İSTANBUL TEKNİK ÜNİVERSİTESİ
BİLGİSAYAR VE BİLİŞİM FAKÜLTESİ

İTÜ-RISC İşlemcisi Benzetim Programı

Bitirme Ödevi

Umut BALKIŞLI
040100805

Bölüm: Bilgisayar Mühendisliği
Anabilim Dalı: Bilgisayar Bilimleri

Danışman: Dr. Feza BUZLUCA

Haziran 2013

İSTANBUL TEKNİK ÜNİVERSİTESİ
BİLGİSAYAR VE BİLİŞİM FAKÜLTESİ

İTÜ-RISC İşlemcisi Benzetim Programı

Bitirme Ödevi

Umut BALKIŞLI
040100805

Bölüm: Bilgisayar Mühendisliği
Anabilim Dalı: Bilgisayar Bilimleri

Danışman: Dr. Feza BUZLUCA

Haziran 2013

Özgünlük Bildirisi

1. Bu çalışmada, başka kaynaklardan yapılan tüm alıntıların, ilgili kaynaklar referans gösterilerek açıkça belirtildiğini,
2. Alıntılar dışındaki bölümlerin, özellikle projenin ana konusunu oluşturan teorik çalışmaların ve yazılım/donanımın benim tarafımdan yapıldığını bildiririm.

İstanbul, Haziran, 2013

Umut BALKIŞLI

İmza

İTÜ-RISC İşlemcisi Benzetim Programı

(ÖZET)

Yapılan çalışma, RISC mimarisi konusunda bilgi sahibi olmak isteyen kişiler için uygun bir benzetim ortamı sunarak eğitim sürecini kolaylaştırmayı amaçlamaktadır. Uygulama sayesinde bilgisayar mimarileri, iş hattı yapıları ve farklı iş hatlarının kullanılmasıyla ortaya çıkan performans farklılıkları gerçek donanımlara ihtiyaç duymaksızın gözlemlenebilecektir. Gerçek donanımlar yerine benzetim programının kullanılması, tüm öğrenciler için gerçek RISC tabanlı donanımlar bulma maliyetinin yanı sıra laboratuvar kurma ve geliştirme ortamı hazırlama maliyetine de çözüm getirecektir. Böylece kişisel bilgisayar sahibi olan insanlar ek bir gereksinim duymadan, yalnızca uygulamayı kullanarak rahatlıkla çalışmalarını yapabileceklerdir.

Projede, kullanıcıların uygulama geliştirebilmesi için kullanacakları güçlü bir editör sağlama, bir RISC tabanlı mikroişlemci davranışını modelleme ve yazılan uygulamaları makine diline çevirerek sözde mikroişlemci üzerinde çalışmasını sağlayacak bir derleyici hazırlama gibi problemler üzerinde çalışılmıştır. Bu çalışmaların sonucunda .NET çatısında C# diliyle kullanıcılar için uygun bir editör, farklı iş hattı yapılarına kolay değişiklik yapılabilecek bir RISC mikroişlemci yapısı, değiştirilmeye uygun bir komut seti, bellek, saklayıcılar, güçlü bir hata kontrol birimi ve derleyici hazırlanmıştır.

A Simulation Software for ITU-RISC Microprocessor
(SUMMARY)

This paper describes a study on compiler design and microprocessor simulation. Research has been made on RISC microprocessors. Using Scintilla.NET and DockPanel Suite components a development and simulation environment has been developed under C#2.0 and .NET framework.

There is no actual compiling and linking stages in complier and also there is no actual microprocessor to run commands on it. System is behaving like a compiler and microprocessor. Because of that the system is called as “simulation”. These two components are the base of this project. When user wants to run the written program compiler will check for errors than codes will be converted to binary system for microprocessor. After that operation commands will be loaded to microprocessors iteratively to run. These commands can run on microprocessor but without memory, registers and status flags there is no meaning to do that. So, these are the other components of a microprocessor simulation software.

Compiler

Dealing with machine level languages is really hard for programmers. Without keywords and variables writing or maintaining a big program is almost impossible. Programmers use programming languages and compilers to solve that problem. Programming languages has instruction sets and these instructions are named respectively to their ability. Programmers write their program with a language, than compiles it with a compiler. Compiler gets written code and converts it to machine level. After that conversion program can run on microprocessor. Of course there may be some syntax errors when writing program, compiler must also detect errors and warn user. If there is an error, than compiling must be canceled.

In this study, a simple compiler has been developed. This compiler takes the program which was written by a user and checks for syntax errors. To do that, it first removes comment, then finds *op-code* and parameters. If there is nothing wrong with the command, compiler converts it to binary format. Original commands and binary converted commands are stored in a list. At last compiler returns this list for pipeline to use.

Register

There are usually many registers in RISC architecture. Because there is no direct memory-to-memory commands or stack for every subroutine call. Instead of them there is sliding window mechanism. Here, there are three types of register: global, local and shared. Global registers can be accessed by every subroutine. Local registers can be accessed by only current subroutine. Shared registers are used to pass parameters between subroutines. And there is a current window pointer which holds the information of current subroutine number.

In this study, there are 138 registers. For every window 10 global registers, 12 shared registers (6 for previous, 6 for next subroutine) and 10 local registers. Window length is,

local registers + total shared registers + global registers = 32

There can be 7 subroutine calls for that system. Usually, if there are more than seven subroutine calls, next window's values can be saved to stack. That makes system work under this circumstances. But for that simulation, there is a restriction.

Microprocessor

According to their instruction structures there are two different microprocessors, CISC (Complex Instruction Set Computer) and RISC (Reduced Instruction Set Computer). Each has some specialties, advantages and disadvantages. Here, some differences between CISC and RISC are shown in table below.

CISC	RISC
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes, high cycles per second	Low cycles per second, large code sizes
Transistors used for storing complex instructions	Spends more transistors on memory registers

In this study, RISC architecture is used. Instruction set has 28 different command and is inspired from Berkeley RISC-I. Instruction set is not static, users may change commands, their behaviors or binary definitions from XML file. That makes system more flexible for instruction sets. Users may add/change/delete commands if they want.

Pipeline

Improving effectiveness of resources is an issue on computer science. For microprocessors pipeline structure is used to do that. Commands can be run one by one on microprocessor but instead of holding CPU for a long time, commands are partitioned to atomic sentences. We have *n* layer pipeline which means every command has *n* atomic sentences and on microprocessor there can be *n* different command at time *t* (where $t>=n$). For each microprocessor cycle one atomic sentence will run for every command. So instead of waiting for *n* clock cycle for every command, microprocessor can finish a command in every cycle after time *t* (where $t>=n$).

In this study, system allows different kind of pipelines. Initially, three layer pipeline is implemented. Layers are respectively, *Instruction Fetch*, *Decode-ALU Operation* and *Data*.

Syntax errors are named as compile-time errors. There is another kind of error in pipeline, run-time errors or hazards. Compiler cannot detect hazards because compiler does not try to understand program's behavior. It just converts commands to binary format. Definition of hazard is trying to reach same register at the same cycle. For three layer

pipeline, that occurs when two load/store commands are back to back and using same registers or after one load/store command another register command with same register. This simulation system detects hazards when they occur and adds a NOP (*No-Operation*) command to solve.

Memory

For that study, global memory system is used which means user can set memory values before running program and can reach them after program is over. That usually helps users about setting initial memory values. There is also *clear* option at anytime for memory, registers and status flags.

System

System has a flexible structure. Application language support, instruction set support and pipeline support are main indicators. By using XML files users can add new application languages. All fields are reading their texts from XML file according to language. Initially there are two different languages (Turkish and English). Just like in application language, users also can change instruction set from XML file. If they want they also can add new commands. Commands are not static in software. Every command has a definition in XML file, pipeline gets that definition creates a new instance of C# application with that definition and runs it on that instance. Than it get return value as a result. That system allows more flexibility for users to create their own commands.

Some system parameters like command length, memory size, register count, default start label are also defined in same XML file like instruction set and conditions. Conditions are easily interested in status flags. So from XML file we can add definitions for that conditions like instructions. And with same strategy for conditions pipeline will create a new instance of C# application and run it under that. Return value will be condition's result. The only difference between command running and condition running is that, conditions needs an *if* statement while instructions does not.

Other than back-end system, there is and front-end development with Scintilla.NET and DockPanel Suite open source projects. Scintilla.NET provides a powerful editor with highlighting and marker functionality. Markers are used to show current line on pipeline and breakpoints. DockPanel Suite is used to make forms look like Visual Studio style. And screen is used effectively with that open-hide-change location forms. It also gives an impression of looking *cooler* and *more professional* to users.

İÇİNDEKİLER

1. GİRİŞ

Performansı arttırmaya dayalı çalışmalara günümüzde her zamankinden fazla önem verilmektedir. Bu gelişim göz önüne alındığında performansın önemi gelecekte de artmaya devam edecektir. Bilgisayar mimarileri, bir bilgisayar sisteminin performansında oldukça önemli bir role sahiptir. Benzer bir amaç için farklı bilgisayar mimarileri kullanılabilir ancak bunlar arasında işlemcinin yapısı, hızı, sistemin çalışma şekli, bellek organizasyonu gibi faktörleri göz önüne alarak en uygun sistemi seçmek performansta büyük etkilere neden olacaktır.

Bilgisayarların işlem yapma yeteneği için kullandığı Merkezi İşlem Birimi (MİB) komut yapılarına göre *CISC (Complex Instruction Set Computer)* ve *RISC (Reduced Instruction Set Computer)* olarak ikiye ayrılır.

CISC merkezi işlem birimleri, geniş ve karmaşık bir komut setine sahiptir. Çok fazla adresleme kipi vardır. Komut yapıları değişken uzunluktadır, bu nedenle komut çözme işlemi karmaşıktır. Daha fazla bellek erişimine dayalı çalışmayı benimsemiştir. Komut setinin ve adresleme kiplerinin geniş olması daha yüksek düzeydeki programlama dillerine (ör: C) benzemesini sağlamış ve programcı açısından uygulama geliştirmeyi kolaylaştırmıştır. RISC merkezi işlem birimlerinde ise çok daha küçük bir komut seti bulunur (mikro işlemler). Bellek erişimli komutlar bulunmaz. Az sayıda ve basit adresleme kipleri vardır. Ayrıca komut uzunlukları sabit olduğu için komut çözme işlemi daha kolaydır. **[1][2][3]**

CISC ve RISC mimarilerinin ikisinde de zaman zaman görülebilen ancak RISC için daha önemli olan bazı özellikler de vardır. Kayan saklayıcı yapısı sayesinde altprogram çağrılarında verileri belleğe kaydetmeden eski verileri de parametre olarak geçerek programların çalışması sürdürülebilir. Kayan pencere yapısının etkili olabilmesi için de yüksek sayıda saklayıcıya sahip olmak önemlidir. Komut ve veri belleklerinin ayrı olması da bellekten veri erişiminin hızlanması için önem taşımaktadır. **[1][2][3]**

Bunun yanında bilgisayar sistemlerinin çok önemli performans faktörlerinden biri de paralel veri işlemedir. Günümüz bilgisayar sistemlerinin tümü paralel veri işleme için "iş hattı" yapısını kullanmaktadır. Bir komutun yürütülmesi için gerekli olan bazı işlemler vardır. İş hattı yapısında, kullanılan modele göre alt görevler tanımlanır ve iş hattına komut geldiğinde her saat işaretinde komutun bir alt görevi gerçekleştirilir. Bu sayede k adet katmanlı bir iş hattında belli bir t anında k adet komuta ait alt görevler gerçekleştirilmiş olur. Segman sayısını arttırmak ve bir komutu çok küçük parçalara bölmek fiziksel nedenlerden ötürü verimliliği arttırmayacağı için sistemdeki gecikmelerin iyi hesaplanmış olması ve buna göre bir iş hattı tanımlanması gerekmektedir. **[1]**

1.1 Proje Kapsamı

RISC işlemciler daha basit bir yapıya sahip olduğu için günümüz kişisel bilgisayarlarında pek tercih edilmemektedirler. Ancak mobil teknolojinin gelişmesiyle birlikte RISC işlemcilerin kullanımı hızla artmaktadır. Bu nedenle yapılan bitirme çalışmasında RISC mimarisinin kullanımı tercih edilmiştir. Projede, kullanıcıların uygulama geliştirebilmesi için kullanacakları güçlü bir editör sağlama, bir RISC tabanlı mikroişlemci davranışını modelleme ve yazılan uygulamaları makine diline çevirerek sözde mikroişlemci üzerinde çalışmasını sağlama gibi problemler üzerinde çalışılmıştır. Bu çalışmaların sonucunda kullanıcılar için uygun bir editör, farklı iş hattı yapılarına kolay değişiklik yapılabilecek bir RISC mikroişlemci yapısı, değiştirilmeye uygun bir komut seti, bellek, saklayıcılar, güçlü bir hata kontrol birimi ve derleyici hazırlanmıştır.

1.2 Bu Alanda Yapılmış Çalışmalar

Bu alanda daha önceden yine İTÜ Bilgisayar Mühendisliği öğrencisi olan Gökhan Akın Şeker'in bitirme çalışması**[4]** yer almaktadır. Bu çalışmada,

- Berkeley RISC-I mimarisi,
- 4 katmanlı iş hattı,
- derlenme sonrası (yalnızca yazılan kod ile) erişilebilen bellek,
- 138 adet saklayıcı ile kayan pencere yapısı,
- sabit komut seti,
- sabit uygulama dili,
- oldukça basit bir uygulama geliştirme editörü,

kullanılmıştır. Bu çalışma özellikle 3 katmanlı iş hattı desteğini veremediği için eğitim amaçlı kullanımında problemler yaşanmaktadır. Bu ve diğer eksiklikler göz önüne alınarak hem kullanıcı dostu hem de değişimlere karşı sorun çıkarmayacak, farklı koşullarda çalışmasını sürdürebilecek bir sistem tasarlanması hedeflenmiştir.

1.3 Çalışma ve Sonuçlar

Benzetim programını gerçeklemek için geliştirme ortamı olarak Microsoft Visual Studio 2010, programlama dili olarak ise C# 2.0 kullanılmıştır. Bunun dışında Scintilla.NET açık kaynak yazılımı sayesinde uygulama geliştirme editörü, DockPanel Suite açık kaynak yazılımı sayesinde de form yapısı hazırlanmıştır. Yazılıma esneklik katmak adına parametrik verilerin tümü XML dosyalarında saklanmıştır.

Yapılan çalışma sonucunda İngilizce ve Türkçe dil desteğine sahip bir benzetim programı hazırlanmıştır. Programda Scintilla.NET açık kaynak yazılımı sayesinde uygulama geliştirmeyi kolaylaştıran renklendirmelere ve otomatik tamamlama özelliklerine sahip güçlü bir editör, derlenme aşamasından önce de değer atamalarına imkan veren bellek yapısı, kayan pencere yapısına uygun saklayıcılar ve üç katmanlı iş hattı hazırlanmıştır.

