Designing and Optimizing the Fetch Unit for a RISC Core

Mojtaba Shojaei, Bahman Javadi*, Mohammad Kazem Akbari, Farnaz Irannejad

Computer Engineering and Information Technology Department, Amirkabir University of Technology, Tehran, Iran

Received 18 December 2008; revised 15 September 2009; accepted 21 September 2009

Abstract

Despite the extensive deployment of multi-core architectures in the past few years, the design and optimization of each single processing core is still a fresh field in computing .On the other hand, having a design procedure)used to solve the problems related to the design of a single processing core (makes it possible to apply the proposed solutions to specific-purpose processing cores .The instruction fetch, which is one of the parts of the architectural design, is considered to have the greatest effect on the performance .RISC processors, which have architecture with a high capability for parallelism, need a high instruction width in order to reach an appropriate performance .Accurate branch prediction and low cache miss rate are two effective factors in the operation of the fetching unit .In this paper, we have designed and analyzed the fetching unit for a 4-way) 4-issue (superscalar processing core .We have applied the cost per performance design style and quantitative approach to propose this fetch unit .Moreover, timing constrains are specially analyzed for instruction cache to enable the proposed fetch unit to be in a superpipeline system .In order to solve the timing problem, we have applied the division method to the branch prediction tables and the wave pipelining technique to the instruction cache.

Keywords: Instruction fetch, Branch prediction, Instruction cache, Cost, Performance, Timing.

1. Introduction

In the past few years, the multi-core architecture has been proposed to ameliorate the performance of microprocessors and the users' need for high speed [1, 2]. In this architecture we achieve a higher performance by using multiple microprocessors in a chip. The microprocessors are independent and physically separated from each other. They are connected to each other in a multiprocessor environment which is placed in a single chip. On the other hand, there are a lot of requests for the specific-purpose processing cores like DSP, encryption and games. The cell processor is an example of such cores that has been constructed via the collaboration of the IBM, Toshiba and Sony companies [3]. All of these architectures have one thing in common; that is an optimized design for each processing core in order to reach the best performance in the multi-core environments.

Thus, the design and optimization of each processing core is still an important issue in the world of computer architecture. So the RISC architecture has been considered by the microprocessor designers due to its simple structure and great potential for performance improvement. The instruction level parallelism (ILP) can be used in this architecture in order to improve the performance of the processor. Thus the superscalar and superpipeline techniques, which can enable the instructions to be executed simultaneously, have the greatest effect on performance improvement. In superpipelined processors, the clock cycle is reduced by dividing the pipeline into smaller steps so that more instructions could run simultaneously [4]. A superscalar processor runs multiple operations simultaneously on separated pieces of hardware units. This is done by issuing multiple instructions for different execution units and executing them simultaneously in a single cycle.

The examination of the instruction level parallelism has shown that 4 to 8 instructions can be executed by using limited hardware and a compiler. The main significant factors in reaching this potential are the components and the techniques used in the processor. The instruction fetch efficiency, branch prediction, data and instruction caches, resource allocation, decode width, issue width and hardware constrains are the factors which have a great effect on utilizing this potential [5]. Among these factors, the instruction fetch has the greatest effect on the way parallelism is utilized in programs [6]. For this reason, the instruction fetch step has to be designed according to the

*Corresponding author. E-mail: javadi@aut.ac.ir

existing problems and limitations. Considering the relating to the instruction fetch, we have designed and optimized this unit for a RISC core. We have applied the cost per performance design style and quantitative approach to propose this fetch unit.

The next section will be an introduction to the benchmarks and the tools used in the simulation. In the third section, we will analyze the problems that exist in the instruction fetch and also the solutions that have been proposed to solve them. The instruction cache and its configuration that has great effect on the instruction fetch are discussed in section 4. The mechanism which is used for branch prediction is explained in section 5. The section 6 of this paper is focused on the method used for the instruction prediction. In section 7, we will discuss about the timing, the delay and access time of each component in the instruction fetch that must be placed in a superpipelined system. Finally the conclusion and the references are stated at the end of the paper.

2. The Simulation and Benchmarks

2.1. The Simulator

A cycle level simulator is used in the different sections of this paper. The simulator is called HydraScalar [7] and is the modified version of the sim-outorder simulator (from the SimpleScalar set) [8]. The main configuration is the same model as Alpha 21264 processor [9]. The simulation is an out-of-order execution. It uses an issue queue with a length of 64 instructions, an issue width of 4 for the integer instructions and of 2 for the floating-point ones. The memory has a two-level and non-blocking hierarchy. In the first level, we have a 2-way set-associative data cache with the size of 64 KB and an access time of two cycles. In the second level, we have an 8 KB direct mapped cache with an access time of 12 cycles.

The cache miss rate is calculated by using the Single-path method and the Sim-Cheetah simulator which is a part of the SimpleScalar set. The simulations are run by using the LRU replacement algorithm for both of the direct mapped and 2-way set-associative caches on a benchmark of 1500 million instructions in each program set, with a 32 to 256 bytes block and an 8 to 512 KB cache.

