEN(CALC_TIME,a,<>)
PROF_GEN(n,a,as) = if (n!=0) then PROF_GEN(n-1, a,<a,a,a>^as)
                          else as

IMP_USR_DMDS = ( usr_dmd?x.y.z -> DMDS_SPLIT(x,y,z) ; IMP_USR_DMDS)
                          [] core_sync -> IMP_USR_DMDS

DMDS_SPLIT(x,y,z) = (       DMDS_COLLECT_AXES(x,0)
                            [|{|core_sync,core_sync2|}|]
                            DMDS_COLLECT_AXES(y,1)
                          )
                            [|{|core_sync,core_sync2|}|]
                            DMDS_COLLECT_AXES(z,2)

DMDS_COLLECT_AXES(dmds,inst) = prof_ask_posi.inst!1 -> to_prof_posi.inst?curr_posi ->
                                        DISTR(prof_dummy(dmds,curr_posi),inst)

DISTR(<>,inst) = SKIP

DISTR(<vel,acc,posi>^as, inst) = core_sync2 -> to_core_prof_vel.inst!vel ->
                                    to_core_prof_acc.inst!acc -> to_core_prof_posi.inst!posi ->
                                    core_sync -> DISTR(as,inst)
```

Code 3-11 Refinement of IMP_USR_DMDS for SPEC_USR_DMDS

As the user demands may either come or not in a running cycle, in order to prevent the potential deadlock caused by *usr_dmd*, at the beginning of control flow there is a *external choice* for *usr_dmd* and *core_sync.* If the *usr_dmd* could not arrive within a

specific time, the *core_sync* would allow the core to continue running without blocking the *Core* by letting it wait for the *usr_dmd.*

With the specification *SPEC_USR_DMDS*, the *usr_dmd* will not be released until after several cycles indicated by *CALC_TIME* if if *usr_dmd* arrives in current cycle. In order to get a valid refinement for the specification, in the implementation the distribution of the profile is toggled with *core_sync* and *core_sync2*.

The *core_sync2* is used to select the branch in the *Core*. If the profile is ready, the *core_sync2* signal will be arrived prior to *core_sync* as shown in *DISTR ()*. If the profile is empty and there is no usr_dmd, the *core_sync2* will not be activated. Therefore, with *core*_sync and core_sync2, the *Core* would know whether it should be ready for the profile or not. The modules in *IMP_USR_DMDS* are synchronised on *core_sync2* and *core_sync* which are a part of barrier synchronization in the *Core.*

By hiding the additional events, the *IMP_USR_DMDS* is principally a valid refinement for *SPEC_USR_DMDS.*

The resource requirements for *IMP_USR_DMDS* are 18 channels and 4 threads.

Refinement for SPEC_DATA:

For the refinement of *SPEC_DATA*, the control flow of IMP_DATA is shown in Figure 3-5

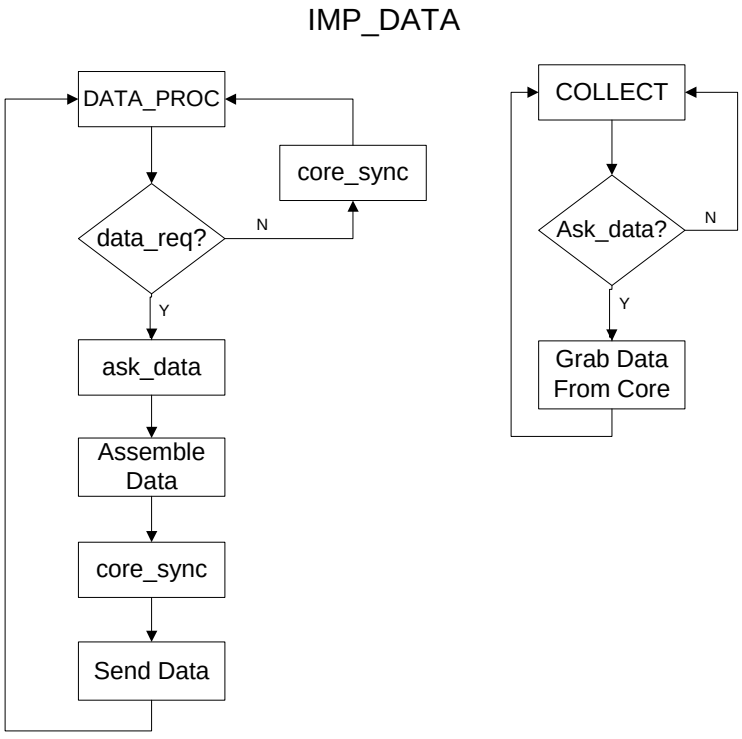


Figure 3-5 Flow chart for IMP_DATA

The refinement consists of the following modules:

1. *COLLECT* is the master process for indexed processes *DATA_GRAB()*
2. *DATA_GRAB ()* is used collect data from the *Core.* It receives the data from the *Core* in every cycle through the channel *core_to_data*. If there is a data requirement, it sends the data to ASM_DATA () through the channel *data_self_grab*. The branch of *non_data* does nothing, and it just offers an alternative path in case of no data requirement, in order to prevent the potential deadlock caused by the data requirement.
3. *PROC_DATA* is the master process for process *ASM_DATA ()* and *ASM_DATA_NON_DATA()*. When the data requirement signal arrives, it follows the branch of ASM_DATA (), otherwise follows the branch of ASM_DATA_NON_DATA ().
4. *ASM_DATA()* collects data from DATA_GRAB() through the channel *data_self_grab* and assemble it in an array. Finally sends the data back to user via channel *data_trn.*
5. *ASM_DATA_NON_DATA()* does not do substantive work. It is used to provide alternative path in order to prevent deadlock.
6. *IMP_DATA* is the master process for *PROC_DATA* and *COLLECT*, and organises them using *sharing parallel* [14].

The simplified CSP code is shown in Code 3-12

```
data(dats)= 0

COLLECT = [|{|data_self_sync|}|] inst:Axes @ DATA_GRAB(inst)

DATA_GRAB(inst) = core_to_data.inst?dat -> data_self_sync ->
                  (
                    ask_data.inst?abc -> data_self_grab.inst!dat ->  DATA_GRAB(inst)
                  []
                    non_data.inst?abc  -> DATA_GRAB(inst)
                  )

PROC_DATA = data_req?req -> ASM_DATA(0,<>)
            [] ASM_DATA_NON_DATA(0,<>)

ASM_DATA (n,xs) = if (n==NUM_AXES) then core_sync -> data_trn!data() -> PROC_DATA
                  else
                     ask_data.n!0 -> data_self_grab.n?dat -> ASM_DATA(n+1,<(n,dat)>^xs)

ASM_DATA_NON_DATA (n,xs) =  if (n==NUM_AXES) then core_sync -> PROC_DATA
                            else
                               non_data.n!0 -> ASM_DATA_NON_DATA(n+1,<(0)>^xs)

IMP_DATA = PROC_DATA [|imp_data_sync_alpha|] COLLECT
```

Code 3-12 Refinement of IMP_DATA for SPEC_DATA

By hiding the additional events, the *IMP_DATA* is principally a valid refinement for *SPEC_DATA.*

The resource requirements for *IMP_DATA* are 16 channels and 7 threads.

Until now, all the components of the *Servo Clock* have been refined. The last stage is to integrate those components to form a system. the simplified CSP code is shown in Code 3-13

```
IMP_DEVICES = IMP_AXES [| {|start|} |] IMP_PROBE

SRV_CLK1 = (CORE_ALGS [| sync_alphabet_1 |] IMP_USR_DMDS)

SRV_CLK2 = SRV_CLK1 [|sync_alphabet_2|] IMP_DEVICES

SRV_CLK3 = SRV_CLK2 [|sync_alphabet_3|] IMP_DATA

SERVO_CLOCK = SRV_CLK3
```

Code 3-13 Refinement of SERVO_CLOCK for SPEC_SERVO_CLOCK

SERVO_CLOCK is the master process for all the components, and it interconnects those modules with *sharing parallel* [14].

The total amount of resource requirements are 24 threads and 80 channels which could be possibly accommodated by XC-1 Development Kit.

3.2.3 Refinement Validation

Until now, we have the full refinement for the *Servo Clock*. But we don't know whether what we have refined is exactly what we wanted. The formal evidence will be given out in this section.

For the simple refinement like *IMP_PROBE*, it is easy to analyse whether it refines the specification *SPEC_PROBE* by hand. But for the complex refinement, such as *IMP_AXES, IMP_DATA* and *IMP_USR_DMDS*, it is a heavy load of work to analyze them by hand and mistakes may be easily brought to analysis. Additionally, to some extent, to analyse the determinism and freedom of deadlock and livelock is not an easy job, as there are various interconnections among modules.

As introduced in section 2.1.1, FDR2 can formally prove whether the implementations refine the specifications with the *failures-divergence* and *traces* model, and check determinism and freedom of deadlock and livelock as well. Thus FDR2 can give the formal evidence of the validity and the feasibility of the refinement. With the validation principle and procedure introduced in [30], the refined components of *Servo Clock* described CSP have been verified by FDR2. The

verification was divided into two steps. The first step was to verify each module individually with FDR2 for determinism, freedom of deadlock and livelock. After this, check the refinement against the specification with *failures-divergence* and *traces* models in FDR2. The second step was to integrate all the components to form the full *Servo Clock* and verify it with FDR2.

As the all the modules were attached to the *Core* for data exchange and synchronization, the event-order of the modules was constrained by the *Core.* Thus, the determinism, freedom of deadlock and livelock of the *Core* should be checked first.

The scripts[2] for checking determinism, deadlock and livelock freedom are given below,