Kullanıcı uygulamayı yazdıktan sonra derleme aşamasına geçilmektedir. Derlenmeden önce bir hata ayıklayıcı çalışmakta ve söz dizimi hataları tespit edilip, tümü

listelenir. Hatasız bir uygulama yazılmış ise tüm komutlar iki sayı düzenindeki karşılıklarına çevrilerek makine düzeyindeki komutlara çevrilir ve belleğe yüklenir. Bunun ardından kullanıcı programı ister adım adım, isterse de baştan sona çalıştırabilir. Bunun dışında uygun gördüğü yerlere durak noktaları yerleştirerek programın o noktaya kadar çalışmasını sağlayabilir. Her saat işaretinde yeni bir komut çevrime alınır ve iş hattına koyulur. Ekrandaki iş hattı bölümünde yeni gelen komutlar rahatlıkla görülebilir. Aktif olan iş hattına bağlı olarak uygun işlemler her saat darbesiyle birlikte iş hattında bulunan komutlara uygulanır. Belleğe, saklayıcılara yazma gibi işlemlerden sonra anlık olarak değişimler ekranda gözlemlenebilir. Programın çalışması bittikten sonra da bellek, saklayıcı ve durum kütüklerinin içerikleri erişilebilir durumda kalır.

1.4 Rapor Organizasyonu

Hazırlanan raporun ikinci bölümünde projenin tanımı ve iş planı verilmiş; üçüncü bölümünde projenin gerçeklemesinde kullanılan kuramsal bilgiler aktarılmış; dördüncü bölümünde analiz ve modelleme kısımları UML diyagramlarıyla aktarılmış; beşinci bölümünde tasarım ve gerçekleme kısmında yapılan çalışmalar aktarılmıştır. Altıncı ve yedinci bölümlerde ise yapılan çalışmadan elde edilen sonuçlar ve öneriler belirtilmiştir.

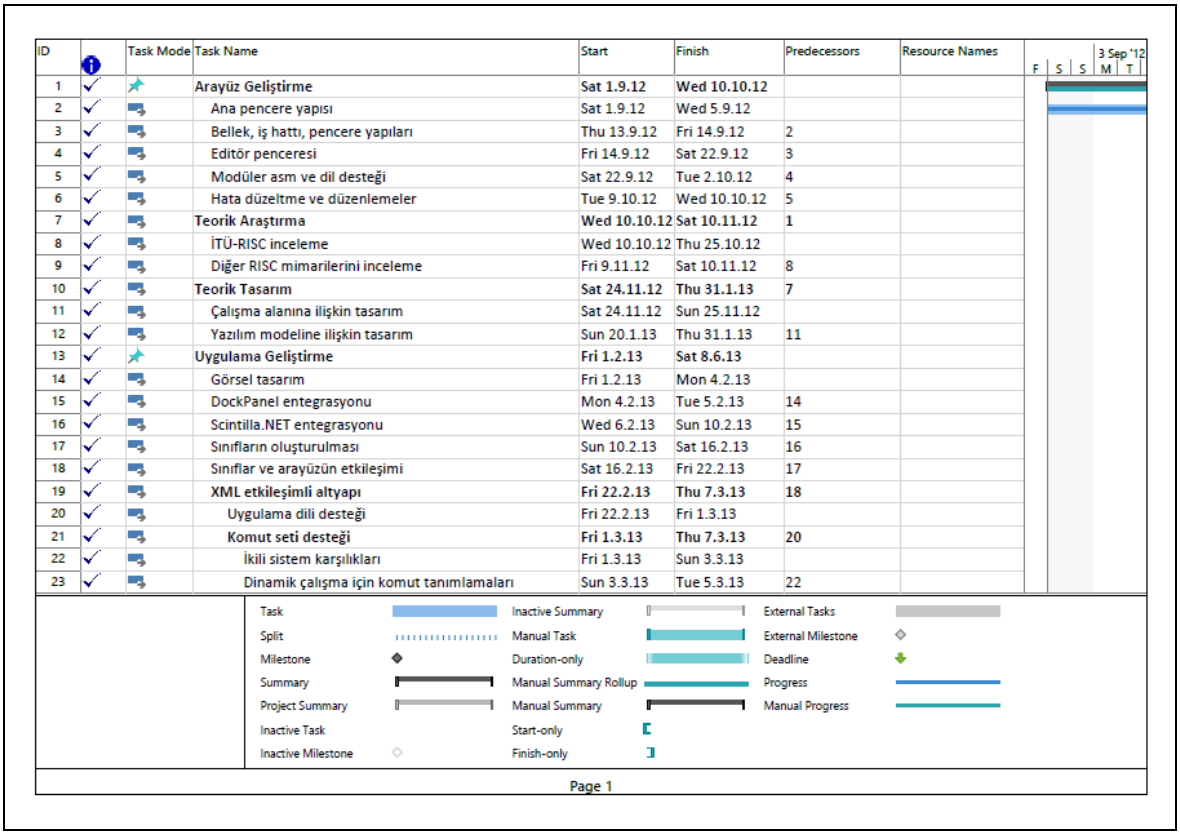
2. PROJENİN TANIMI VE PLANI

2.1 Proje Tanımı

Yapılan çalışma, kullanıcıların uygulama geliştirebilmesi için kullanacakları güçlü bir editör sağlama, bir RISC tabanlı mikroişlemci davranışını modelleme ve yazılan uygulamaları makine diline çevirerek sözde mikroişlemci üzerinde çalışmasını sağlama gibi problemleri kapsamaktadır. Proje bellek, iş hattı, saklayıcı, komut seti, editör modüllerinden oluşmaktadır.

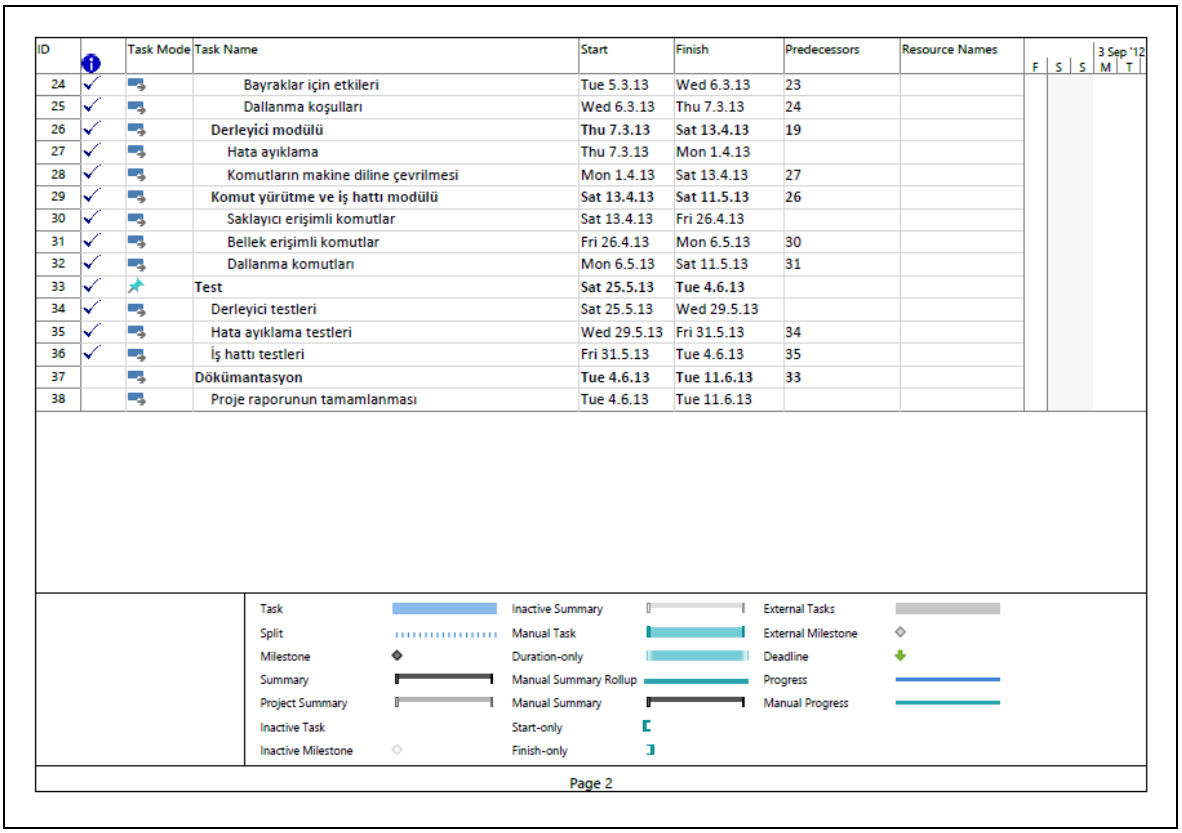
2.2 Proje Planı

Proje, kullanıcıların uygulama geliştirme ara yüzü, geliştirilen uygulamayı derleme ve sözde mikro işlemci üzerinde yürütme bölümlerinden oluşmaktadır. Projenin gerçeklemesine ilişkin yapılan zaman paylaşımı aşağıdaki proje planı ve GANTT diyagramıyla gösterilmiştir.



ID	Task Mode	Task Name	Start	Finish	Predecessors	Resource Names
1		**Arayüz Geliştirme**	**Sat 1.9.12**	**Wed 10.10.12**		
2		Ana pencere yapısı	Sat 1.9.12	Wed 5.9.12		
3		Bellek, iş hattı, pencere yapıları	Thu 13.9.12	Fri 14.9.12	2	
4		Editör penceresi	Fri 14.9.12	Sat 22.9.12	3	
5		Modüler asm ve dil desteği	Sat 22.9.12	Tue 2.10.12	4	
6		Hata düzeltme ve düzenlemeler	Tue 9.10.12	Wed 10.10.12	5	
7		**Teorik Araştırma**	**Wed 10.10.12**	**Sat 10.11.12**	**1**	
8		İTÜ-RISC inceleme	Wed 10.10.12	Thu 25.10.12		
9		Diğer RISC mimarilerini inceleme	Fri 9.11.12	Sat 10.11.12	8	
10		**Teorik Tasarım**	**Sat 24.11.12**	**Thu 31.1.13**	**7**	
11		Çalışma alanına ilişkin tasarım	Sat 24.11.12	Sun 25.11.12		
12		Yazılım modeline ilişkin tasarım	Sun 20.1.13	Thu 31.1.13	11	
13		**Uygulama Geliştirme**	**Fri 1.2.13**	**Sat 8.6.13**		
14		Görsel tasarım	Fri 1.2.13	Mon 4.2.13		
15		DockPanel entegrasyonu	Mon 4.2.13	Tue 5.2.13	14	
16		Scintilla.NET entegrasyonu	Wed 6.2.13	Sun 10.2.13	15	
17		Sınıfların oluşturulması	Sun 10.2.13	Sat 16.2.13	16	
18		Sınıflar ve arayüzün etkileşimi	Sat 16.2.13	Fri 22.2.13	17	
19		**XML etkileşimli altyapı**	**Fri 22.2.13**	**Thu 7.3.13**	**18**	
20		Uygulama dili desteği	Fri 22.2.13	Fri 1.3.13		
21		**Komut seti desteği**	**Fri 1.3.13**	**Thu 7.3.13**	**20**	
22		İkili sistem karşılıkları	Fri 1.3.13	Sun 3.3.13		
23		Dinamik çalışma için komut tanımlamaları	Sun 3.3.13	Tue 5.3.13	22	

Şekil 1 – Proje planı-1



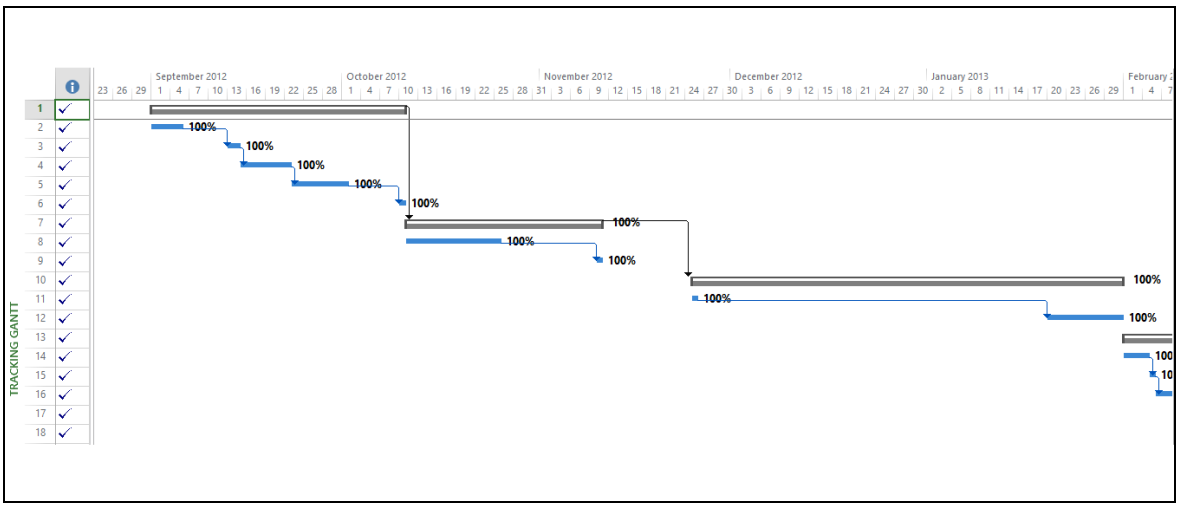
ID	Task Mode	Task Name	Start	Finish	Predecessors	Resource Names
24		Bayraklar için etkileri	Tue 5.3.13	Wed 6.3.13	23	
25		Dallanma koşulları	Wed 6.3.13	Thu 7.3.13	24	
26		**Derleyici modülü**	**Thu 7.3.13**	**Sat 13.4.13**	19	
27		Hata ayıklama	Thu 7.3.13	Mon 1.4.13		
28		Komutların makine diline çevrilmesi	Mon 1.4.13	Sat 13.4.13	27	
29		**Komut yürütme ve iş hattı modülü**	**Sat 13.4.13**	**Sat 11.5.13**	26	
30		Saklayıcı erişimli komutlar	Sat 13.4.13	Fri 26.4.13		
31		Bellek erişimli komutlar	Fri 26.4.13	Mon 6.5.13	30	
32		Dallanma komutları	Mon 6.5.13	Sat 11.5.13	31	
33		**Test**	**Sat 25.5.13**	**Tue 4.6.13**		
34		Derleyici testleri	Sat 25.5.13	Wed 29.5.13		
35		Hata ayıklama testleri	Wed 29.5.13	Fri 31.5.13	34	
36		İş hattı testleri	Fri 31.5.13	Tue 4.6.13	35	
37		**Dökümantasyon**	**Tue 4.6.13**	**Tue 11.6.13**	33	
38		Proje raporunun tamamlanması	Tue 4.6.13	Tue 11.6.13		

