ISTANBUL TECHNICAL UNIVERSITY
FACULTY OF COMPUTER AND INFORMATICS

Design and Implementation of Fully Associative Instruction Cache Memory on a Processor

Graduation Project

Berkay KÖKSAL
150120025

Department: Computer Engineering

Advisor: Assoc. Prof. Feza BUZLUCA

June 2017

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, 05.06.2017

Berkay KÖKSAL

Table of Contents

Table of Figures

This page is left blank intentionally.

Design and Implementation of Fully Associative Instruction Cache Memory on a Processor (SUMMARY)

Processors should use the main memory smart to keep it available for other hardware as much as the CPU. Over the years, CPU's had many disabilities that have prevented them to benefit their components' full potential. The conductivity technologies of transistors and silicon components have not improved as fast as the sequential circuit technology. Thus, created a severe problem that processor clock frequency was able to work much faster than the transmission frequency of data through the main memory. Processors were forced to wait for the transmission and stay idle for relatively long intervals. To benefit from this wait state, some architectures aim to reduce clock speed or voltage to lower the energy consumption. However, a proper solution was required to solve this problem instead of trying to benefit from this certain drawback. The most solid approach in this matter is that, another memory unit called the cache memory is located very close to the processor would lower the latency. Cache memory would keep the data that the processor needed from the memory recently, considering they might be required in the present future as well. These repetitive requests identified as "hits" would transmit much faster, thus increase the performance of the system.

A cache memory unit is designed and connected to work mutually with a single instruction processor designed on the Logisim simulation environment. For the replacement algorithm of the cache memory the common high performance approach of "Least Recently Used" is used which brings the need of aging bits with it. Aging is important to keep record of what data was needed and when was it needed to not evict them if they are frequently needed by the processor. The cache memory built in the project is an instruction cache that will not edit the data. Therefore, the dirty bits were not necessary and a write-through method was used. The size of this cache is 256 bits in total which is relatively small. The search logic is quite simple as the fully-associative cache addressing is used to make any address to be cached anywhere in the memory. The system is tested with different programs to benefit from the cache and the performance is reported in the report with all required design information.

Design and Implementation of Fully Associative Instruction Cache Memory on a Processor (ÖZET)

Bir bilgisayarı bilgisayar yapan parçanın mikro işlemcisi olduğunu söylemek kesinlikle yanlış bir ifade olmaz. Bir insan için beynin insanın tüm vücudunun hareketinden ve çalışmaya devam etmesinden sorumlu olduğu gibi işlemci de bilgisayarın çalışır durumda kalmasını sağlayan parçası olarak görev yapar. Daha önceden tanımlanmış veya tasarlanmış komutları yürüterek gerekli aritmetik, mantıksal veya giriş çıkış işlemleri ile bilgisayar programlarının tamamlanmasını sağlayan parçasıdır. İşlemci terimi ilk olarak Martin H. Weik tarafından 1961 yılının başında yayınlanan bir raporda kullanılmıştır (Weik, Mart 1961)[1]. Rapor komut işlenmesinin otomatik hale getirilmesi ve verilerin hesaplanmasını hızlandıracak pratik bir çözüm arayışı ile yazılmıştır. Raporda aslında bahsi geçen araştırma bugün hala günümüzde kontrol ünitesi olarak adlandırdığımız, işlemcinin komutların ve verinin beraber yürütülmesini kendi kontrolünde tutan bölümüdür.

İlk üretimlerde kullanılan parçaların zamanla küçültülmesi ve transistörlerin mikro bilgisayarlarda kullanılmaya başlaması maliyeti düşürerek, işlemcilerin kişisel ve endüstriyel alanda pazarlanmaya başlamasını beraberinde getirdi. Böylece bugün hala devam etmekte olan teknoloji yarışı da başlamış oldu. Bu alanda çalışma gösteren şirketler ve kuruluşlar mevcut teknolojinin nasıl daha iyi bir yere getirilebileceği ile ilgili devasa yatırımlar yaparak bu alanın ilerlemesini sağladılar. Temel olarak bakıldığında işlemci teknolojisinin geliştirilebilmesi için iki ana yöntem gösterilebilir. İlki işlemcinin içerisinde yer alan parçaların teknolojilerini geliştirmektir. İkincisi ise mevcut tasarımın işleyişi ile ilgili gelişmeler yapmaktır. İşlemcilerin içerisinde yer alan parçaların daha az yer kaplayacak şekilde ufaltılması, tümleşik devrelerin yayılması gibi çalışmalar ilk çözüm için fayda sağlamıştır. Ancak yalnızca fiziksel olarak parça malzemeleri değişikliği ile bu değişiklikten sağlanan faydanın zamanla araştırma maliyetine değer bulunmaması yüzünden ikinci çözüm çok daha cazip görülmeye başlanmıştır. Bu zaman diliminde üretilen işlemciler, bulundurdukları parçaların tam potansiyelini kullanmaktan tasarımları yüzünden acizlerdi. İyi mühendislik ve yaratıcı çözümlerin uygulamaya dökülmesi ile performansın parça değişikliğine gerek dahi kalmadan arttırılabileceği açık bir gerçekti.

İşlemcilerde çalışmayı yavaşlatacak, zorlaştıracak ve hatta tamamen durdurabilecek tasarımsal engeller mevcuttu. İşlemci parçaları bir diğer bilgisayarda kullanılmaya uygun değildi, saat çevrimlerinde gelinen frekans aralığı sonucu belleğe yazmak için yavaş kalmaktaydı ve işlemci parçaları diğer parçaların işlerini bitirmesini beklemek zorunda bırakılmaktaydılar. Uyumluluk sorununun çözümü için ilk çalışma 1962'de IBM tarafından çıkartılan "işlemci aileleri" çalışması olmuştur. Aynı yazılımın aile içerisindeki farklı

işlemcilerde çalışabilir olduğunu kanıtlayan IBM System 360 işlemcisinin başarısı kullanıcıların daha önceden bilgisayarlar için yapmış oldukları yatırımlarını tekrar kullanılabilir hale getirerek büyük bir başarıya imza atmış ve işlemcilerin uyumluluk sorunlarının ileride neredeyse tamamen ortadan kaldırabilecek çözümlerin önünü açmıştır. Ancak işlemcilerin zamanı kullanma konusundaki sorunları hala devam etmekteydi. Zamanlama ile ilgili en büyük problem işlemcinin bir işlemi yapma süresinin, o işlemi bellekten okuma süresine göre çok daha hızlı olmasından kaynaklı bekleme durumu idi. Bugün kullandığımız işlemci ve belleklerde dahi işlemci saat frekansı yaklaşık olarak 0.3 nanosaniye civarlarında iken, işlemin bellekten okunması 30 nanosaniyeye kadar varan süreler almaktadır. Görüldüğü üzere günümüz teknolojisi ile bile bu sorun tam anlamıyla çözülememiş ancak yapılan araştırmalar ile en aza indirgenmeye veya gizlenmeye çalışılmıştır. Bu alanda yapılan birkaç önemli araştırmaya bu raporun da yazılmasına olanak sağladıkları düşünülerek bu bölümde yer verilecektir.

1980'li yılların başında Berkeley Üniversitesi bellek erişimlerini azaltmak için daha basit komutlar kullanmayı ve belleğe erişecek komutları bu diğer basit komutlardan faydalanarak sınırlandırmayı öne süren RISC yaklaşımını sunmuştur. RISC'de yazılan programlar eskiden kullanılan komplike komutlara göre çok daha uzun olsalar da işlemci tarafından yürütülme süreleri çok daha hızlıdır. RISC işlemcilerinin fikrinde ve tasarımlarında büyük rol oynayan iki büyük bilgisayar mimarisi fikri bulunmaktadır. İlki olan Harvard Mimarisi komutların ve verinin yazıldığı belleklerin birbirinden fiziksel olarak tamamen ayrılmalarını önermiştir. Bu sayede işlemcinin hem yeni komutu okuması hem de veriyi belleğe yazmasının aynı anda yapılabilir olmasını mümkün oluyordu. Bu tasarımda eski tasarımlardan farklı olarak, veri ve komutlar için işlemciye ulaşan iki farklı veri hattı oluyordu. Aynı zamanda komutlar ve verilerin farklı boyutlara ve uzunluklara sahip olabilir kılıyordu. Bu alandaki ikinci çalışma olan Von Neumann Mimarisi ise buna gerek olmadığını, komut ve verilerin bir arada tutulabilir olduğunu savunuyor ve belleğin belirli bölümlerinin komutlara ayrılmasını öngörüyordu. Bu yaklaşım işlemcinin komut ve veri erişimlerinde beklemesine sebep olarak dar boğaz yaratıyor ve işlemciyi yavaşlatarak performanstan ödün vermesine sebep oluyordu.

Tüm bu çalışmalara rağmen daha önceden de bahsedilmiş olan işlemcinin diğer parçaları beklemek zorunda olması problemi hala devam etmekteydi. İletkenlerin ve yarı iletkenlerin fiziksel yapılarının yüksek frekanslarda yeterli hıza ulaşamaması yüzünden bu sorun günümüzde dahi çözülememiştir. İşlemcinin saat frekansı ve belleğin erişim süresi arasındaki farkın yarattığı bu bekleme durumunu çözmek amaçlı olmayan ancak bu durumu faydalı bir durum olarak kullanmayı veya kısmen gizlemeyi hedefleyen bazı çalışmalar bulunmaktadır. Faydalı durum için bir örnek vermek gerekirse; bu bekleme süresi boyunca işlemci saatinin yavaşlatılması veya voltaj şiddetinin azaltılması sağlanarak gereksiz enerji tüketimi engellenebilirdi. Bekleme süresini olumlu kullanmayı hedefleyen tasarımlar performans açısından fayda sağlamadıklarından, bilgisayar bilimi daha çok gecikmenin azaltılmasını veya görünmez olmasını sağlayacak çözümlere odaklanmıştır.

Bilgisayarların bekleme sorunları çözmeye dayalı yapılmış çalışmalara örnek olarak iş hattı yaklaşımı gösterilebilir. Komutların işlenmesinin bir iş hattındaki segmanlara bölünmesi

sağlanarak farklı parçaların sürekli olarak veriyi işlemesi sağlanır. Komutun işlemi yapılırken bir sonra gelecek komutun bellekten alınmaya başlanması sağlanarak, ilk komutun sonucu hesaplandığı anda sonraki komut neredeyse hazır hale getirilir böylece erişim süreleri arasındaki bekleme durumu görünmez denebilecek seviyede azaltılır. Alt parçalara bölünebilecek her işlem iş hattı mantığı ile tasarlanabilir. Birbiri ile yakın karmaşıklığa sahip olan segmanlar, aynı işlemi farklı verilere uygulamayı sürekli olarak devam ettirerek iş hattının devamlılığını sağlarlar. İş hattı yaklaşımı, aynı zamanda paralel yürütme gibi birçok özelliği de mümkün kılarak sistemin performansına katkı sağlar.

Bu alanda yapılan bir diğer başarılı çalışma ise cep bellek yaklaşımıdır. Belleğe göre erişim süresi daha kısa olacak mesafede ve yapıda ufak boyutlu ve hızlı bellekler işlemciye yakın olarak konumlandırılır. Bellek tarafından yakın zamanda kullanılan veriler bu alanda saklanarak tekrar ihtiyaç duyulması halinde bellek erişimine gerek kalmadan daha hızlı bir şekilde iletilirler. Bu noktada cep bellekte gerçekleşen vuru veya ıska oranları cep belleğin sistem performansına olan katkısını etkileyen en büyük etkendir. Cep belleklerin ilk pazarlanan modeli 8kB ve 16kB cep bellek gömülü modelleri ile 1985 yılında üretilen intel 80386 olmuştur. Bu projenin tasarımsal kısmının cep belleklerle ilgili olmasından dolayı araştırmanın devam eden bölümü cep belleklerin tasarımları ve teknolojilerindeki ilerlemeler üzerine olan araştırmalar odaklanılarak devam edilecektir.

Logisim ortamında tasarlanmış olan tek komutlu işlemci için yapılacak olan cep belleğin çalışma ve adresleme fonksiyonları ile ilgili araştırmalar yapılmıştır. Aslında “first-in, first-out” yani ilk kaydedilenin ilk dışarı atılan veri olması prensibi tasarım olarak daha kolay gerçeklenebilmesine rağmen, günümüz cep belleklerinin sıkça kullandığı “Least Recently Used” mantığına göre daha düşük başarı oranına sahiptir. Bu sebepten dolayı LRU yönteminin tasarımın performansı ve projenin mühendislik değeri açısından daha uygun olacağı düşünülerek tasarım bu metoda göre şekillendirilmiştir. Tasarımda, doğrudan yazma yöntemi kullanıldığından, “kirli” bit bayrağına ihtiyaç olmamıştır, zira cep bellek içerisine veri yazılmamış olup sadece komutlar yazılmıştır. Bu sebeple de cep bellek ve normal bellek arasında veri farklılığı yaşanması mümkün olmamaktadır. Ancak, LRU yaklaşımında hangi satıra ne zaman başvurulduğu önem kazandığından, kayıt tutulması için yaşlandırma bitleri kullanılmıştır.

Cep belleğin boyutunun logaritmasına karşılık yaşlandırma bitlerinin sayısı değişmektedir. Projenin başarılı olması için gerekli bulunan 8 satırlık cep bellek, 3 bitlik yaşlandırmaya ihtiyaç duyar. Yaşlandırma bitleri ıska durumunda yaşı en yüksek olanı dışarı atarak yerine yeni veriyi yaşı sıfır olacak şekilde alır. Vuru olması durumunda ise vuruya maruz kalan satırın yaşı sıfırlanır, yaşı bu satırdan düşük olanları yaşı arttırılırken, yüksek olanların yaşları sabit tutularak yaş çakışmaları bitler bazında engellenerek sistemin sürekli farklı yaşlara sahip satırlara sahip olması sağlanır. Bahsi geçen yaşlandırma algoritmasının gerçeklenmesinde ise ortaya bir çok başka problem çıkmıştır. Öncelikle aynı saat çevrimi içerisinde hem vuru olan satırın yaşının sıfırlanması hem de bu satırın yaşının diğerleri ile kıyaslanması mümkün olmamaktadır. Bu sebeple vuru olan satırın yaş sıfırlanma sinyali gecikmeye uğratılarak önce diğer satırların işlemden geçmesi sağlanmıştır. Bu gecikme

işlemcinin saat çevriminin her zaman iki katı yüksek frekansa sahip olacak şekilde ayarlandığından kapıların çalışmasını engelleyecek frekanslara yükselmediği sürece, işlemcinin sonraki saat çevriminden önce yapılacağından çalışmayı etkilemeyecektir.