```
assert CORE_ALGS :[ deterministic [FD] ]
assert CORE_ALGS :[ deadlock free [FD] ]
assert CORE_ALGS :[ divergence free ]
```

The *CORE_ALGS* is the master process (or entry) of the *Core* and *divergence free* means livelock free. The results from FDR2 are shown in Figure 3-6.

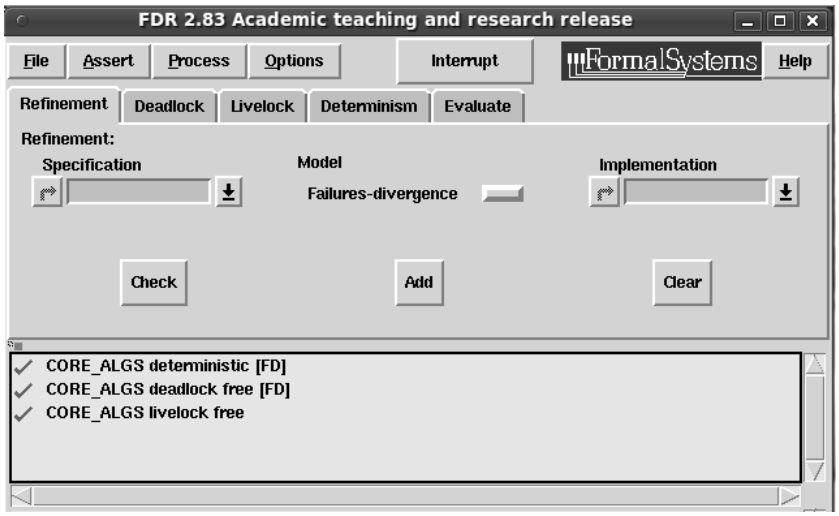


Figure 3-6 FDR2 checking results for the Core

The results shows the *Core* is deterministic and free of deadlock and livelock which means the control flow of the *Core* is valid. But we still don't know whether the control flow acts as what we intended. In this case, as introduced in *Section 2.1.2* ProBe could be used to check the traces of the *Core.* It was an informal check, so we practiced 5 cycles' check with *CORE_ALGS.* The results show the traces of *CORE_ALGS* were the same as the flow chart shown in Figure 3-3.

[2] The assertion scripts are used to generate checking-list for FDR2. The checking-list could also be generated manually from FDR2 user interface.

As the *Core* itself does not have a specification, it is not able to check the refinement validity.

For each module, the order of some of the events is constrained by the *Core.* Each module should be analyzed along with the *Core.* For the verification on *IMP_USR_DMDS*, firstly it should be integrated with the *Core.* The code for integration is shown below.

CHK_USR_DMDS = CORE_ALGS [| sync_alpha |] IMP_USR_DMDS

CHK_USR_DMDS is a new model for the verification on *IMP_USR_DMDS.* It only integrates the *Core (represented by CORE_ALGS)* and *IMP_USR_DMDS.* The *sync_alpha* is the alphabet for synchronization between *CORE_ALGS* and *IMP_USR_DMDS.* The check of determinism, deadlock and livelock freedom is the same as that of *CORE_ALGS,* and the script is shown below.

```
assert CHK_USR_DMDS :[ deterministic [FD] ]
assert CHK_USR_DMDS :[ deadlock free [FD] ]
assert CHK_USR_DMDS :[ divergence free ]
```

The assertions above were passed in FDR2 which means *IMP_USR_DMDS* can work along with the *Core* without deadlock, livelock and non-determinism.

Until now, there is no evidence for the validity of the refinement. As mentioned in *Section 2.1* the FDR2 provides two models (*failures-divergence model* and *traces* model) to check the validity of the refinement. The *traces* model (denoted as [T= [3]) validates the safety of the refinement that make sure the refinement does not do anything beyond the specification. The *failures-divergence* model (denoted as [FD=) validates the liveness of the refinement that make sure the refinement could do things required by the specification. We can say a refinement is valid only if the checks with these two models are passed, as mentioned in *Section 2.1.1.3.* The script for the assertion with these two models is shown below.

```
assert SPEC_USR_DMDS\ {|sync|}      [T=CHK_USR_DMDS \  hidden_alpha
assert SPEC_USR_DMDS\ {|sync|}      [FD=CHK_USR_DMDS \ hidden_alpha
```

The event '*sync*' is hidden in both assertions. That is because the *sync* is only used to constrain the order of events in specifications and should not be seen from the environment. Otherwise, the specification would require the refinement to perform 'sync' which does not exists in the refinement. The '*hidden_alpha*' is the sub-alphabet of *CHK_USR_DMDS* which containing the events and channels that in *CHK_USR_DMDS* but not in the alphabet of *SPEC_USR_DMDS.* According to the

[3] The syntax for FDR2 could be found in Appendix A

principle mentioned at the beginning of *Section 3.2.2*, the additional events represented by *hidden_alpha* should be hidden. The checking results of *CHK_USR_DMDS* from FDR2 are shown in Figure 3-8.

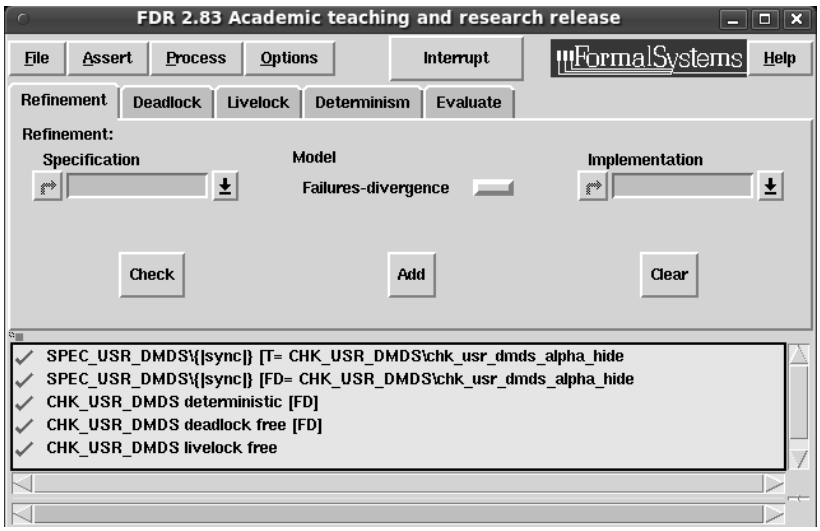


Figure 3-7 The checking results of *CHK_USR_DMDS*

The result shows the integrated module is deterministic, and free of deadlock and livelock. It also shows the integrated module is a valid refinement according to the discussion in *Section 2.1.1.3.* The verification on the rest of the modules is similar to the procedure above. Due to space limit, the details could be found in the CSP source code.

The correctness of each module does not necessarily mean the system formed by these modules is correct. The potential problems could be introduced by the interaction among different modules. Thus, the verification of on the whole system formed by these modules is necessary. The refinement of *Servo Clock* is an integration of *CORE_ALGS, IMP_USR_DMDS, IMP_DATA, IMP_PROBE* and *IMP_AXES.* The integration is shown in Code 3-13. The verification procedure is similar to that of IMP_USR_DMDS, and the result is shown in Figure 3-8 on the next page.

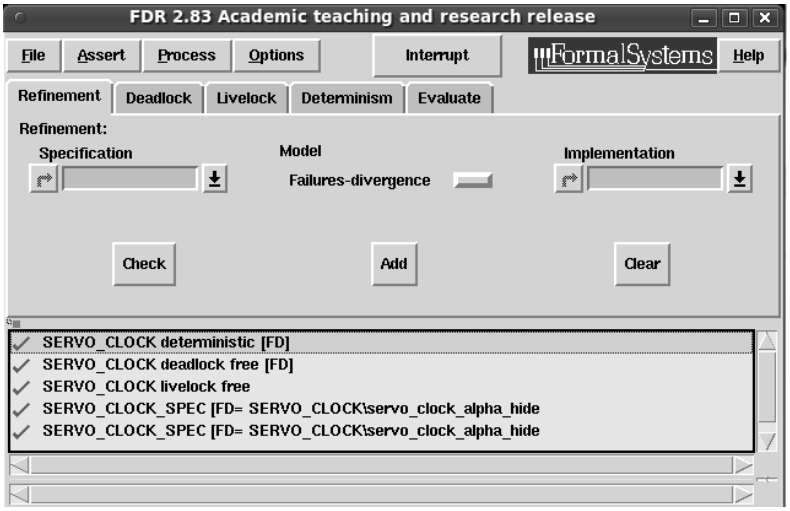


Figure 3-8 The checking result of *SERVO_CLOCK*

All the components and the *Servo Clock* implementation are deterministic and free of deadlock and livelock. The FDR2 result of refinement validation is shown in Figure 3-9.

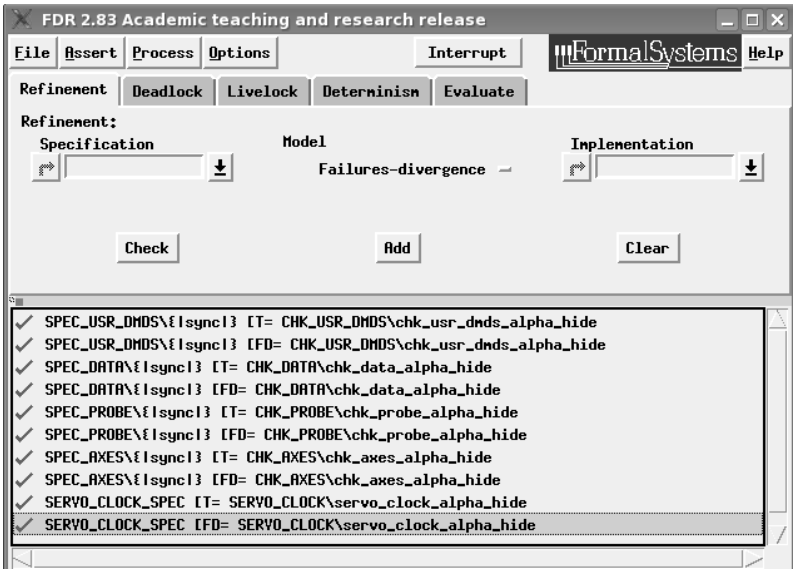


Figure 3-9 Refinement Validation for all modules

With the discussion above, the refinement is valid, which means the communication protocol for the whole system is correct and it acts as the specification required. The result given by FDR2 is formal evidence and it is the guideline for converting work. If the XC application converted from CSP could not work properly, the problem should be in the XC part, not in the communication protocol. The overall diagram of the *Servo Clock* could be found in Appendix C.

3.3 Implementation in XC

In this section, the *Servo Clock* would be implemented in XC and run on XC-1 Development Kit. As mentioned in *Section 3.2* CSP is only used to design the communication protocol for this project, in this section the detailed algorithm will take the place of dummy functions of the CSP implementation.

The refinement of the specification in CSP would be converted into XC as the framework of the *Servo Clock* and the algorithm is a development of the framework to achieve the real functionality of the Servo *Clock.*

3.3.1 Converting CSP into XC

Conversion of CSP is another major task of this project. Before converting, the system described in CSP should apply the *Conversion Rules* in *Section 3.1*. If not, the CSP script should be refined into lower-level until the *Conversion Rules* could be applied.

The refinement in *Section 3.2* was optimised by considering *Conversion Rules* and *XC-1* resource limitation, and it is appropriate for converting.

Before we start, we need to allocate the resources on XC-1 Development Kit for each module. The resource requirement of each module is shown in Table 3-1

Table 3-1 Resource requirement of each module

Module Name	No. of Channels	No. of Threads
CORE_ALG	23	4
IMP_AXES	19	4
IMP_PROBE	4	4
IMP_DATA	16	7
IMP_USR_DMDS	18	4
SERVO_CLOCK	80	23+1

Each XCore could accommodate up to 8 threads, and provide up to 32 channels. With the information from Table 3-1, the location of each module in XC-1 Development kit is shown in Table 3-2

Table 3-2 Resource allocation

Location	Modules
Core [0]	IMP_AXES
Core [1]	CORE_ALG
Core [2]	IMP_USR_DMDS and IMP_PROBE
Core [3]	IMP_DATA

As we can see from Code 3-8 The CSP implementation of *CORE_ALG* has 4 processes, one master process *CORE_AXES* and 3 children processes *CORE_AXIS().* The processes were implemented as functions in XC.

In *CORE_AXIS (),* the events and channels are running sequentially in each axis and there are three *CORE_AXIS* running in parallel. Because there are two places need barrier synchronization and CORE_ALG could be accommodated in an XCore, the implementation of *CORE_AXIS()* in XC could apply *Conversion Rule Three.*