3 Sep '12 — F S S M T

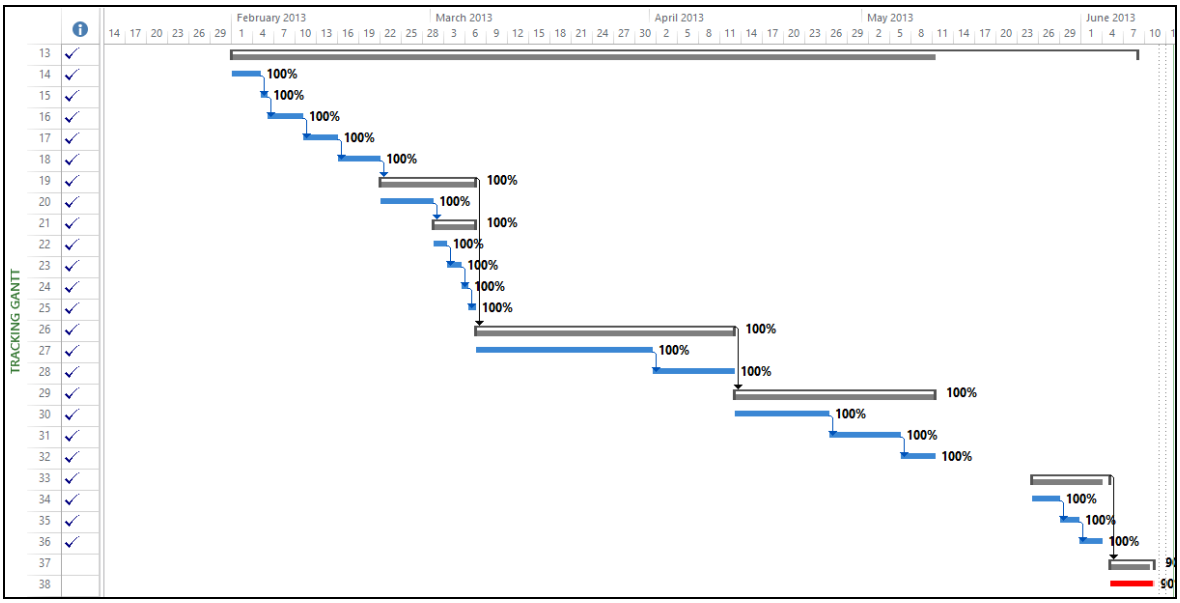
Task — Inactive Summary — External Tasks
Split — Manual Task — External Milestone
Milestone — Duration-only — Deadline
Summary — Manual Summary Rollup — Progress
Project Summary — Manual Summary — Manual Progress
Inactive Task — Start-only
Inactive Milestone — Finish-only

Page 2

Şekil 2 – Proje planı-2



Şekil 3 – GANTT diyagramı -1



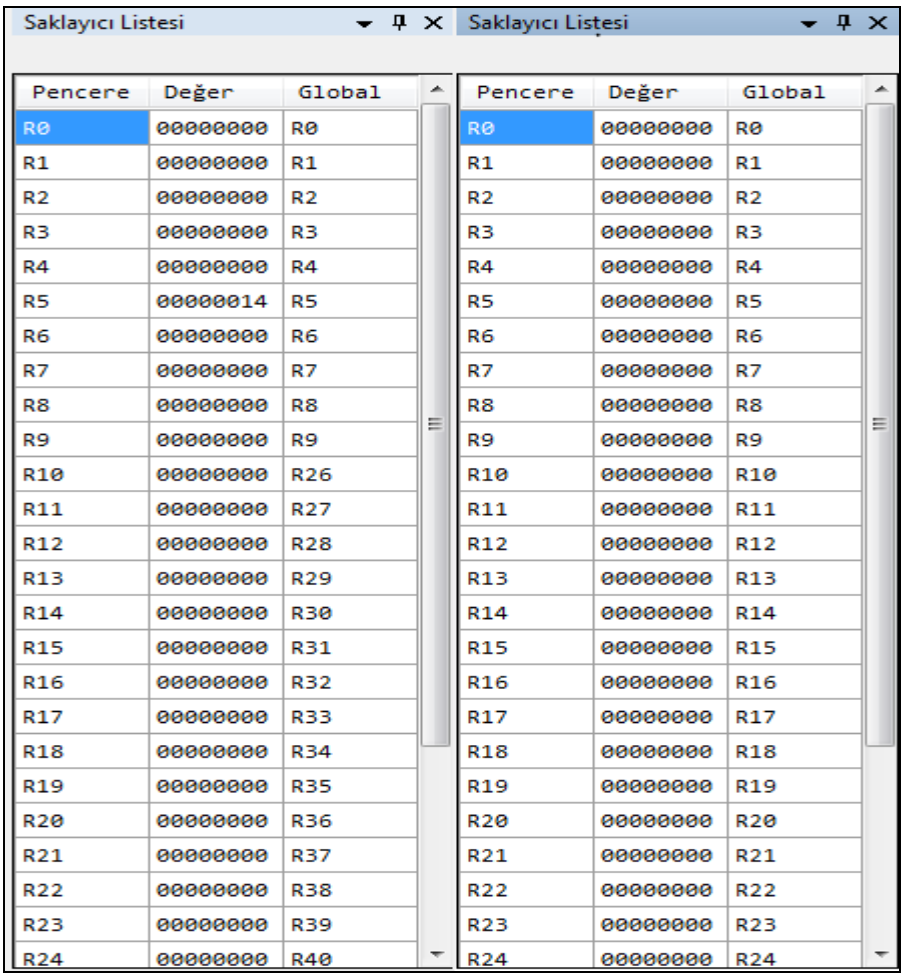
Şekil 4 – GANTT diyagramı - 2

3. KURAMSAL BİLGİLER

Projeye başlamadan önce kullanılacak bilgisayar sisteminin özellikleri belirlenmiştir. Buna göre kullanılan merkezi işlemci komut yapısına göre incelendiğine RISC tipinde, parametre sayısına göre üç parametreli bir sistem tercih edilmiştir. Komut ve veri bellekleri ayrı olarak düşünülmüş ve Harvard mimarisi kullanılmıştır.

3.1 Saklayıcılar

Altprogram çağrıları için kayan saklayıcı yapısı kullanılacaktır. Bu yapı altprogram çağrılarında belleğe ihtiyaç duymadan parametre aktarımları yapılmasını ve yerel değişkenlerin tutulmasını sağlar. Bellek erişiminin bu şekilde azaltılması verimliliği de arttır. Her program kendi içerisinde belli sayıda saklayıcıya erişebilmektedir. Bu saklayıcı kümesine o programın *pencere*si denir. Yeni bir altprogram çağrısı geldiğinde pencere kaydırılır ve yeni altprogram başka bir pencereyi kendi saklayıcıları olarak kullanır. Bu nedenle altprograma gitmeden önce verileri belleğe (yığın) yazmak ve geri dönüşlerde verileri yığından geri almak işlemlerine ihtiyaç duyulmaz. Bunun dışında pencereler kaydırılırken bazı saklayıcılar altprogramın da görebileceği şekilde bırakılır. Bu sayede programlar arası parametre aktarımlarında da bellek kullanımına ihtiyaç ortadan kalkar.



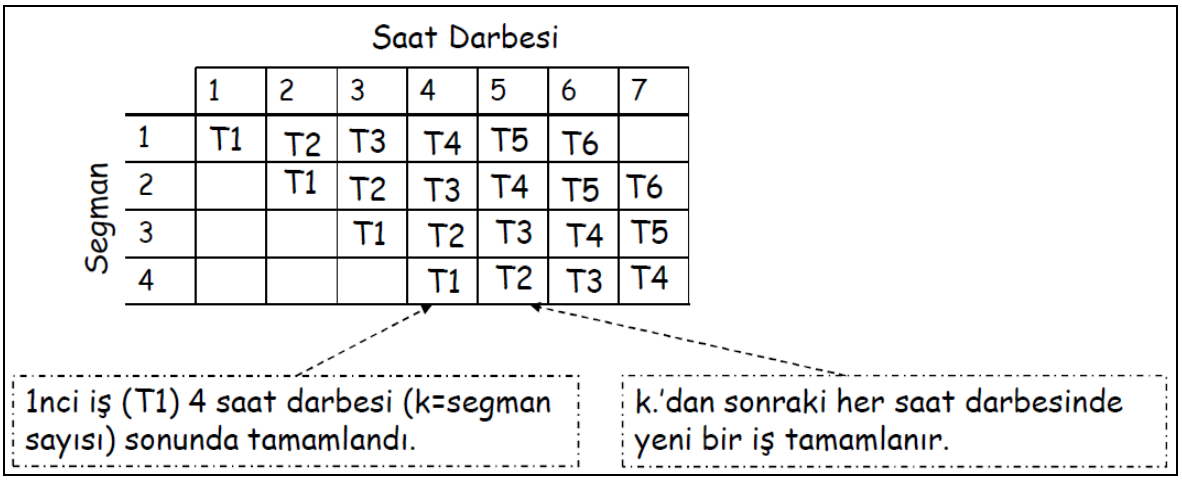
Pencere	Değer	Global
R0	00000000	R0
R1	00000000	R1
R2	00000000	R2
R3	00000000	R3
R4	00000000	R4
R5	00000014	R5
R6	00000000	R6
R7	00000000	R7
R8	00000000	R8
R9	00000000	R9
R10	00000000	R26
R11	00000000	R27
R12	00000000	R28
R13	00000000	R29
R14	00000000	R30
R15	00000000	R31
R16	00000000	R32
R17	00000000	R33
R18	00000000	R34
R19	00000000	R35
R20	00000000	R36
R21	00000000	R37
R22	00000000	R38
R23	00000000	R39
R24	00000000	R40

Pencere	Değer	Global
R0	00000000	R0
R1	00000000	R1
R2	00000000	R2
R3	00000000	R3
R4	00000000	R4
R5	00000000	R5
R6	00000000	R6
R7	00000000	R7
R8	00000000	R8
R9	00000000	R9
R10	00000000	R10
R11	00000000	R11
R12	00000000	R12
R13	00000000	R13
R14	00000000	R14
R15	00000000	R15
R16	00000000	R16
R17	00000000	R17
R18	00000000	R18
R19	00000000	R19
R20	00000000	R20
R21	00000000	R21
R22	00000000	R22
R23	00000000	R23
R24	00000000	R24

Şekil 5 - Altprogram çağrısı yapıldığında program yerel olarak yine aynı bellek numaralarını kullanmakta ancak global olarak baktığımızda farklı saklayıcılara erişmekte.

3.2 İş Hattı

Paralel veri işlemenin etkili yöntemlerinden biri iş hattı kullanmaktır. İş hatları temel olarak bir ana işi küçük alt işlere bölme prensibine (böl ve yönet) dayalıdır. İşlemciye gelen her komut için ortak bazı mikro işlemler tanımlanır ve her saat darbesinde bu işlemler gerçekleştirilir. İş hattına yeni bir komut geldiğinde ilk katmana yerleştirilir ve her saat darbesiyle birlikte bir sonraki segmana kaydırılır. Birinci katmanda yer alan komut saat darbesiyle ikinci katmana geçtiğinde yeni bir komut (eğer varsa) gelmiş olacaktır ve eski komutun yerine birinci katmana yerleşir. Bu şekilde ilerlediğinde paralel olarak katman sayısı kadar komut çalıştırılabilir. Ancak burada daha fazla komutu paralel yürütebilmek için katman sayısını çok fazla arttırmak avantaj sağlamaz. Belli bir verimlilik yükselişinden sonra iş hattındaki mikro işlemleri küçültmek işlerin uzamasına, gecikmelerin artmasına sebep olacaktır.



Şekil 6 - Dört katmanlı bir iş hattı örneği **[1]**

Hat No	Değer	Kod
P1	00000010010100000010000000000010	ADD R0, S0, R10
P2	00000010010110000010010100000000	ADD R0, S1, R11
P3	00000010011000000010000000000011	ADD R0, S2, R12

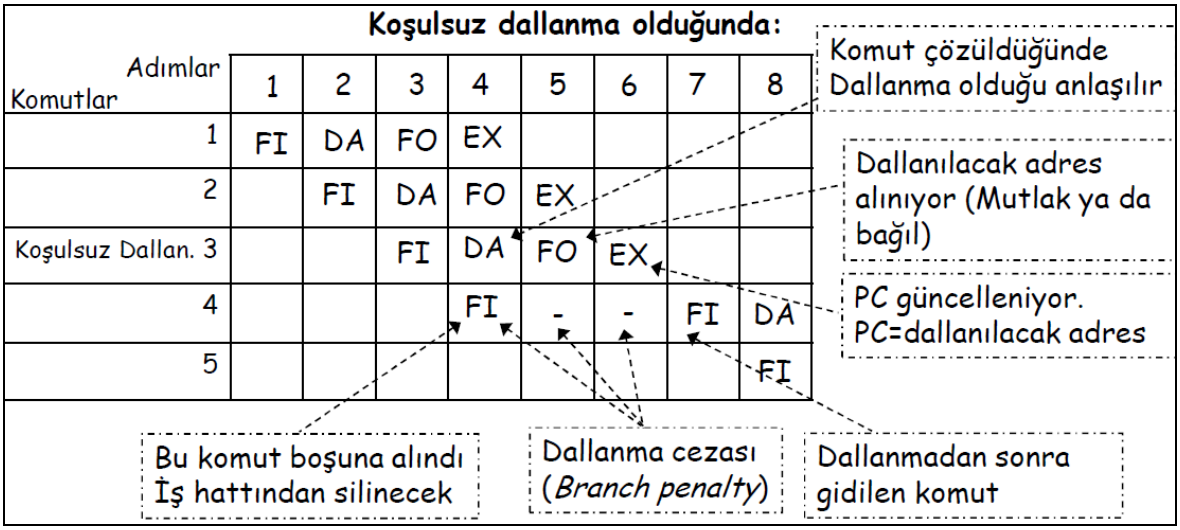
Şekil 7 - Üç katmanlı iş hattında komutların çalıştırılma anı

Her komut çevriminde program sayacı (PC) değişir. Bunun yanında kimi komutların yürütülmesi de durum saklayıcılarını etkiler. Bayrakların değişimi özellikle dallanma komutları için önemlidir çünkü koşullu dallanma işlemlerinin yapılıp yapılmayacağı kararı bayraklara göre verilir.