Sistem ilk çalışmaya başladığında cep bellek içerisindeki adresler sıfır olarak başlamaktadır. İlk komut sisteme alınmaya çalışıldığında bu sebepten dolayı cep bellek vuru olduğunu düşünerek işlemciye yanlış veriyi göndermekteydi. Sorunun çözülmesi için sıfır adresi es geçilebilir veya cep bellek gecikme ile çalışmaya başlatılabilirdi. Ancak bu çözümler sistemin çalışma yeteneklerini kısıtladıklarından ve kötü mühendisliği beraberinde getirdiklerinden, daha başarılı ve genel bir çözüm olan "geçerli" bitlerinin satırlara yerleştirilmesi uygun bulunmuştur. Satırlar başlangıçta geçersiz olarak bayraklanmış ve ancak geçerli bir veri ile yüklendiklerinde vuru sinyali verebilecek duruma getirilerek başlangıçta sıfır olan adreslerin sisteme müdahale etmeleri engellenmiştir. Geçerli bitleri aynı zamanda sıfır adresinin de cep belleğe doğru yazılmasını sağlayarak cep belleğin tüm bellek adreslerini kapsamasını sağlamıştır.

Cep belleğin sistem içerisine gerçeklenmesi için çalışmalar başladığında, adresleme algoritması ile ilgili de çalışmalar yapılmıştır. Doğrudan haritalama ve tam ilişkili önbellek arasında bir kıyaslama yapılmıştır. Doğrudan haritalama tasarımsal olarak en basit çözümdür, vuruyu test etmek için her adresin yerleşebileceği alan kısmen belli bir blok içerisinde olduğundan arama fonksiyonu oldukça başarılıdır. Ancak yine aynı sebepten dolayı kayıt performansı düşük olduğundan vuru performansı en düşük tasarımdır. Tam ilişkili önbellek ise herhangi bir adresin cep bellekte herhangi bir yere kaydedilebilmesini mümkün kılar. Bu sayede paralel olarak vuru kontrolü yapılırken, geçerli veriye sahip tüm satırlar aranmak zorunda olduğundan zamandan zarar edilse de kayıt ve vuru performansı olarak doğrudan haritalamaya göre daha başarılıdır. Proje tasarımınca tam ilişkili önbellek yaklaşımı kullanılarak indekslemeye ve bloklamaya olan ihtiyaç ortadan kaldırılmış ve bellek içerisinde aramanın paralel yapılması için paralel komplike mantıksal devreler kullanılmıştır.

Cep bellek modülü, sayfa değiştirme ve adresleme algoritmalarını başarılı bir şekilde kontrol edebilir hale getirilmiştir. Cep belleğin bir satırı 3 yaşlandırma biti, 1 geçerlilik biti, 12 bitlik adres alanı ve 16 bitlik veriyi tutan toplamda 32 bitlik bir alana sahiptir. Proje kapsamında 8 satır kullanılarak toplam 256 bitlik bir cep bellek tasarlanmıştır. Tasarlanmış bellek modülü kendisi ile paralel olarak bağlanarak çalışma yeteneğine sahip olduğundan çoğaltılarak kullanılabilmektedir. Test sonuçlarının daha iyi gözlemlenebilmesi için her modül içerisine çalıştığı süre zarfında gerçekleşen vuru ve ıska sayılarını kaydeden bölümler yerleştirilmiştir.

1. Introduction

There is no doubt that the most significant component in a computer is the Central Processing Unit (CPU) which plays the role of the brain of a computer. The component that executes predefined or predesigned instructions to complete computer programs by performing basic arithmetic, logic, input and output operations. The term CPU was first referred by Martin H. Weik in early 1961 on a survey report that was focused on a practical solution to accomplish automatic computing and data processing. The report was aimed to make progress on designing of the first Control Unit (CU) which is still the term as we call it today to provide automatic instruction running and processing data alongside with it (Weik, March 1961) [1].

After the CPU's to become commercially available for users the technology race has started to improve previous designs. There are many ways to improve current performance of a CPU. One is to change their component material which was effective during early builds. After fully implementation of transistors and silicon components the component change became inefficient due to the high costs and lack of further technology in the relevant field. Therefore, the second method of improvement which is the change of design has taken the throne and kept its lead until today. Early CPU's were most likely to not use their parts perfectly efficient. It was known that with proper engineering, creative solutions could be implemented which would increase the performance even embedded with same parts.

Early CPU's had plenty of design disabilities. Such as; every part was unique for one CPU which was incompatible for any other, the clock frequency was not enough to process enough data for some operations and most important issue was that the CPU parts were waiting for the other parts to finish their work. The incompatibility problems were partially fixed by IBM's new approach of "family of computers" which was introduced in 1962 to ensure same software to work on future builds to not waste users' previous investments. The first CPU family's System/360 success leads more families to follow after it to solve this problem completely (Bosworth) [2]. However, the designs were still not timely efficient considering the access time for Random Access Memory (RAM) was still too high to handle between instructions. Many researches were made in this field that should be mentioned in this part, considering them having a huge impact to make this project possible.

In early 1980's Berkeley University has started its researches to minimize the RAM accesses by removing instructions that uses RAM access too much by much more simpler instructions which are faster and inexpensive. This approach is called Reduced Instruction Set Computing (RISC). On RISC approach the loading and storing to RAM is made with only a few instructions, and with divided instruction the CPU can work on higher speed even though the required code becomes relatively larger than a CISC code equivalent (Flynn, 1995) [3]. Two more very important suggestions followed up the RISC approach to optimize the RAM usage: The Harvard Architecture and Von Neumann Architecture. The Harvard Architecture proposes that keeping instructions and data in the same memory unit creates a bottleneck on accessing them and intends to physically separate them into two different memory units

which would make simultaneously access possible. Therefore, having two busses makes CPU able to read an instruction and also read/write data to these memory units at the same time. It also brings the capability of having different size, width, timing for instructions and the data. Von Neumann Architecture defends another approach that instructions and data can be kept together which makes new instruction fetch and data operations to be made simultaneously impossible due to the one bus access design that creates von Neumann bottleneck. The bottleneck creates a shortage of performance on the system which proves that Harvard architecture process data much faster than Neumann architecture (Harris, 01 Jan. 1970)[4].

There is no doubt that researches above have made CPU performance increase compared to very early designs. However, the biggest problem was still not solved by any of the approaches above. The problem of "wait state" is the state in which the delay happens between the memory and the CPU that originates from the frequency difference between the CPU core clock and the access time between RAM-CPU. Today, on modern desktop CPUs the achieved several GHz on CPU clock means about 0.3 to 0.5 nanoseconds is still much faster than a RAM access of about 15-30 nanoseconds (Taylor, 2015)[5]. Some computers aim to reduce clock speed or clock voltage during the waiting period to reduce energy consumption (Meng, 2013)[6]. The proper solution or a valid design to this problem is still not discovered due to the lack of conductivity for parts built even today. However, again creative researches are made to provide significant reduction to almost make it invisible on the output performance. Some of solid approaches in this purpose are; the cache memory and operand forwarding on the instruction pipeline. Considering the fact that this project aims to design a cache memory to solve the memory access latency, the further research will be focusing on the cache memory that also reduces access time dramatically.

The projects aims to design a cache memory module for a single instruction processor designed on Logisim environment (Burch, 2014)[7]. Research focused on the page replacement and addressing algorithms to decide how to build this memory module. The "first-in, first-out (FIFO)" approach is the simplest solution found for the page replacement. Due to its low hit performance and less scientific value, a more complex and common principle of "Least Recently Used (LRU)" is decided to be used. For the design the approach of "write through" is used which means the data will be written to the cache and the memory together. Making them having different values impossible, a need of a "dirty bit" in the design was not necessary. The LRU algorithm makes the timing of reaching per line important, which brings the need of "aging bits" that was used in the design.

The number of aging bits will increase with the logarithm of the size of the cache memory. For the project design to be successful, 8 lines of cache memory were found to be sufficient that would need only 3 bits for aging. When a miss occurs in the cache memory, the line with the maximum age will be evicted and replaced with the new data while its age will be set to zero. When a hit occurs, the line with the hit will have its age cleared to zero while others will adjust themself according to their previous age. The other lines that have less age than the hit line will have their ages increased while the ones that were older will hold their current ages. This algorithm guarantees to have different ages for each line no matter a miss

or hit occurs. The implementation of the aging bits however brings many problems to the design. It was not possible to clear the hit age and compare it with the other lines in the same clock signal. Therefore, the comparison is made first while the clear operation will be delayed. This delay could not be more than the clock frequency of the CPU due to the fact that it might cause a failure on the cache memory. Thus, the delay was set to be a half clock time of the CPU by using the master slave D-latch design. As long as this boosted CPU frequency is enough to keep the logic parts working, the processor will not notice any of this delay whatsoever.

The system starts for the first time, and the cache memory has the addresses of zero saved in each line. Because of this, when the first instruction arrives to the memory unit, the cache will act like there was a hit scenario on the address of zero and feeds the bus with its dirty, wrong zero value. A few solutions were found for this problem such as, ignoring the address of zero or a delay to make the cache memory enabled after the address of zero is passed. These solutions limit the capabilities of the memory unit and also they cause bad engineering for the system. Any of these solutions will make the system not be able to save the address zero even if it was necessary or even collapse the system in case of a branch to the address of zero. Therefore, a more successful and general solution called “valid bit” is used to solve the problem. A valid bit that starts with invalid in the start is inserted into each line of the cache memory. The cache lines will only be able to react to a hit scenario when their value is “valid” after they are loaded with a value. Valid bit approach solves the problem of false hit on zero address and makes the cache memory to work on every address of the memory.

After the page replacement algorithm is tested and before the implementation is done for the main system, more research was necessary to decide the addressing functionality of the cache memory. A decision was required to be making between the direct-mapped or fully associative cache memory. The direct mapping was the simplest solution to design, where the addresses are saved in a particular place in the memory and easy to be searched. However, due to this feature it has the worst performance of hit ratio since the addresses have to be evicted when another close one arrives to the system. On fully-associative addressing, an address can be saved anywhere in the cache memory which brings the need of parallel search over the whole cache memory for each line. Even though the search takes more time, the hit ratio is much higher than the direct-mapping since they can be placed anywhere in the cache memory. During the implementation, the fully associative approach is used which makes indexing not required but needed a complex logical circuit of its own to search the whole cache lines parallel.

The design cache memory module was implemented to the main system that is capable of making page replacement and addressing successfully. Each line of the cache memory will consist of, 3 bits of aging, 1 bit of valid, 12 bits of address and 16 bits of data value that adds up to a total of 32 bits. Project module will have 8 lines which has in total of 256 bits of cache memory size. This cache module is capable of being connected parallel to another one in order to increase the size of the cache memory. Each cache memory module is inserted

with special type of registers to count the total occurrence of hit and miss while it's working to make testing results be clearer.

2. Project Description and Plan

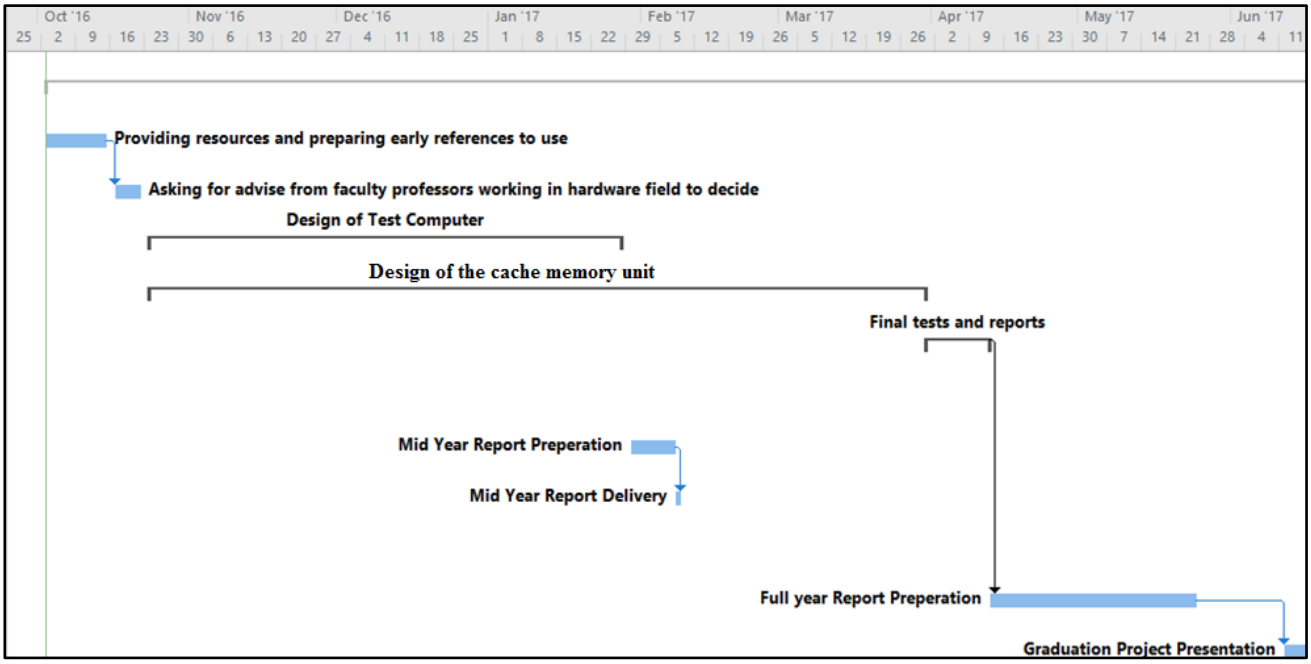


Figure 2.1 Full year gantt chart

Full Year Plan

The project started before October as the planning part. But the given date of first project plan is October; it is not mistaken to state that it has started officially on October 2016. The project has 2 main separate parts to be dealt with. First, design of test computer and second, the design of the cache memory has to be completed. It is already stated that test computer plays a very sufficient role in the project, which is why it consumed almost all first semester. The planning of cache memory also started to be dealt with in semester one. Before the end of first semester; the test computer was ready to be implemented. The planning and simulation of the first prototype of project was also ready to be implemented. For the spring semester the project design part fastened up and made the implementations and brings the project to the test stage. Then after the tests are done and results are collected for this final report to be written.