```
//CORE_AXES
par(int i=0;i<NUM_AXES;i++)
  {
     // A part of CORE_AXIS(inst)
     {to_core_posi[i]:>curr_posi[i];
       to_core_probe_defns[i]:>defns[i];
       core_to_data[i]<:curr_posi[i];}
   }
// Sync Point A
```

```
CORE_AXIS(inst)  =
to_core_probe_defns.inst?defns ->
to_core_posi.inst?curr_posi ->
core_to_data.inst!curr_posi ->  …
```

Code 3-14 Conversion of the Core

Before the *external choice* of *core_sync2* and *core_sync*, the communication channels to_*core_probe_defns, to_core_posi and core_to_data* could be converted into XC by applying *Conversion Rule One* and *Two*. The conversion is shown in Code 3-14

Actually the *external choice* of core_sync2 and core_sync are barriers, so the choice is made on barriers. As the processes in the module of IMP_USR_DMDS which was allocated on a different XCore also synchronize on the same barriers of core_sync and core_sync2, the *Conversion Rule Three* could not be applied. The reason is the condition of "*with*in an XCore" in *Conversion Rule Three* could not be met. In this case, we could implement the *distributed barrier synchronization* using a hybrid approach denoted as ***hybrid barrier synchronization***.

```
CORE_AXIS(inst) = to_core_probe_defns.inst?defns -> to_core_posi.inst?curr_posi ->
                    core_to_data.inst!curr_posi ->
          (
                    core_sync2 -> core_data_sync ->   -- comments: two 2-node barriers.
                   to_core_prof_vel.inst?prof_vel ->
                    to_core_prof_acc.inst?prof_acc ->  to_core_prof_posi.inst?prof_posi ->
                    core_sync ->
                    core_trn_vel.inst!calc_velo () -> CORE_AXIS(inst)
           []
                    core_sync ->  core_data_sync ->    -- comments: two 2-node barriers.
                    core_trn_vel.inst!calc_velo() -> CORE_AXIS(inst)
           )
```

Code 3-15 Modified CORE_AXIS()

The *hybrid barrier synchronization* is implemented by hardware and software implemented barrier synchronization. The "*Sync Point A*" after *par* statement could be treated as a node of software implemented barrier synchronization. Thus, synchronise on "*Sync Point A*" is equivalent to synchronise on the barrier. The behaviour of *hybrid barrier synchronization* has been check by experiments in XDE with step-by-step execution. The processes could not continue executing until after they all synchronize at "*Sync Point A*".

As there are three modules (*CORE_ALG, IMP_DATA* and *IMP_USR_DMDS*) synchronize on "*Sync Point A*", for simplicity, we didn't use 3 nodes software implemented barrier synchronization. Instead, we split the 3-node barrier (*core_sync2 and core_sync* in Code 3-8) into two 2-node barriers (see the comments in Code 3-15 in previous page). But this will lead to some issues which will be discussed in Chapter 4.

```
// Sync Point A
select{
    case core_sync2:>abc:
            sel=1;
            break;
    case core_sync:>abc:
          core_data_sync<:1;
            sel=0;
            break;
          }
           // the branches for choices
           if (sel)
           {
               par(int i=0;i<NUM_AXES;i++)
               {
                  { to_core_prof_vel[i]:>prof_vel[i];
                    to_core_prof_acc[i]:>prof_acc[i];
                   to_core_prof_posi[i]:>prof_posi[i]; }
               }
               core_dmds_sync:>abc;
               core_data_sync<:1;
               par(int i=0;i<NUM_AXES;i++)
               { core_trn_vel[i]<:calc_velo(curr_posi[i], prof_vel[i], prof_acc[i], prof_posi[i],
                                                    defns[i]);
               }
           }
           else
           {
             par(int i=0;i<NUM_AXES;i++)
            {core_trn_vel[i] <: calc_velo(curr_posi[i], 0, 0, curr_posi[i], 0);}
           }
   }
}
```

Code 3-16 Conversion of the Core

The conversion could be made by considering *Conversion Rule Two* and *Four* which is shown in Code 3-16 (in previous page).

Similar to *CORE_ALG*, *IMP_USR_DMDS* has 4 processes, one master processes and three children processes. The conversion of IMP_USR_DMDS is similar to CORE_ALG, and the CSP script could be easily converted into XC by applying the *Conversion Rules* except for the *external choice* of usr_dmd and core_sync which is shown in Code 3-17.