Bayraklar	Değer
PC	$000C
Z	0
C	0
V	0
N	0

Şekil 8 - Bayrak değerleri ve program sayacı

İş hatlarının en büyük düşmanı dallanma komutlarıdır. Eğer bir dallanma işlemi gerçekleşecek olursa katman sayısına bağlı olarak bir *dallanma cezası* oluşur. Çünkü gelen komutun dallanma olduğunu ve dallanmanın gerçekleşeceğini fark edene kadar iş hattı yeni komutları almaya devam edecektir. Bunun sonucunda dallanmanın gerçekleşmesi gerektiğini fark ettiğinde, o zamana kadar almış olduğu komutların tümünü iş hattından kaldırması gerekir. Buradan da kolaylıkla fark edilebileceği gibi çok fazla katmana sahip bir iş hattının dallanma cezası da fazla olacaktır.



Şekil 9 - Dört katmanlı bir iş hattında dallanma cezası örneği **[1]**

3.2 Bellek

Bellek birimi yazılan programların ve verilerin saklanması için kullanılmaktadır. Kullanıcılar belleğe programlarına gerekli olan ön değerleri atmak için programlarının başına belleğe yazma komutları kullanmak yerine global bellek yapısı sayesinde derlenme işleminden önce de belleğe erişebilir ve değer atamalarını doğrudan belleğe yapabilir. Bu sayede ilk baştaki değer atama iş yükünden kurtulmuş olur.

Bellek Haritası

$****	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$000_	02	60	20	03	02	58	25	00	02	50	20	02	15	62	C0	00
$001_	02	68	20	03	10	72	80	00	00	00	00	00	00	00	00	00
$002_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
$003_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
$004_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
$005_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
$006_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
$007_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
$008_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
$009_	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Şekil 10 - Bellek yapısı (her hücre için varsayılan değer $00)

3.3 Komut Seti

Kullanıcıların uygulama geliştirebilmesi için bir komut seti gereklidir. Projede varsayılan olarak gelen komut seti Berkeley RISC-I'den alınmıştır. Her komuta karşılık gelen bir ikilik sistem kodu da bulunmaktadır. Uygulama komutları bir XML belgesinden alıp işleme soktuğu için komut isimleri değiştirilse de uygulama sorunsuz bir şekilde çalışmaya devam edecektir. Bu da uygulamaya farklı komut setleri için destek sağlama gücü vermekte ve esnekliğini arttırmaktadır. Şu anda kullanılmakta olan komut seti ve komutların özellikleri aşağıda listelenmiştir.

Komut	**Parametre**	**Açıklama**
ADD	Rs,S2,Rd	Rd = Rs + S2
ADDC	Rs, S2, Rd	Rd = Rs + S2 + C
SUB	Rs, S2, Rd	Rd = Rs - S2
SUBC	Rs, S2, Rd	Rd = Rs + S2 – C
SUBR	Rs, S2, Rd	Rd = S2 – Rs
SUBCR	Rs, S2, Rd	Rd = S2 – Rs – C
AND	Rs, S2, Rd	Rd = Rs & S2
OR	Rs, S2, Rd	Rd = Rs | S2
XOR	Rs, S2, Rd	Rd = Rs xor S2
SLL	Rs, S2, Rd	Rd = Rs << S2
SRL	Rs, S2, Rd	Rd = Rs >> S2 (mantıksal)
SRA	Rs, S2, Rd	Rd = Rs >> S2 (aritmetik)
LDL	(Rx)S2,Rd	Rd = M[Rx+S2] (long)
LDSU	(Rx)S2,Rd	Rd = M[Rx+S2] (ushort)
LDSS	(Rx)S2,Rd	Rd = M[Rx+S2] (short)
LDBU	(Rx)S2,Rd	Rd = M[Rx+S2] (ubyte)
LDBS	(Rx)S2,Rd	Rd = M[Rx+S2] (byte)
STL	(Rx)S2,Rm	M[Rx+S2] = Rm (long)
STS	(Rx)S2,Rm	M[Rx+S2] = Rm (short)
STB	(Rx)S2,Rm	M[Rx+S2] = Rm (byte)
JMP	COND, S2(Rx)	PC = PC + S2
JMPR	COND, Y	PC = PC + Y
CALL	S2(Rx), Rd	CWP--; Rd = PC; PC = Rx + S2
CALLR	Y, Rd	CWP--; Rd = PC; PC = Rx + Y
RET	(Rx)S2	PC = Rx+S2; CWP++

3.4 Derleyici

Derleyici, yazılan komutları yürütülebilir hale getirmek için kullanılan bir birimdir. Derleyici kullanıcının verilen komut setiyle yazmış olduğu uygulamayı (bkz. Şekil 11) alıp mikroişlemcinin anlayabileceği şekilde makine diline dönüştürür (bkz. Şekil 12).

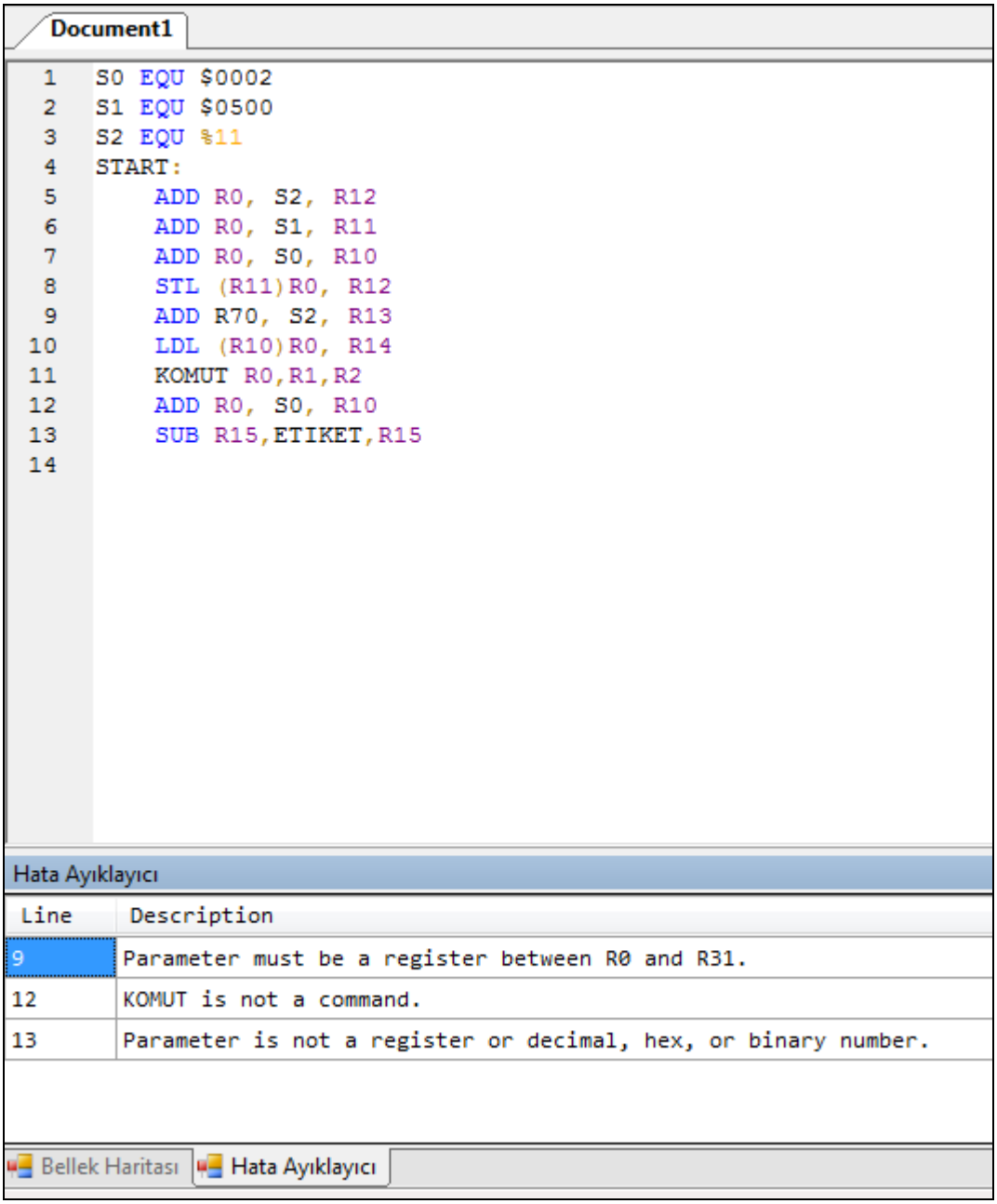
```
1   S0 EQU $0002
2   S1 EQU $0500
3   S2 EQU %11
4   START:
5       ADD R0, S2, R12
6       ADD R0, S1, R11
7       ADD R0, S0, R10
8       STL (R11)R0, R12
9       ADD R0, S2, R13
10      LDL (R10)R0, R14
11
```

Şekil 11 - Editör kullanılarak yazılan bir uygulama örneği

```
1   //ADD R0, S2, R12
2   00000010011000000010000000000011
3   //ADD R0, S1, R11
4   00000010010110000010010100000000
5   //ADD R0, S0, R10
6   00000010010100000010000000000010
7   //STL (R11)R0, R12
8   00010101011000101100000000000000
9   //ADD R0, S2, R13
10  00000010011010000010000000000011
11  //LDL (R10)R0, R14
12  00010000011100101000000000000000
13
```

Şekil 12 - Derlenen komutlar ve ikili sayı karşılıkları

Bunun yanında yazılan programda hata olup olmadığını da denetlemek derleyicinin görevidir. Buna göre kullanıcı hatalı bir komut girişinde bulunduğunda derlenme aşamasında tüm komutlar taranır, hatalı olarak görülen satırlar belirlenir ve kullanıcıya hatanın ne olduğuna dair bir uyarı mesajı gösterir. Bu sayede gözden kaçan hataları tespit etmek ve düzeltmek kullanıcı için kolaylaşır.



Şekil 13 –Derleme işlemini başlattıktan sonra uygulamanın söz dizimi hatalarını tespit etme ve kullanıcıyı uyarma mekanizması

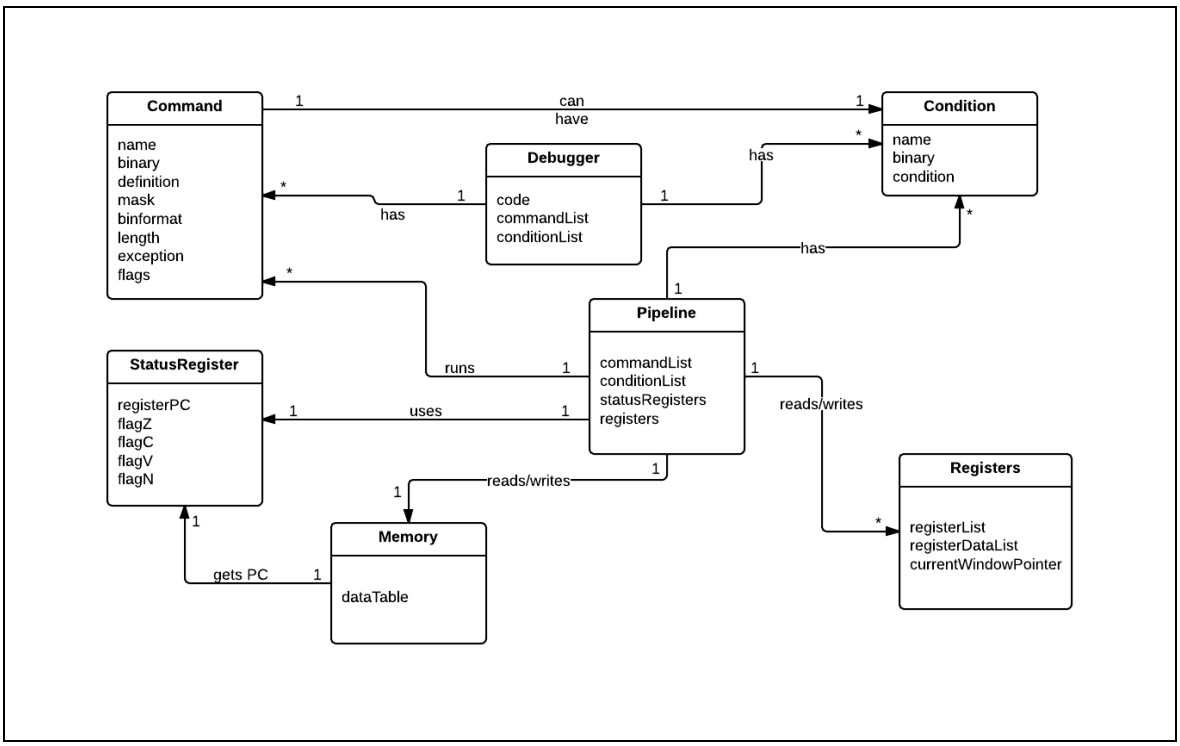
4. ANALİZ VE MODELLEME

4.1 Analiz

Probleme ilişkin yapılan analizler sonucunda çözüm için gerekli olan gerçek dünya nesneleri tespit edilmiştir. Bunlar;

- Komutlar
- Koşullar
- Derleyici
- Bellek
- İş hattı
- Saklayıcılar

olarak listelenebilir. Kullanılması gereken nesnelerin belirlenmesinin ardından bu nesnelerin birbirleriyle olan ilişkileri ve bağlılıkları tanımlanmalıdır.