The CACTI 2.0 simulator is used to measure the memory access time [10]. An analytical model is used for the computation of the transistor level delay of cache [6]. The simulator is also capable of calculating the power consumption of the memory structure. The only thing that is taken into account in the calculation of the access time in the tables is the output data time of the simulations, while the comparison time of Tag is ignored. The CMOS 0.15um is the technology which is used over all the steps.

2.2. Benchmarks

The evaluations are performed using the SPEC2000 benchmarks [11]. This set consists of SPECint 2000 (14 integer programs) and SPECfp 2000 (12 floating-point programs) [12]. Among all these programs, we have chosen 7 integer programs and 3 floating-point programs. All of them are compiled using the gcc 2.7.2.3 compiler on a PISA set of instructions [13]. Also 100 million instructions are used from each benchmark.

3. Instructions Fetch Analysis

Considering the importance of the instruction fetch, the factors that limit the instruction fetch have to be recognized and elevated. Branch instructions have two drawbacks on instruction fetch. Primarily since the right fetching path is not known at the time of conditional branch, we have an unknown state for the fetching instruction. Also the destination address of a taken branch instruction is not defined and should be predicted. After a branch operation, the order of the access to the instructions is destroyed. So we need to access another line of the cache memory. Moreover at least some cycles are needed to find out that the predicted path or the indirect branch address is incorrect. These few cycles are called the misprediction penalty. This penalty dramatically diminishes the performance due to the high number of cycles [11].

Besides the incorrect branch prediction, control transfer can also deteriorate the performance. By control transfer of a taken conditional branch, we mean the calls and unconditional branches. Figure 1 illustrates how a straightforward fetching mechanism would handle the control instructions. As we can see, the two instructions following the taken branch are discarded in the first instruction block. The branch transfers the control to the second block, but the starting point is not the beginning of the block. As a result, the first instruction of the block is also discarded. Another control transfer exists in the second block. That is why another instruction is also discarded from this block. Four instructions are remained, while we have a fetching potential of 8 instructions.

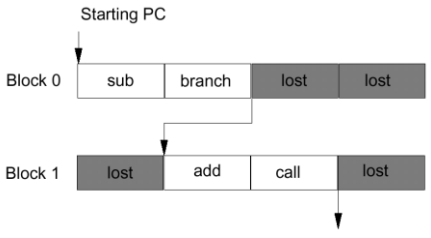

Fig. 1. Simple instructions fetch [5].

This example demonstrates that transfer control instructions cause two problems for the straightforward fetching mechanism. The first one is the alignment branch, which is caused when the destination address of the branch is not found at the beginning of the destination block. This problem can be solved in hardware level. The second problem occurs when a sequential access to the instructions of a line in the cache is stopped by the control transfer

instruction. So a new line should be read. Unlike the alignment branch problem, this one greatly limits the instructions placed in each fetching block.

3.1. Instruction Fetching Model

Figure 2 illustrates the different steps of the fetching operation. The instruction cache reads the requested block of width *q* and delivers it to the instruction fetcher. The instruction decoder receives a block of width *n*. When we use prefetching, then *q* new instructions of the instruction fetcher enter the prefetching queue of length *p* and *n* instructions come out of it. The inequality *q*>*n* can be deduced from the diagram. If prefetching is not used, the fetching and the decoding widths will be equal, and the instruction fetcher will directly transfer the instructions to the decoder. The instruction fetcher has the responsibility of determining the new instruction counter and transferring it to the instruction cache in each cycle. If we have a dynamic branch prediction, it will be used here. The new instruction counter should be determined in the same cycle. Also, after the targeted block has been received from the instruction cache, a primary decoding is performed for the type recognition of the instructions. Instructions following the first transfer control instruction will be invalidated as well.

4. Instruction Cache

In this section we will discuss about the hardware techniques that have been designed and optimized for the instruction cache.

4.1. Simple Cache

A simple technique used for fetching the instructions from the instruction cache is to have equal line size and width for the fetching block. In this technique, if the starting address of the instruction counter is not located at the beginning of the cache line, all the instructions preceding this one are invalidated and the number of the returned instructions will be less than the fetching width. In all techniques a control transfer instruction would invalidate all the other instructions. Figure 2 illustrates an example of the fetching mechanism which is explained in the previous section.

4.2. Self-aligned Cache

Destination alignment can be solved by using a self-aligned cache. In this method the instruction cache reads two lines in a single cycle and puts them together. Thus it can always return *n* instructions. This technique can be implemented in two ways [14]:

- Using a cache with two ports and having two accesses in each cycle.
- Designing a 2-way interleave cache.

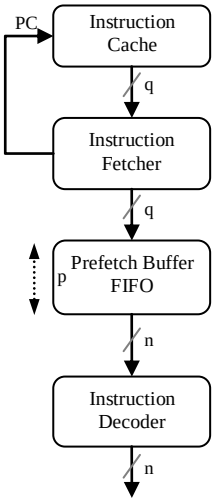

Fig. 2. Block fetch diagram.

