# Aspect-Oriented Refactoring of Visual MIPS Datapath Simulator

Hidayet Aksu, Özgür Bağlıoğlu, and Mümin Cebe
*Department of Computer Engineering, Bilkent University*
*Bilkent 06800, Ankara, Turkey*
*{haksu,ozgurb,mumin}@cs.bilkent.edu.tr*

## Abstract

**EZ_MIPS is a visual MIPS datapath simulator, produced for educational purposes, which aims to make understanding of MIPS working principle more easily. In this paper, we analyze and refactor the EZ_MPIS to enhance its quality and add some new features. The restructuring process consists of two steps. For this, we first introduce a set of object-oriented patterns. Then the tool is analyzed from the aspect-oriented perspective. We have identified the crosscutting concerns and we have refactored these concerns to aspects. In addition, we have applied enforcement aspects to ensure that the adopted polices would enhance the development process both in time and quality. We also refactor some other scattered code segments using aspects. We also add a new concern profiling to our program in order to get the execution information about the simulation. Based on our experiences we describe our lessons learned. The OOP and AOP approaches will absolutely enhance the process of design and implementation.**

## 1. Introduction

MIPS (Microprocessor without Interlocked Pipeline Stages) which is developed a team led by John L. Hennessy at Stanford University in 1981, based on RISC architecture. The basic attempt was able to enhance performance through the implementation of instruction pipelines, which is well known but difficult to actualize [3]. MIPS focused on almost entirely on the pipeline making sure it could be run fully integrated. The most important reason for this is all the instructions need of one cycle to complete its cycle in CPU. This problem is solved by pipelined architecture that increases speed with reduced delays and CPU is used as much as possible without any gaps between instructions. MIPS has a wide variety of usage area in today's computer technology including DVD players, Networking devices such as WLAN Access points, televisions, portable devices such as digital cameras, video game consolers such as play station etc[4]. As seen from these example usage areas, we can see the impress of MIPS in every part of electronic devices. For this reason it is important to visualize the MIPS architecture in order to fully understand and become professional in MIPS architecture which is widely used.

The domain of simulations takes much attention in order to make the problem more understandable and to monitor the feasibility of the problem. With all these contributions of MIPS in mind, it is important to develop a visual MIPS for students to grasp the concepts of MIPS architecture fully. To support education activities o simulators we have developed a MIPS simulator by using object-oriented techniques which is commonly used in software development process. EZ_MIPS aims to make his task easier with a visual datapath simulator. EZ_MIPS will be helpful in understanding the MIPS architecture which is widely discussed in [5] has been taught in most of Computer Architecture courses.

In general, object-oriented approaches providing many advantages by breaking systems into unit parts to manage the whole system easily [1], but it is inadequate for encapsulating the concerns that are scattered over whole components such as concurrency, failure handling and logging [2]. These kind of scattered concerns break down the integrity and cause the coupling of different components. This causes in code scattering, and code tangling which in turn software becomes hard to maintain, because of low-cohesion and high coupling. Aspect-Oriented programming (AOP) proposes powerful solutions to this difficulty by identifying these concerns and encapsulating them efficiently. AOP efficiently separates the crosscutting concerns, by providing explicit concepts to modularize them and integrates them with the system components.

EZ_MIPS is first implemented in 2003 with agile programming. In our experiences maintaining the EZ_MIPS was not trivial, because it is implemented with agile development. In this implementation, there are some design defects and bugs. As such, we decided to analyze the EZ_MIPS and refactor it to reduce complexity and enhance maintainability. Then, the design of modules is improved by using OOP and AOP principles.

The process applied in this paper with OOP and AOP approaches, software restructuring, is an essential activity in software engineering, according to Lehman's second law of software evolution, which address increasing complexity: As a

program evolves, its becomes more complex unless work is done to maintain or reduce it[ 11].

Shortly, in this paper, the EZ_MIPS will be refactored from the perspective of object-oriented approaches, and whenever it is not sufficient aspect-oriented approaches. The purpose is to make the program more stable by using de-facto patterns and cleaning code from crosscutting concerns, scattering code segments finally purifying design by means of aspect-oriented approaches.

The rest of this paper is organized as follows. In the next section, we will give a short introduction of selected case, EZ_MIPS, for reengineering with OOP and AOP approaches. The object-oriented reengineering process using well known design patterns applied to EZ_MIPS are given in section 3. Then we continue with detailed identification of crosscutting concerns and scattered code segments in the simulated model and we discuss the solution architecture of these concerns using aspect-oriented approaches. In section 5, we mention related work, and finally in section 6, we provide our concluding remarks.