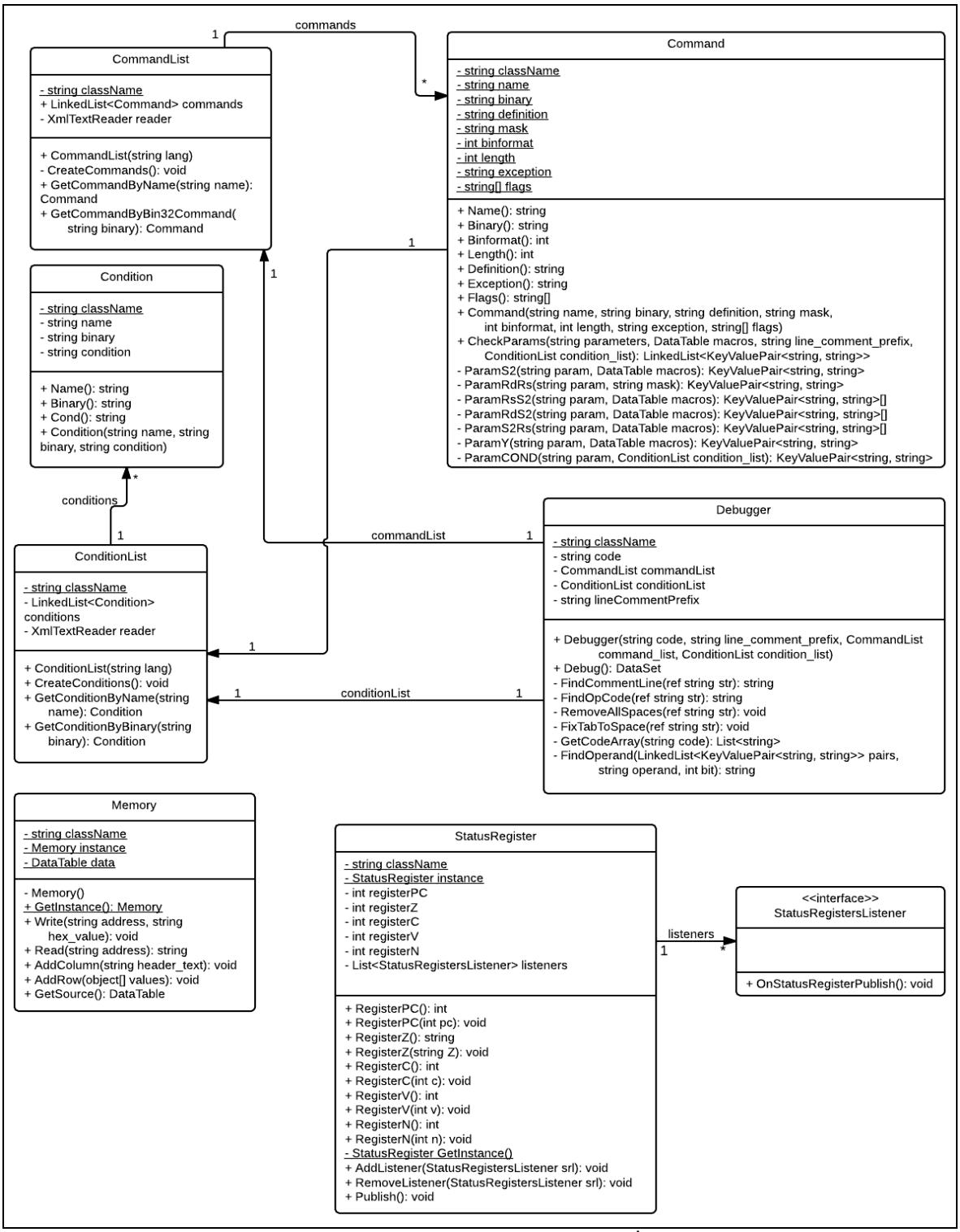


Şekil 14 – Gerçek dünya modeli UML diyagramı

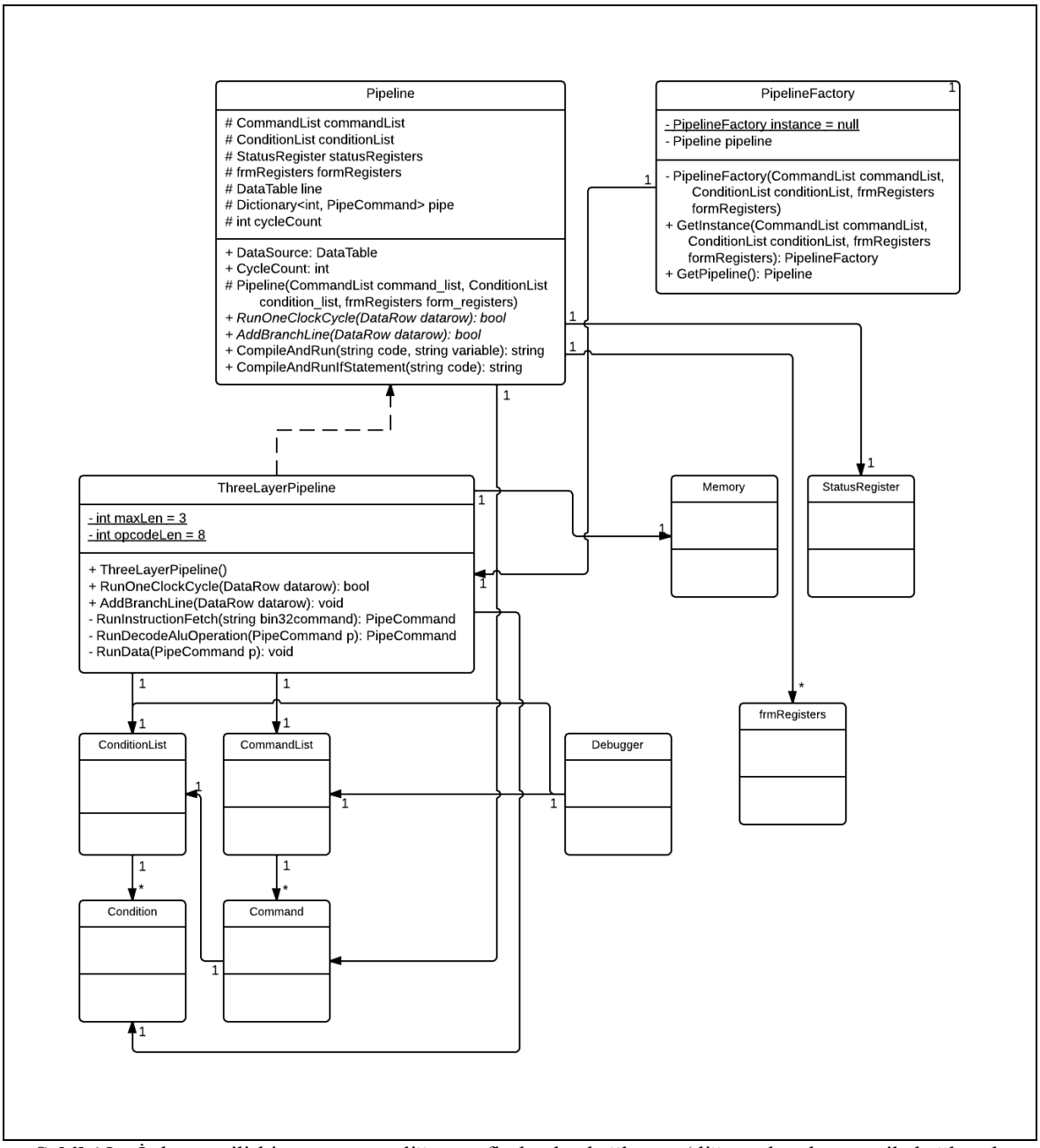
Sistem için için iş hattı *(Pipeline)* ve derleyici *(Debugger)* nesneleri diğerlerine göre daha büyük ve karmaşık nesnelerdir. Özellike iş hattı nesnesinin çok fazla nesneye ihtiyaç duyması sistem için ek bir yük oluşturmaktadır. Ancak komutlar ile ilgili yürütme, bellek erişimi, saklayıcı erişimi, dallanma gibi kararlar ve işlemler burada yapıldığı için bu nesne diğerlerinden daha büyük olmak zorundadır. Diğer yandan derleyici nesnesinin başka nesnelerle çok büyük bir bağımlılığı yoktur. Bunun sebebi kullanıcının yazmış olduğu "ham kodu" alıp kendi içindeki mantıksal süzgeçlerden geçirerek makine diline çevirmesidir. Bu aşamada başka nesnelere ihtiyaç duymasa da çok fazla işlem yapacaktır.

4.2 Modelleme

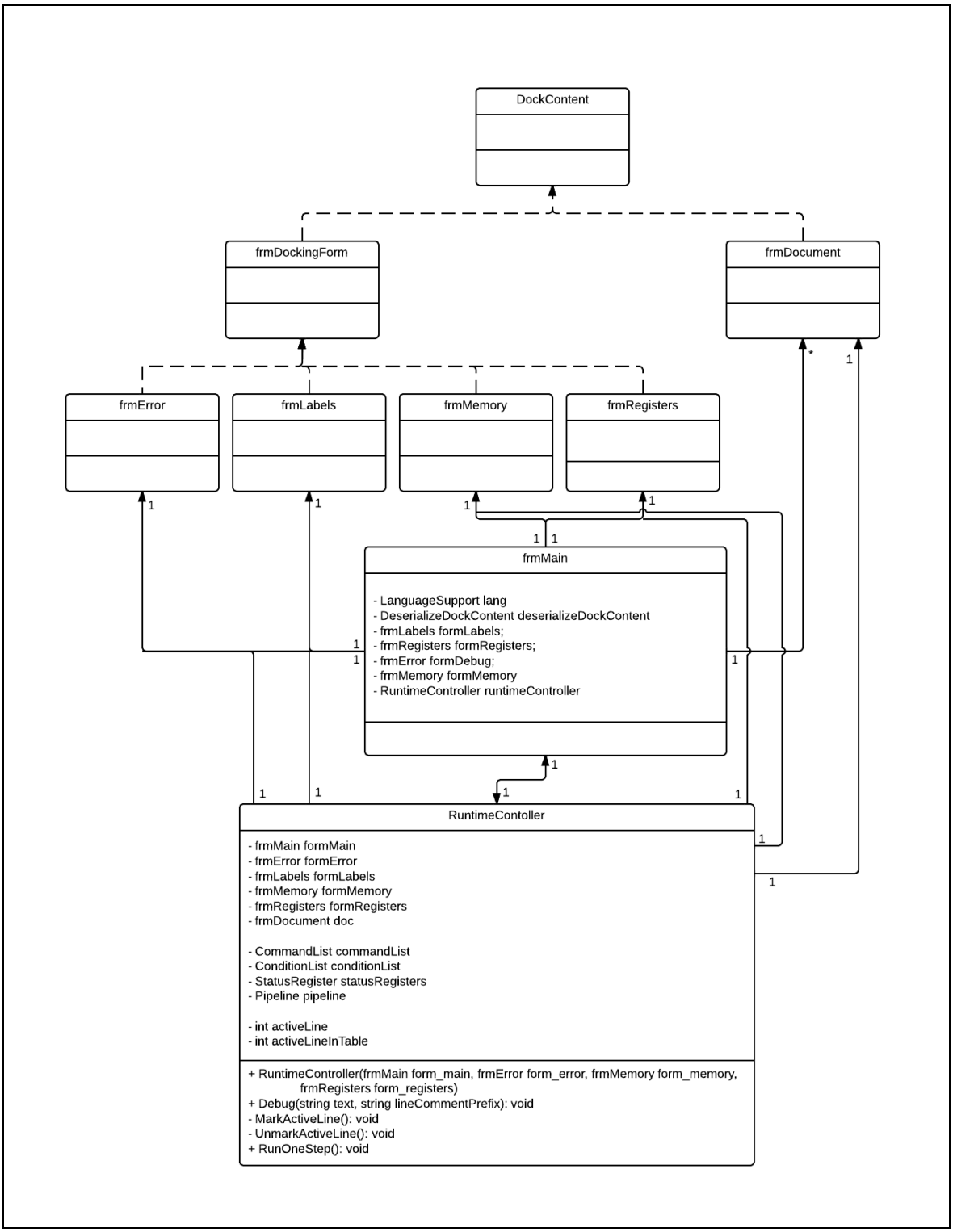
Analiz aşamasında tespit edilen nesnelerin yazılım dünyasına aktarılması ve doğrudan yazılım haline getirilebilecek bir yapının oluşturulması modelleme bölümünde gerçekleştirilecektir. Problemin çözümü için kurgulanan yapı UML diyagramlarıyla gösterilmiştir.



Şekil 15 – Mantıksal sınıflara ilişkin UML diyagramları (İş hattı bölümü hariç)



Şekil 16 – İş hattına ilişkin tasarım ve diğer sınıflarla olan bağlantısı (diğer pek çok nesne ile bağlantılı olduğu için gösterimi bu şekilde ayrılmıştır). Daha önceki UML diyagramlarında detaylı açıklaması yapılan sınıflar için temsili gösterim kullanılmıştır.



Şekil 17 – Ara yüz nesnelerinin ilişkileri ve kontrol nesnesiyle olan bağlantısı

5. TASARIM, GERÇEKLEME VE TEST

Tasarımda nesneye yönelik yaklaşım benimsenmiş ve gerçek dünyadaki her nesne için tasarımda da bir "nesne" karşılık düşmüştür. Yazılım kalitesi ve tasarımı açısından bazı yapay nesnelere de başvurulmuştur. bunun yanında tasarım şablonları yardımıyla yazılım kalitesi güçlendirilmiştir.

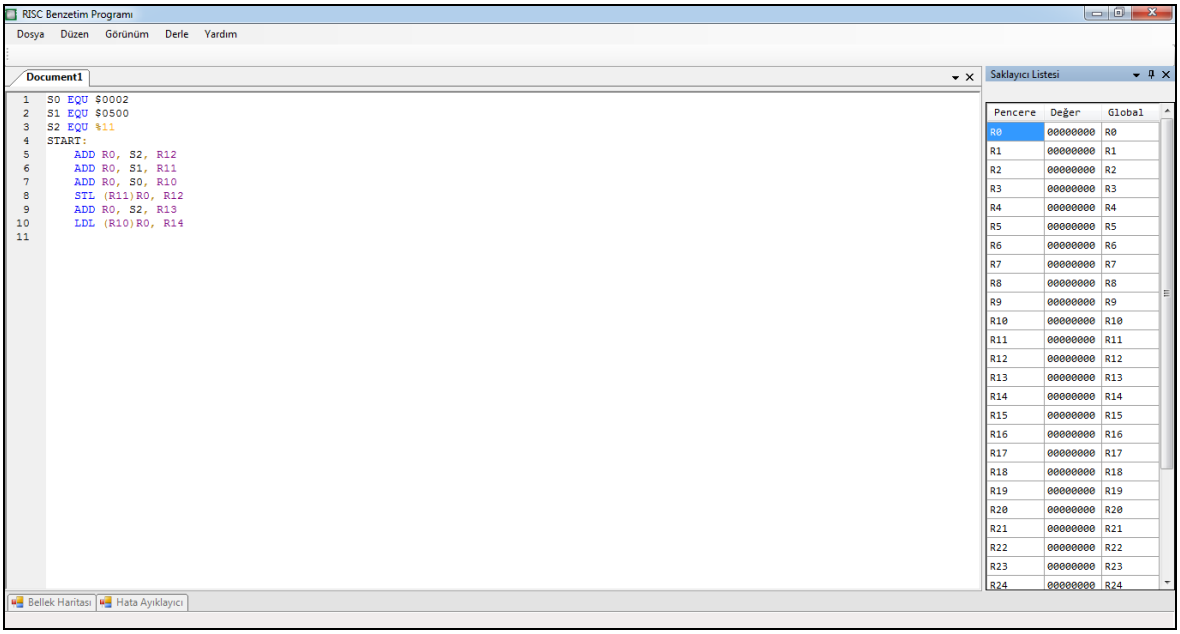
Grafik ara yüz ile uygulama lojiğini birbirinden ayırmak için ara yüz nesnelerine mantıksal görevler atanmamıştır. "frm" ön eki ile başlayan tüm nesneler görsel ara yüz (form) nesnesi olarak oluşturulmuştur ve asıl verilerini başka bir nesnede tutmaktadır. Bu sayede görsel değişimlerin uygulamayı etkilemesi önlenmiştir. Yine de görsel nesneler ile mantıksal nesneler arasında bir etkileşime ihtiyaç duyulmaktadır. Bunu sağlamak için bir kontrol sınıfı tasarlanmıştır (RuntimeController). Kullanıcı uygulamasını geliştirip "derle" veya "yürüt" demediği sürece bu sınıfa ihtiyaç duyulmaz. Ancak derlenme ve yürütme aşamasına geçildiğinde artık ara yüzde hazırlanmış olan kodların alınması gerçek nesnelere aktarılması, bellek, saklayıcı, iş hattı, durum kütüğü değerlerinin değişmesi gerekmektedir. Bunlar için kontrol sınıfı bir aracı rolü üstlenmiştir. **[5]**