Fall Semester

Task Name	Duration	Start	Finish
Providing resources and preparing early references to use	10 days	Mon 3.10.16	Fri 14.10.16
Asking for advice from faculty professors working in hardware field to decide	5 days	Mon 17.10.16	Fri 21.10.16
Design of Test Computer	**70 days**	**Mon 24.10.16**	**Fri 27.01.17**
Design of single instruction CPU	**50 days**	**Mon 24.10.16**	**Fri 30.12.16**
Decision making on the design units	5 days	Mon 24.10.16	Fri 28.10.16
Designing and prediction possible casualties	25 days	Mon 31.10.16	Fri 2.12.16
Adding pieces together to make CPU work	15 days	Mon 5.12.16	Fri 23.12.16
Error fixing	5 days	Mon 26.12.16	Fri 30.12.16
Planning of replacement algorithm	**40 days**	**Mon 24.10.16**	**Fri 16.12.16**
Planning the aging and dirty bits	5 days	Mon 24.10.16	Fri 28.10.16
Simulation and error fixing of first prototype	25 days	Mon 31.10.16	Fri 2.12.16
Mid-Year Report Preparation	9 days	Mon 30.01.17	Thr 9.02.17
Mid-Year Report Delivery	1 day	Thr 9.02.17	Thr 9.02.17

Figure 2.2 Fall semester schedule

The first part of the project, the design of the single instruction CPU was made in the fall semester. All units of the computer was built and tested with programs written in its own machine language. However, only designing the test computer would push the main cache memory design to a very small time in the second semester and clearly was not enough to write the mid-year report. Therefore, the theoretical design of the cache replacement algorithm has already started in the fall semester.

Spring Semester

Task Name	Duration	Start	Finish
Preparing the prototype to be connected to the test computer	10 days	Mon 5.12.16	Fri 16.12.16
Implementation of first prototype	**20 days**	**Tue 28.02.17**	**Mon 27.03.17**
Implementing the design to test computer	5 days	Tue 28.02.17	Mon 6.03.17
Error fixing and redesigning if necessary	15 days	Tue 7.03.17	Mon 27.03.17
Preparing for testing	3 days	Tue 28.03.17	Thu 30.03.17
Final tests and reports	**9 days**	**Fri 31.03.17**	**Wed 12.04.17**
Collecting anonymous code blocks to test	3 days	Fri 31.03.17	Tue 4.04.17
Converting code blocks to machine code	5 days	Wed 5.04.17	Tue 11.04.17
Running codes to test RAM usage drops	1 day	Wed 12.04.17	Wed 12.04.17
Full year Report Preparation	30 days?	Thu 13.04.17	Wed 24.05.17
Graduation Project Presentation	5 days?	Mon 12.06.17	Fri 16.06.17

Figure 2.3 Spring semester schedule

By the time that the spring semester started, the test computer was able to work by itself perfectly and the cache memory was ready to be implemented over it with its own tested algorithm. After the implementation is done, an extra time required to fix the errors of the design and finally in March 2017 the final design was ready to be tested. The tests are done using various kinds of programs and results are collected to be given in this report.

3. Theoretical Information

In this part of the report, every academic term regarding the design found while doing literature review will be given and explained.

Hardware Loop Buffer

In case of a loop detection the loop buffer saves every instruction and the branch conditions if size of the program does not exceed buffer size, in order to use it again but without branch prediction required. The loop buffer speeds up the loop execution time by eliminating branch decisions and comparisons from the instructions partially. A well designed hardware loop buffer unit would not bring any extra complexity to the system while providing upto 6.2% speedup of execution time for the processor (Scott DiPasquale, 2003)[8].

Cache Write Policy

There is more than one way to manage replacement timing of data cached in the cache memory. The most common methods when it comes to writing are “write-through” and “write-back”. In order to design a cache on a simulation, a writing method has to be decided. Therefore, both methods are examined in details.

Write-Through

Write request writes the new data on cache and memory simultaneously. By doing soi the system guarantees the values to be synchronous. Meaning that no dirty read or dirty write would be possible on a system using write-thought mythology. When a value is caches, and requested again by the processor, it would be fed to the system bus using the cache memory and not the main memory. That is why using a write-through actually does an extra unnecessary main memory writing operation which brings access latency.

Write-Back

Write operations are only made over the cache memory and not the main memory. This approach results the main memory and cache memory to hold different values for the same address at the same time. When a hit occurs on such address, it will be fed from the cache memory which is in fact the most recent version of this value. The value of the main memory is only adjusted when the cached line has to be evicted. Therefore, there will be only one main memory write operation, once the cache line is being evicted. The write-back approach has lower latency and has a better performance on a system that edits values of memory cells frequently.

Considering their advantages and disadvantages, a decision is made to use write-through on the project design. The write-back approach is mainly used for data caches, where the data is constantly being edited and recalled by different instructions to be read from the cache memory. However, the project aims to design an instruction cache memory. The instruction codes are not likely to be changed by any instruction whatsoever. Considering they will not be edited, the advantages of write-back would be unnecessary.

Cache Replacement Policy

When all lines of a cache memory is full, some lines have to be evicted and be replaced with the new incoming data. Cache replacement algorithm decides which lines will be replaced according to its principal. There are many different methods to discuss, but some are much popular and will be explained here to choose the proper method for the project design.

FIFO and LIFO

First in first out and last in first out also would be represented with queue and stack mythology were proposed to be the first solution for the replacement problem. These algorithms do not keep records of how many or how often any line was addressed in the cache. They intend to replace a line when its life cycle is over as it can either be the first or the last to be evicted.

Least Recently Used

Least recently used method intends to keep records of which line is addressed when in order to decide. When a line has to be replaced, the line which was not addressed for the longest time will be the one to be replaced. The detailed version of the LRU mythology is also given in the analysis part of this report.

Random Replacement

Random Replacement, as can be understood from its name, intends to replace a random line in the cache to cache the incoming value. Depending on how “lucky” it could be defines its success. A random replacement algorithm could either evict bad lines and end up with a terrible performance, or evict the good lines and perform better hit ratio than any other method as well. RR’s simplicity made it quite handy for small sized cache designs and actually was used in ARM-R Series Cortex processors (ARM Limited, 2001)[9].

Replacement algorithm is the most important part in the design of the cache memory considering it would be the part to fulfill the purpose of this design. Least Recently Used algorithm is used with proper aging bits and valid bit being implemented in the project.

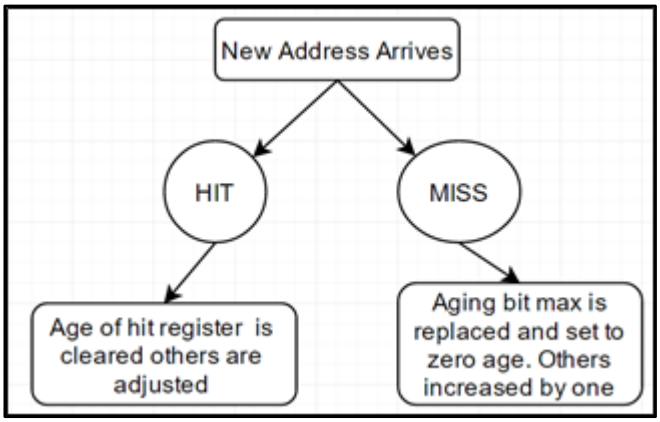


Figure 3.1 Hit and miss flowchart

There are two possible situations that can occur once a new address arrives. Either it was already registered or it is a new address. The term used for these cases are “hit” and “miss”. The replacement algorithm works under the condition of a miss occurrence. More detailed explanation of this is given on the modeling part of the report.

4. Analysis and Modeling

Build Analysis

As the project is based on a design of a cache memory, there must be a test computer built to use this cache unit. For the design of the test computer, Berkeley 1 is used since it is a proper RISC system that is easy to be implemented that has a single instruction processor.

- The system will have a memory unit that has a width of 12 bits on the addresses while the data will be 16 bits as it was in the original design.
- A 16 bit ALU is built and tested to be used to manipulate the AC of the computer.
- 12 bit address register, 12 bit program counter, 16 bit data register, 16 bit accumulator, 8 bit input, 8 bit output, 16 bit temporary register, 16 bit instruction register is built and connected as registers.
- The control unit is built which is controlled with a 4 bit sequence counter to make fetch, decode and operations using the timing of the sequence counter.
- The flags of start, interrupt and data values of the accumulator are implemented as well.
- The system is tested with simple programs and was able to work successfully.

After the test computer was built and tested the cache memory was ready to be designed.

- The 3 aging bits were implemented and tested simultaneously. Increase, hold and clear signals are logically expressed and implemented.
- The valid bit is designed to be only valid after the first load occurs on the line. The values used over the design will have to be selected only if they are valid.
- The aging bits are tested with various possible input cases. Such as, the hit scenario, the miss scenario, the half full hit/miss or the eviction due to aging.
- The hit and miss counters are implemented to make testing results to be observed easily.
- The tests are done to report the performance of the system.

ALU Design Models

Below is given the modeling of ALU inputs by using 5 select bits. A simple 16 bit ALU that can manage RISC operations with only 16 operations will be largely sufficient for this project's needs.

S3	S2	S1	S0	C_{in}	Operation	Symbol
0	0	0	0	0	Transfer A	F < A
0	0	0	0	1	Increment A	F < A + 1
0	0	0	1	0	Addition	F < A + B
0	0	0	1	1	Add with carry	F < A + B + 1
0	0	1	0	0	Subtract with borrow	F < A + B'
0	0	1	0	1	Subtract	F < A + B' + 1
0	0	1	1	0	Decrement	F < A – 1
0	0	1	1	1	Transfer A	F < A
0	1	0	0	X	AND	F < A ∧ B
0	1	0	1	X	OR	F < A V B
0	1	1	0	X	XOR	F < A ⊕ B
0	1	1	1	X	Complement	F < A'
1	0	0	0	X	Logical Right Shift	F<shrA, A(15)L=0
1	0	0	1	X	Arithmetic Right Shift	F<shrA, A(15)L=A(15)
1	0	1	X	X	Circular Right Shift	F<shrA, A(15)=A(0)
1	1	0	0	X	Logical Left Shift	F<shlA, A(0)=0
1	1	0	1	X	Arithmetic Left Shift	F<shlA, A(0)=0
1	1	1	X	X	Circular Left Shift	F<shlA, A(0)=A(15)

Figure 4.1 ALU Input Table

Cache Write Methods

There are two main methods called write through and write back that could be used in the design. Write through intends to write the input value to the cache and the memory at the same time, where on write back the value is only written to the cache and gets written to the memory only when it is being replaced. Write through benefits from the situations where there is a miss since it does not need to write any data back to the main memory, where write back benefits from not writing the same block repeatedly to the memory (Evans, 2017)[10].

As my design would only work on the instruction cache side and not the data caching, the write back would be unnecessary since the values of instruction codes will not be edited whatsoever. The instructions are read from the memory in case of a miss or from the cache in case of a hit occurrence but they are not edited. Therefore, the write through would be sufficient on an instruction cache as it was mainly used in many common cache memory designs.

The cache designed for this project will operate on a write-through basis with a random placement algorithm of fully-associative addressing.

Hit and Miss Algorithm

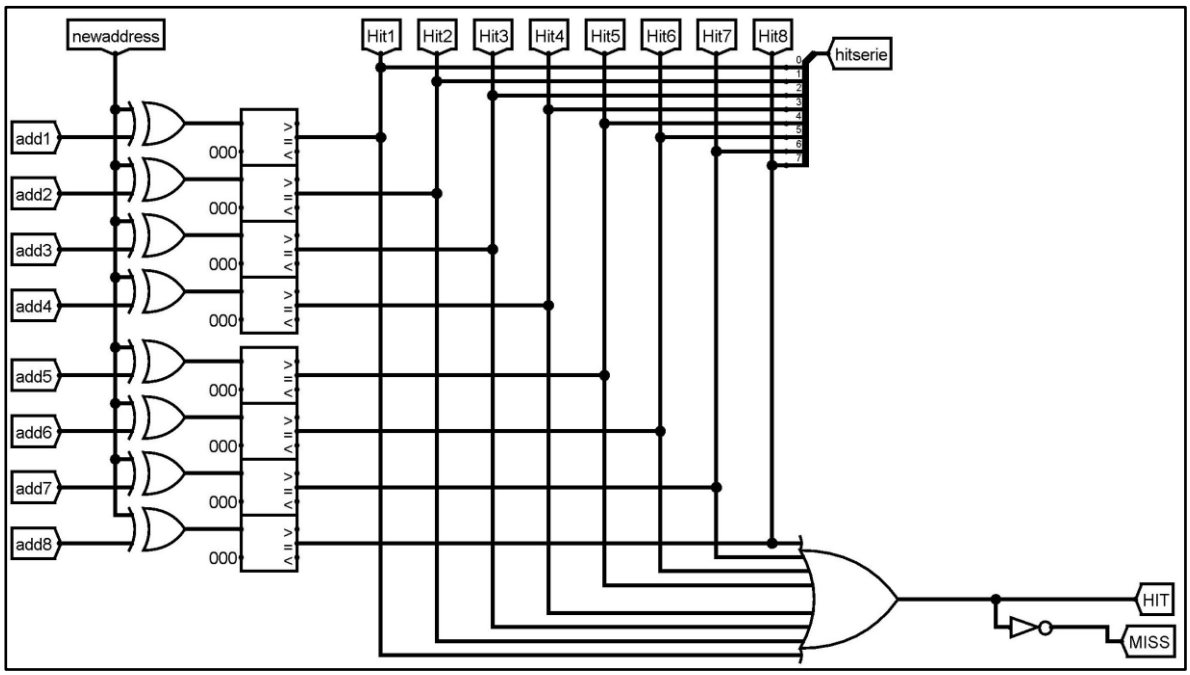


Figure 4.2 Hit and Miss Logical Expression

The new address is made XOR with every registered address and the results are compared to see if any of them was indeed equal to the incoming new address or not. The results of every "if equal zero" is also important for the decision of what will happen in case of a HIT afterwards.