## 2. Problem Case: EZ_MIPS

As explained before the simulation model discussed in this paper is based on the MIPS instruction set. Having an idea about how a computer datapath works and in what manner a datapath could be improved, are of fundamentals of computer science. However, sometimes, it becomes difficult to understand the manner in which a given datapath works through the assistance of books. At that times usage of some tools, with which the user enable to interact, become valuable to overcome these difficulties. EZ_MIPS was designed to be one of these tools. EZ_MIPS is, simply, a simulation of the datapaths that are discussed in computer architecture courses. EZ_MIPS makes it exciting to study on and easy to understand MIPS datapaths. For each datapath (Single Cycle Datapath, MultiCycle Datapath, and Pipelined Datapath [5]) there is a dedicated edition of EZ_MIPS.

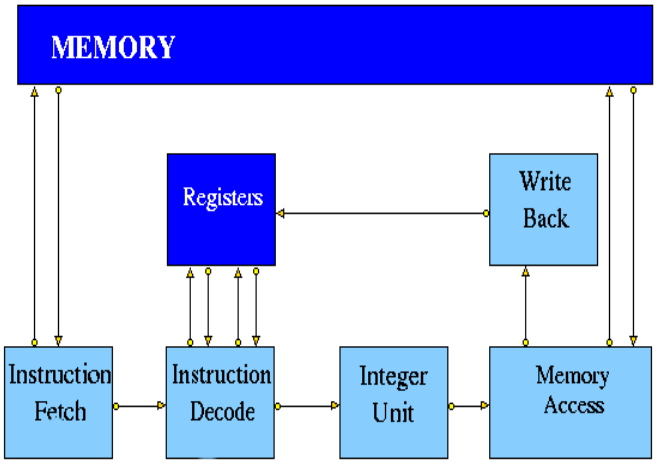


**Figure 1. Typical MIPS Pipeline**

As visualized in

Figure 2 a MIPS processor can overlap the execution of several instructions, potentially leading to big performance gains by using pipelined approach. MIPS composed of several parts that process the instructions in a pipelined manner. The fundamental parts are instruction memory, registers, arithmetic logic unit, forwarding unit, data memory, and multiplexers [6]. The architecture composed of 32 general purpose registers. MIPS registers take both less time to access and have a higher throughput than memory The data types used in MIPS are byte, halfword (2 bytes), and word(4 bytes). As seen from

Figure 1 the instruction is fetched by MIPS, then it is decoded according to instruction type and the necessary values are read from registers. Then in integer unit (ALU) the execution of instruction is done. ALU is used for data and address arithmetic, so load and store instructions are processed by ALU before sent to memory access unit and memory. Also, ALU is responsible from the additions required for integer test and relative branch instructions so that the Memory Access Unit executes branches. Lastly, if required the memory access is called and the results from both arithmetic/logic and load instructions are written to registers by write back unit [7].