Birden fazla iş hattı yapısı entegre edilebilsin diye iş hattı nesnesi fabrika tasarım şablonu yardımıyla oluşturulmuştur. Bu sayede yeni bir iş hattı yapısı sistemin işleyişini hiç değiştirmeden eklenebilecektir. **[5]**

Bazı nesneler için de "tekillik" (Singleton) prensibi benimsenmiştir. Bunlar sistemde yalnıza bir defa oluşturulması gereken/yeterli olan nesnelerdir. Buna örnek olarak iş hattını seçen fabrika verilebilir. Bir defa iş hattını seçtikten sonra uygulama o iş hattına uygun olarak çalışacaktır. **[5]**

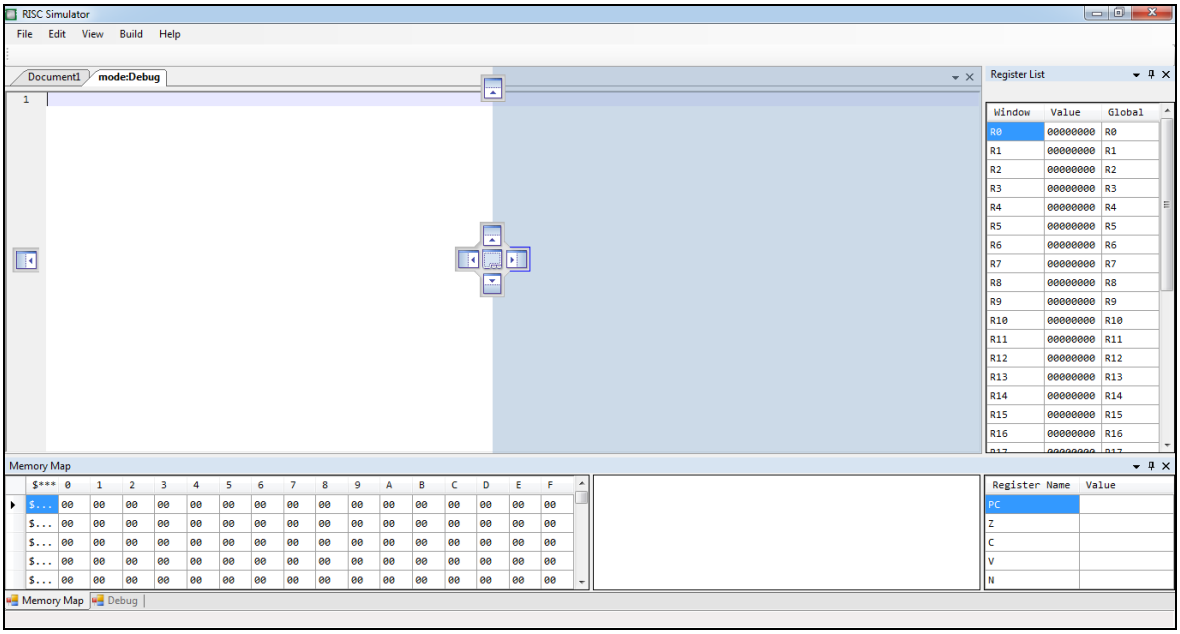
Benzetim programını gerçeklemek için geliştirme ortamı olarak Microsoft Visual Studio 2010, programlama dili olarak ise C# 2.0 kullanılmıştır. Bunun dışında Scintilla.NET açık kaynak yazılımı sayesinde uygulama geliştirme editörü, DockPanelSuite açık kaynak yazılımı sayesinde de görsel açıdan güç katılmıştır. Yazılıma esneklik katmak adına parametrik verilerin tümü XML dosyalarında saklanmıştır.

Kullanımı kolaylaştırmak ve estetik bir ara yüz sunabilmek adına DockPanelSuite kullanılarak Visual Studio görünümünde bir uygulama hazırlanmıştır. Scintilla.NET editörü sayesinde de komutlar renklendirilmiştir.

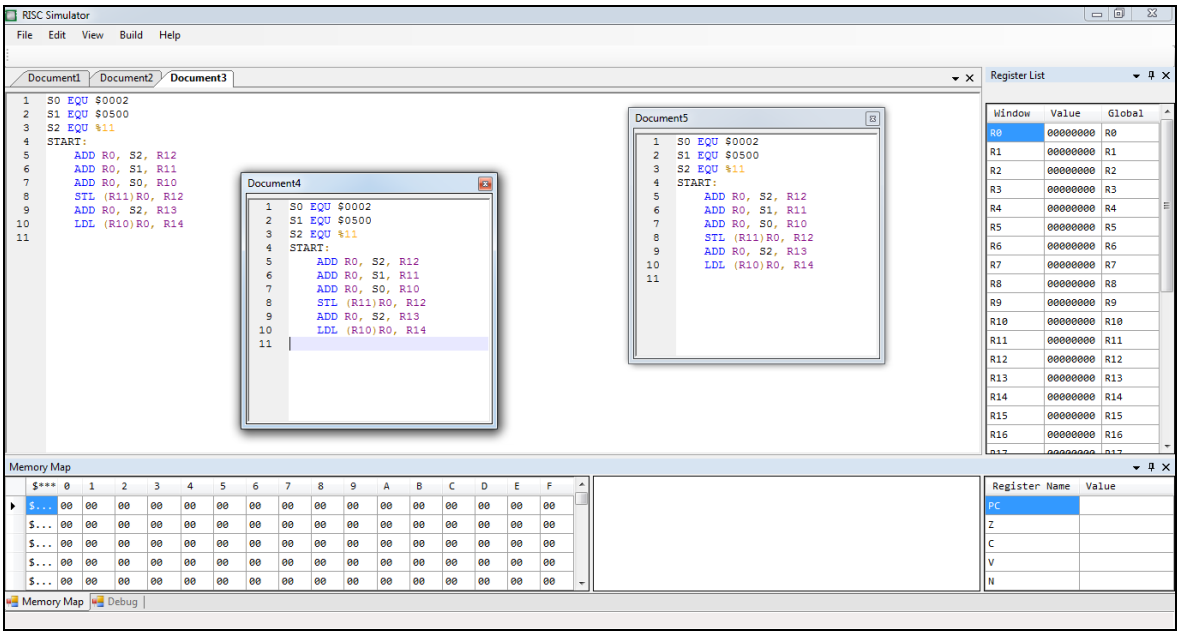


Şekil 18 – Visual Studio tipi form yapısı

Form nesnelerinin yerleri istenilen şekilde değiştirilebilir ve uygulama görsel değişimlerdeki son ayarları kaydederek bir sonraki açılışta bu seçimleri hatırlar. Böylece kullanım kolaylığı sağlanmış olur.

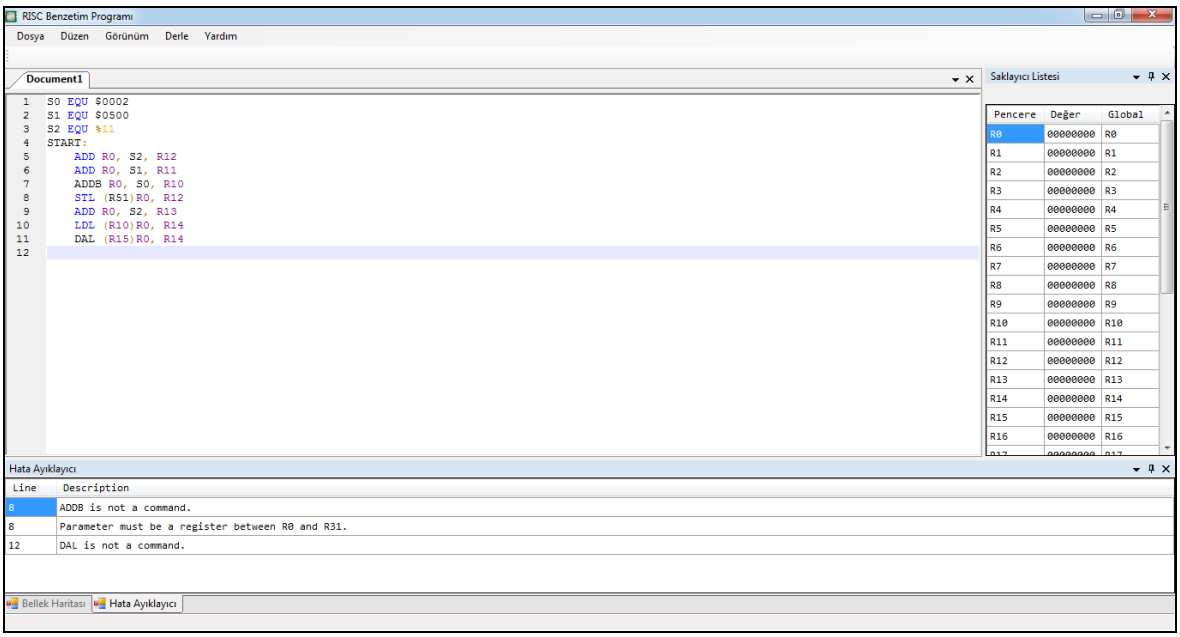


Şekil 19 – Bir form nesnesinin yeri istenildiği gibi değiştirilebilir



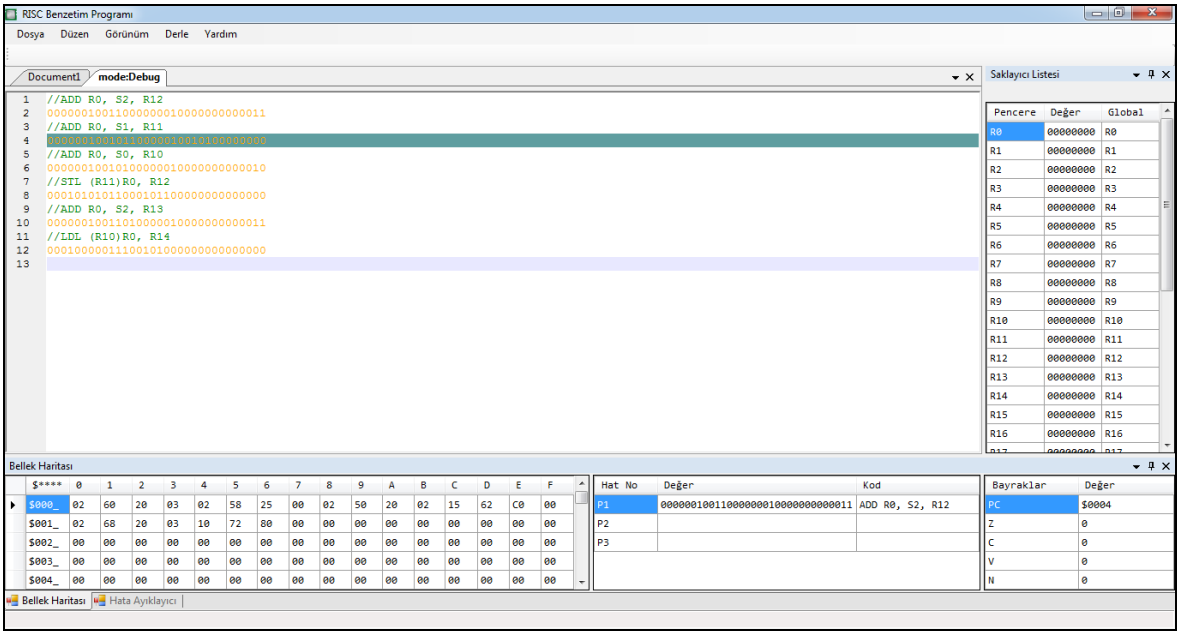
Şekil 20 – Birden fazla dosyayla çalışırken istenilen sayfaların boyutlarıyla oynanabilir

Derlenme sırasında çıkan hatalar "hata ayıklayıcı" sekmesinde listelenir ve kullanıcı hatalarını kolaylıkla görüp düzenleyebilir.

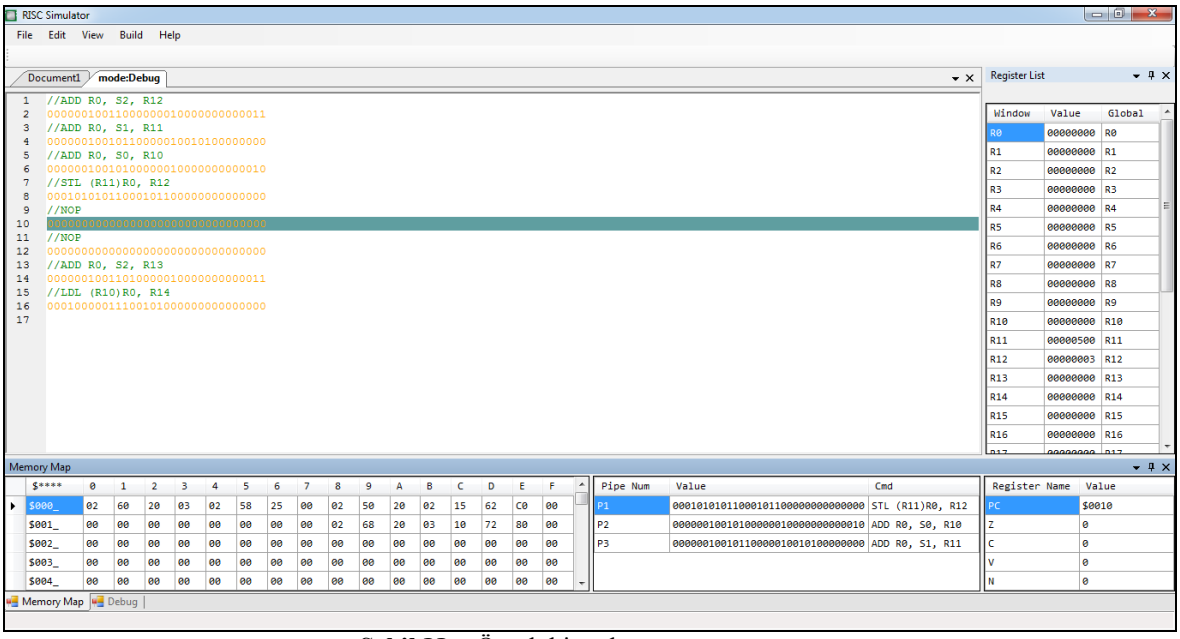


Şekil 21 – Hata ayıklayıcı

Derlenme gerçekleştikten sonra iş hattı, saklayıcılar ve durum kütüğü hazırlanır. Komutların kendisi ve ikili karşılığı editör üzerinde gösterilir. Bir sonraki adımda çalıştırılacak olan satır koyu renkle boyanmıştır.