```
IMP_USR_DMDS = (usr_dmd?x.y.z -> DMDS_SPLIT(x,y,z) ; IMP_USR_DMDS)
                        [] core_sync -> IMP_USR_DMDS
```

Code 3-17 Conversion of IMP_USR_DMDS

As the CORE_ALG also synchronizes on *core_sync* and the channel-end of *core_sync* in CORE_ALG has been configured as an input, the other channel-end of *core_sync* must be configured as an output. But considering the *Conversion Rule Four*, the *guards* in a *select* statement must not be output. It is also against the *Conversion Rule One*, if configure the *core_sync* in IMP_USR_DMDS as an input.

Considering *Conversion Rule Four,* the *case* statement of a *guard* could be any input event. Hence, we can use other types of input event to trigger the synchronization instead of a channel event.

The operation of the choice between *usr_dmd* and *core_sync* is to wait *usr_dmd* from user. If there is no demands position from user, the system would engage *core_sync* to let the whole system continue running. Thus we can use a timer to trigger the synchronization on *core_sync.* The conversion of the *choice* is shown in Code 3-18

```
sel=1;
tmr:>time;
time+=100;
select{
      case usr_dmd:>dmds_temp :
            sel=1;
            break;
      case tmr when timerafter(time) :> void :
             sel=0;
             core_sync<:1;
             break;
}
```

Code 3-18 Conversion of IMP_USR_DMDS

The conversion of the rest of the CSP script is similar to the conversion discussed above. Due to space limitations, the conversion details could be found in the source code of this project.

The integration of the XC modules applies the strategy which firstly forms the system with single axis and then extends it to 3 axis parallel system. In the development phase with XC, the *debug perspective* and *XMOS Simulator* mentioned in *Section 2.2.3* could be used to track traces of the XC application in order to sort out the bugs. The full diagram of the *Servo Clock* could be found in *Appendix C*.

3.3.2 Numerical Algorithm for Movement

The peripheral device is directed by a profile when it is running in a 3-dimension space. In *Servo Clock* each axis has its own individual profile. A profile is generated from the data of current coordinates and target coordinates .The profile contains the information of desired velocity, acceleration and position. The algorithm of the *Servo Clock* compares the current state of the peripheral device and the profile, and gives out the control signal. Hence, the profile is the strategy for the peripheral device to move from one position to another position.

The strategy is shown in Figure 3-7.

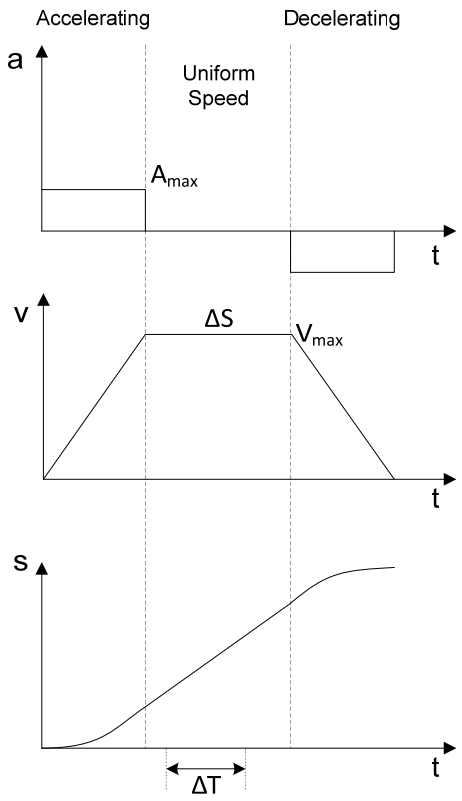


Figure 3-10 Moving strategy

It can be observed from Figure 3-10, the moving strategy is divided into three steps, accelerating, constant velocity and decelerating. Due to physical nature, the device has max velocity and max acceleration limit denoted as V_{max} and A_{max}.

Suppose the device is running at max acceleration rate when it is accelerating and denote the time period as Δt. The discrete equations for the profile could be derived from Figure 3-10.

If ΔS=0 there are only 2 parts in the moving strategy: accelerating and decelerating. Thus the equation for accelerating part and decelerating part is shown in Equation 3-1.

$$\frac{1}{2}\mathbf{A_{max}} * (\mathbf{n\Delta t})^2 = (\mathbf{S_{demand}} - \mathbf{S_{current}})/2$$

$$\mathrm{Vn} = \mathrm{Amax}^* \mathbf{n\Delta t}$$

$$\mathrm{Sn} = \mathrm{Sn\text{-}1} + \mathrm{Vn\text{-}1}^* \Delta\mathbf{t} + 0.5^*\mathrm{Amax}^* \Delta\mathbf{t^2} \qquad \text{Equation 3-1}$$

Where S_{demand} is demanded position, $S_{current}$ is current position and n is the cycles of iterations. S_0=0.

If ΔS>0 there are 3 parts in the moving strategy: accelerating, uniform velocity and decelerating. The velocity is increasing at the maximum accelerating rate before becoming uniform. Thus, the equations for accelerating part is shown in Equation 3-2

$$\mathbf{n} = \frac{\mathbf{V_{max}}}{\mathbf{A_{max}} * \Delta\mathbf{t}}$$

$$\mathbf{V_n} = \mathbf{A_{max}} * \mathbf{n} * \Delta\mathbf{t}$$

$$\mathbf{S_n} = \mathbf{S_{n-1}} + \mathbf{V_{n-1}} * \Delta\mathbf{t} + 0.5^*\mathbf{A_{max}}^* \Delta\mathbf{t^2} \qquad \text{Equation 3-2}$$

The equation for uniform velocity part is shown in Equation 3-3,

$$\mathbf{m} = \frac{\Delta\mathbf{S}}{\mathbf{A_{max}} * \Delta\mathbf{t}}$$

$$\mathbf{V_m} = \mathbf{V_{max}}$$

$$\mathbf{S_m} = \mathbf{S_{m-1}} + \mathbf{V_{m-1}} * \Delta\mathbf{t} \qquad \text{Equation 3-3}$$

where *m* is the cycles of iterations.

The equation for decelerating part is similar to Equation 3-2. The accelerating rate is opposite to that of accelerating part in Equation 3-2. The equations are shown in Equation 3-4.

$$\mathbf{n} = \frac{\mathbf{V_{max}}}{\mathbf{A_{max}} * \Delta\mathbf{t}}$$

$$\mathbf{V_n} = \mathbf{V_{max}} - \mathbf{A_{max}} * \mathbf{n} * \Delta\mathbf{t}$$

$$\mathbf{S_n} = \mathbf{S_{n-1}} + \mathbf{V_{n-1}} * \Delta\mathbf{t} - 0.5 * \mathbf{A_{max}} * \Delta\mathbf{t^2} \qquad \text{Equation 3-4}$$

The algorithm is implemented by the function of *prof_generator()* in XC , and the detailed implementation can be found in Part1 of Appendix B.

The close-loop control algorithm shown in Figure 2-16 could simply implemented by Equation 3-5,

$$\mathbf{V_{axis}} = \mathbf{V_p} + (\mathbf{S_p} - \mathbf{S_c}) * \mathbf{F} \qquad \text{Equation 3-5}$$

where V_{axis} is the velocity to axis, V_p is the velocity from profile, S_p is the demanded position from profile, S_c is current position from device and F is the scaling factor.

The control algorithm is implemented by function of *calc_velo()* in XC.

Chapter 4 Experiment Results and Issues

This chapter evaluated the performance of the parallel design and discussed about the issues during the converting work.

4.1 Experiments with Ideal Calculation

The performance of the parallel *Servo Clock* is measured by a set of experiments comparing the execution time of the parallel design against the sequential design with the same calculation load.

The calculation load is implemented by iterations of subtractions and additions in the functions of *prof_generator()* and *calc_velo(),* which could be found in *calc_func.c* file.

The results are shown in Table 4-1.

Table 4-1 Experimental results with ideal calculations

Calculation Load (unit)	Parallel Execution Time (us)	Sequential Execution Time (us)	Sequential/Parallel
0	9.8	19.6	2
1	12.8	27.4	2.140625
2	15.8	35.2	2.227848
3	18.8	43	2.287234
4	19.32	47.94	2.481366
5	21.8	55.72	2.555963
6	24.52	62.54	2.550571
7	27.1	70.32	2.594834
8	28.72	77.14	2.685933
9	31.32	83.94	2.680077
10	33.92	91.72	2.704009
11	36.52	98.54	2.698248
12	39.1	106.32	2.719182
13	40.72	113.14	2.778487
14	43.32	119.94	2.768698
15	45.92	127.72	2.781359
20	57.92	163.72	2.826657
80	201.92	595.72	2.950277

The results indicate the average execution time for one cycle. It can be observed that, the performance of the parallel design is better than that of the sequential design. When applying 0 unit of calculation load, then time taken by parallel and sequential design is 9.8us and 19.6us respectively due to the inter-communication among modules. Thus real execution time for calculation should be the execution time in the table subtracted by the time spent on inter-communication.

The time cost by *Servo Clock* is distributed into three parts: algorithm, data collection and data return. Denote T_a, T_{pc} and T_{ps} as the time spent on algorithm, data collection and data return for parallel design, and T_a, T_{sc} and T_{ss} for that of sequential design. The time traces are shown in Figure 4-1

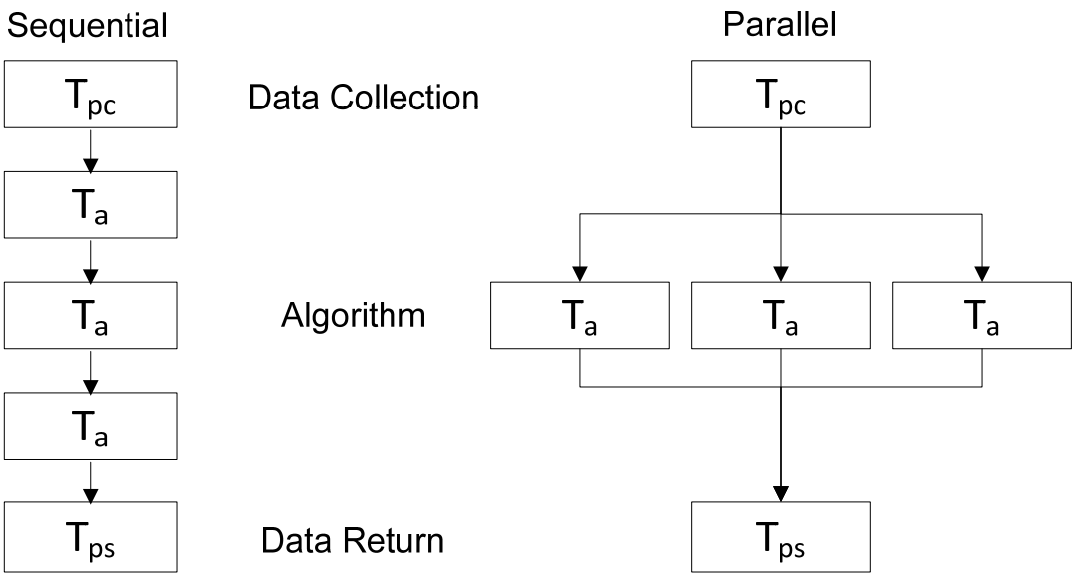


Figure 4-1 Time trace of parallel and sequential design

Thus, the cycle time of each design could be derived as follows:

$$T_p = T_a + T_{pc} + T_{ps}$$

$$T_s = T_a * 3 + T_{sc} + T_{ss} \qquad \text{Equation 4-1}$$

Where T_s is the cycle time for sequential design, and T_p is the cycle time for parallel design. When changing calculation load, the time $T_{pc}+T_{ps}$ and $T_{sc}+T_{ss}$ for data collection and data return of both designs are constants. Thus the ratio of T_s and T_p can be represented as Equation 4-2

$$A = \frac{T_s}{T_p} = \frac{T_{sc}+T_{ss}+3*T_a}{T_{sc}+T_{ss}+T_a} \qquad \text{Equation 4-2}$$

With increasing calculation load (T_a), the ration will be approaching 3, but never reach 3. Figure 4-1 represents the data from Table 4-1 and shows the trends with increasing calculation load. The *Experimental Result* curve represents the data in *Table 4-1,* and the *Ideal Result* curve represents the result from Equation 4-2 with $T_{sc} + T_{ss} = 1$.

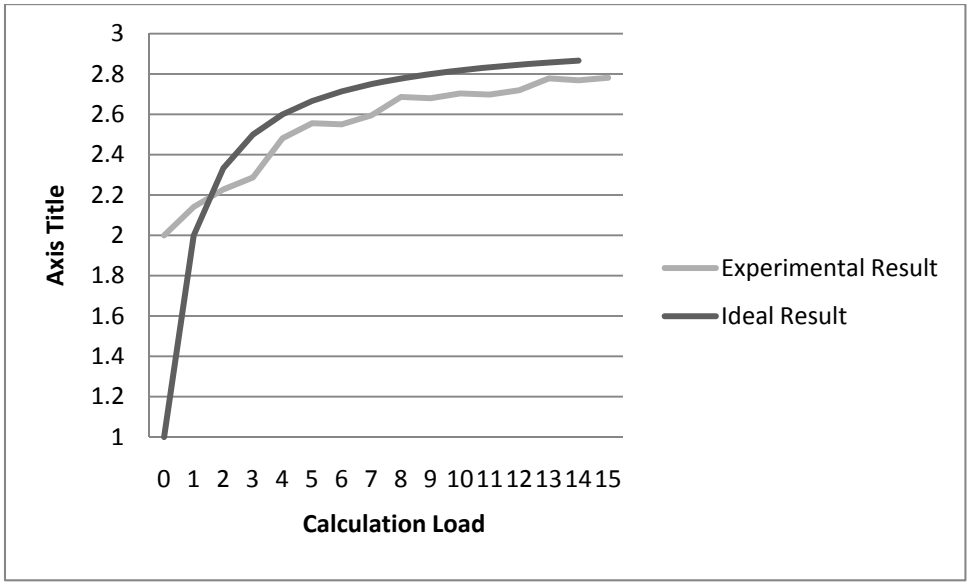


Figure 4-1 Trends of performance speedup

From the discussion above, the analysis results matches the experimental results. More benefit will be gained from parallelism with the increasing calculation load. In this project, we just distributed the main calculation task into 3 parallel calculation tasks. If we could distribute the main task into more parallel sub tasks, the performance would be enhanced but there would be an explosive growth of resource requirement.

According to *Amdahl's Law* [1], with a parallelism the performance speedup is limited due to the sequential part. If the ratio of the time spent on parallel calculations to the time spent on sequential calculations increases in a system, the performance of the system could be enhanced to a higher level. But this also means the cost on hardware will increase. In, this project, the performance of the parallel *Servo Clock* is limited within 3 times of that of the sequential *Servo Clock* but used almost all the resources on the XC-1 Development Kit.

With the discussion above, when designing a parallel system, *Amdahl's Law* should be taken into account in order to make a balance between performance and cost.

4.2 Experiment with Simple Control Algorithm

In order to test the full functionality of the *Servo Clock*, this experiment is taken with a real control algorithm. The dummy functions of *prof_generator()* and *calc_velo()* were replaced by the real algorithms discussed in *Section 3.3.2*.

The test bench consists of virtual device and virtual user interface. The *Servo Clock* receives the required data from virtual environment, and sends the control data to the virtual peripheral devices.

The function of the virtual device is to receive the demanded velocity from the *Servo* Clock and send back 'current count' to the *Servo Clock* after the transformation from *velocity* to *count* so as to simulate the real device. The function of virtual user interface is to simulate the behaviour of user whom sends data requirement and demanded position at any time. The construction of the virtual axis is shown in Code 4-1,