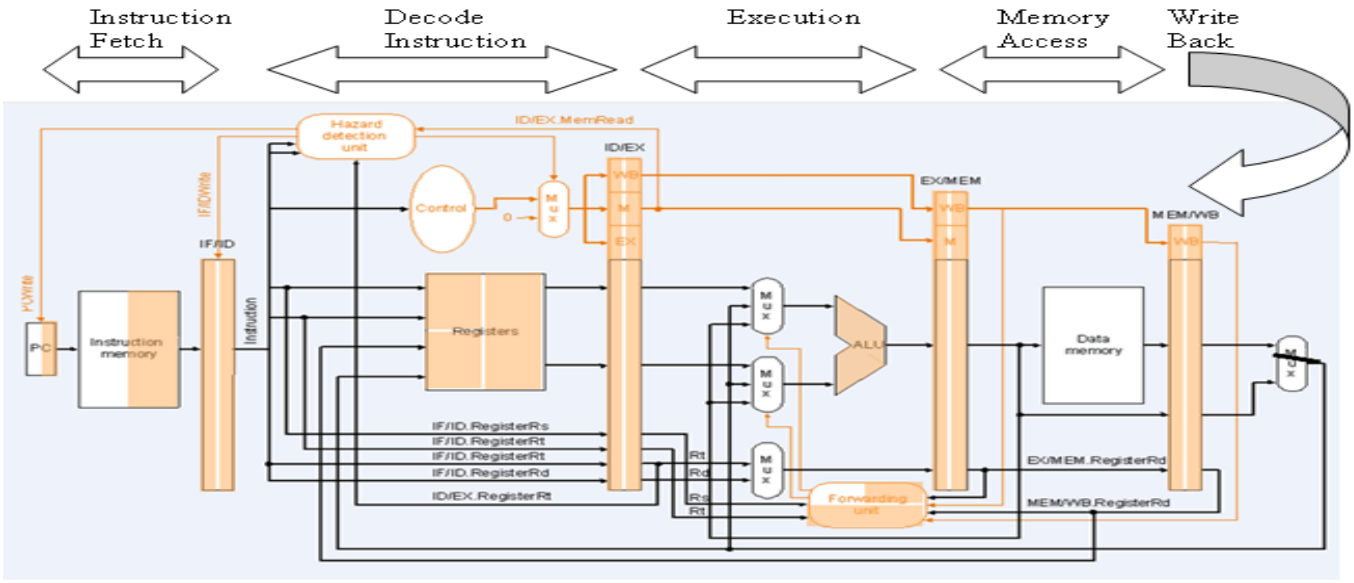


**Figure 2. MIPS Visual Datapath [Taken from EZ_MIPS]**

The projection of

Figure 1 is implemented in

Figure 2 which is implemented by EZ_MIPS. During the execution of instructions, all these units are mostly in busy state because of pipelined structure. The instructions are executed as shown in

Figure 3.

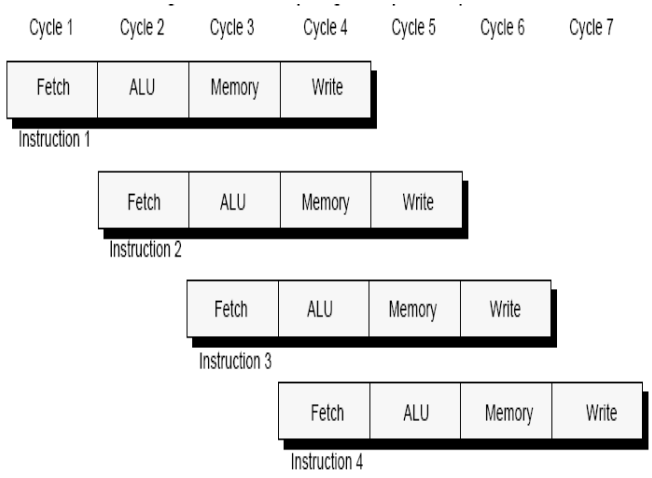


**Figure 3. Pipelining of instructions via MIPS[8]**

Lastly, EZ_MIPS gives the full details of all cycles going in all units of MIPS by datapath visualization tool, data memory and register values. Also the ALU execution can be monitored within any time in execution of programs via tooltips and external windows. Also the simulator has a variety of look and feel options that provides more appealing interfaces.

The main features of EZ_MIPS are as follows:

- Control of execution speed, including single step to variable speed

- Thirty-two registers and 1KB memory unit visible at the same time, selectable via tables,
- "spreadsheet" (WYSIWYG) modification of values in registers and memory and instruction sets,
- Selection of data value display in binary, decimal or hexadecimal formats
- "surfing" through memory using memory window
- Toolbar icons for every menu item
- Visual Datapath simulator with various tooltip options to simulate instuctions at different speeds.
- Different look-and-feel options.
- Have opportunity to start execution of instructions from selected instruction step of instruction set.
- Modification of instructions read from file in anytime.
- Showing the status of program in information bar.
- Profiling of environment variables in simulation (new feature comes from AOP approach)

After defining the properties and features of simulator, the next sections mention about analysis of its design from various perspectives.

## 3. Restructuring with Design Patterns

When we look at the current implementation of EZ_MIPS there are some cases where best practices of engineering have not been adopted. By using the reverse engineering approaches, some of the patterns will enhance the usability of codes and makes system more stable, less coupled and cohesive. In these kinds of situations improving the quality of design is done by adopting

existing design patterns to these problematic parts. In this section, the problems with current design or the demands needed about current design are analyzed and adopted patterns are explained in detail.

For reengineering of patterns, we followed a migration strategy which is an incremental approach widely used in reengineering of current systems. The strategy starts with analyzing the system from the object-oriented perspective, finding the demands and problems in current design and prototyping the target solution for the defective parts and incrementally migrating from current to target solution. In this process, we always have a running version in all steps of migration. And with each unit of change in design we tested the system's stability. After attaching new patterns we have more cohesive and less coupled code. In the following lines, we present our changes by using existing patterns.

## 3.1. Abstract Factory Pattern

In EZ-MIPS simulation, we provide to users to change GUI themes easily. As the simulator aims to teach MIPS structure easily, we want to user can change the theme of GUI that he/she feel more appealing and comfortable with it. Considering the desired functionality, each time we want to create different types of themes it would require a considerable amount of afford to add new theme to simulator. By using the Creational Design Patterns, the amount of afford is going to decrease seriously. Also, our main concern is creating different themes with separating the details of implementation of a set of themes from the execution of system.