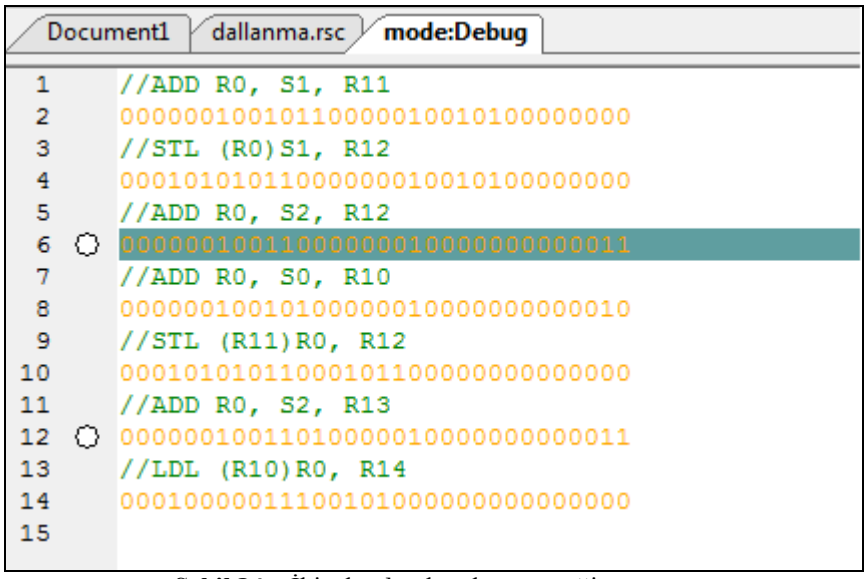


Şekil 22 – Derlenmeden sonra uygulamanın görünümü



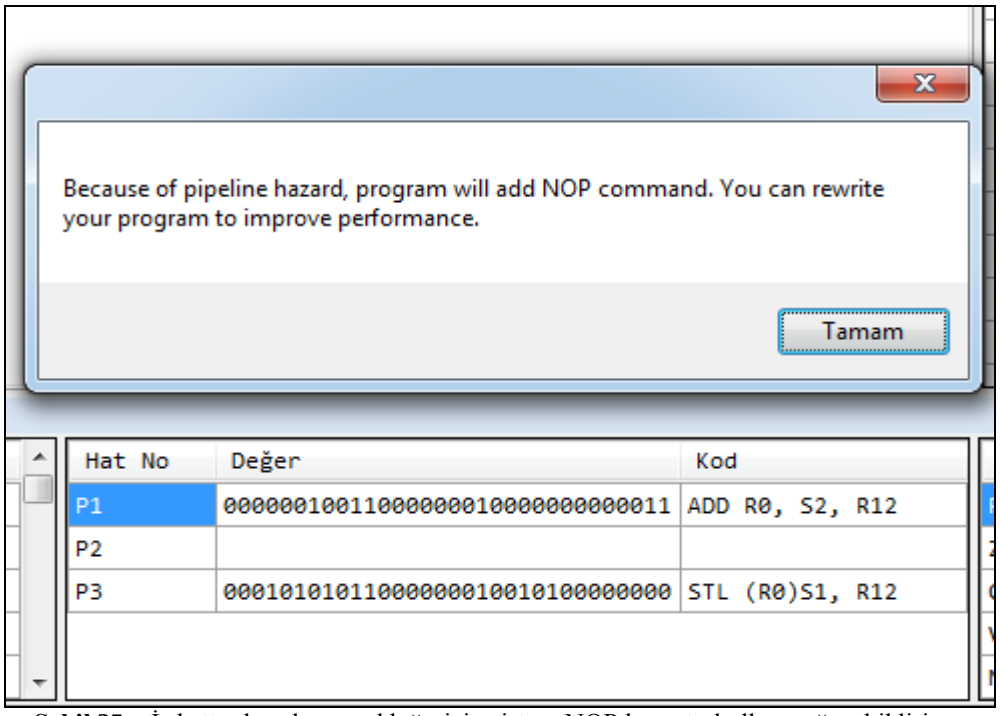
Şekil 23 – Örnek bir çalışma anı görüntüsü

Beyaz yuvarlak imler ile uygulamaya “durak noktaları” koyulabilmektedir. Bu imler sayesinde kullanıcı programı en baştan başlatıp istediği bir noktaya gelene kadar çalıştırarak o anda saklayıcı/bellek değerlerini kontrol edebilir veya adım adım yürütmeye devam edebilir.



Şekil 24 – İki adet durak noktası örneği

İş hattı çakışmalarına yol açacak bir uygulama yazılırsa derlenme aşamasında bu hatalar fark edilemez. Çünkü bunlar “çalışma zamanı” hataları olarak isimlendirilir. Bu gibi hataları çalışma anında fark edip işleyişi bozmamak için bir hata düzeltici sistem eklenmiştir. Aşağıda sistemin çalışması esnasında *ADD R0,S2,R12* ve *STL (R0)S1,R12* komutlarının arka arkaya gelmesinin iş hattı çakışmasına neden olduğu tespit edilmiş ve sistem araya NOP komutu koyarak müdahale edeceğini bildirmiştir.

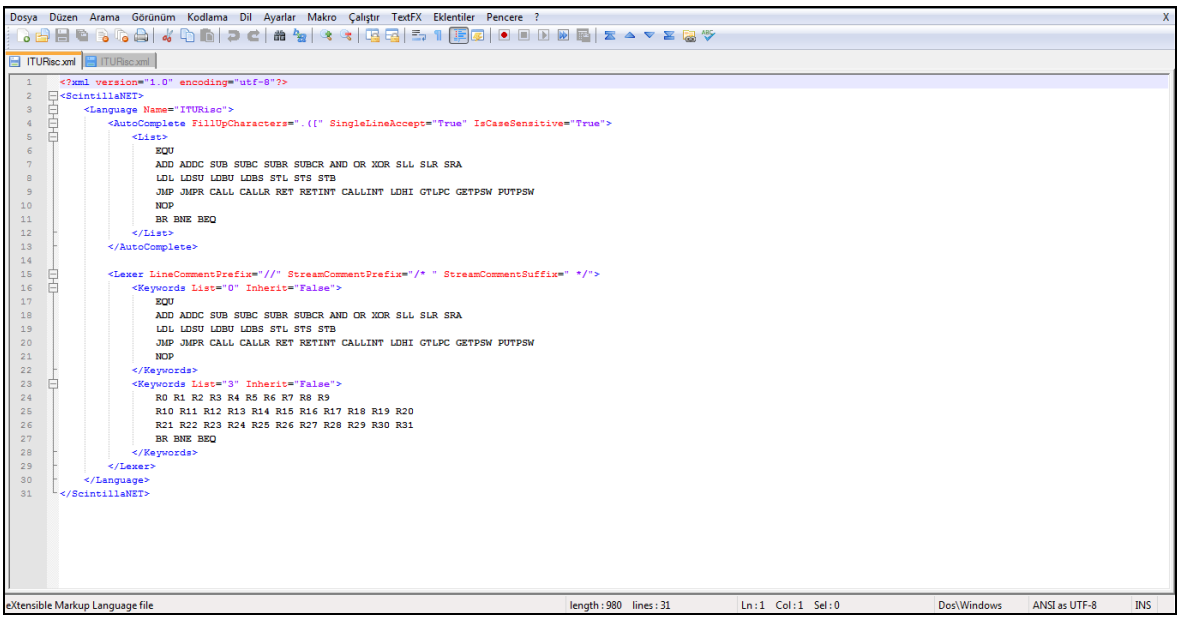


Şekil 25 – İş hattında çakışma olduğu için sistem NOP komutu kullanacağını bildiriyor

Hat No	Değer	Kod
P1	00000010011000000010000000000011	ADD R0, S2, R12
P2	00000000000000000000000000000000	NOP
P3	00010101011000000010010100000000	STL (R0)S1, R12

Şekil 26 – Sistem çakışmayı NOP ekleyerek çözdü

Otomatik tamamlama ve editör renklendirme özelliklerinden faydalanmak için XML dosyası üzerinden "uygulama dili" tanımlamaları yapılabilir. Bu sayede CTRL+Space tuş kombinasyonu kullanılarak komutların listesi getirilebilir. Tüm komutların isimlerinin ezberlenmesine gerek kalmaz.



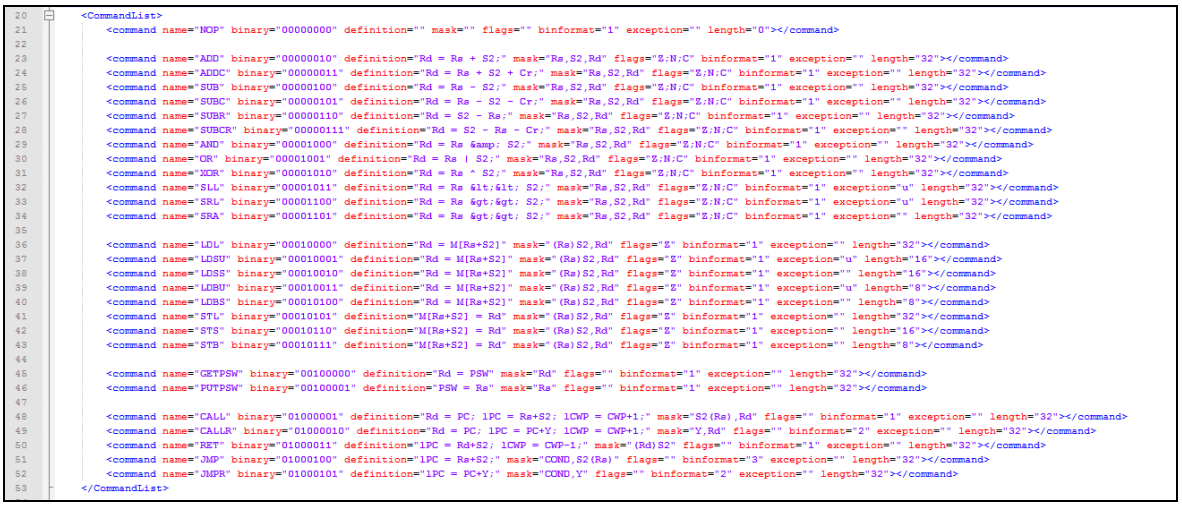
Şekil 27 – Uygulama dili tanımlaması için XML dosyası

Dallanmalar esnasında kullanılan koşullar da ayrı bir XML dosyası üzerinden parametrik olarak tanımlanır. Dallanma koşulları bayraklara baktığı için burada bayraklar üzerinden yeni koşullar eklenebilir veya var olanlar düzenlenebilir.

```
55      <ConditionList>
56          <condition name="BR" binary="00000" condition="1==1"></condition>
57          <condition name="BNE" binary="00001" condition="Z==0"></condition>
58          <condition name="BEQ" binary="00010" condition="Z==1"></condition>
59      </ConditionList>
```

Şekil 28 – XML üzerinden koşul tanımlamaları

Komut seti tanımlamaları da koşulların yer aldığı XML dosyaları üzerinden yapılmaktadır. Buradan komutların isimleri ikili düzendeki karşılıkları ve komutların davranışları değiştirilebilir. Buna ek olarak yeni komutlar da yazılabilir. Eklenen/değiştirilen komutlara ilişkin uygulama içinden herhangi bir düzenlemeye ihtiyaç yoktur. Doğru söz dizimleri kullanıldığı taktirde uygulama yeni komut setleriyle de sorunsuz çalışabilir.



```
<CommandList>
    <command name="NOP" binary="00000000" definition="" mask="" flags="" binformat="1" exception="" length="0"></command>

    <command name="ADD" binary="00000010" definition="Rd = Rs + S2;" mask="Rs,S2,Rd" flags="Z;N;C" binformat="1" exception="" length="32"></command>
    <command name="ADDC" binary="00000011" definition="Rd = Rs + S2 + Cr;" mask="Rs,S2,Rd" flags="Z;N;C" binformat="1" exception="" length="32"></command>
    <command name="SUB" binary="00000100" definition="Rd = Rs - S2;" mask="Rs,S2,Rd" flags="Z;N;C" binformat="1" exception="" length="32"></command>
    <command name="SUBC" binary="00000101" definition="Rd = Rs - S2 - Cr;" mask="Rs,S2,Rd" flags="Z;N;C" binformat="1" exception="" length="32"></command>
    <command name="SUBR" binary="00000110" definition="Rd = S2 - Rs;" mask="Rs,S2,Rd" flags="Z;N;C" binformat="1" exception="" length="32"></command>
    <command name="SUBCR" binary="00000111" definition="Rd = S2 - Rs - Cr;" mask="Rs,S2,Rd" flags="Z;N;C" binformat="1" exception="" length="32"></command>
    <command name="AND" binary="00001000" definition="Rd = Rs &amp;amp; S2;" mask="Rs,S2,Rd" flags="Z;N;C" binformat="1" exception="" length="32"></command>
    <command name="OR" binary="00001001" definition="Rd = Rs | S2;" mask="Rs,S2,Rd" flags="Z;N;C" binformat="1" exception="" length="32"></command>
    <command name="XOR" binary="00001010" definition="Rd = Rs ^ S2;" mask="Rs,S2,Rd" flags="Z;N;C" binformat="1" exception="" length="32"></command>
    <command name="SLL" binary="00001011" definition="Rd = Rs &lt;&lt; S2;" mask="Rs,S2,Rd" flags="Z;N;C" binformat="1" exception="u" length="32"></command>
    <command name="SRL" binary="00001100" definition="Rd = Rs &gt;&gt; S2;" mask="Rs,S2,Rd" flags="Z;N;C" binformat="1" exception="u" length="32"></command>
    <command name="SRA" binary="00001101" definition="Rd = Rs &gt;&gt; S2;" mask="Rs,S2,Rd" flags="Z;N;C" binformat="1" exception="" length="32"></command>

    <command name="LDL" binary="00010000" definition="Rd = M[Rs+S2]" mask="(Rs)S2,Rd" flags="Z" binformat="1" exception="" length="32"></command>
    <command name="LDSU" binary="00010001" definition="Rd = M[Rs+S2]" mask="(Rs)S2,Rd" flags="Z" binformat="1" exception="u" length="16"></command>
    <command name="LDSS" binary="00010010" definition="Rd = M[Rs+S2]" mask="(Rs)S2,Rd" flags="Z" binformat="1" exception="" length="16"></command>
    <command name="LDBU" binary="00010011" definition="Rd = M[Rs+S2]" mask="(Rs)S2,Rd" flags="Z" binformat="1" exception="u" length="8"></command>
    <command name="LDBS" binary="00010100" definition="Rd = M[Rs+S2]" mask="(Rs)S2,Rd" flags="Z" binformat="1" exception="" length="8"></command>
    <command name="STL" binary="00010101" definition="M[Rs+S2] = Rd" mask="(Rs)S2,Rd" flags="Z" binformat="1" exception="" length="32"></command>
    <command name="STS" binary="00010110" definition="M[Rs+S2] = Rd" mask="(Rs)S2,Rd" flags="Z" binformat="1" exception="" length="16"></command>
    <command name="STB" binary="00010111" definition="M[Rs+S2] = Rd" mask="(Rs)S2,Rd" flags="Z" binformat="1" exception="" length="8"></command>

    <command name="GETPSW" binary="00100000" definition="Rd = PSW" mask="Rd" flags="" binformat="1" exception="" length="32"></command>
    <command name="PUTPSW" binary="00100001" definition="PSW = Rs" mask="Rs" flags="" binformat="1" exception="" length="32"></command>

    <command name="CALL" binary="01000001" definition="Rd = PC; lPC = Rs+S2; lCWP = CWP+1;" mask="S2(Rs),Rd" flags="" binformat="1" exception="" length="32"></command>
    <command name="CALLR" binary="01000010" definition="Rd = PC; lPC = PC+Y; lCWP = CWP+1;" mask="Y,Rd" flags="" binformat="2" exception="" length="32"></command>
    <command name="RET" binary="01000011" definition="lPC = Rd+S2; lCWP = CWP-1;" mask="(Rd)S2" flags="" binformat="1" exception="" length="32"></command>
    <command name="JMP" binary="01000100" definition="lPC = Rs+S2;" mask="COND,S2(Rs)" flags="" binformat="3" exception="" length="32"></command>
    <command name="JMPR" binary="01000101" definition="lPC = PC+Y;" mask="COND,Y" flags="" binformat="2" exception="" length="32"></command>
</CommandList>
```