Figure 3 illustrates an example of this technique. Since the program counter starts from the third instruction in line zero, the first two instructions are not used. Since the next line can be executed in the same cycle, four instructions (which are composed of two instructions of the line zero and two instructions of line one) are returned.

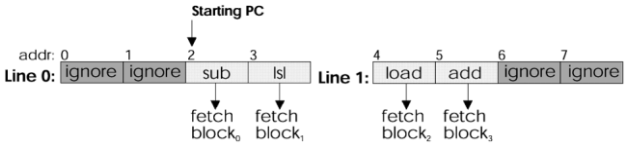

Fig. 3. Self-aligned fetch *n* = 4 [5].

4.3. Expected Instruction Fetch

Following are the number of the expected instructions for different fetching mechanisms based on an analytical model. By using this method, we can find the performance of the mechanisms [14].

4.3.1. Simple Cache

Assume that *n* is the width of a memory block and *b* is the probability of the control to be transferred. Let L_i be the probability of a control transfer to occur at position *i*. Also let E_i be the probability of the starting address in the block to be at the position *i*. Upon a control transfer, if the target address is equally likely to enter any position in a block, then:

$$E_1{}^{simple}(n,b) = 1 - \frac{n-1}{n} C^{simple}(n,b)$$

$$E_1{}^{simple}(n,b) = \frac{c^{simple}(n,b)}{n}, 2 \le i, \le n \quad (1)$$

$$L_i^{simple}(n,b) = \sum_{j-1}^{i} b(1-b)^{i-j} E_j^{simple}(n,b) \quad (2)$$

Where *C(n, b)* is the probability of a control transfer in a block.

$$C^{simple}(n,b) = \sum_{i=1}^{n} L_i^{simple}(n,b) = \frac{n}{\frac{1}{b}+n-1} \quad (3)$$

The total number of the expected instructions, fetched in each cycle for a simple fetch is as follows:

$$F^{simple}(n,b) = \sum_{i=1}^{n} E_i^{simple}(n,b) r(i,b)$$
$$= \frac{C^{simple}(n,b)}{b} = \frac{n}{1+b(n-1)} \quad (4)$$

Equation (4) is the weighted sum of the expected number of instructions to be fetched for each possible starting position.

4.3.2. Self-aligned cache

The probability of the control to be transferred in a block of cache is:

$$C^{align}(n,b) = 1-(1-b)^n \quad (5)$$

The expected fetch for the self-aligned cache in each cycle is equal to the expected execution of the block instructions with the size of *n*.

$$F^{align}(n,b) = r(n,b) = \frac{1-(1-b)^n}{b} \quad (6)$$

because *n* instructions will always be read from the instruction cache.

4.4. Evaluation of the Instruction Fetch Mechanisms

Figures 4 and 5 demonstrate an evaluation of the theory related to the instruction fetch mechanisms, where $b=1/8$. No prefetching is used and it is considered for diverse values of the decoder width (*n*). The value of *b* and the probability of control transfer are chosen to conform to RISC architecture [4]. Figure 4 indicates the number of the expected instructions to be fetched and in Figure 5, *C* indicates the probability of a control transfer in a block.

For *n=64* the fetching rate is approximately equal to $1/b$. Even though having a large fetching width would ameliorate the fetching performance, it cannot be implemented in hardware [15]. Figure 4 illustrates the expected values to be fetched for three types of cache, when *b = 1/8*. The best case occurs for a block size of n and when the instruction rate is equal to n in each cycle. It is demonstrated that the difference between the best case and the real case increases as n increases. In this case, the curve approximates 1/8. The main drawback of the simple cache techniques is the slow rate they have to reach this value. This Figure demonstrates that we need to have a large value for n to achieve the expected performance.

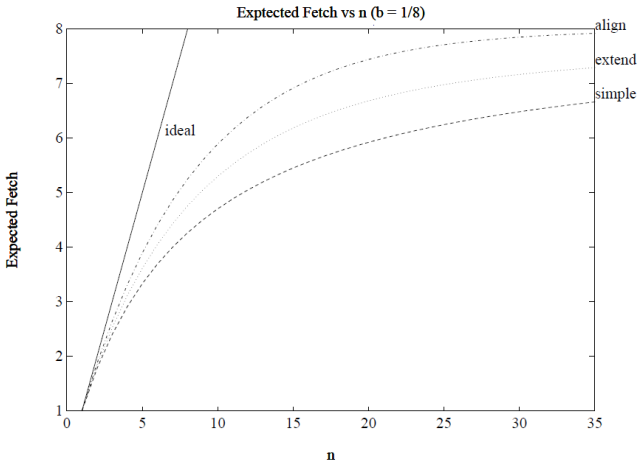

Fig. 4. Expected instruction fetch without prefetching.

Figure 5 illustrates the expected values for the instructions to be fetched in a self-aligned cache, for different values of *q* and *p*, with prefetching capability, and where $b=1/8$. At *q=6* and *p=8*, the expected fetch is already over 3.95.