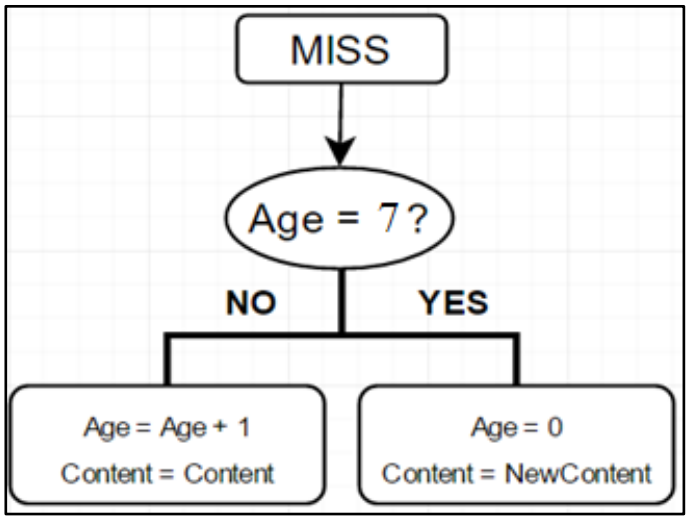


Figure 4.3 Miss Flowchart

MISS Situation

There will always be a line with age 7. During a miss, the line with age 7 will launch the LOAD value for itself which will replace its own values with the incoming data. This line will also CLEAR its age back to zero due to the fact that it is now the most recently used. Every other line that had a miss will only INCREASE their age by one and HOLD their contents for the next operations to come.

HIT Situation

For the hit the situation is a bit complicated. There are 8 different possibilities in the HIT address when there are 3 bit aging bits implemented. To understand the algorithm better a simulation for a 2 bit aging can be observed below. On a 2 bit aging bit there will always be a line with 00, 01, 10 and 11 ages in the system and the hit can occur on any of them.

	OLD VALUES				NEW VALUES			
	Age 1	Age 2	Age 3	Age 4	Age 1	Age 2	Age 3	Age 4
HIT AGE 0	00	01	10	11	00	01	10	11
HIT AGE 1	00	01	10	11	01	00	10	11
HIT AGE 2	00	01	10	11	01	10	00	11
HIT AGE 3	00	01	10	11	01	10	11	00

CLEAR HOLD INCREASE

Figure 4.4 Hit Age Adjustment Example

To generalise the problem, when a hit occurs on the age of that line will have its age cleared to become the most recently used one. For the other lines the decision will be made by looking their age compared to the one that got hit. If the age of the line is less than the hit age it will increase, or else it will hold its value. This algorithm is used to always keep ages different from each other in order to prevent any multiple maximum age situation to occur.

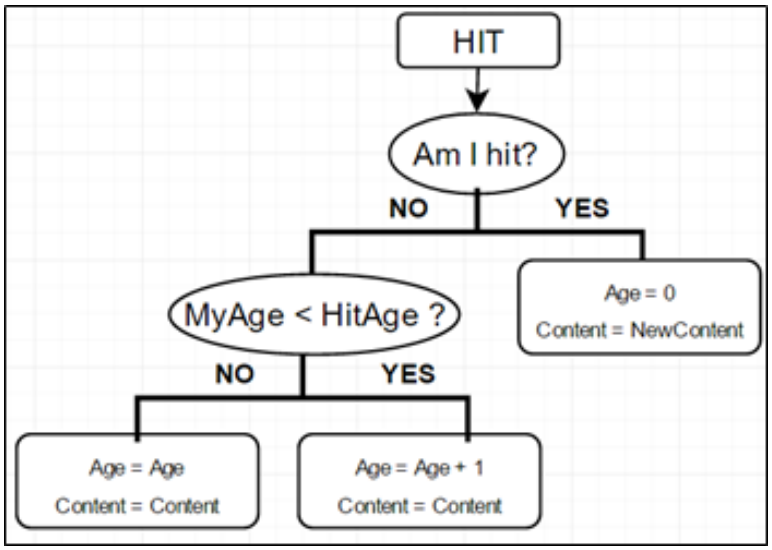


Figure 4.5 Hit Flowchart

Cache Addressing

The cache addressing method defines the slots of each entity to be placed in the cache memory. A decision was to be made between direct mapped or fully associative mapped cache. Direct mapped cache would intend to place addresses into a certain place in the cache memory. The search would be easy since the address should be in that block, which makes searching the whole cache unnecessary. However, again because of this reason some addresses evict each other even there is still space left in the cache memory, which reduces the hit ratio dramatically. For direct mapped addressing to be useful the size of cache should be relatively higher than what is planned for this project. Therefore, direct mapped cache was found to be unnecessary for this project's need.

On fully associative cache, an address can be cached anywhere. However, there is a tradeoff to consider here. In order to increase the hit ratio by being able to cache randomly, the system will have to search the entire cache. Searching would create a delay if the size is bigger. How this project, on an 8 line cache this delay is very small compared to the benefit it would provide over the hit ratio. Therefore, fully associative cache mapping is used in the project.

5. Design, Implementation

Arithmetic Logic Unit

The ALU from its name, it consists of two parts, the Arithmetic Unit and the Logical Unit. The basic designs of both and a combined ALU designs will be given here.

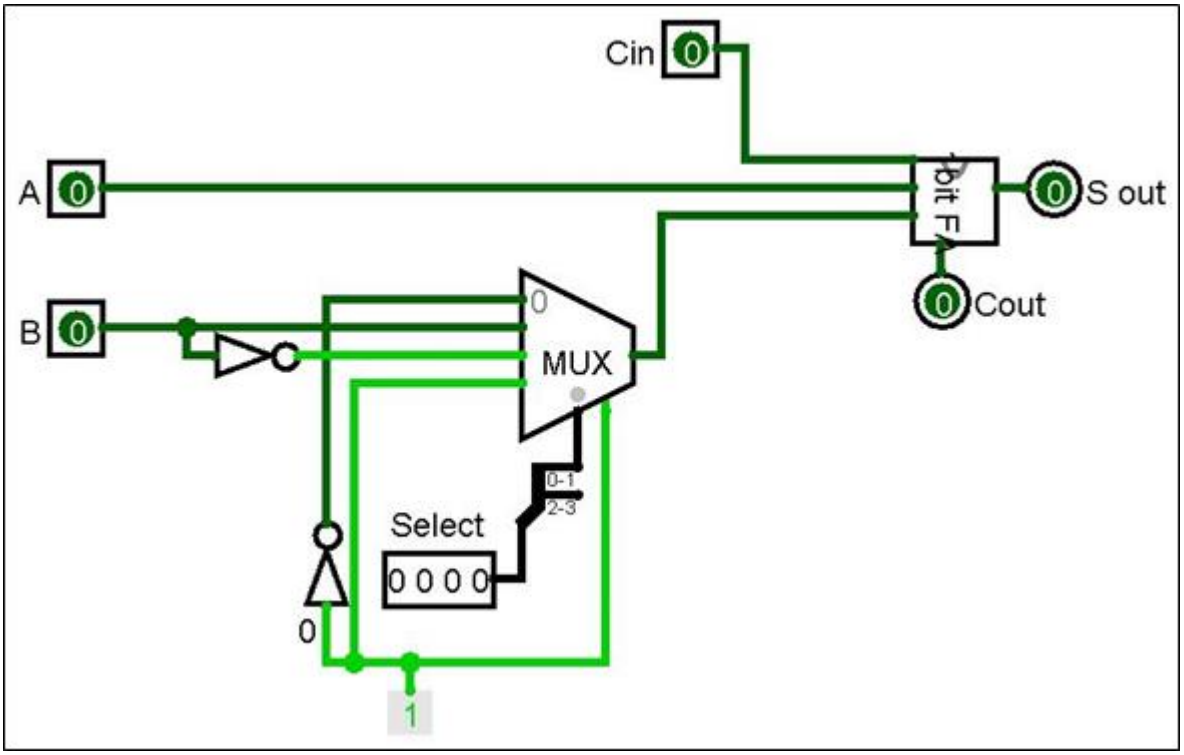


Figure 5.1 One Bit Arithmetic Unit

The figure above is the Arithmetic Unit of the ALU designed for the project. The arithmetic unit is in charge of adding operations with using 2 input signals and 5 select signals as 1 C_{in} and 4 as S3 S2 S1 S0. The 1 bit full adder will calculate the S_{out} and C_{out} for this design.

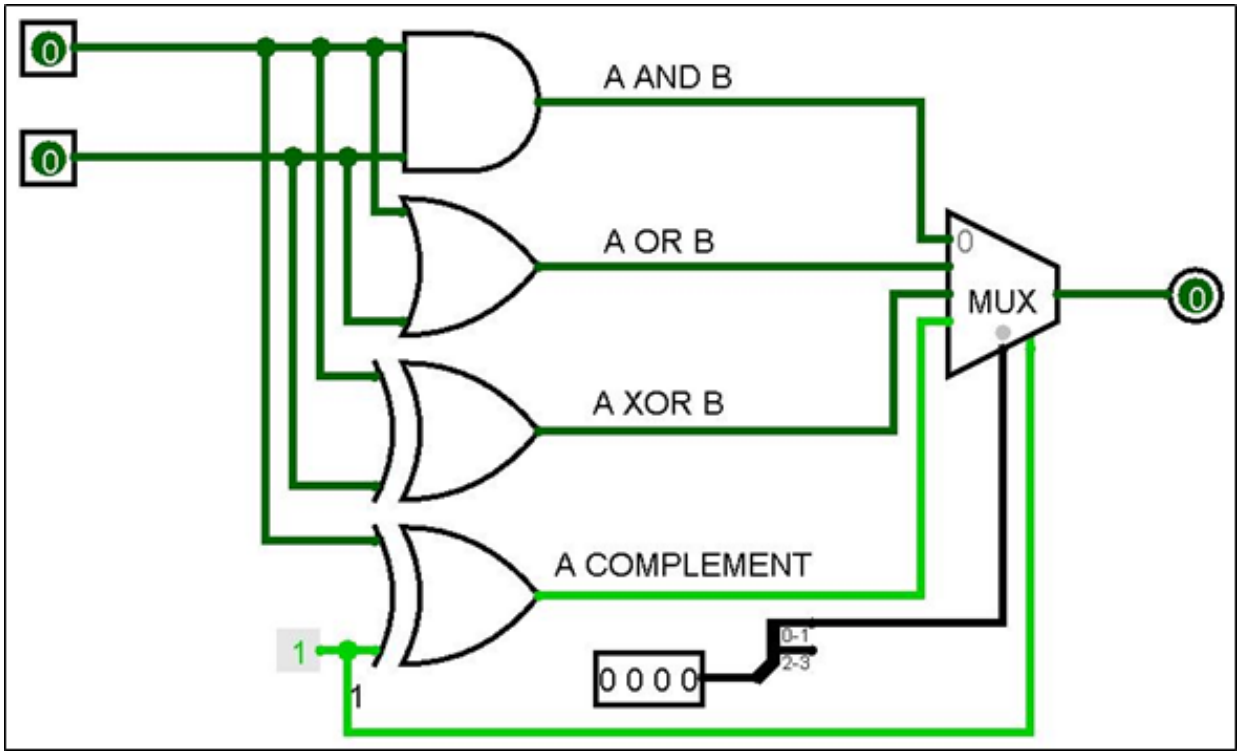


Figure 5.2 One Bit Logical Unit

The figure above is the Logic Unit that will make logic operations for the ALU with using 4 selects as S3 S2 S1 S0. The 5th select signal (C_{in}) will not be necessary for these operations mostly. A combined 1 bit ALU unit is given above that successfully can operate the operations given in the table above has a design as shown. The ALU is capable of processing 16 of the operations with no error and is tested before the next part of the project has started.

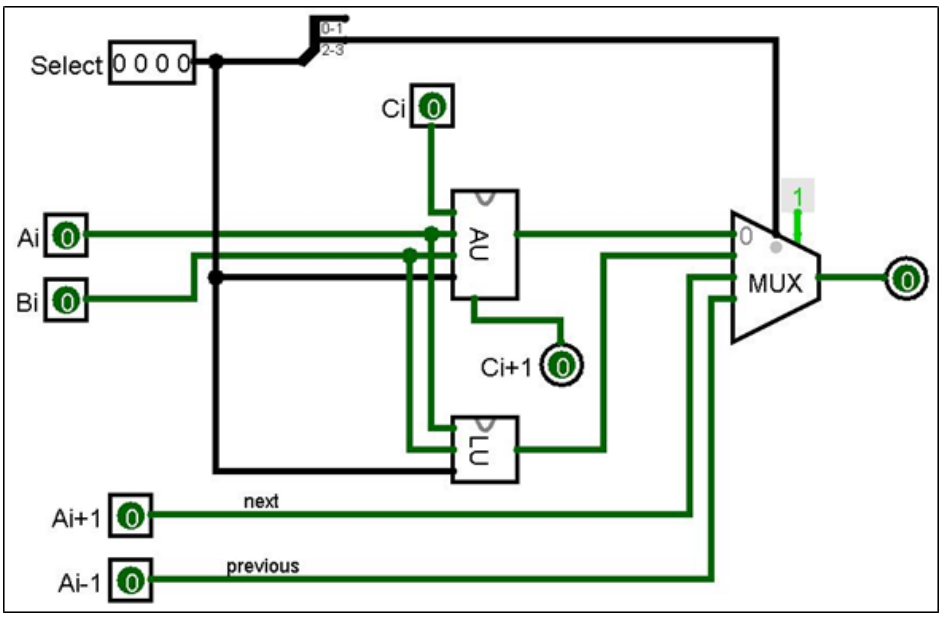
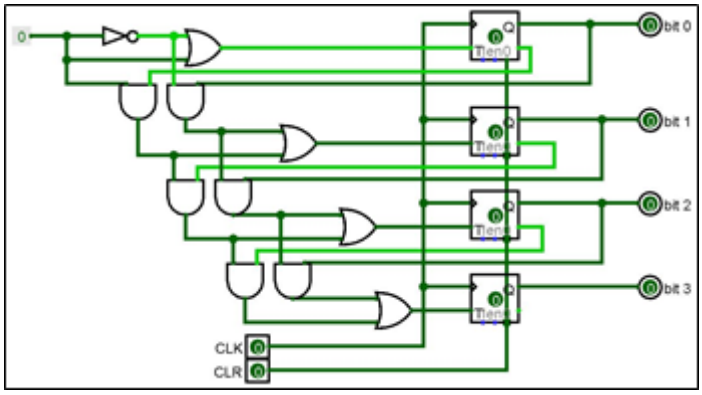


Figure 5.3 One Bit ALU

These two units will be combined to form the Arithmetic Logic Unit that is capable of doing all operations on a 1 bit size. 16 of these modules will be combined to form the 16 bit ALU.