Şekil 29 – XML üzerinden komut seti tanımlaması

Komut setine sağlanan bu esneklik benzer şekilde uygulama dilinde de geçerlidir. Uygulama dil desteğini yine bir XML dosyası üzerinden sağlamaktadır. Bu nedenle sisteme yeni bir dil desteği eklemek oldukça kolaydır. Sistem varsayılan olarak Türkçe ve İngilizce dil desteği vermektedir.

```
<?xml version="1.0" encoding="utf-8"?>
<RISCSimulator lang="tr">

    <form id="1" name="frmMain" value="RISC Benzetim Programı">
        <object parent_id="1" name="dockPanel" value="Sürüklenebilir Panel" />
        <object parent_id="1" name="toolBar" value="Araç Kutusu" />
        <object parent_id="1" name="mainMenu" value="Ana Menü" />
        <object parent_id="1" name="statusBar" value="Durum Çubuğu" />
        <object parent_id="1" name="menuItemFile" value="Dosya" />
        <object parent_id="1" name="menuItemNew" value="Yeni" />
        <object parent_id="1" name="menuItemOpen" value="Aç..." />
        <object parent_id="1" name="menuItemSave" value="Kaydet" />
        <object parent_id="1" name="menuItemClose" value="Kapat" />
        <object parent_id="1" name="menuItemCloseAll" value="Tümünü Kapat" />
        <object parent_id="1" name="menuItemExit" value="Çıkış" />
        <object parent_id="1" name="menuItemEdit" value="Düzen" />
        <object parent_id="1" name="menuItemView" value="Görünüm" />
        <object parent_id="1" name="menuItemRegisters" value="Saklayıcı Listesi" />
        <object parent_id="1" name="menuItemLabels" value="Etiket Listesi" />
        <object parent_id="1" name="menuItemDebug" value="Hata Ayıklayıcı" />
        <object parent_id="1" name="menuItemMemory" value="Bellek Haritası" />
        <object parent_id="1" name="menuItemApplicationLang" value="Uygulama Dili" />
        <object parent_id="1" name="menuItemTurkish" value="Türkçe" />
        <object parent_id="1" name="menuItemEnglish" value="İngilizce" />
        <object parent_id="1" name="menuItemAssemblyLang" value="Derleyici Dili" />
        <object parent_id="1" name="menuItemITURisc" value="ITU Risc" />
        <object parent_id="1" name="menuItemSchemaNew" value="Yeni Stil Görünüm" />
        <object parent_id="1" name="menuItemSchemaOld" value="Eski Stil Görünüm" />
        <object parent_id="1" name="menuItemHelp" value="Yardım" />
        <object parent_id="1" name="menuItemAbout" value="Simülasyon Hakkında" />
        <object parent_id="1" name="menuItemBuild" value="Derle" />
        <object parent_id="1" name="menuItemStartDebugging" value="Derlemeye Başla" />
        <object parent_id="1" name="menuItemRun" value="Çalıştır" />
        <object parent_id="1" name="menuItemStepByStep" value="Adım Adım Çalıştır" />

        <object parent_id="1" name="toolBarButtonNew" value="Yeni" />
        <object parent_id="1" name="toolBarButtonOpen" value="Aç..." />
```

Şekil 30 – Türkçe dil desteği

```xml
<?xml version="1.0" encoding="utf-8"?>
<RISCSimulator lang="en">

    <form id="1" name="frmMain" value="RISC Simulator">
        <object parent_id="1" name="dockPanel" value="Dock Panel" />
        <object parent_id="1" name="toolBar" value="Tool Bar" />
        <object parent_id="1" name="mainMenu" value="Main Menu" />
        <object parent_id="1" name="statusBar" value="Status Bar" />
        <object parent_id="1" name="menuItemFile" value="File" />
        <object parent_id="1" name="menuItemNew" value="New" />
        <object parent_id="1" name="menuItemOpen" value="Open..." />
        <object parent_id="1" name="menuItemSave" value="Save" />
        <object parent_id="1" name="menuItemClose" value="Close" />
        <object parent_id="1" name="menuItemCloseAll" value="Close All" />
        <object parent_id="1" name="menuItemExit" value="Exit" />
        <object parent_id="1" name="menuItemEdit" value="Edit" />
        <object parent_id="1" name="menuItemView" value="View" />
        <object parent_id="1" name="menuItemRegisters" value="Register List" />
        <object parent_id="1" name="menuItemLabels" value="Label List" />
        <object parent_id="1" name="menuItemDebug" value="Debugger" />
        <object parent_id="1" name="menuItemMemory" value="Memory Map" />
        <object parent_id="1" name="menuItemApplicationLang" value="Application Language" />
        <object parent_id="1" name="menuItemTurkish" value="Turkish" />
        <object parent_id="1" name="menuItemEnglish" value="English" />
        <object parent_id="1" name="menuItemAssemblyLang" value="Assembly Language" />
        <object parent_id="1" name="menuItemITURisc" value="ITU Risc" />
        <object parent_id="1" name="menuItemSchemaNew" value="New Style" />
        <object parent_id="1" name="menuItemSchemaOld" value="Old Style" />
        <object parent_id="1" name="menuItemHelp" value="Help" />
        <object parent_id="1" name="menuItemAbout" value="About Simulation" />
        <object parent_id="1" name="menuItemBuild" value="Build" />
        <object parent_id="1" name="menuItemStartDebugging" value="Start Debugging" />
        <object parent_id="1" name="menuItemRun" value="Run" />
        <object parent_id="1" name="menuItemStepByStep" value="Step by Step" />

        <object parent_id="1" name="toolBarButtonNew" value="New" />
        <object parent_id="1" name="toolBarButtonOpen" value="Open..." />
```

Şekil 31 – İngilizce dil desteği

Tüm bu esneklik sayesinde sistem üzerinde değişiklik yapmak oldukça kolaylaşmıştır. Uygulama dili, komut seti gibi değişimlere imkan sağlanmaktadır. Bu sistem özellikle farklı dillerde komut setleri kullanmak isteyen kullanıcılar için faydalıdır. Yeni bir uygulama dili ekledikten sonra yalnızca komutların isimlerini de o dile göre değiştirerek sistemde hiç değişiklik yapmadan kullanabilirler.

6. DENEYSEL SONUÇLAR

Arayüz için farklı dış kaynaklı projelerden yararlanılmış olsa da uygulama ekranının yanıt verme performansında bir sorun olmadığı açıkça gözlemlenebilmektedir. Açılıp kapanan, yeri değiştirilebilen formlar problem çıkarmamakta ve tüm görsel değişimler kaydedildiğinden kullanıcı ekranı bir defa ayarladıktan sonra aynı şekilde sürekli kullanabilmektedir. Editördeki renklendirmeler de performansa olumsuz etki etmemektedir.

Yazılan kodların derlenmesi aşamasında katarlar üzerinde fazla işlem yapılıyor ancak milisaniye bazında sürelerde hata ayıklama işlemleri yapılmakta ve yazılan kod ikili sisteme dönüştürülmektedir. Çok büyük programlar yazılmadığı sürece derlenme aşamasında da sistem problem yaşamamaktadır.

Yürütme zamanında her komutun davranışı karakter katarından alınıp yeni bir C# uygulaması olarak arka planda çalıştırıldığından bazı zamanlarda ufak takılmalar görülmektedir. Yine de kullanımı ciddi olarak etkileyecek veya kullanıcıya sorun yaratacak kadar ciddi bir sorun göze çarpmamıştır.

Önceden yapılmış olan çalışmaya kıyasla uygulama daha fazla disk ve bellek işgal etmektedir. Buna karşın daha güçlü, esnek bir sistem olarak hizmet verdiği ve kullanıcı dostu bir ara yüze sahip olduğu göz önüne alındığında bu farklar göz ardı edilebilir düzeydedir. Eski çalışmayla olan bazı farkları sıralamak gerekirse;

Eski çalışma	**Yeni çalışma**
Uygulama için tek dil desteği	Çoklu dil desteği
Sabit komut seti	XML üzerinden eklenebilir/değiştirlebilir komut seti
Sabit komutlar	Desene sadık kalarak yeni komutlar yalnızca XML dosyasını değiştirerek eklenebilir
Sabit iş hattı	Eklenebilir/Değiştirilebilir iş hattı
Yalnızca komutlarla erişilebilir bellek	Global bellek yapısı ile uygulama derlenmeden/yazılmadan önce bellek ile ilgili değer atamalarını yapmak mümkün
Basit editör	Güçlü bir editör ile kullanıcılar için daha rahat uygulama geliştirme imkanı
Basit ara yüz	Gelişmiş ara yüzü ile daha profesyonel bir IDE görünümü ve kullanım şekli

7. SONUÇ ve ÖNERİLER

Bu çalışmada RISC mimarisinin çalışma mekanizması, RISC mimarisinde uygulama geliştirme, derleyici tasarımı konuları hakkında araştırmalar yapılmış ve bu konular üzerinden proje gerçeklenmiştir. Komut seti yardımıyla geliştirilen uygulamaların mikro işlemcinin anlayabileceği düzene geçişi, komutların iş hattına yerleştirilmesi ve yürütülmesi gibi konularda araştırmalar yapılarak çözüme gidilmiştir.

Yapılan projenin en büyük önemi sistemin tamamen parametrik ve değişimlere karşı hazırlıklı olmasıdır. Buna göre kolaylıkla yeni bir iş hattı yapısı eklenebilir, bellek boyutları düzenlenebilir, uygulamaya yeni komut setleri ve komutlar eklenebilir, uygulama dili desteği genişletilebilir. Bu sayede tek bir okula veya bilgisayar mimarisine yönelik olmak yerine bazı değişiklikler yaparak farklı sistemlere de uyarlanabilir ve global kullanıma açıktır.

Projenin esnek ve global olmasının yanında donanım ve laboratuvar gibi maliyetlere de olumlu yönde etkisi büyüktür. RISC mimarisi hakkında eğitim verilmek istendiğinde farklı mimarileri gösterebilmek için bunlara uygun donanımlar ve satın alınmalı, derleyiciler bulunmalı ve laboratuvarlar kurulmalıdır. Ayrıca bozulan donanımlar için bakım ve yeniden edinme gibi problemler de olacaktır. Bunun yerine uygulama kullanıldığında bazı uyarlamalar ile pek çok farklı yapı için tek bir uygulama üzerinden eğitimler verilebilir, aralarındaki benzerlik ve farklılıklar ortaya konulabilir. Eğitim alanlar da sadece kişisel bilgisayarları yardımıyla bu uygulamayı kullanabilecekleri için her an denemeye ve öğrenmeye devam edebilme imkanına sahip olurlar. Donanımın bozulması eskimesi gibi riskler de yok olur.

8. KAYNAKLAR

[1] Feza Buzluca, "Bilgisayar Mimarisi Ders Notları", 2005,
http://ninova.itu.edu.tr/tr/dersler/bilgisayar-bilisim-fakultesi/22/blg-322/ekkaynaklar/

[2] Bora Uğurlu, "Mikroişlemciler ve Mikrobilgisayarlar", 2007,
http://members.comu.edu.tr/boraugurlu/courses/bm307/content/week4/hafta4_b.pdf

[3] Eric Roberts, "RISC Mimarisi Ders Notları", 2006,
http://www-cs-faculty.stanford.edu/~eroberts/courses/soco/projects/risc/risccisc/

[4] Gökhan Akın Şeker, "Berkeley RISC-I İşlemcisi İçin Platform Bağımsız Benzetim Programı", 2008,
http://www.buzluca.info/tezler/b08_1.pdf

[5] Feza Buzluca, "Object-Oriented Modeling & Design Ders Notları", 2012,
http://ninova.itu.edu.tr/tr/dersler/bilgisayar-bilisim-fakultesi/2097/blg-468e/ekkaynaklar/

[6] Neil Hodgson, "Açık kaynak kodlu bir editör, ScintillaNET dokümantasyonu", 2012,
http://scintillanet.codeplex.com/documentation

[7] Mark Twombley, Steve Overton and Weifen Luo, "Açık kaynak kodlu ara yüz tasarımları için yapışkan paneller, DockPanel Suite", 2012,
http://sourceforge.net/projects/dockpanelsuite/