```
void VIRT_AXIS(chanend snd_count,chanend rec_vel)
{
   int dmd_vel,count=0;
   while(1)
   {
      rec_vel:>dmd_vel; // receive velocity from Servo Clock
      count=(count+dmd_vel*DELTA_T)%MAX_COUNT;
      snd_count<:count;// return velocity to Servo Clock
   }
}
```

Code 4-1 Virtual axis

where DELTA_T is the period of one cycle (1ms), MAX_COUNT is defined to simulate the physical features of the device.

With the timing analyzing method introduced in *Section 2.2.4,* we can track the detailed status of the peripheral device and the timing in real-time. The initial position of the peripheral device is (0, 0, 0). Set the demanded position to (500, 600, 700) and apply it to the *Servo Clock.* It takes 16 steps to move from (0,0,0) to the final position (499, 601, 699), and each cycle takes an average time of 28us. The result is shown in Figure 4-2.

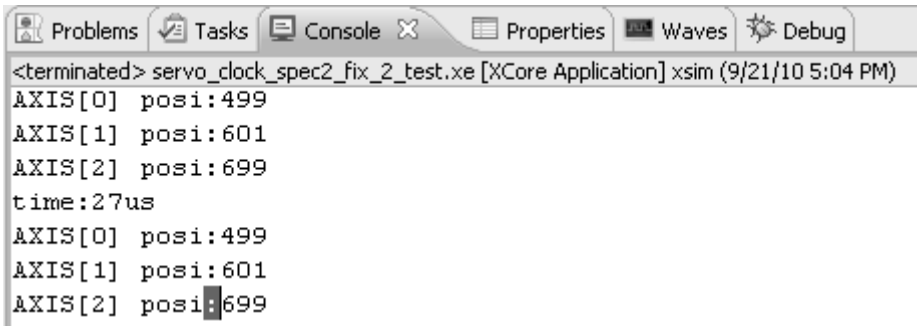


Figure 4-2 Test results from XDE

The functionality of the parallel *Servo Clock* was successfully validated, and the cycle time is much less than the requirement which is 1ms per cycle. The high-performance parallel *Servo Clock* enables more accurate algorithm to operate, and has the potential to run under frequency higher than 1 KHz. There is also a potential

to increase the device sampling rate to increase the accuracy according to *Shannon Theorem*[29].

4.3 Issues

Issue 1: Conflicts between formal verified design and efficient design

In chapter 3, we modified the verified CSP implementation (Code3-8) of CORE_ALG for conversion simplicity. In the verified implementation, all of the three modules *IMP_USR_DMDS, IMP_DATA* and *CORE_ALG* synchronize on the events of *core_sync* and *core_sync2.* Thus it requires 3-node barrier synchronization which will consume 3 channels and it will spend at least 6 steps to finish the barrier.

In practice, the 3-node barrier is split into two 2-node barrier (Code 3-15), and it just requires two steps to finish these 2 barriers. But this will cause the problem of validity of the refinement.

We checked the modified *CORE_ALG* refinement with FDR2, the *Failures-divergence* refinement check was failed, but the *Traces* refinement check was passed. This means the refinement does not match the full cases of the specification but only a part of it, and the behaviour of the refinement is within the specification constrain. From the FDR2 feedback, the problem is that in some cases the operation of returning data from the *Core* could not happen before other operations after *sync* indicated in the specification.

In this case, it doesn't affect the full functionality of the *Servo Clock* and it improves the resource efficiency and program efficiency. Hence, we deiced to use the modified *CORE_ALG* refinement, although it doesn't match the full cases of the specification.

The issue could be resolved by either updating the specification in CSP or implementing the inefficient 3-node barrier synchronization in XC.

Issue 2: Formal verification in CSP V.S. extensions in XC

In the formal verified refinement in CSP, all the data types for communication are in the range of {0..1}. The range of the data is relevant to the complexity of communication. If the range was enlarged, the communication complexity would significantly increase. The result below is of the refinement checking with *failures-divergence model* by FDR2 with the data range of {0..1}.

*+.**
+.... 7,925,836
Refine checked 7,925,836 states
With 31514897 transitions
Took 82(74+0) seconds

It can be observed that there are millions of states and transitions in the check, and the refinement check takes about 2 minutes. With the data range of {0..2}, the result is shown below.