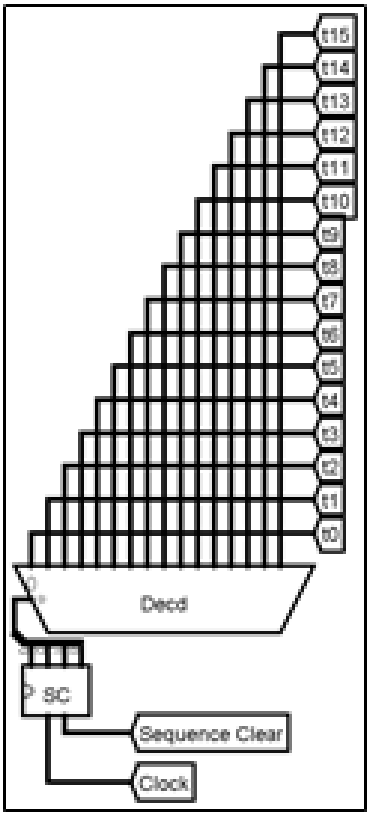
Sequence Counter

Considering the project will work with a line that will make different operations on different time panels, to process instruction timing mechanism was necessary. Therefore a 4 bit sequence counter is found to be efficient in this matter.



The figure left shows a simple 4 bit sequence counter using T flip-flops capability of toggling with time. This sequence counter will only have a CLK signal to keep counting or a CLR signal to restart itself.

Figure 5.4 Four Bit Binary Counter



The timing of each operation will be done using T0 to T15 different time frames. First 3 clock cycles are for the fetch, the 4th one for the indirect addressing, following 5 clocks will be used for the execution of the instruction. Additional timing frames will not be necessary due to the sequence clear will occur in the middle to prevent any over counting for the system.

Figure 5.5 Four Bit Sequence Counter

Bus Design

Current state of the project can be handled by using only one bus for everything. Therefore a bus is designed and its select signals are set to carry out necessary registers information to the bus.

X2	X1	X0	Function
0	0	0	Clear data bus. Bus <= 0
0	0	1	Load AR data to bus. Bus <= AR
0	1	0	Load PC data to bus. Bus <= PC
0	1	1	Load DR data to bus. Bus <= DR
1	0	0	Load AC data to bus. Bus <= AC
1	0	1	Load IR data to bus. Bus <= IR
1	1	0	Load TR data to bus. Bus <= TR
1	1	1	Load Memory data to Bus. if memory read is 1. Bus <= M[AR]. If memory write is 1. Bus <= **** (unwanted situation)

Figure 5.6 Data Bus Selection Signals

Instruction Set

Type	Symbol	Hex code	Description	Register Transfer Language
REGISTER	CMA	7200	Complement AC	AC <-- not AC (1's complement)
	CME	7100	Complement E	E <-- not E
	CIR	7080	Right circular shift AC and E	AC (0-14) <-- AC (1-15) AC (15) <-- E E <-- AC (0)
	CIL	7040	Left circular shift AC and E	AC (1-15) <-- AC (0-14) AC (0) <-- E E <-- AC (15)
	INC	7020	Increment AC	AC <-- AC + 1
	SPA	7010	Skip if AC is non-negative	if (AC(15) = 0) PC <-- PC + 1
	SNA	7008	Skip if AC is negative	if (AC(15) = 1) PC <-- PC + 1
	SZA	7004	Skip if AC is zero	if (AC = 0) PC <-- PC + 1
	SZE	7002	Skip if E is zero	if (E = 0) PC <-- PC + 1
	HLT	7001	Halt computer	S <-- 0
INPUT AND OUTPUT	INP	F800	Input character to AC	AC <-- INPR
	OUT	F400	Output character from AC	OUTR <-- AC
	SKI	F200	Skip on input flag 1 means valid input is available	if (FGI = 1) PC <-- PC + 1
	SKO	F100	Skip on output flag 1 means register is clear for new output	if (FGO = 1) PC <-- PC + 1
	ION	F080	Interrupt on	IEN <--- 1
	IOF	F040	Interrupt off	IEN <-- 0

Figure 5.7 Instruction Set I

Type	Symbol	Hex code blue - direct yellow-indirect		Description	Register Transfer Language
M E M O R Y	AND	0xxx	8xxx	AND memory word with AC	AC <-- AC and M[xxx] or M[M[xxx]]
	ADD	1xxx	9xxx	ADD memory word with AC	AC <-- AC + M[xxx] or M[M[xxx]]
	LDA	2xxx	Axxx	Load AC from memory word	AC <-- M[xxx] or M[M[xxx]]
	STA	3xxx	Bxxx	Store AC to memory word	M[xxx] <-- AC or M[M[xxx]]
	BUN	4xxx	Cxxx	Branch unconditionally	PC <-- M[xxx] or M[M[xxx]]
	BSA	5xxx	Dxxx	Branch and save return address	M[xxx] <-- PC or M[M[xxx]] PC <-- M[xxx] + 1 or M[M[xxx]]
	ISZ	6xxx	Exxx	Increment and skip if 0 Does not involve AC	M[xxx] <-- M[xxx] + 1 or M[M[xxx]] ^ if (M[xxx] = 0) then PC <-- PC + 1 or M[M[xxx]]

Figure 5.8 Instruction Set II

The instructions designed are exactly the same as the ones that Morris Mano design was using (Mano, 1993)[11]. It is decided to use them and nothing extra to keep the system still capable of running with RISC mentality. There are three different types of instructions which are; memory, register and input/output operations. For memory operations proper time is needed to process the operation. However, for register operations and I/O operations no clock cycle is needed.

Control Unit

Since the approach used for the project is SIMD, the control unit plays a very important role to keep everything under its supervision. Therefore, fields implemented are designed separately to provide any further implementation easier.

Select Handler

Select handler is the part of CU that gives proper inputs to selecting the BUS and ALU multiplexers. Operation codes of ALU are synchronous with the instructions of the system to make ALU complete the operations it is supposed to.

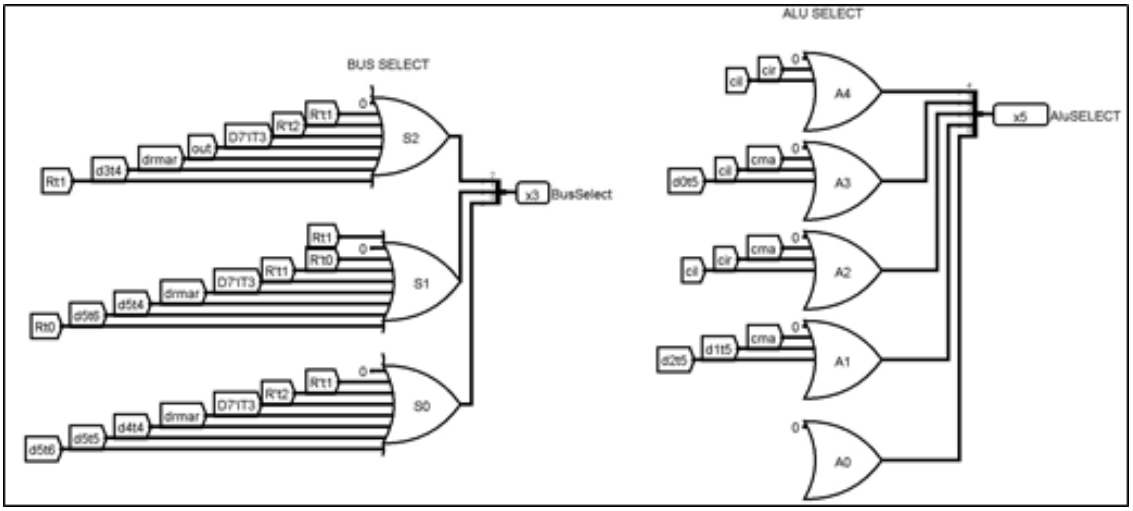


Figure 5.9 Select Handler

Register Mode Setter

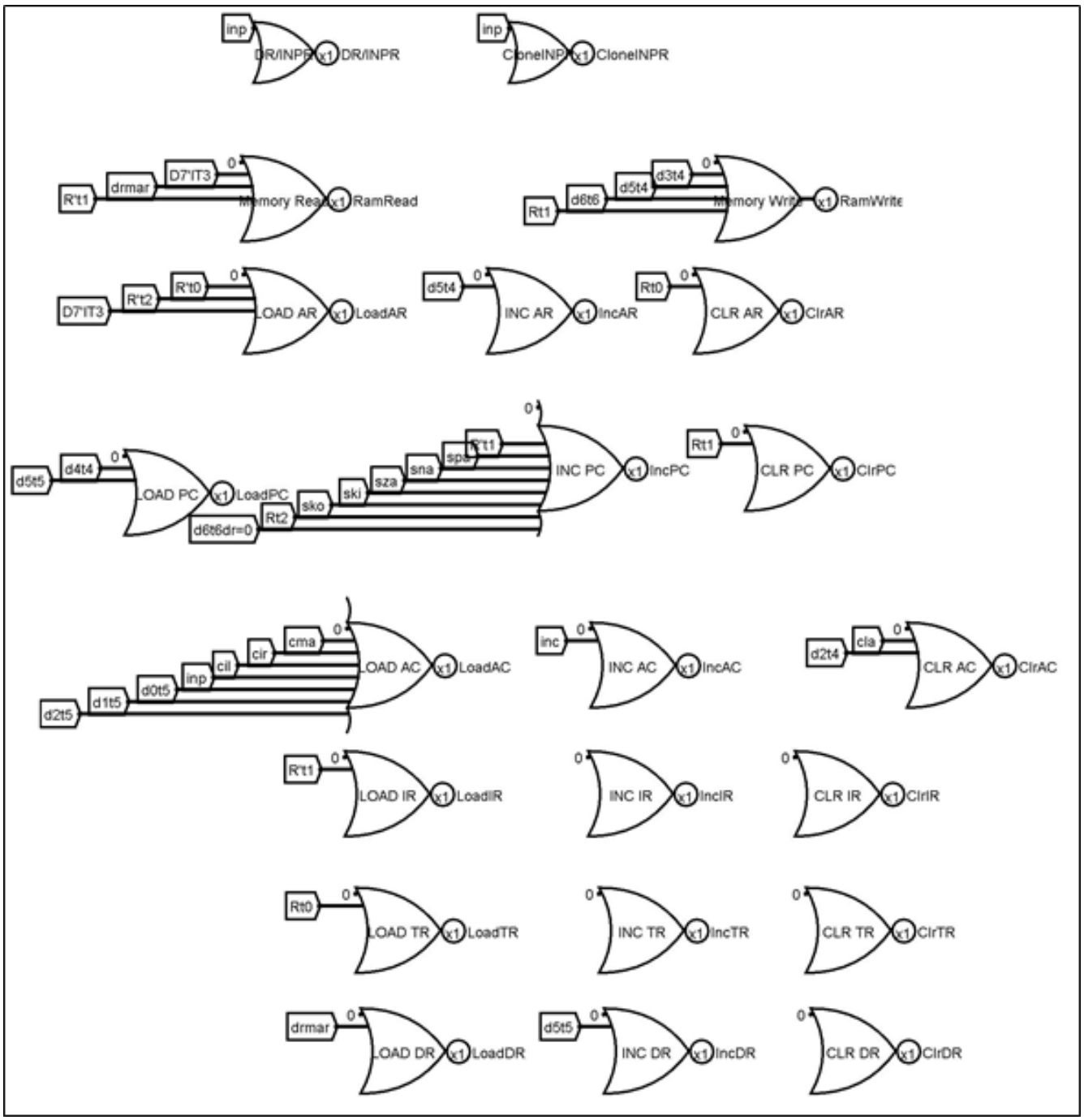


Figure 5.10 Resigter Mode Setter

Almost every instruction in the system either loads, increases or clears registers' content in a relevant time frame. Therefore the "T" tags are added at the end of tunnel names to specify exactly what time period it refers to. The flow of an instruction can be seen by looking to the mode setter as well as the instruction executioner.

Instruction Executioner

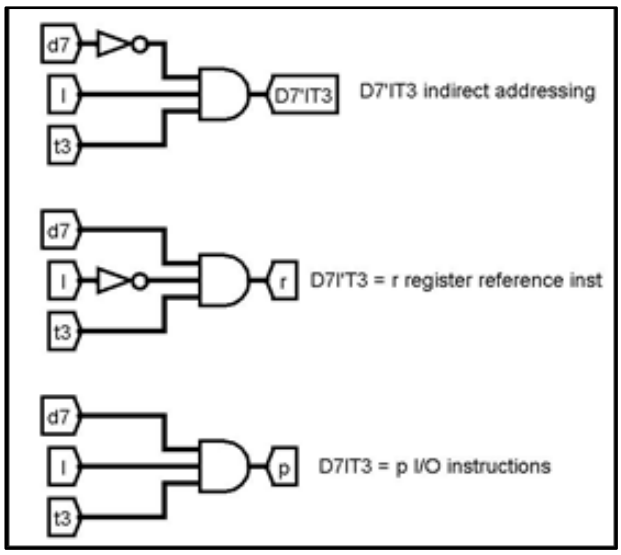


Figure 5.11 Main Instruction Types

The instruction registers values are checked to decide what operation is indeed being executed. “R” is for register operations, “p” is for I/O operations and “D7’IT3” is for indirect addressing. The decision of instruction will be made by looking what sort of instruction is being executed afterwards.

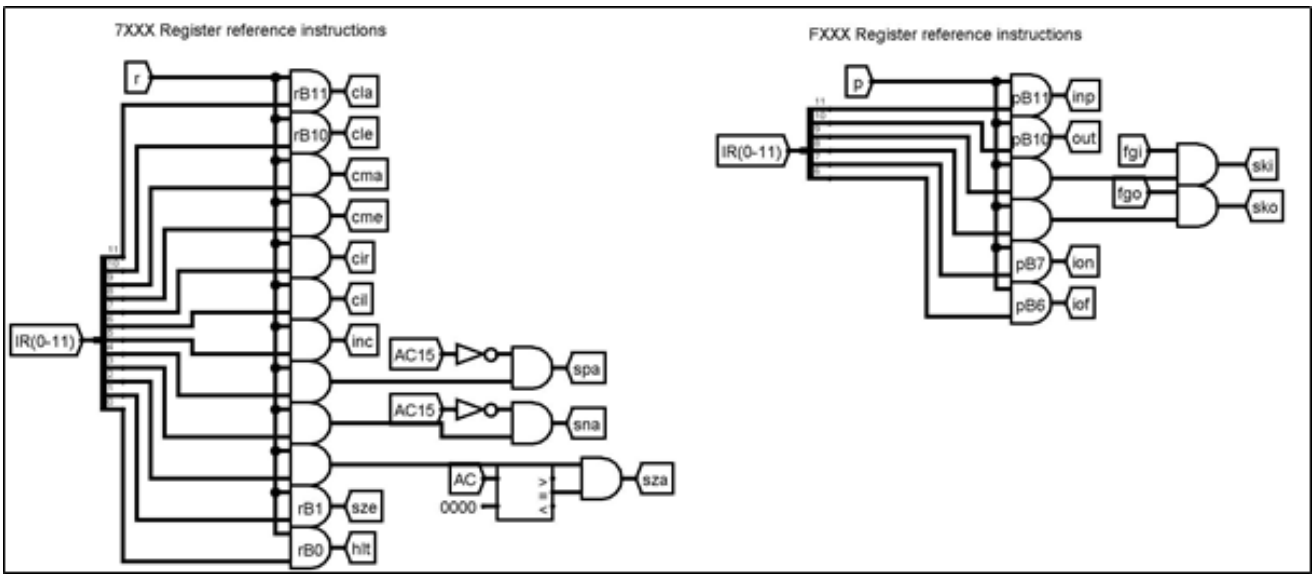


Figure 5.12 Instruction Type Tunnels

The names of operations are given to the tunnel names to keep control unit simpler for the designer. Instruction register and the instruction type decide what the operation is.