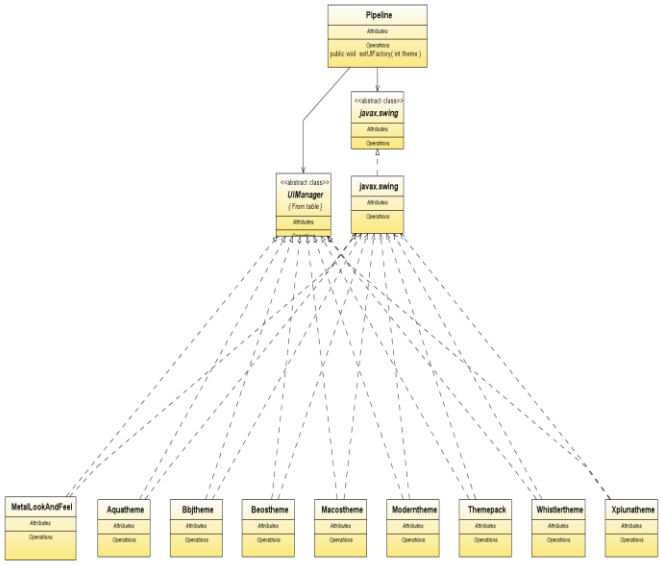


**Figure 4. EZ_MIPS Abstract Factory Implementation**

Abstract Factory promises us the encapsulation of a group of individual factories that have a common theme and the usage of generic interface to create concrete objects that are part of the theme [9]. By using Abstract Factory pattern, the client does not know the theme of concrete objects since it uses only the generic interfaces of their products. We provide to use a set of theme factories by defining UIManager class that declares an interface for creating different types of themes. Thus, by only selecting the specific theme option user will stay independent of the implementation details how it is performed. Figure 4 shows the class diagram of the abstract factory design of the classes.

## 3.2. Observer Pattern

The observer pattern (sometimes known as publish/subscribe) is a software design pattern in which an object maintains a list of its dependents and notifies them automatically of any state changes, usually by calling one of their methods [9].

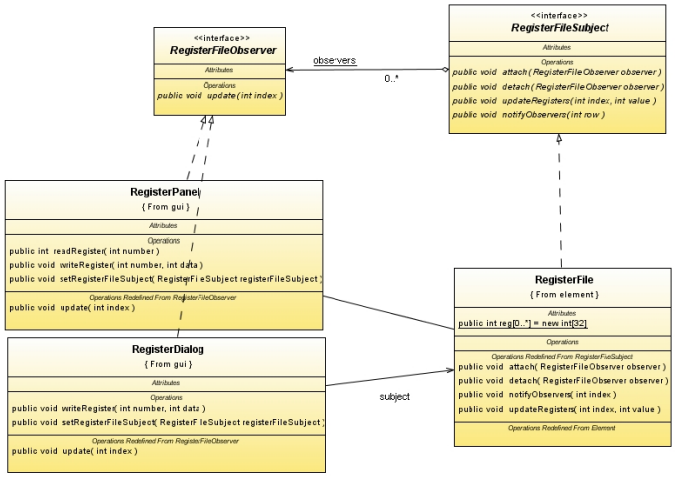


**Figure 5. EZ_MIPS Observer Pattern**

In EZ-MIPS simulation, we have two windows (Register Panel and Register Dialog) to show the value of registers at the time of the current instruction set. Throughout the step by step execution of instruction set, these two windows should be notified about the new values on the registers. To provide this functionally, we use Observer Pattern. Before applying Observer Pattern, we face many difficulties. For each element in MIPS, we should keep tracks and updates the associated windows values. This cause to hard to manage to code, when there is a need for modification. However, in this pattern there are two components subjects and observers [9]. Subjects

represent the core data and when a modification is occurred on the data, subject notifies all the attached observers to update themselves.

Figure 5 shows the diagram of pattern in our design.

### 3.3. State Pattern

In EZ_MIPS simulation environment, users have the chance of tracing the given instruction set by using three kinds of number states. These states are: binary state, decimal state, and hexadecimal state. The number states are implemented in all elements of EZ_MIPS Simulator. Without this pattern we have to add all the elements switch statements, which makes code less readable. To solve this problem we proposed a state pattern which is a family of behavioral pattern. As Gamma mentioned, this is a clean way for an object to partially change its type at runtime which is exactly our case [9]. The modified UML diagram is shown in Figure 6.