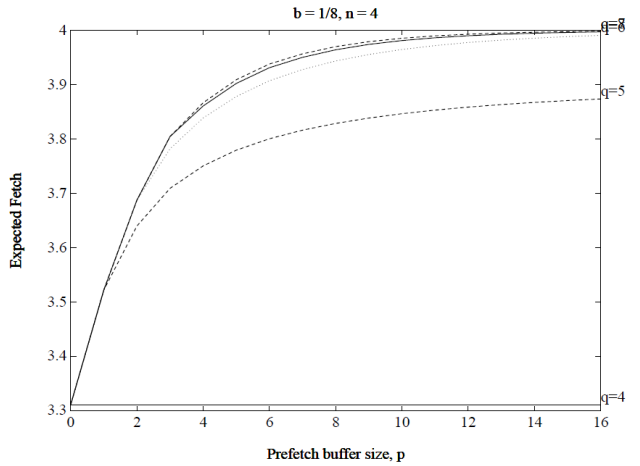

Fig. 5. Self-Aligned expected instruction fetch with prefetching (*n=4*).

It has been shown that for processors with high issue width (*n≥8)*, prefetching is not sufficient and we need to fetch two blocks in each cycle. Considering the results extracted from the analysis of this mathematical model [5], and as we are designing a 4-path processor, we can say that it is better to have a fetching length of 8 cycles, a prefetching queue and a self-aligned cache to achieve higher performance.

4.5. Instruction Cache Configuration

In the previous section we discussed about the techniques used in the fetching unit of the processors and the impact they have on the design of cache. In this section, we analyze and simulate the different configurations of the instruction cache.

We should consider the length of the fetching instructions in each cycle prior to the analysis of the results. Since 32 bytes are fetched each time (using two simultaneous accesses and 4 instructions in each access), the smallest size of the cache block is 32 bytes. This size is used in the simulation.

Figure 6 illustrates the average miss rate for different block sizes for 16 programs versus instruction cache size. As we can see, the miss rate is very high for the blocks with size of 32 bytes. This rate is a result of the fact that all data in a block having this size are read in a single fetching cycle. This allows for exploiting the locality in an optimized fashion.

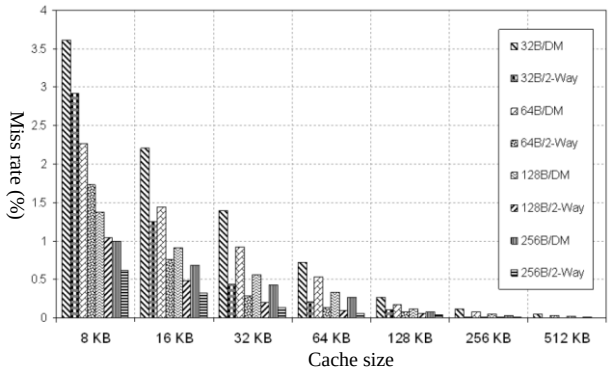

Fig. 6. Average miss rate for instruction cache with different configurations.

The design decisions applied to the instruction fetch unit of a RISC pipeline are based on the performance rather than the cost. The higher costs for the instruction cache would be acceptable to get the better performance as it is the first stage in the pipeline. This fact is more obvious for the superscalar processors. These processors fetch multiple instructions in each cycle and if a cache miss occurs in the instruction cache, a long interrupt happens and the performance is deteriorated. Figure 6 demonstrates that the cache miss rate in a direct mapped cache is much higher than the expected limit and we cannot get a suitable efficiency out of this type of memory.

Obviously, this core needs an instructions cache with a very low miss rate. Two instructions are accessed simultaneously in each fetching cycle and in more than 10% of these accesses (this value can reach 50% based on the block size) we need to read two blocks simultaneously. This can cause the miss probability in each cycle to increase. There is also the possibility of a miss to occur simultaneously for both of the blocks. This can cause a high penalty for the system. It is also predictable to have a very low miss rate for caches having a size of 256 bytes (especially for the large ones). But this will cause two problems that cannot be solved easily.

First the cache miss penalty increases as the block size becomes lager. Also as the block size becomes larger, the L2 cache will need larger blocks, therefore its miss rate will be increased. Enlarging its block size causes the miss penalty to increase. Besides, the power consumption has become approximately twice and this cannot be ignored for a block having a size of 256 bytes [14].

5. Branch Prediction

Branch prediction is the mechanism used for predicting the jumping path prior to the execution. As explained before, if branches are mispredicted, the performance of the processor is extremely deteriorated. So the first significant factor in the selection of a prediction method is the high precision. Also the technique that is applied must be fast enough to be used in a super pipelined system.