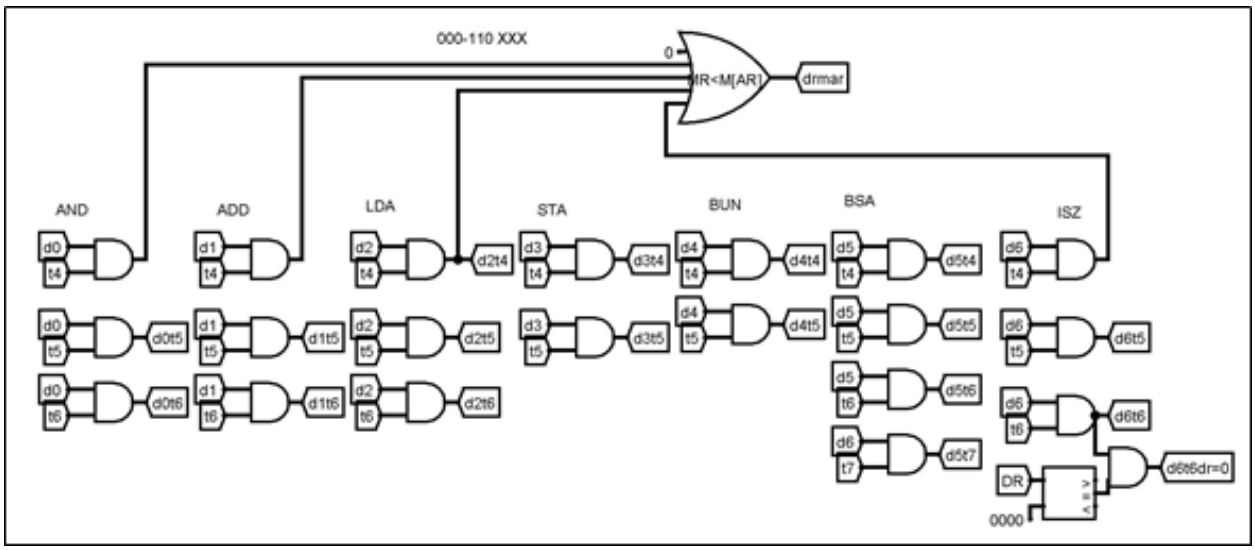


Figure 5.13 Instruction Timings

The time period of instructions are given in the figure above. The shortest operations are STA and BUN commands with only 2 clock cycles need that ends at T5, and the longest is the BSA command that takes 4 clock cycles to end at T7. The Timing frames after T7 are pointless in this matter since the instruction will be divided into storing and executing afterwards when the system is being converted to pipeline mentality. The timing will be handled with segment differentiations once the system is complete.

Interrupts

IEN being the interrupt enabled permission, fgi and fgo are the situation of an input or output to the system; the interrupts will only be accepted if the time period is not t0-t1-t2 due to the fact that an interrupt in the middle of fetching would crash the system and also either ein,fgi and fgo flags should be enabled.

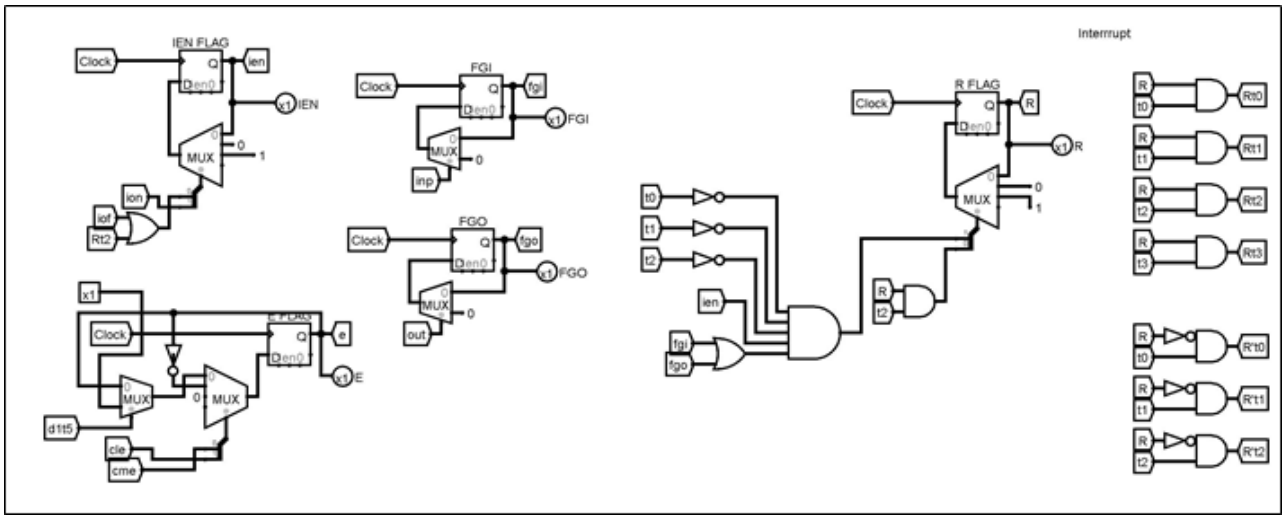


Figure 5.14 Interrupt Flags

Cache Memory Unit

Aging Register

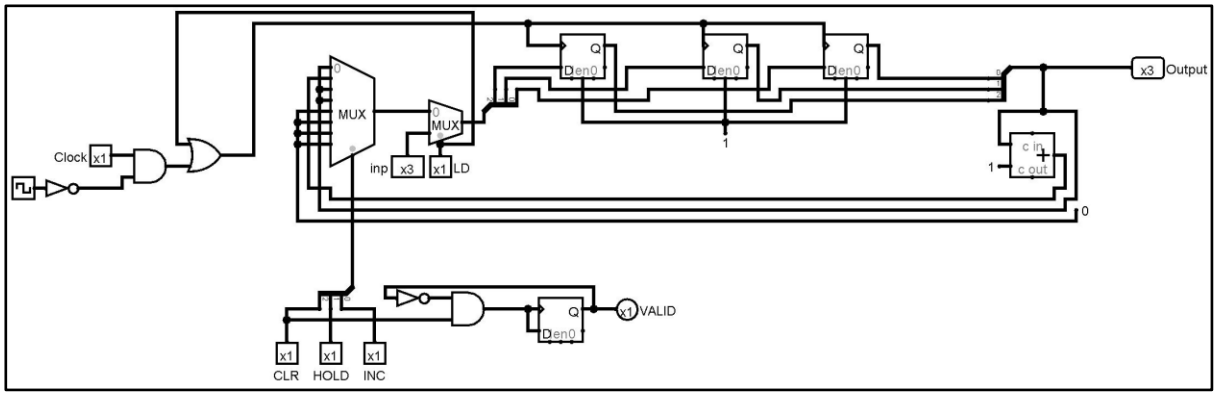


Figure 5.15 Aging Register

The aging register will have the capability of clearing, holding and increasing depending on the situation of the address that arrives. When a value needs to be loaded the age will be cleared, when a line is loaded it will also set the valid bit of this line that will stay as it is. If there was a hit on some other line the age of other lines can be either held or increased.

Hold and Increase Control

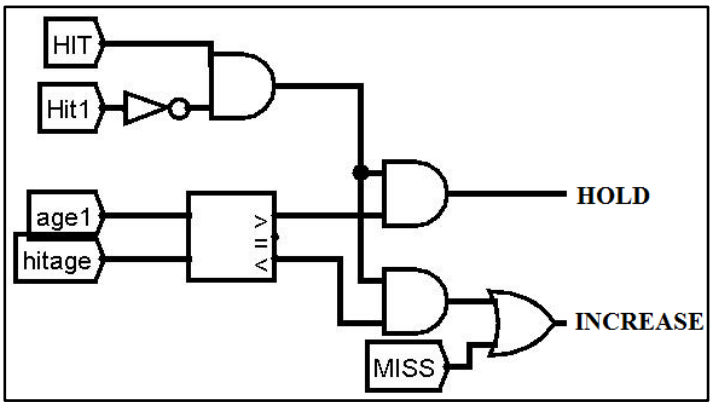


Figure 5.16 Hold and Increace for aging

The hold and increase will only occur in case of a hit occurrence in some other line. If the current age of a line is higher than the hit age it will hold its value, if not it will increase its value. There is no possibility that they can be equal since that would mean a hit on that line which would automatically prevent the hold and increase.

Load Control

There are two possible situations that can load content to a line in a cache memory. Either there must be a miss occurrence and the age of the line should be maximum, or there must be a hit scenario that will clear the age but not change the data in it. These two scenarios are held in an 8 bit data set called agemaxserie and the hitserie. Two of these scenarios cannot occur at the same time considering a miss has to occur to load to the highest age, and a hit has to occur to generate the hitserie. Therefore, a logical OR operation can decide the age of the block depending on the result of these two.

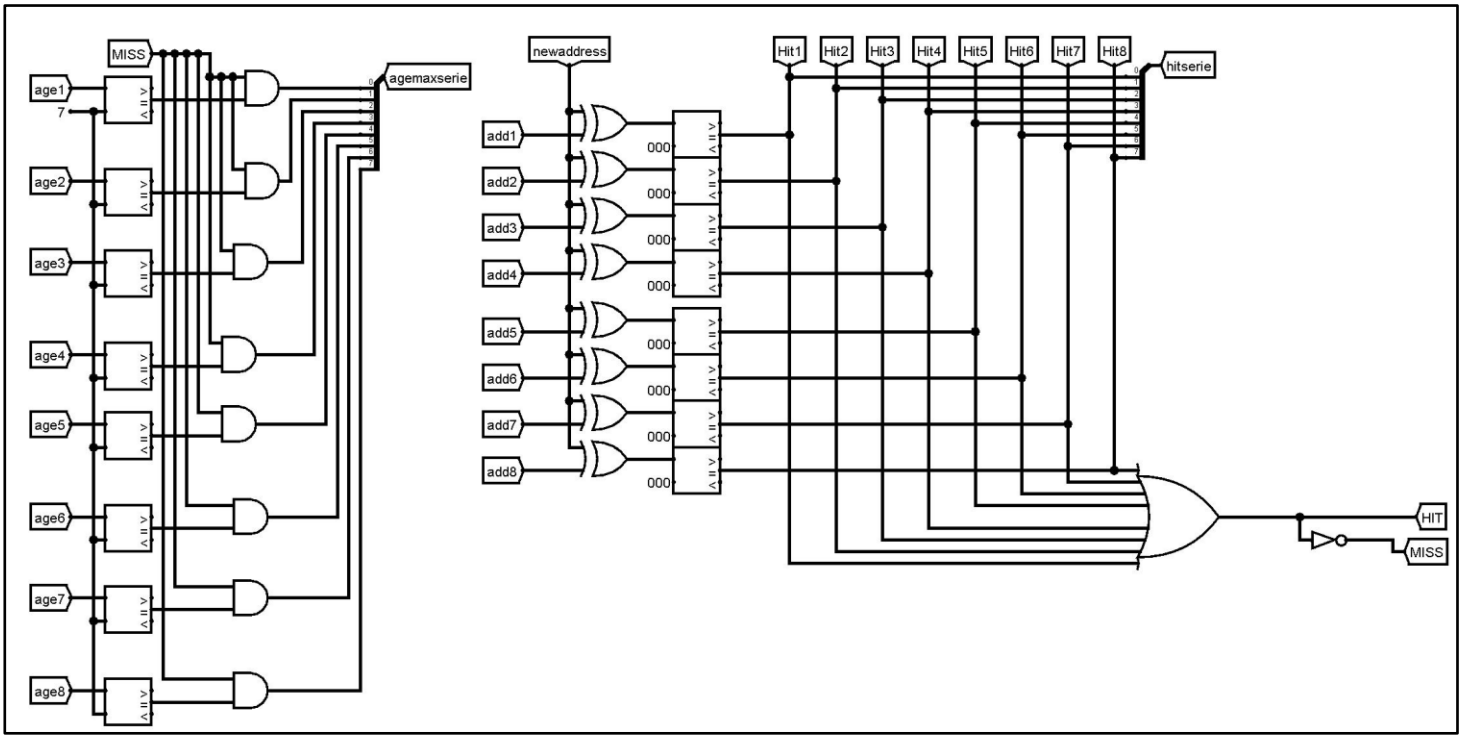


Figure 5.17 Load Control I

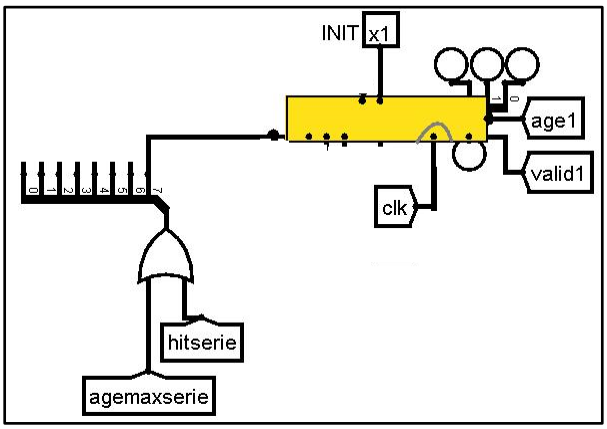


Figure 5.18 Load Control II

The OR operation result of agemaxserie and the hitserie can be used to feed the clear inputs of the aging registers in order to adjust their ages accordingly. By doing so an age can only be cleared if there is a miss and its age is 111, or if there is a hit on it.