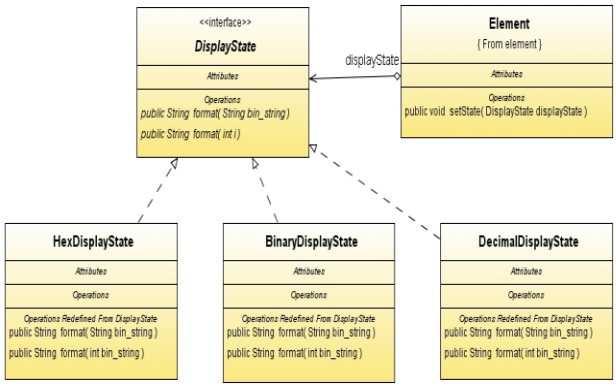


**Figure 6. EZ_MIPS Sample State Pattern**

## 4. Introducing Aspects

Aspect-oriented software development is a relatively new approach to software development and design, which points out limitations of object-oriented software development. Aspect-oriented software development adds crosscutting concerns to address those features that cut across modules [10].

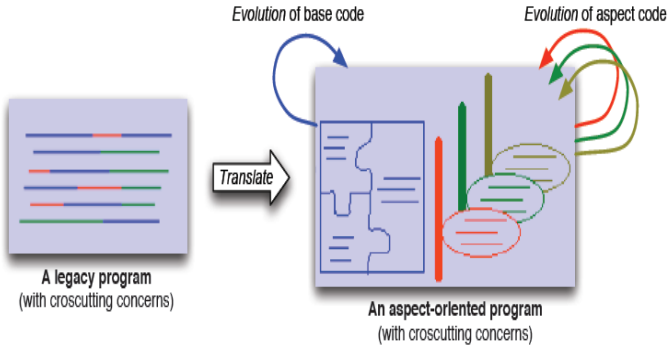


**Figure 7. Cross fertilization of software code with AOP[11]**

As visualized in Figure 7, in order to perform aspect-oriented development, we have to translate existing code into their aspect-oriented equivalents and re-factor them continuously [11]. This refactoring process starts with identification of concerns scattered in implementation, redesigning of them using AOP principles and implementing these concerns.

The reverse engineering of EZ_MIPS implied that there are some crosscutting concerns and scattering code in the simulator. Although we apply OOP paradigms to EZ_MIPS simulation, we can not exactly provide encapsulation and separation of concerns. These concerns should be removed to provide modularity and clarity of the implementation.

In this section, we will mention two kinds of aspects; development and production aspects. Development aspects which can be easily removed from production builds aims to the development process easier and faster. Production aspects are intended to be included in the production builds of an application. Production aspects insert functionalities to an application rather than only adding more visibility of the internals of a program [12].

### 4.1. Policy Enforcement

The policies and contracts are the core components in the project's life cycle. These rules make all the development more concrete, organized, and systematic. So the enforcement policies are necessary for robust development and coordination. In our reengineering process, we have defined some rules to improve code in a team. By these rules, the development process becomes more comfortable. Some of these implementation policies are as follows:

1. There should not be no System.out.{print*(),error*()} in the project. Because this project is GUI based, there is no

meaning of these calls. We give warning when there is a call.

2. The Class names, method names and variable names should be compliant with Java's coding conventions. This is for making code more readable.

Another problem is some of the object creations should not be called from some modules. This may include the creation and execution of MIPS elements being in one object (e.g. DataPath class). This is a powerful mechanism for development of project. Enforcing this constraint makes the program more stable and unnecessary or wrong creation of MIPS Datapath elements will not be allowed. Note that these policies and enforcements are regarded as development aspects which are so important in the process of project implementation lifecycle. Here is the following sample showing policies that are discussed:

```
pointcut consoleout():(get(* System.out) || get(* System.err)) &&
!within(aspects..*);
public pointcut classWithLowerCase() :
        initialization( pipeline..a*.new(..))  ||
        initialization( pipeline..z*.new(..));
public pointcut methodWithUpperCase() :
        execution(* pipeline..*+.A*(..))  ||
        execution(* pipeline..*+.Z*(..));
declare warning :
    consoleout() :
    "output to console not allowed. Consider using logger.";
declare warning :
        classWithLowerCase() :
        "class names should not begin with lower case letter.";
declare warning :
        methodWithUpperCase() :
        "method names should not begin with upper case letter.";
```

## 4.2. Consistency Verification

Another crosscutting concern is for providing the correctness of execution of each element. Each time we need to control whether the code of element fine or not, we must put some checker into pre and post conditions of each operations. So this means each time we want to check if the retrieved input is correct, and the result of the operation is done correctly, we must put the validation control into these methods or operations. This is why we implement an aspect called Consistency Verification Aspect to encapsulate common issues associated with this concern.