```
+.*
+.... 106,904,754
Refine checked 106,904,754 states
With 467169183 transitions
Took 2243(2218+3) seconds
```

The number of checking states is significantly increased, and the check takes about 40 minutes. Thus, for the full range of integers, it might take even one or more days to finish the check. So we decided to formally check the refinement with a small range, and extend the data range to full integer in XC. The final application in XC was proved to work properly.

Until now, we didn't find a formal way to prove the correctness of this procedure, although the final result is what we wanted. Thus, there is still a need to prove the correctness.

Issue 3: Alternative configuration of *select* construction in XC

For the synchronization on *core_sync* between *CORE_ALG* and *IMP_USR_DMDS* in XC (Code 3-16 and Code 3-18), the channel-end in *CORE_ALG* was configured as input while the other channel-end in *IMP_USR_DMDS* was configured as output. In practice, we tried the opposite configuration in which the CORE_ALG channel-end was configured as output, and that of IMP_USR_DMDS was configured as input. The code is shown in Code 4-2

```
//Code in CORE_ALG
tmr[i]:>time[i];
sel[i]=1;
time[i]+=15000;
select{
  case to_core_prof_vel[i]:>prof_vel[i] :
  //CASE A
     to_core_prof_acc[i]:>prof_acc[i];
     to_core_prof_posi[i]:>prof_posi[i];
     break;
  case tmr[i] when timerafter(time[i]) :> void :
  //CASE B
     sel[i]=0;
    core_sync:>1;
    break;
}
```

```
//Code in IMP_USR_DMDS
sel=0;
select{
  case usr_dmd[0]:>dmds_temp :
  //CASE C
     sel=1;
     break;
  case core_sync:>abc:
  //CASE D
     core_sync<:1;
     break;
}
```

Code 4-2 Alternative conversion

We observed the synchronization with choice in CSP is different from that in XC. In CSP, when an *external choice* engages in a signal, the other *external choices* synchronized with it will always engages in the same signal and choose the relevant branch. But in XC, so far as we know, it lacks the equivalent module to implement it.

The *CASE A* in Code 4-1 means the profile is ready, and the *Core* engages in this branch and does relevant operations. The *CASE B* will be engaged if the profile was still not ready after a time limit. The *CASE C* will be engaged if user demands come in. The *CASE D* is coupled with CASE B, if the *Core* times out, it will be engaged in.

Practically, the approach shown in Code 4-1 will potentially lead to deadlock if the time limit is shorter than the time taken to generate profiles.

Suppose the user demands signal comes in and the timer expires at the same time, the *CORE_ALG* would engage in *CASE B* which needs to synchronize on *core_sync* while the *IMP_USR_DMDS* would engage in *CASE C* which firstly needs to synchronize on *to_core_prof_vel[i]*. As the channels in XC are lossless [17], the program will be blocked the channel if the other channel-end is not matched. Therefore, in this case, *CORE_ALG* will be blocked at *core_sync* and *IMP_USR_DMDS* will be blocked at *to_core_prof_vel[i].* Consequently, deadlock is introduced which means it is not a valid design.

Issue 4: A shortage of the parallel *Servo Clock*

The design of *IMP_USR*_DMDS can let the *Core* continue running when there are no user position demands. But when there are user position demands, the *Core* should have to wait for the profile to be generated before continue running. As the *Servo Clock* is running at the frequency of 1 KHz, this issue may potentially lead to a problem when the time for profile generation exceeds 1ms.

Chapter 5 Conclusion and Future Work

5.1 Conclusion

With current state-of-the-art of industry, parallel computing is an alternative way to speedup performance. Developing a parallel system is more difficult than that of a traditional sequential system. Deadlock, livelock and determinism are commonly met in design phase. As the title suggests, this project has two main tasks. One is to refine the specification into implementation level appropriate for converting into XC, and the other one is to convert the CSP implementation into XC.

In chapter 3, we analyzed the behaviour of the *Servo Clock,* designed the *Core* of the *Servo Clock,* refined the specification and verified it with FDR2. As discussed in chapter 3, there were various ways to convert CSP models into XC models. The decision of which way to take depends on programming complexity in XC programming language, execution efficiency and resource efficiency. The *Servo Clock* specification is not complex in this project and only one-level refinement is appropriate for converting into XC. When designing the parallel system, the resource limit should be taken into account. Otherwise the design for the target device may not be valid. The design flow is proved to be successful.

The four *Conversion Rules* in *Chapter 3* have been validated informally by observing the behaviours both in CSP and XC. These four conversion rule at the moment could be used in manual conversion work with specific conditions. Converting CSP script is relevant easy by applying the *Conversion Rules* due to the equivalence between both languages. During the design phase, CSP is good at designing communication protocol, but not well at calculations, especially for shared data processing. That is because CSP is algebra for processes that have their own memory spaces and exchange data via channels. Thus we used dummy functions in CSP and implement the details in XC instead of the dummy functions in order to reduce development difficulty.

The success of chapter 3 is quite useful to parallel system design as the software complexity is suffering an explosive increase. Design and verify a parallel embedded system becomes more and more difficult and time consuming. The design flow introduced in chapter 3 could potentially lower down the design difficulty and reduce product to market time.

In chapter 4, we did a set of experiments to evaluate the performance of both parallel design and sequential design. The comparison on the execution time of parallel design and that of the sequential design demonstrated how much benefit the parallel design can get. The experimental results showed that there was a

limitation of performance speedup brought by parallelism, and the mathematical analysis based on the results explained why there is a performance speedup limitation. In terms of cost efficiency, *Amdahl's Law* should be applied to make a balance between performance and parallelism. The experiment with simple control algorithm successfully proved the functionality of the *parallel Servo Clock.*

Although the parallel design can work properly, there is still shortage which could be improved by future development. When apply a more advance moving strategy in the profile, the calculation time for the profile will increase and it will probably exceeds the maximum allowance of 1ms. During profile generation the servo clock could do nothing until the generation is completed. The potential solution of this issue will be in next section.

5.2 Future Work

Although this project achieved a certain success, there are still improvement could be made.

5.2.1 Advance Parallel Architecture

The *issue 3* discussed in *Chapter 3* showed the shortage in current *parallel Servo Clock* design. The *Core* could not continue working before the completion of profile generation. This could be resolved by designing an individual profile generator, which could be running concurrently with *IMP_USR_DMDS*. The new profile generator is running individually from *IMP_USR_DMDS.* After the completion of the profile generation, it sends the ready signal to IMP_USR_DMDS, and then the *IMP_USR_DMDS* grabs the profile from the generator. The flow chart is shown in Figure 5-1 on next page.

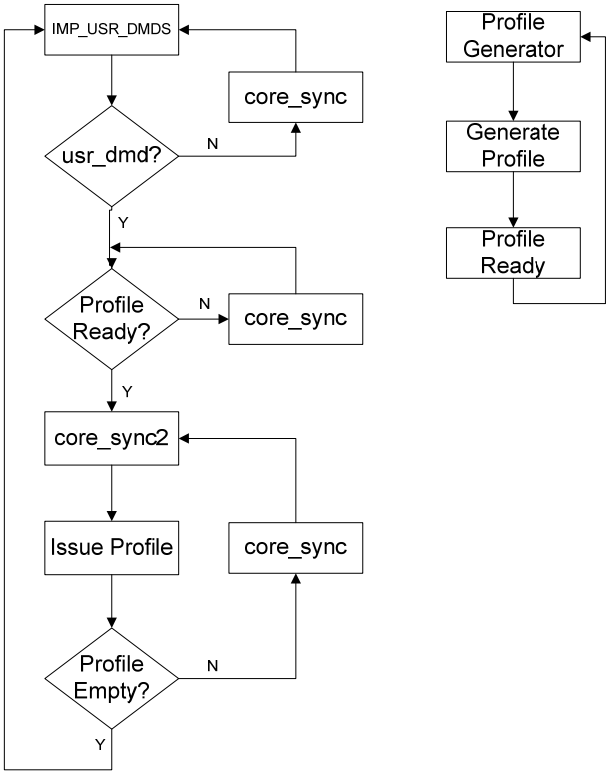


Figure 5-1 Flow chart of new IMP_USR_DMDS

This approach requires extra resources as the new profile generator adds additional parallelism. Also, there would be a conflict for the refinement and the specification.

In the specification, the *usr_dmd* channel could be released after a certain *CALC_TIME*. The *CALC_TIME* could be either treated as the time for profile generation or the time for profile issuing. But when the profile generator runs individually apart from *IMP_USR_DMDS*, the time used to issue profile will be an arbitrary time. The specification didn't give out the arbitrary time for profile issuing. In order to resolve the problem, either the specification could be updated or re-design the whole *Servo Clock* in future.

5.2.2 Advanced Conversion Models for Automatic Translator

All the conversion work of this project was done manually by applying the *Conversion Rules*. When the software complexity increases, it is inevitable to make mistakes during converting and it is time consuming.

Thus, more advanced conversion models could be built in order to form an automatic translator. With the advantage of CSP and the automatic translator, it can greatly reduce the development time and difficulty for parallel system design.

Conversely, if the XDE could integrate a tool which is similar to FDR2 to analyse the deadlock and freelock freedom, it will also reduce the development time and effort which is needed in industry.

Bibliography

[1] G.Amdahl, "The validity of the single processor approach to achieving large-scale computing capabilities," in *Proceedings of AFIPS Spring Joint Computer Conference*, Atlantic City, (April 1967), pp. 483–85.

[2] Blaise Barney, "Introduction to Parallel Computing," Lawrence Livermore National Laboratory, Retrieved 2007-11-09.

[3] Pedersen J.B., "Classification of Programming Errors in Parallel Message Passing Systems," in *Proceedings of Communicating Process Architectures 2006*, 2006, pp. 363-376.

[4] C.A.R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666-667, August 1978.

[5] G.Barrett, "Model checking in practice: The T9000 Virtual Channel Processor," *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 69-78, Feburary 1995.

[6] Buth Bettina, Kouvaras Michel, Peleska Jan, and Shi Hui, "Deadlock Analysis for a Fault-Tolerant System," in *Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology*, 1997, pp. 60-74.

[7] Buth Bettina, Peleska Jan, and Shi Hui, "Combining Methods for the Livelock Analysis of a Fault-Tolerant System," in *Proceedings of the 7th International Conference on Algebraic Methodology and Software Technology*, 1998, pp. 124-139.

[8] J.C.M.Baeten, "A Brief History of Process Algebra," *Theoretical Computer Science*, vol. 335, no. 2-3, pp. 131-146, 2005.

[9] K.Eder, "Concurrency Part II Lecture Notes in COMS22101: Concurrency and Communications," Computer Science Department, Universiti of Bristol, 2010.

[10] S.Schneider, *Concurrent and Real-time System The CSP Approach*.: JOHN WILEY & SONS, 2000.

[11] Hilderink Gerald, André Bakkers, and Jan Broenink, "A Distributed Real-Time Java System Based on CSP," in *The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, California, 2000, pp. 400-407.

[12] N.Parashkevov Atanas and Jay Yantchev, "ARC - a tool for efficient refinement and equivalence checking for CSP," in *IEEE Int. Conf. on Algorithms and Architectures for Parallel Processing ICA3PP*, 1996, pp. 68-75.

[13] Jun Sun, Liu Yang, and Song Dong Jin, "Model Checking CSP Revisited: Introducing a Process Analysis Toolkit," *Proceedings of the Third International Symposium on*

Leveraging Applications of Formal Methods, Verification and Validation, no. Communications in Computer and Information Science, pp. 307-322, 2008.

[14] Formal Systems (Europe) Ltd., "FDR2 Manual," 2005.

[15] D.May, "The XMOS XS1 Architecture," XMOS Ltd., 2009.

[16] D.May, "XMOS Architecture XC language," XMOS Ltd., 2010.

[17] D.Watt, "Programming XC on XMOS Devices," XMOS Ltd., 2009.

[18] "XMOS Simulator Tutorial," XMOS Ltd., 2009.

[19] "XMOS Timing Analyzer Whitepaper," XMOS Ltd., 2010/05/18.

[20] Hilderink Gerald, Broenink Jan, Vervoort Wiek, and Bakkers Andre, "Communicating Java Threads," University of Twente, dept. EE, Control Laboratory, 1997.

[21] C.A.R. Hoare, "Monitors: An Operating System Structuring Concept.," *An Operating System Structuring Concept*, vol. 17, no. 10, pp. 549-557, 1974.

[22] Brown Neil and Welch Peter, "An Introduction to the Kent C++CSP Library," Computing Laboratory, University of Kent, Kent, 2003.

[23] V.Raju, L.Rong, and G.S.Stiles, "Automatic Conversion of CSP to CTJ,JCSP and CCSP," in *Communicating Process Architectures*, 2003.

[24] Freitas Angela and Cavalcanti Ana, "Automatic Translation from Circus to Java," *Lecture Notes in Computer Science*, vol. 4085, pp. 115-130, 2006.

[25] Seidel Karen, "Case Study: Specification and Refinement of the PI-BUS," Programming Research Group, Computing Laboratory, Oxford University,.

[26] (2010, Sep) CSP for Java (JCSP). [Online]. http://www.cs.kent.ac.uk/projects/ofa/jcsp/explain.html

[27] Xu Hong, K.Mckinley Philip, and M.Ni Lionel, "Efficient Implementation of Barrier Synchronization in Wormhole-Routed Hypercube Multicomputers," *Journal of Parallel and Distributed Computing*, vol. 16, no. 2, pp. 172-184, October 1992.

[28] Jamie Hanlon, "XK-XMP-64 Performance Measurements," XMOS Ltd., 2010.

[29] Raymond W. Yeung., "Information Theory and Network Coding," *Springer*, 2008, 2002.

[30] Lawrence Jonathan, "Practical Application of CSP and FDR to Software Design," IBM United Kingdom Ltd., Winchester,.

Appendix A: Notations

A.1 Syntax for machine readable CSP:

Standard CSP	CSP for FDR2	Explanation
STOP	STOP	No actions (Deadlock)
SKIP	SKIP	Successful termination
P \ a	P \ a	Hiding
$P \Box Q$	P [] Q	External choice
$P \mid\mid\mid Q$	P ||| Q	Interleaving
$P \mid_{\{c\}} \mid Q$	P[|{|a|}|]Q	Interface parallel (*sharing*)
$P_A \mid\mid_B Q$	P[a||a']Q	Alphabetised parallel
$\mid_{A_{i \in I}} \mid P_i$	[|a'|]x:a@p	General interface parallel
P ; Q	P ; Q	Sequential composition
$P \sqsubseteq_T Q$	*P [T= Q*	*Traces* refinement
$P \sqsubseteq_{FD} Q$	*P [FD= Q*	*Failures-divergence* refinement

Appendix B: XC Source Code Sample

This part only gives out the essential part of the source code.

Part 1: Moving Strategy for Profile Generator

The function prof_generator() is used to apply the moving strategy in section 3.3.2 to generate profiles for 3 individual axis. It is called by IMP_USR_DMDS for each of the 3 axes.