Performance Test Registers

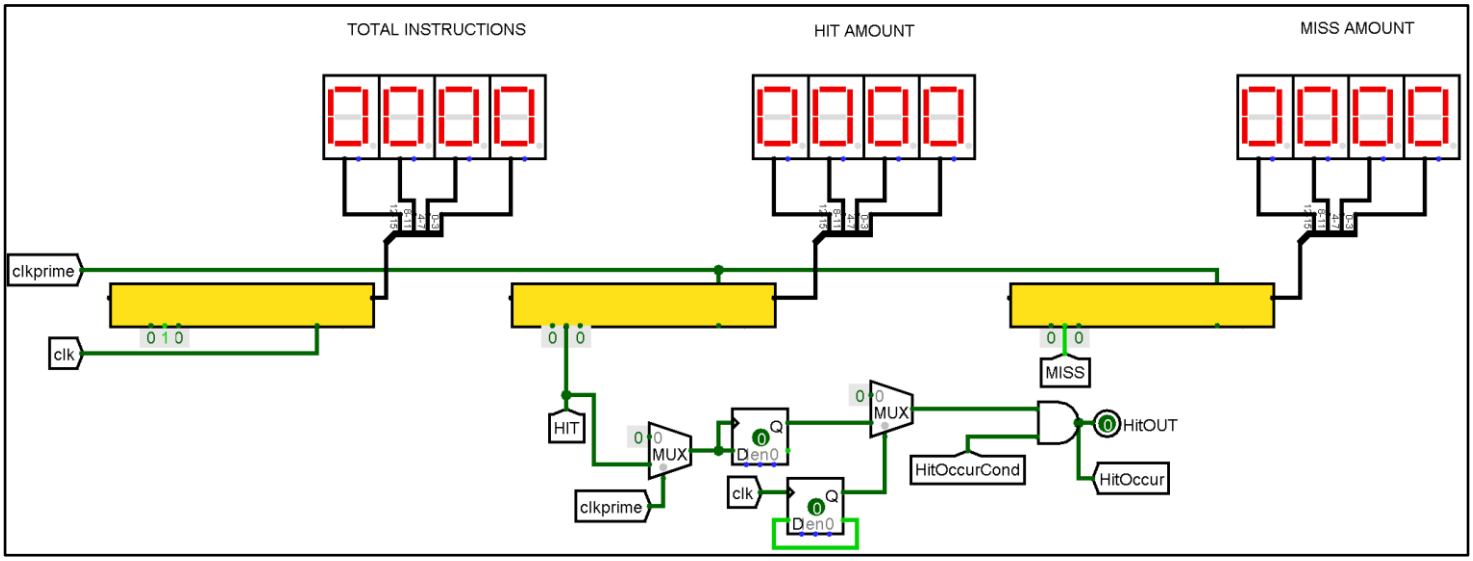


Figure 5.19 Performance Test Registers

When the CPU goes into fetch sequence, the cache memory will count the state result of hit or miss at the same time. The total instructions will increase by one no matter what occurs. The miss amount will increase if there is a miss. The hit amount will increase when there is a hit, and it will also set the flag of Hit_{OUT} as an output of this module in order to make an outside impact of the cache unit. This flag will remain high until the fetch sequence is over, and during this period this signal will block the CPU's RAM access request which will make the memory reachable for the other units in the system. During this period, the cache result will feed the system since there is a hit occurrence. The Hit_{OUT} signal will remain set for x2 times of the CPU clock by using a D-Latch master and slave circuit. By doing so the cache can have full access over the memory until the full fetch process is over for an instruction.

Hit Age Decision

When a hit occurs, as a disadvantage of fully associative cache addressing, it has to search the whole cache in order to find the age that got hit.

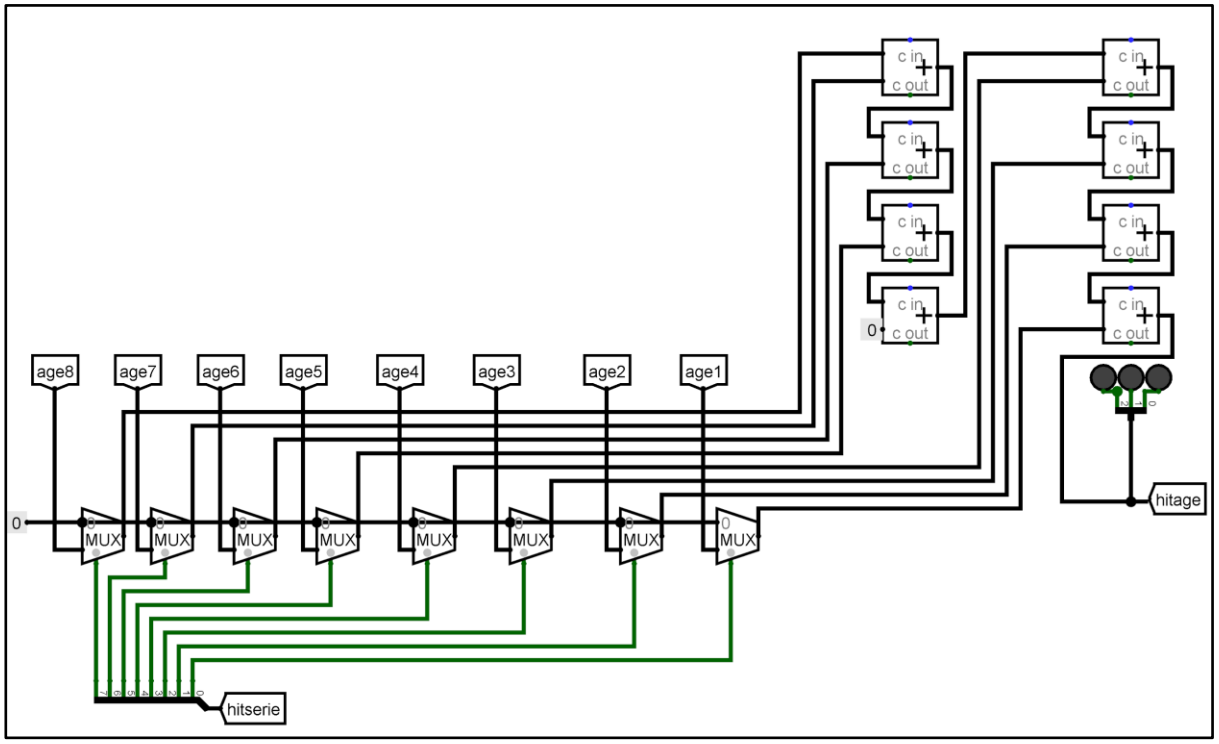


Figure 5.20 Hit Age Decision

The circuit shown above only lets an age pass to the summing part if it has been hit. There can only and only be one age that can cross the multiplexers since one address is only kept once in the cache. After this one address is passed the rest will output a zero state which is not effective in the sum operation. The hitage is calculated by letting only this one age to the output.

Hit Value Decision

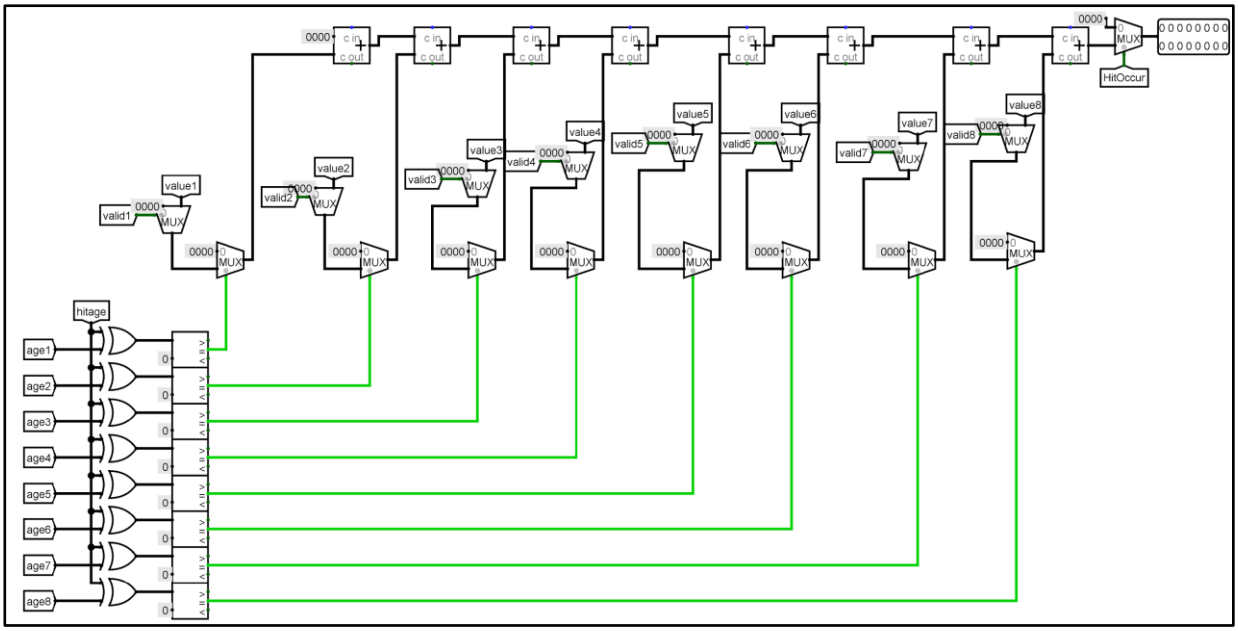


Figure 5.21 Hit Value Decision

When it comes to value that should go out the valid bits come handy. An output of a value is only important when there is a hit scenario; therefore there is a hitoccur condition at the end that prevents from non-hit values to cross. The rest of the value is calculated by only letting a value out if it is valid and there is a hit on it. Using these two multiplexer conditions and summing up the results, again only one will cross to the output as the result of value that got hit. This value will be used to feed the data bus of the system during the fetch period instead of the main memory.

6. Experimental Results

The performance of the built-in cache module has to be tested under different circumstances. A various types of programs are written to test the performance where some benefit from the cache memory more than usual, and some do not. On this part of the report, the results of experiments will be given.

Loop Experiment

Looping is a very common mentality when it comes to computer programs. Many programs intend to repeat the same sequence of a code block until a condition is satisfied.

```
LDA $10     001 2010
INC         002 7020
STA $10     003 3010
LDA $10     004 2010
SZA         005 7004
BRA $001    006 4001
HALT        007 7001

010 FFFD    010 FFFD
```

Figure 6.1 Test 1 Machine Code

A program is written to the ram addresses from 1 to 7 while a data value is used that is placed on address of 010.
The code intends to increase the value of FFFD (010) one by one each loop, and when it reaches 0000, the condition of Skip-If-Zero will jump the branch line to halt the computer.

On the first time all of the instructions will face a miss, since they are not recorded in the cache memory. But when the code loops back on another time, they will be fed from the cache memory to the bus meaning there will be hit on all of them.

```
MISS  LDA $10
MISS  INC
MISS  STA $10
MISS  LDA $10
MISS  SZA
MISS  BRA $001
HIT   LDA $10
HIT   INC
HIT   STA $10
HIT   LDA $10
HIT   SZA
HIT   BRA $001
HIT   LDA $10
HIT   INC
HIT   STA $10
HIT   LDA $10
HIT   SZA
MISS  HALT
```

Figure 6.2 Test 1 Code Sequence

Program will loop three times where the first run as mentioned before will result in a total miss. On the second and the third loop the code will catch the hits and count them in. In the end the result should give 11 hits and 7 misses in 18 instructions.

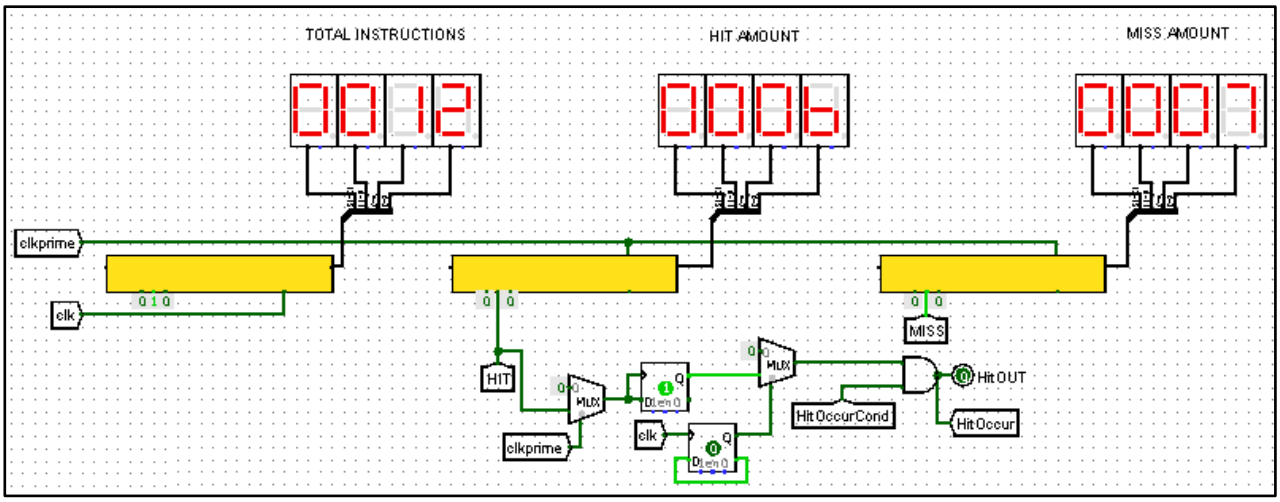


Figure 6.3 Test 1 Results I

The results are kept perfectly correct by the cache counters. Only with three loops the system have provided a roughly 61% hit rate.

Assuming that the loop was larger than three sequences, which run the program with ten loops to observe the boost of performance. There will be a total of 61 instructions where again 7 will be miss but the remaining 54 will be hit.