Branch prediction could be performed *statically* and *dynamically*. When performed statically, each branch is predicted as a taken or not taken, and when performed dynamically the prediction takes place at the run time and is based on the behavior of the branches. Dynamic predictors are used in advanced processors due to their high precision [11]. A Pattern History Table (PHT) is used in this method for maintaining each branch's state. The content of each entry in this table is a 2-bit up-down saturating counter. The addressing method for the PHT is widely proposed in the literature. In the simplest method addressing can be done by using the program counter. This simple mapping causes a noticeable Interference. Branch correlation and two-level adaptive prediction mechanisms are used for the effective use of a PHT. Figure 7 illustrates the basic diagram for the two-level adaptive branch prediction method.

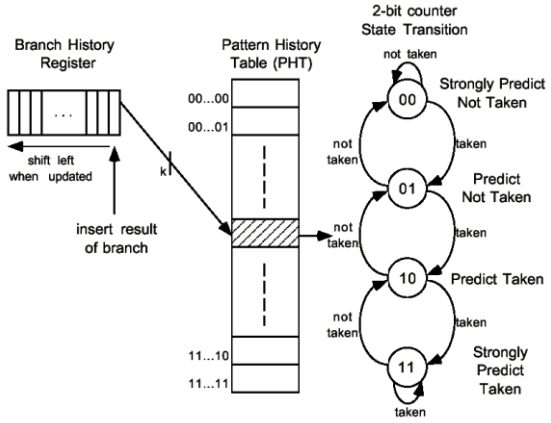

Fig. 7. 2-level adaptive branch prediction.

In this method a *k* bits branch history register is used for addressing a PHT of the size 2^k. As you can see, each entry

of the PHT contains a 2 bits counter. The prediction state is specified based on these two bits in the format of a finite state machine. The history register can be a global, single register or a Branch History Table (BHT) of registers that are addressed using the instruction counter. One of the solutions used to solve the problem of interference is to add a number of branch addressing bits to the history register for addressing the PHT. In this method, two independent branches which have the same history are mapped to different entries of the PHT. That is why they have different addresses [11, 16, 17].

5.1. Improvement of Branch Prediction Methods

The first problem designers found in the existing branch prediction structures was the conflict in PHT. But the main problem in the two-level structures is the wrong-history misprediction [18]. This is due to the existing local and global correlation in this structure. The local correlation performs the branch prediction based on the past behavior of the same branch. In the local correlation the branch prediction is performed based on the past behavior of all the previous branches. This problem arises due to the fact that in most of the programs, some of the branches are predicted using the global history and some others are predicted using the local history. According to studies, 35 to 50 percent of the total number of the mispredictions occurs in the global predictors related to the wrong-history misprediction, while the problem of interference occurs for 15 to 20 percent of the cases.

A hybrid predictor can be used as a solution for this problem[16]. In this method an appropriate structure is created by using the combination of one or multiple predictors and also a mechanism for selection of the component used for the branch prediction. If one of the components is a global predictor and the other one is a local predictor, the two types of predictors can be applied together [16]. The structure of this predictor is illustrated in Figure 8.

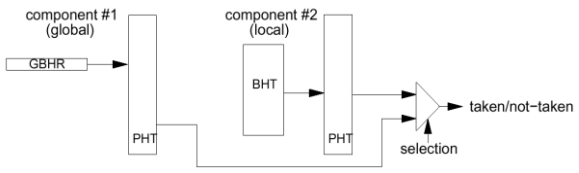

Fig. 8. Hybrid branch prediction.

If the selection mechanism between the components is well-done, this method alleviates the problem of wrong-history misprediction. An instance of such a structure is used in the Alpha 21264 processor [9]. The main problem of the hybrid predictor is its high hardware cost. This problem is due to the fact that the hardware is divided into various components and these components can work well only if they are big enough. For example, in the Alpha 21264 processor, more than 28Kbits has been expended for this section [9].

Other problem with hybrid method is how to design the appropriate selection mechanism. A method called the alloyed branch prediction has been proposed to solve this problem [19]. In this method, we concatenate the global and local histories to a number of branch address bits to perform the PHT addressing. These changes solve the two problems of the hybrid predictors. Moreover, in addition to the use of local and global history, there is no need for any selection mechanism. The structure of this mechanism is illustrated in Figure 9.

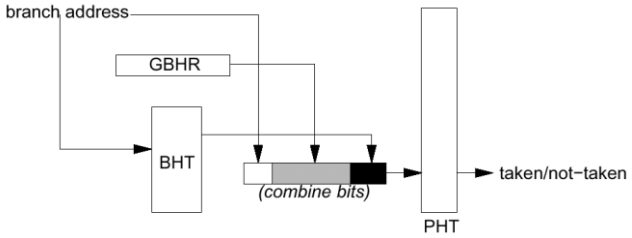

Fig. 9. Alloyed branch prediction.

5.2. Evaluation of the Branch Prediction Methods

In the following, we compared the two mechanisms for branch prediction as hybrid and allotted branch predictions. The best possible configurations are used in each structure to compare these two methods [19]. These configurations are demonstrated in Table 1 and Table 2. Three configurations are described in these tables with the cost equal to 64 Kbits, 8Kbits and 2Kbits. Since when the cost is equal to 2Kbits, the hardware components existing in the hybrid method are very small, this cost is infeasible for this mechanism.