```
unsigned prof_generator(int dmds, int curr_posi,
                        int prof_vel[],int prof_acc[], int prof_posi[])
{

        int delt_s,i,max_acc;
        unsigned p_size;
        if (dmds>curr_posi) //get the distance with uniform velocity
        {
                delt_s = (dmds-curr_posi)-(MAX_VEL*MAX_VEL)/MAX_ACC;
                        max_acc=MAX_ACC;
        }
        else
        {
                delt_s = (curr_posi-dmds)-(MAX_VEL*MAX_VEL)/MAX_ACC;
                max_acc=-MAX_ACC;
        }
  if (delt_s>0) // moving strategy with uniform velocity part
  {
        unsigned  n, m,tmp;
        prof_vel[0]=0;prof_acc[0]=max_acc;prof_posi[0]=curr_posi;
        n=MAX_VEL/MAX_ACC;                  //get the time for accelerating
        m=delt_s/MAX_VEL/DELTA_T;           // get the time for uniform velocity.
        tmp=n+m+n; // the time for whole moving strategy.
        for (i=1;i<tmp;i++)
        {
        prof_vel[i]=prof_vel[i-1]+DELTA_T*prof_acc[i-1];
        if (i<n) prof_acc[i]=max_acc;            // accelerating at the maximum rate
        else if (i<m+n)
                {
                        prof_acc[i]=0;           // uniform velocity
                }
                else prof_acc[i]=-max_acc;    //decelerating at maximum rate
        prof_posi[i]=prof_posi[i-1]+ prof_vel[i-1]*DELTA_T+
                        prof_acc[i-1]*DELTA_T*DELTA_T/2; // demanded position
        }
        p_size=tmp;

  }
  Else // moving strategy without uniform velocity part
     {
int n;
```

```
n = (sqrt(abs((dmds-curr_posi)/MAX_ACC/DELTA_T/DELTA_T))+0.5);
// the time for acceleration or deceleration
prof_vel[0]=0;prof_acc[0]=max_acc;prof_posi[0]=curr_posi;
 for (i=1;i<2*n;i++)
        {
          prof_vel[i]=prof_vel[i-1]+DELTA_T*prof_acc[i-1];
          if (i<n) prof_acc[i]=max_acc;              // accelerating
          else prof_acc[i]=-max_acc;                 // decelerating
          prof_posi[i]=prof_posi[i-1]+prof_vel[i-1]*DELTA_T+
                        prof_acc[i-1]*DELTA_T*DELTA_T/2;
        }
        p_size=2*n;

  }
  for (i=p_size;i<ARR_SIZE;i++)              //fill the rest with 0
  {
            prof_vel[i]=0;
            prof_acc[i]=0;
            prof_posi[i]=dmds;
  }
  return p_size;
}
```

Part2: Implementation for CORE_ALG