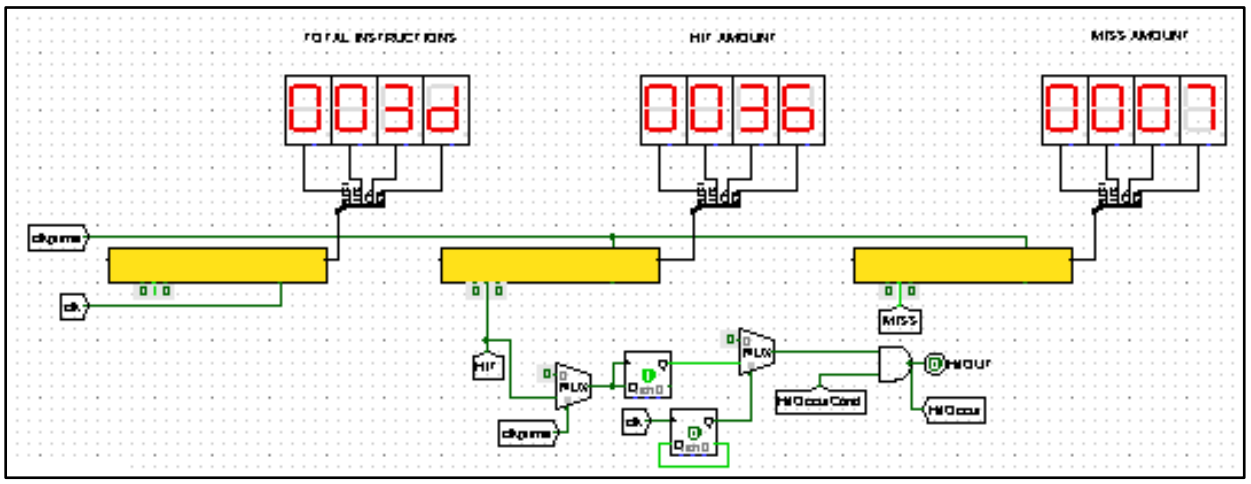


Figure 6.4 Test 1 Results II

The results have proven the success of loop experiment where the cache memory can be useful when it comes to looping.

Oversized Loop Experiment

The cache designed for this project can only hold up to 8 instructions that are recently used. If loop instructions would have a size more than 8 instructions, after the 8th one gets cached, the cache memory will start to replace them with the next instructions as it goes. Therefore, when it is meant to loop to the beginning of the loop, no hit will occur since the cache memory will only consist of the last 8 instructions of the loop. For this purpose, the code from the 6.1 experiment can be extended and still used to prove the point made.

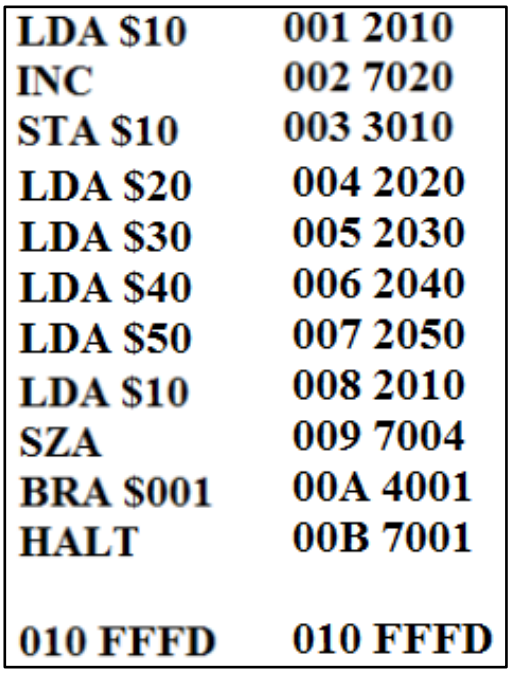


Figure 6.5 Test 2 Machine Code

The code chosen for this experiment has a loop inside that starts from the address of 001 and branches at 00A meaning 9 instructions are in the loop. This clearly extends the cache size. Let us monitor the cache behavior on such situation.

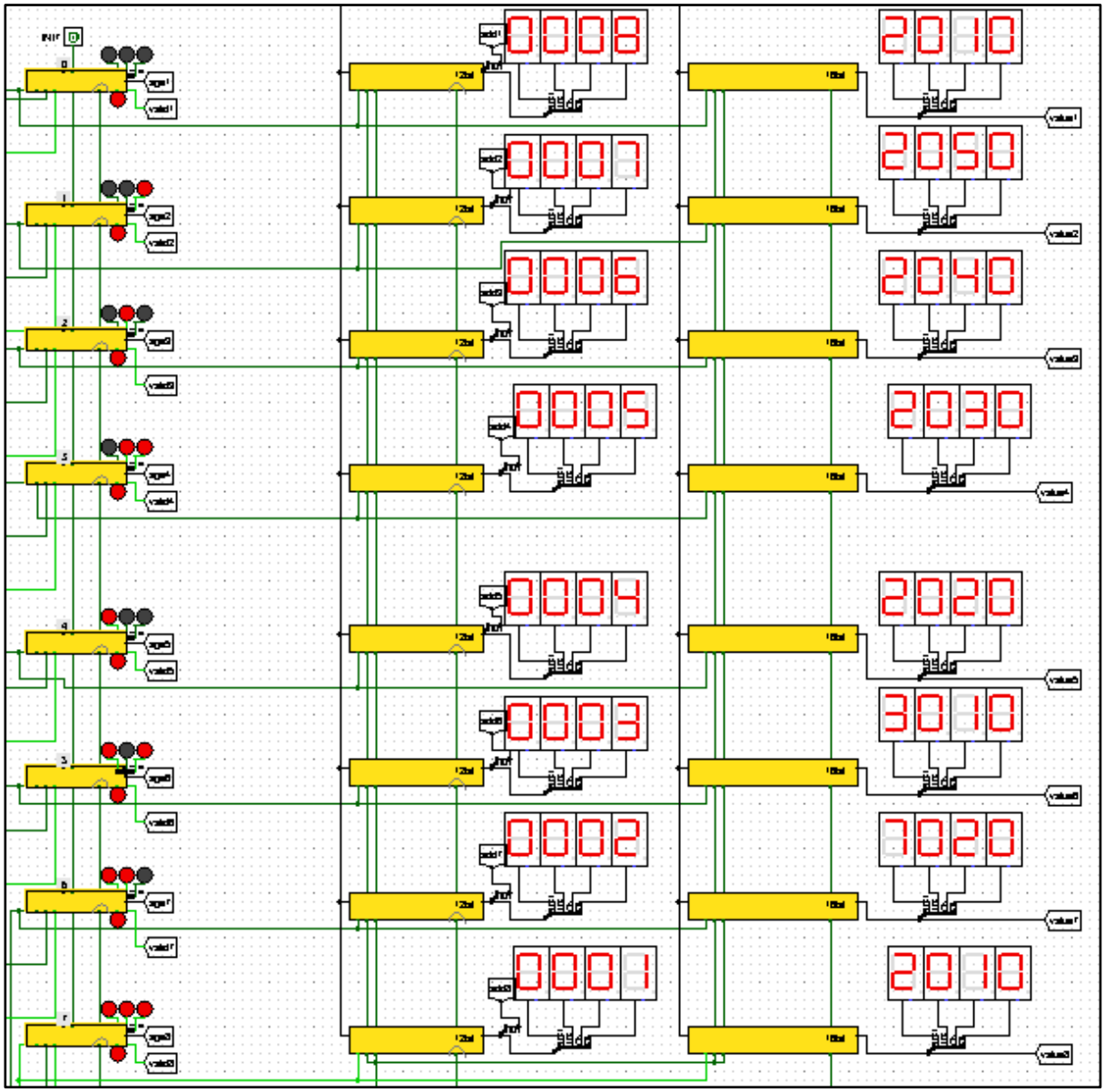


Figure 6.6 Cache Status I

After first 8 instructions are processed the cache memory have cached them properly inside. However the loop did not occur yet. There is still the last instruction called SZA required to make the loop back to address of 001. When this SZA is processed, a miss will occur that will replace the line with the highest age of 111 which was indeed the start of the loop.

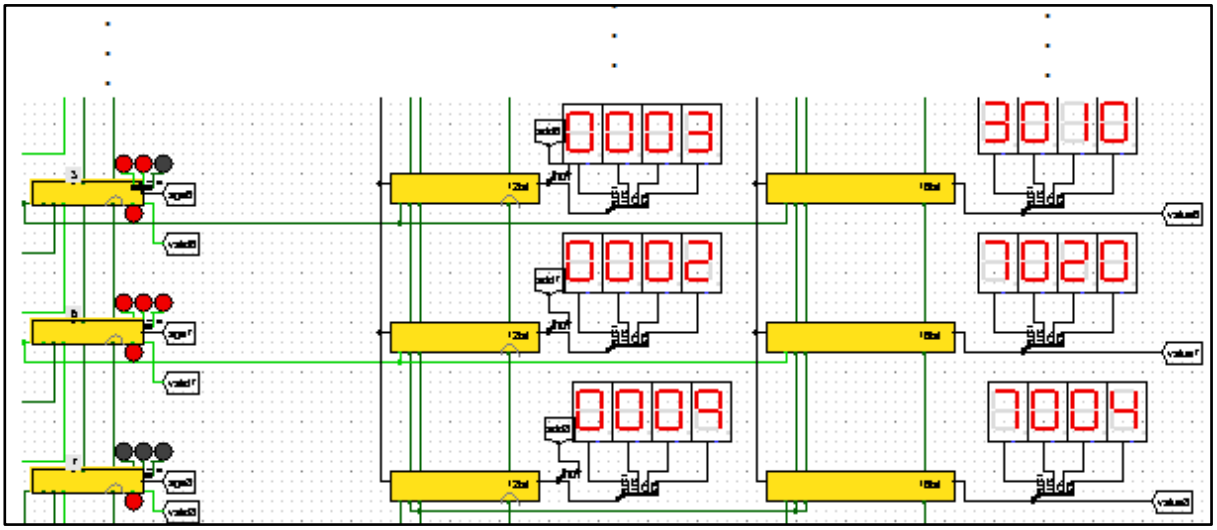


Figure 6.7 Cache Status II

As can be seen this SZA instruction is placed at the bottom of the cache where it will make the next instruction of 001 evicted. Thus, causes the entire loop to face a miss occurrence.

```
MISS  LDA $10    001 2010
MISS  INC        002 7020
MISS  STA $10    003 3010
MISS  LDA $20    004 2020
MISS  LDA $30    005 2030
MISS  LDA $40    006 2040
MISS  LDA $50    007 2050
MISS  LDA $10    008 2010
MISS  SZA        009 7004
MISS  BRA $001   00A 4001
MISS  LDA $10    001 2010
MISS  INC        002 7020
MISS  STA $10    003 3010
MISS  LDA $20    004 2020
MISS  LDA $30    005 2030
MISS  LDA $40    006 2040
MISS  LDA $50    007 2050
MISS  LDA $10    008 2010
MISS  SZA        009 7004
MISS  BRA $001   00A 4001
MISS  LDA $10    001 2010
MISS  INC        002 7020
MISS  STA $10    003 3010
MISS  LDA $20    004 2020
MISS  LDA $30    005 2030
MISS  LDA $40    006 2040
MISS  LDA $50    007 2050
MISS  LDA $10    008 2010
MISS  SZA        009 7004
MISS  HALT       00B 7001
```

Figure 6.8 Test 2 Code Sequence

Because that the last part of the loop have evicted the beginning instructions of the loop from the cache, a hit case was not possible to occur. There were total of 30 instructions and they all resulted in a miss.

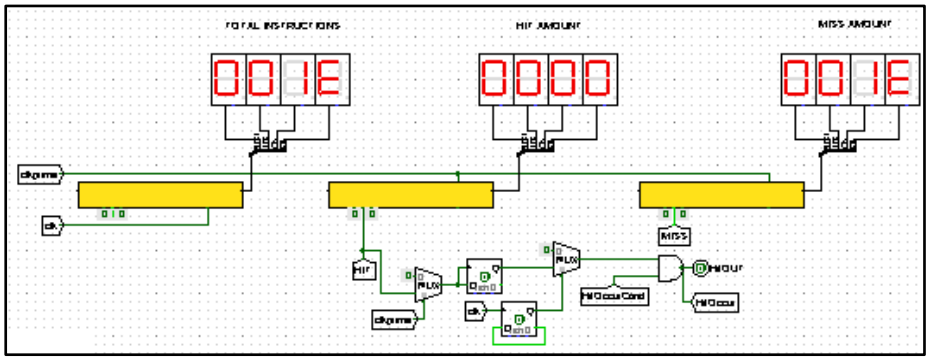


Figure 6.9 Test 2 Results

The cache counters have shown the result in hexadecimal format as all instructions caused a miss. Meaning the cache memory did not provide any performance boost over the system.

7. Conclusion and Suggestions

Cache memory approach was a stepping stone for computer architecture. Conductivity of silicon components was creating the technology bottleneck by causing the latency for the processor. Thus, cache memory came to rescue by physically being placed close to the processor to reduce the latency. Over the years, many different algorithms are implemented to the cache memories and make them capable of deciding which data is to be cached and which is to be evicted.

For this project, a cache memory unit is designed and implemented that uses aging bits and valid bit to benefit from a LRU approach for replacement, with a write-through writing policy. The cache module is designed separately from the processor in order to test executively. Once the tests were complete, the cache module is connected to the main processor and the connections are completed. In the end, an instruction cache was successfully implemented that is capable of holding 8 lines with their address and values as well as their aging's and validation bits that adds up to a total 256 bits of cache memory size.

Looping is a very common operation in computer science. Almost every computer program consists of loops of instructions which is nothing but a repetitive executions over the processor. Instruction cache memories are able to record these instructions in order to retrieve in the recent future when required again by the processor to boost the performance. The designed cache module was able to work properly to record up to 8 instructions that were recently addressed. The hit and miss scenarios are being successfully executed and aging bits are adjusted accordingly to the status. The module is also implemented with a 16x3 bit recording unit that is in charge of keeping hit and miss amounts. After the tests are complete it is proven that, cache memories can boost the performance of a system when the program that is being run has repetitive sequences of code blocks.

8. References

[1] Weik, M. H. (March 1961). *A Third Survey of Domestic Electronic Digital Computing Systems.* Maryland: Ballistic Research Laboratories.

[2] Bosworth, E. L. (No Date). Overview of Computer Architecture The IBM System/360. Columbus, GA: Department of Computer Science Columbus State University.

[3] Flynn, M. J. (1995). *Computer architecture: pipelined and parallel processor design.*

[4] Harris, D. H. (01 Jan. 1970). *Digital Design and Computer Architecture.*

[5] Taylor, M. B. (2015). CSE240 Lecture Notes - Memory Design. San Diego.

[6] Meng, B. M. (2013). *On Understanding the Energy Consumption of ARM-based Multicore Servers.* Singapore: National University of Singapore.

[7] Burch, C. (2014). Logisim (Computer Program).

[8] Scott Di Pasquale, K. E. (2003). *Hardware Loop Buffering.* Houston: Rice University.

[9] ARM Limited. (2001). ARM720T Technical Reference Manual. *Revision: r4p3*.

[10] Evans, C. (2017). Cache is vital for application deployment, but which one to choose:. *Write-through, write-around or write-back cache? We examine the options*. http://www.computerweekly.com/feature/Write-through-write-around-write-back-Cache-explained.

[11] Mano, M. M. (1993). *Computer System Architecture* (3rd Edition b.). Prentice Hall.