In the system all the wires are implemented as an object. For example ALU unit must get 2 objects created by 2 MUX elements shown in Figure 2. Also the inputs must have some characteristics (such as the format of incoming elements). As this example other elements have some pre and post conditions (object creation, object format of input and output elements). These concerns are done by consistency verification. And this aspect eases the debugging and verification of the flow of simulation. Note that the example that is discussed is implemented in the following aspect:

```
pointcut aluWork(ALU alu ):this(alu) &&
(execution(void ALU.work(boolean)));

Wire aluOut = WireSet.wires[35];
Wire aluZeroOut  = WireSet.wires[36];
Wire aluInA = WireSet.wires[32];;
Wire aluInB = WireSet.wires[34];;

before  (ALU alu ): aluWork(alu) {
    // null check
    if ( aluInA == null || aluInB == null )
        throw new IllegalArgumentException("ALU inputs are not valid.");
    // bounds check
    if ( aluInA.getValue().length() > 32 || aluInB.getValue().length() > 32 )
        throw new IllegalArgumentException("ALU inputs are is out of bounds.");
}

after  (ALU alu ): aluWork(alu) {
    // null check
    if ( aluOut == null || aluZeroOut == null  )
        throw new IllegalArgumentException("ALU output is not valid.");
    // bounds check
    if ( aluOut.getValue().length() > 32 )
        throw new IllegalArgumentException("ALU output is out of bounds.");
    // bounds check
    if ( aluZeroOut.getValue().length() > 1 )
        throw new IllegalArgumentException("ALU Zero output is out of bounds.");
}
```

Note that there are two verification points, before ALU execution process and after ALU execution process. This aspect makes the input and output bounds checking and validations.

## 4.3. Tooltip Concern

Another crosscutting concern that we have faced during reengineering is originated from the need to inform user the current state of each element in EZ_MIPS simulation. To do so, we show a tool tip panel that represents previous and current status of element at any clock cycle. It interests all or the elements current state. Some of these elements are instruction memory, ALU, registers, hazard detection unit, data memory, forwarding unit. The tooltip includes information about the inputs and

outputs of each unit, the inner status of each element (such as registers and memory segments), and execution status of elements. This makes the maintenance of tooltip harder, if performed with OOP principles. Also when some updates are necessary in tooltip, these modifications affect all parts of various elements in design and implementation which makes maintenance and development harder. Figure 8 shows the effects of this crosscutting concern over the implementation. Here, because the tool tip concern and execution of MIPS concern are indeed separate concerns, these two concerns should be separated to carry out the modularity.

```
if (control == "010") {       // addition
     result = a + b;
     tooltip = "<html>" + colorA + a + End + " + "
     + colorB + b + End + " = " + colorR + result
     + End + ' ';
 } else if (control == "110") {     // substruction
     result = a - b;     //
     tooltip = "<html>" + colorA + a + End + " - "
     + colorB + b + End + " = " + colorR + result
     + End + ' ';
 }
  else if (control == "000") {      // and
     result = a & b;
     tooltip = "<html>" + colorA + a + End + " end
     " + colorB + b + End + " = " + colorR +
     result + End + ' ';
 } else if (control == "001") {     // or
     result = a | b;
     tooltip = "<html>" + colorA + a + End + " or
     " + colorB + b + End + " = " + colorR +
     result + End + ' ';
 }
```

**Figure 8. Crosscutting Code Segment for Tooltip**

To separate ToolTip concern from business logic, we propose to use ToolTip Aspect shown in Figure 9. In this aspect, we define a pointcut called to captures calls to work method of ALU element. After that, we define an advice to simply set the tooltip of ALU element. By the help of AOP technology, we have differentiated the crosscutting tooltip concern from execution logic. To do so, we increase the modularity of the system by encapsulating the aspect code.

```
public privileged aspect TooltipAspect {
        pointcut adderwork(ALU alu ):this(alu) &&
          (execution(void ALU.work(boolean))|| execution(* Element.setState(..)));

        after(ALU alu ): adderwork (alu)
  {
                if(alu.output == null ){
                        alu.setToolTipText("ALU....");
                        return;
                }
                tooltip = "<html>" + colorA + formatted_a + End + op + colorB
                                + formatted_b + End + " ="
                                + colorR + formatted_out + End
                                + ' ' + "<br>aspect ZeroOut: " + colorR
                                + (wireSet.wireSet[36].getValue())  + "</font>"
                                + "</html>";

                alu.setToolTipText(tooltip);

        };
}
```

**Figure 9. Example Tooltip Aspect**

## 4.4. Profiling of Simulation Variants

The next crosscutting concern is for profiling the execution of the each elements of the simulation. To measure the totality of element's behavior from invocation to termination, to make statistical summary of the events observed and to see ongoing interaction, we need a profiler that crosscuts all elements in the EZ_MIPS. Thus the same separation of concerns also is need for the profile concern. Considering this, we decided to develop an aspect called Profiling Aspect. With the help this aspect, there is no need to make any modification of implementation of elements in the simulation. We just implement the desired profiling requests for each element in Profiling Aspect. Figure 10 represents a sample profile window at the end of the execution of EZ_MIPS.