Table 1
Configuration of alloyed branch prediction (g: number of bits in the global history, p: number of bits in the local history, a: number of bits in the branch address)

Alloyed Branch Prediction		64 Kbits	8 Kbits	2 Kbits
MAs	PHT (entries)	16K	2K	512
	BHT (entries)	8K	2K	512
	Index	9g, 4p,3a	7g, 2p, 2a	3g, 2p,4a

Table 2
Configuration of hybrid branch prediction

Hybrid Branch Prediction		64 Kbits	8 Kbits	2 Kbit
Gas	PHT (entries)	16K	2K	--
	Index	7g , 7a	4g , 7a	--
Pas	PHT (entries)	4K	512	--
	BHT (entries)	1K	512	--
	Index	8p, 6a	2p, 7a	--
Selector	(entries) PHT	8K	1K	--
	Index	6g , 7a	3g , 7a	--

The results of the simulation are specified in the form of branch prediction precision for the executed benchmarks in Table 3. Figure 10 illustrates the average accuracy of the branch prediction for various cases. As you can see, when

the cost is equal to 64Kbits, the precision of the hybrid method is slightly less than the alloyed method. In this structure, the hybrid method works well because of the high cost. We can observe the difference of these methods by studying the comparison of the cost equal to 8Kbits. The alloy method works better than the hybrid method, due to its decrease in cost. When cost is low, the alloyed method works as well as a hybrid predictor and when having low cost it works better than the hybrid prediction method.

Table 3
Branch prediction accuracy for hybrid and alloyed branch predictions

	64 Kbits		8 Kbits		2 Kbits
Benchmarks	Hybrid	Alloyed	Hybrid	Alloyed	Alloyed
Bzip2	0.9897	0.9901	0.9890	0.9890	0.9891
Gcc	0.9446	0.9416	0.8690	0.8887	0.7891
Gzip	0.9180	0.9113	0.9115	0.9117	0.9120
Mcf	0.9922	0.9894	0.9784	0.9789	0.9708
Vpr	0.8718	0.8691	0.8605	0.8602	0.8484
Parser	0.9465	0.9393	0.9242	0.9217	0.9140
Vortex	0.9838	0.9817	0.9569	0.9653	0.9198
Ammp	0.9937	0.9884	0.9858	0.9850	0.9806
Equake	0.9939	0.9872	0.9698	0.9719	0.9531
Mesa	0.9930	0.9880	0.9784	0.9791	0.9662

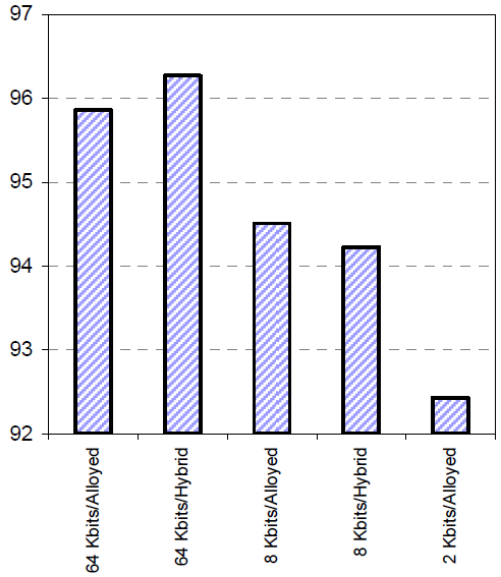

Fig. 10. Average accuracy of branch predictions.

6. Instruction Fetch Prediction

As mentioned before, the branch prediction structure is only for the prediction of branch paths. Thus another mechanism is needed for the prediction of branch target. This mechanism is called the instruction fetch prediction in which the next instruction is predicted to be fetched. The branch target buffer (BTB) is one of the mostly applied mechanisms for this operation [16]. For the taken conditional branches and the unconditional branches a BTB is used for the target address prediction [4]. Another technique applied for the instruction fetch prediction is using an NLS table. In this technique the line address and its set are kept in the instruction cache [11]. Indeed the pointer to the instruction is maintained instead of the instruction itself. The main idea of both methods is alleviating the penalty due to the wrong fetching.

The difference between the NLS and the BTB methods is found in the architecture of NLS which is a table without any tag. This table contains a pointer to the instruction cache which points to the location of the taken target branch. Figure 11 illustrates diagram of this architecture. As we can see in the figure, the next line is computed to be fetched instead of the next address. There are three sources for fetching the next instruction. These sources consist of the NLS predictor, the fall-through line and the return address stack. The NLS predictor consists of the following fields:

- Type field: Table 4 demonstrates the possible sources for prediction according to the Type field. This field is used to find out the suitable prediction mechanism which is specified in Figure 12. If the Type information is extractable from the instruction fetched in the fetching cycle or from the cache (if the information is predecoded), there is no need for this field.
- The line field: This field consists of the line number which should be fetched from the instruction cache. The high bits illustrate the line in the instruction cache and the low bits specify the real instruction in this line.
- The set field: It is possible to find *n* target lines in a set-associative cache. This field is used to show the set number in an instruction cache. When applying a direct-mapped cache there is no need for this field.