```
void CORE_AXES(chanend to_core_posi[], chanend to_core_prof_vel[],
                    chanend to_core_prof_acc[],chanend to_core_prof_posi[],
                    chanend to_core_probe_defns[],chanend core_to_data[],
                    chanend core_data_sync,chanend core_trn_vel[], chanend core_sync,
                    chanend core_sync2)
{
   unsigned curr_posi[NUM_AXES],prof_vel[NUM_AXES];
   unsigned prof_acc[NUM_AXES],prof_posi[NUM_AXES];
   unsigned defns[NUM_AXES],abc,abc1;
   unsigned sel;
while(1)
   {
          //the following part is CORE_AXIS
          par(int i=0;i<NUM_AXES;i++)
                        {
                                {to_core_posi[i]:>curr_posi[i];
                                to_core_probe_defns[i]:>defns[i];
                                core_to_data[i]<:curr_posi[i];
                                }
                        }
          select{
                    case core_sync2:>abc:
                       sel=1;
                       break;
                    case core_sync:>abc:
                     core_data_sync<:1;
```

```
                sel=0;
                break;
            }
        for(int i=0;i<NUM_AXES;i++) // debug: print out axis position
        {
            printstr("AXIS[");
            printint(i);
            printstr("] posi:");
            printintln(curr_posi[i]);
        }

        //the following part is after core_sync2;
        if (sel)
        {
            par(int i=0;i<NUM_AXES;i++)
             {
                 {
                  to_core_prof_vel[i]:>prof_vel[i];
                 to_core_prof_acc[i]:>prof_acc[i];
                to_core_prof_posi[i]:>prof_posi[i];
                 }
             }
            core_sync:>abc;
            core_data_sync<:1;
            par(int i=0;i<NUM_AXES;i++)
            {
            core_trn_vel[i]<:calc_velo(curr_posi[i], prof_vel[i],  prof_acc[i],
                                       prof_posi[i],   defns[i]);
            }
        }
        else
        {
        par(int i=0;i<NUM_AXES;i++)
        {
            core_trn_vel[i] <: calc_velo(curr_posi[i],0, 0,  curr_posi[i], 0);
        }
        }
   }
}
void CORE_ALG(chanend to_core_posi[],chanend to_core_prof_vel[],
            chanend to_core_prof_acc[],chanend to_core_prof_posi[],
            chanend to_core_probe_defns[],chanend core_to_data[],
            chanend core_data_sync,chanend core_trn_vel[],
            chanend core_sync,chanend core_sync2)
{
        CORE_AXES(to_core_posi,to_core_prof_vel,to_core_prof_acc,to_core_prof_posi,
                    to_core_probe_defns,core_to_data,core_data_sync,
                    core_trn_vel,core_sync,core_sync2);
}
```

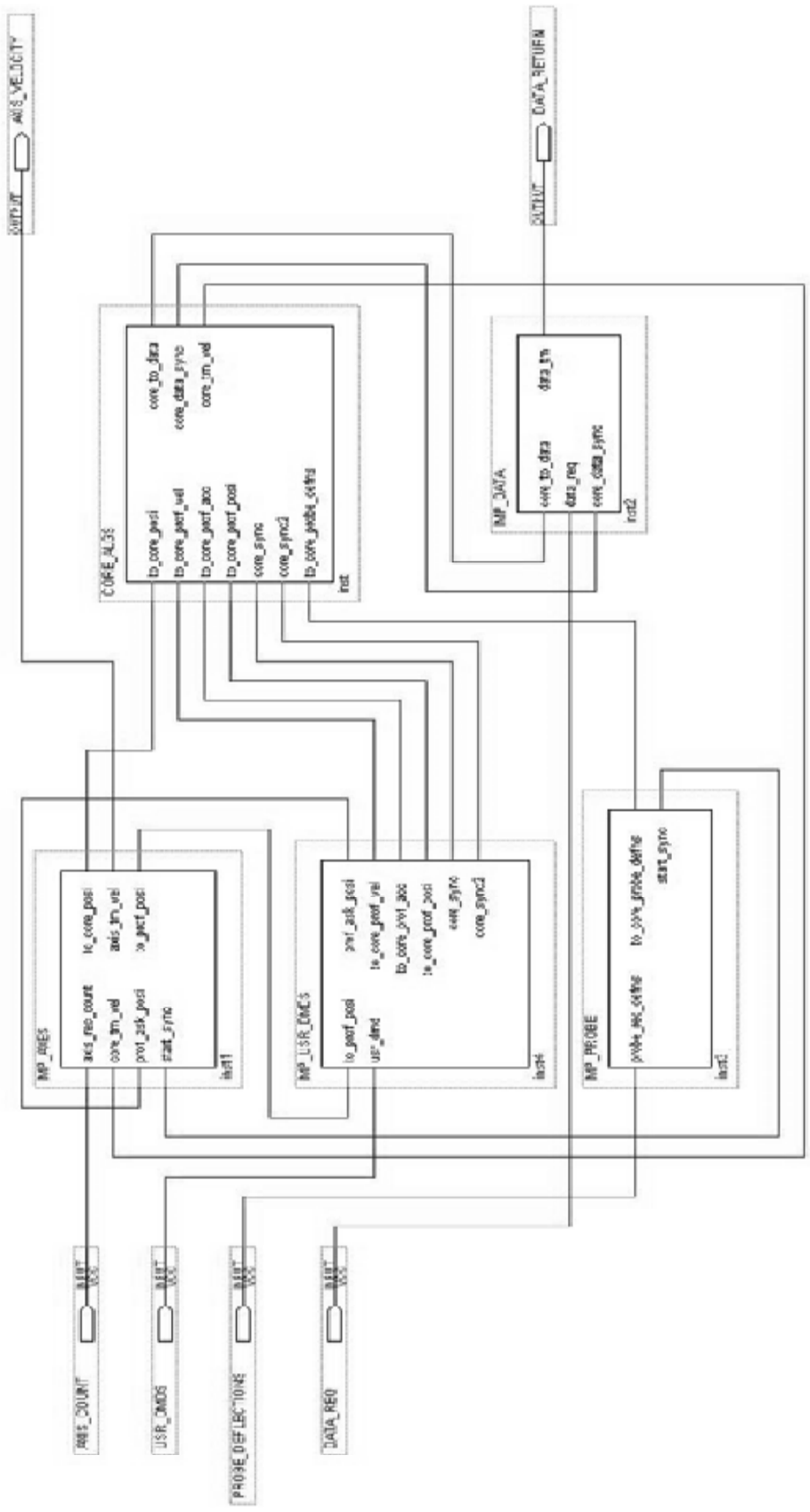
Part 3: Implementation of IMP_USR_DMDS

All the variables use the same name as that in CSP

```
unsigned DMDS_COLLECT_AXES(unsigned inst, int dmds, chanend prof_ask_posi,
                                    chanend to_prof_posi,int prof_vel[],  int prof_acc[],
                                    int prof_posi[] )
{
        unsigned prof_size=0;
        int curr_posi;
        prof_ask_posi<:1;
        to_prof_posi:>curr_posi;
        prof_size=prof_generator(dmds,curr_posi,prof_vel,prof_acc,prof_posi);
         return prof_size;

}

void DISTR(unsigned inst,unsigned vel,unsigned acc,unsigned posi,
                    chanend to_core_prof_vel,
                    chanend to_core_prof_acc,
                    chanend to_core_prof_posi)
{
        to_core_prof_vel<:vel;
        to_core_prof_acc<:acc;
        to_core_prof_posi<:posi;
}

void IMP_USR_DMDS(chanend prof_ask_posi[], chanend to_prof_posi[],
                          chanend usr_dmd,          //x,y,z demands for 3 Axis
                          chanend to_core_prof_vel[], chanend to_core_prof_acc[],
                         chanend to_core_prof_posi[], chanend core_sync,
                          chanend core_sync2)
{
        unsigned prof_size[NUM_AXES];
        unsigned dmds[NUM_AXES];
        unsigned dmds_temp;
        int prof_vel[NUM_AXES][ARR_SIZE],prof_acc[NUM_AXES][ARR_SIZE];
        int prof_posi[NUM_AXES][ARR_SIZE];
        unsigned temp,tmp_count=0,abc,sel;
        timer tmr;
        unsigned time;
        while(1)
        {
         sel=1;
         tmr:>time;
         time+=100;
         select{
           case usr_dmd:>dmds_temp :
                    sel=1;
                    break;
```

```
            case tmr when timerafter(time) :> void :
                        sel=0;
                        core_sync<:1;
                       break;
        }
    if (sel)
    {
        //the following part is DMDS_SPLIT(x,y,z) implementation
        for (int i=0;i<NUM_AXES;i++)
        {
                dmds[NUM_AXES-i-1]=dmds_temp&0x3ff;
                dmds_temp=dmds_temp>>10;
        }
        par{
                par (int i=0;i<NUM_AXES;i++)
                {
                prof_size[i]=DMDS_COLLECT_AXES(i, dmds[i], prof_ask_posi[i],
                                        to_prof_posi[i], prof_vel[i],prof_acc[i],prof_posi[i] );
                }
        }
        temp=get_max(prof_size)+6;
        // 6 more profile issuing in order to cope with algorithm
        tmp_count=0;
        while(temp--)
        {
         core_sync2<:1;
          par{
                par(int i=0;i<NUM_AXES;i++)
                {  DISTR(i,prof_vel[i][tmp_count],prof_acc[i][tmp_count],
                        prof_posi[i][tmp_count],  to_core_prof_vel[i],
                          to_core_prof_acc[i], to_core_prof_posi[i]);
                }
             }
                core_sync<:1;
                tmp_count++;
        }
    }
   }
}
```

Appendix C: Servo Clock Structure



Appendix D: Table of Figures

Appendix E: Table of Code Blocks