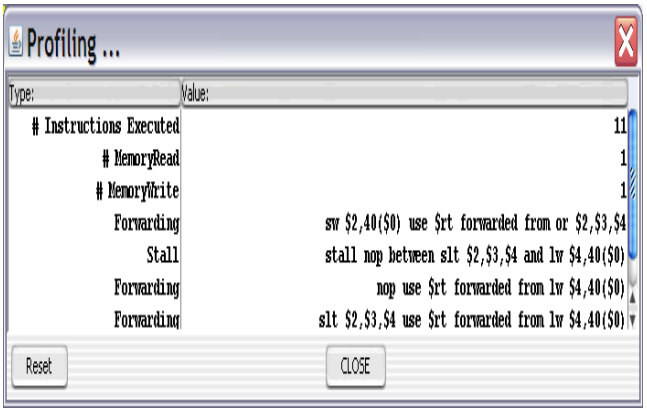


**Figure 10. Example Profiling Aspect**

The implementation details of profiling is done as follows:

```
pointcut endOfMain():
  (execution(* Main.main(..)));
pointcut intructionMemoryWorks(InstructionMemory instr, boolean clockEdge):
    this(instr) && args(clockEdge) &&
execution(void InstructionMemory.work(..));
pointcut dataMemoryRead(DataMemory dataM):
    this(dataM) && execution(* DataMemory.readMemory(..));
pointcut dataMemoryWrite(DataMemory dataM):
    this(dataM) && execution(* DataMemory.updateMemory(..));
pointcut forwardingUnitWorks(ForwardingUnit forwardingUnit):
    this(forwardingUnit) && execution(void ForwardingUnit.work(..));
pointcut hazardDedectionUnitWorks(HazardDedectionUnit hazardDedectionUnit):
    this(hazardDedectionUnit) && execution(void HazardDedectionUnit.work(..));
```

```
void updateInstructionsExecutedCount() {
    profilingMemory.numOfInstructionsExecuted++;
    profilingMemory.model.fireTableDataChanged();
}
void updateMemoryReadCount() {
    profilingMemory.numOfMemoryRead++;
    profilingMemory.model.fireTableDataChanged();
}
void updateMemoryWriteCount() {
    profilingMemory.numOfMemoryWrite++;
    profilingMemory.model.fireTableDataChanged();
}
void addMessage(String type, String msg) {
    profilingMemory.addMessages(new ProfileMessage(type, msg));
}
void processForwardingMessage(ForwardingUnit forwardingUnit) {
    String instr00 = WireSet.labelSet[3].getText();
    String instr01 = WireSet.labelSet[5].getText();
    String instr10 = WireSet.labelSet[4].getText();
    if (forwardingUnit.mux3a.getValue().equalsIgnoreCase("01")) {
        addMessage("Forwarding", "\"" + instr00+ "\" use $rs forwarded from \"" + instr01 + "\"");
    } else if (forwardingUnit.mux3a.getValue().equalsIgnoreCase("10")) {
        addMessage("Forwarding", "\"" + instr00+ "\" use $rs forwarded from \"" + instr10 + "\"");
    }
    if (forwardingUnit.mux3b.getValue().equalsIgnoreCase("01")) {
        addMessage("Forwarding", "\"" + instr00+ "\" use $rt forwarded from \"" + instr01 + "\"");
    } else if (forwardingUnit.mux3b.getValue().equalsIgnoreCase("10")) {
        addMessage("Forwarding", "\"" + instr00+ "\" use $rt forwarded from \"" + instr10 + "\"");
    }
}
void processHazardDedectionMessage(HazardDedectionUnit hazardDedectionUnit) {
    String instr00 = WireSet.labelSet[2].getText();
    String instr01 = WireSet.labelSet[3].getText();
    if (hazardDedectionUnit.mux.getValue().equals("1")) {
        addMessage("Stall", "stall nop between " + instr00 + " and "
                + instr01);
    }
}
```

```
after(InstructionMemory instr, boolean clockEdge):
    intructionMemoryWorks ( instr, clockEdge){
    updateInstructionsExecutedCount();
}
after(DataMemory dataM):
    dataMemoryRead ( dataM ){
    updateMemoryReadCount();
}
after(DataMemory dataM):
    dataMemoryWrite ( dataM ){
    updateMemoryWriteCount();
}
after(ForwardingUnit forwardingUnit):
    forwardingUnitWorks ( forwardingUnit ){
    processForwardingMessage(forwardingUnit);
}
after(HazardDedectionUnit hazardDedectionUnit):
    hazardDedectionUnitWorks ( hazardDedectionUnit ){
    processHazardDedectionMessage(hazardDedectionUnit);
}
```