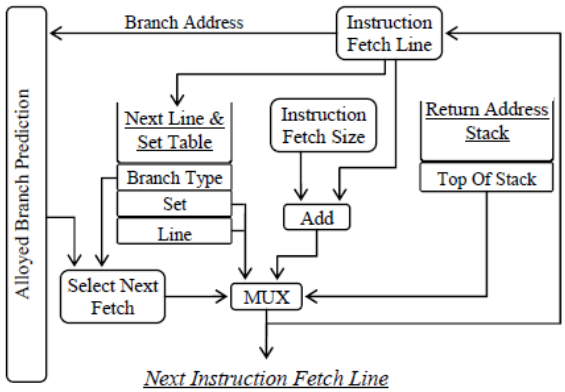

Fig. 11. NLS fetch prediction.

The NLS architecture assumes that it can decide whether an instruction is a branch or is not in the fetch step. Since non-branch instructions fetch the fall-through address and branch instructions use the NLS predictor, adding this information to the instruction can ameliorate the fetching precision of NLS. If the fetched instruction from cache is a branch instruction, the NLS predictor is applied and the Type field is tested for selection of the fetching address. The return instruction also uses the return address stack.

The conditional and indirect branches also use the cache which is specified by the NLS entry. If the Type field specifies an instruction to be of the conditional branch type, the architecture uses the PHT for the prediction of a branch path.

Table 4
Type field in NLS method

Bit 0	Bit 1	Branch type	Prediction source
0	0	Invalid entry	
0	1	Return instruction	Return stack
1	0	Conditional branch	NLS entry, Conditional on PHT
1	1	Other type of branches	Always use NLS entry

If the branch is predicted to be taken, the line and set fields are applied for fetching the instruction cache line. On the other hand, if the conditional branch is predicted not to be taken the fall-through address is used for fetching the next instruction.

The NLS entries are updated after decoding the instruction, specifying the type of the branch and determining the target. The instruction type specifies the Type field, the branch target, the line and set fields. Only the taken branches change the line and set fields. However all the branches change the Type field.

If a conditional branch that is predicted to be not-taken, changes the line and set fields, it will cause those fields to be cleared. One of the methods used for maintaining the NLS is storing the predictors in each line of the instruction cache. This method is called NLS-Cache and is applied in the Alpha 21264 processor [9]. It has been demonstrated that using a table of NLS entries which has no tag is a suitable structure for keeping the NLS predictors [16]. This structure is used in this processor.

6.1. Using the Next Line Addresses with the Instruction Cache

In the NLS architecture, there is no complete target address to be transferred to the instruction cache. In this method only the low bits of the target address (cache line index) exist. This is not considered to be a significant issue for the direct mapped cache, because testing the tag for the target address can be performed at the decoding step of pipeline. But if an associative cache is applied, it should change so that it could use the next address properly. Following are two different methods used for this reason.

In the standard implementation of an associative cache, the appropriate line is selected from a set by comparing the whole tag with various sets. The NLS predictor set field is used for all branch instructions to predict the instruction cache set. But a full comparison of the tag is performed when using the fall-through line address.

The second method applied for using the address of the next line with an associative cache is a little more accurate than the previous method and can ameliorate the performance of the cache. In this method, we assume that each line of the cache has a set field. This field resembles the NLS set field and predicts the set which is related to the location of fall-through line in each cache line. Since the set field is used in each access, only one set of cache can work at a time and the tag comparison operation can be performed at the time of decoding (like the direct mapped method). If prediction of the set is wrong and the tag is not identical to the target address computed in the decoding step, we need to test other sets to find the right entry or to find out a cache miss. This design is suitable for an associative cache L2. If the first prediction is wrong, the remaining set of the instruction is tested. We need to use other methods for caches with an associative level higher than two [14].

6.2. Comparison of BTB and NLS

Compared to BTB, NLS architecture has a better cost per performance ratio. Another advantage of the NLS is that its table should not have any tag, while a BTB must be 2-way or 4-way associative so that it could reach a higher performance. We have approximated the NLS and BTB access times with various sizes by applying the CACTI simulator. As illustrated in Table 5 the access time to a 4-way BTB is approximately two times greater than a NLS table with the same size. So the NLS architecture has a better access time than BTB.

Table 5
Access times for NLS and BTB

Entries	2-way BTB Access Time (ns)	4-way BTB Access Time (ns)	NLS Table Access Time (ns)
128	0.88	0.88	0.38
256	0.91	0.92	0.40
512	0.93	0.96	0.44
1024	0.97	1.00	0.47
2048	1.12	1.08	0.51

7. Fetch Timing

Pipelining a logical circuit is a low level task, for which we need to design all the circuit in a transistor level. Then, we can perform the pipelining operation based on the obtained delays. Considering that the task of designing the instruction fetch step is high level, our approach is only to present an approximate model for the scheduling of the circuits that have been designed. Since the majority of the RISC processors have simple control circuits, the memory and table components (containing branch prediction tables, register file and instruction issue queues) are considered as the critical path. In order to pipeline the fetch step, we assume the access time for the register file to be the processor clock cycle. The register file consists of 80 integer registers, 80 floating-point (64 bit) registers and an

access time equal to 0.95ns [16]. Considering these assumptions, we perform the instruction fetch timing.