In these code segments first pointcuts are introduced which will crosscut all execution parts of EZ_MIPS: data memory, instruction memory, forwarding unit, and hazard detection unit. Then the necessary aspect codes are added for profiling of simulation elements.

## 6. Discussion & Conclusion

As far as we explained in this paper, we have improved existing code quality with object-oriented and aspect-oriented thinking. In the process of restructuring the existing design and implementation we have performed the following two operations:

1. Improved the code quality in order to make code more cohesive, modularized and less coupled. This process is regarded as software refactoring, which aims developing the internal structure of a program without changing its external behavior. This process includes adding new design patterns to our existing design and finding existing concerns or tangling code segments scattered over objects such as tooltip concern.
2. Add new features using aspect-oriented techniques. In our project, we insert profiling feature for the distinct points of simulation in order user to monitor the MIPS behavior better.

Within this process we have visualized that aspect-oriented development eases the implementation of crosscutting concerns and improves reuse of code which is a fundamental principle of software engineering. As explained in [13] and [14] AOP provides the following advantages:

- The components of software are better modularized.
- The modularization provides better separation of concerns and a clearer and cohesive responsibility of the system components, and therefore the end products are better maintainable and reusable.
- These advantages lead to reduction of time-to-market by a better modularized and simpler design, resulting in a reduction of costs.

To conclude, we had a chance to make our hands dirty with experience on AOSD. Also by practical usage of aspects we give a proof of concept that aspect-oriented development will ease the implementation of crosscutting concerns, which are regarded as secondary requirements for modules. We also improved the quality of code by eliminating tangling code segments. In the development

process with AOP techniques, we experienced that the deployment time of the program takes time as addition of new aspects. The only problem may be this concern which makes the development a little bit slower. However in reengineering process AOP development is faster than OOP development, which needs special analysis and management of scattered codes and crosscutting concerns.

## Acknowledgement

Thanks to Asst. Prof. Dr. Bedir Tekinerdogan, who organized the "Third Turkish Aspect-Oriented Software Development Workshop" (TAOSD 2008), for generously sharing his comments and feedback with us.

Also thanks to our class mates and all the participants of the TAOSD 2008, with whom we had a most beneficial workshop, sharing experiences related to AOP.

## References

[1] C. Timothy, "*Object-Oriented Software Engineering"*, Lethbridge and Robert Laganiere, McGraw-Hill.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C.Maeda,C. Opes, J. M. Loingter, J.Irwin, *"Aspect-Oriented Programming"*, In Proceeding of ECOOP'97, Springer-Verlag LNCS, 1241

[3] MIPS Architecture. http://en.wikipedia.org/wiki/MIPS_architecture. Last accessed 15 December 2008.

[4] T. Langens *et al*, "MIPS Architecture",2000.

[5] A. D. Patterson, J. L. Hennessy, "Computer Organization and Design: The Hardware/Software Interface". Third Edition. Morgan Kaufmann. 2004.

[6] S. Jonathan, "http://gicl.cs.drexel.edu/people/sevy/architecture/MIPSRef(SPIM).html." last accessed 15 December 2008. The hardware / software interface

[7] HASE Project, "Simple MIPS Pipeline", Institute for Computing Systems Architecture, School of Informatics, University of Edinburgh

[8] MIPS Technologies Inc**.** MIPS32® Architecture For Programmers, "Introduction to the MIPS32® Architecture"

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley Longman Inc., 1995.

[10] B. Sue, H. Mark, *"Aspect-oriented Software Development: Towards a Philosophical Basis"*, Technical Report. Department of Computer Science, King's College London. 2006.

[11] T. Mens, K. Mens, T. Tourw´e. *"Software Evolution and Aspect-Oriented Programming"*. ERCIM. 2004.

[12] Introduction to AspectJ http://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html. Last accessed 18 December 2008.

[13] R. Laddad, AspectJ in Action, "Practical Aspect-Oriented Programming",Manning Publications Company, 2003.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and G. Griswold, William, *"Getting Started with AspectJ,"* ACM Communications, vol. 44, no. 10, pp. 59–65, 2001.