7.1. Instruction Cache Timing

The AAT (Average Access Time) has an important role in cache (especially instruction cache) design. In the following section, various parameters corresponding to AAT have been simulated and finally AAT has been computed. Considering other effective parameters, the appropriate options will be used for the final decision to be made.

7.1.1 Evaluation of the Access Time

In this section, we will discuss about the timing model for the instruction cache. The cache is the starting point of execution in the instruction fetch unit of a processor. As mentioned before, since the majority of the RISC processors have simple control circuits, only the memory and table components are presented as the critical path for the timing model. The instruction cache is one of the critical paths that can be effective for the performance of the processor.

The instruction cache should be designed in a way that it could obtain the appropriate access time and cycle for the processor to be able to attain the maximum performance.

Table 6 shows various configurations of cache. As you can see the access time of the cache is higher than 1ns, except for the cases where its size is equal to 8KB or 16KB. This fact is very important since the processor cycle (considering no overhead) is accomplished in 0.95ns and all the caches which have an appropriate miss rate have access time of more than two cycles. Of course it is completely natural for the access time to increase for caches with bigger size and higher degree of associativity. Since a cache with a low miss rate is needed for the design of an instruction cache, we need to find an appropriate method for the optimization of the access time.

Table 6
Access time for instruction cache (the access times less than 0.95 ns are highlighted)

	32 Byte		64 Byte		128 Byte		256 Byte	
Bit Rate	DM	2-Way	DM	2-Way	DM	2-Way	DM	2-Way
8 KB	**0.681**	0.956	**0.685**	**0.936**	**0.708**	0.956	**0.752**	1.075
16 KB	**0.762**	1.018	**0.762**	0.998	**0.784**	1.002	**0.828**	1.127
32 KB	**0.933**	1.167	**0.933**	1.114	0.966	1.128	0.962	1.224
64 KB	1.136	1.310	1.136	1.294	1.119	1.343	1.171	1.462
128 KB	1.567	1.585	1.365	1.519	1.347	1.577	1.430	1.799
256 KB	1.767	1.919	1.786	1.886	1.789	1.991	1.927	2.230
512 KB	2.241	2.530	2.255	2.474	2.255	2.580	2.397	2.823

The wave pipelining technique is used as a solution to this problem. This technique is considered to be one of the most suitable methods for the optimization of cache access time.

7.1.2. Wave Pipeline

One of the widely used methods to decrease the execution cycle of a logical circuit is pipelining. Common pipelining methods used for optimization cause an overhead on the delay, cycle time, area and the power consumption. Also a low level design of the circuit has to be presented to obtain the timing parameters and it has to be implemented in transistor level to get useful information about the timing of the pipeline [14].

The extra overhead of the cycle time is due to the extra time needed for the propagation of the signals through the synchronizer, the time needed for the measurement of the components of the synchronizer before the storage of signals, and also the undesirable deviation of the clock pulse in the time of the synchronization signal arrival.

In common pipelines, delay is defined as the time elapsed from readiness of input data (in the first step of the pipeline) till the time when the output data reaches the last step of it. The extra delay is caused by the overhead due to creation of the pipeline. In logical circuits with standard pipelines and common clock pulses for all the synchronizers, we have a partitioning overhead when combinational logical circuits cannot be divided into multiple steps with maximum and equal propagation delays.

The area and the consumed power impose extra overhead which is due to additional transistors and wires, the increase in clock buffer area and power needed for the input of the synchronizer to operate. The additional transistors and wires are used for the implementation of registers and synchronizer latches.

Compared to standard pipelining, wave pipelining is a technique that enables digital systems to attain higher clock pulse. This technique relies on the limited propagation delay of signals in a combinational logical circuit. This delay is used for data storage. Thus multiple waves of data can be propagated in various regions of a logical circuit. Indeed this method exploits the circuit propagation delay efficiently.

Since each data wave is exploited after being sure that there is no interference, the performance yield from the wave pipeline of synchronizer systems is higher than the performance of standard pipelines. Wave pipeline can attain the performance of physical switching.

This optimization is the result of decreasing the extra overhead existing in standard techniques. Since there is no synchronizer, the cycle overhead existing in the previous method is omitted. The circuit cycle is obtained from the changes in the signal propagation delay of the circuit and the delay of the input/output register. In the wave pipeline, since there is no need for dividing the circuit into independent units (by using synchronizer), the extra overhead of this operation is omitted. The area and the power consumed by the circuit are decreased by ignoring

the internal synchronizers and the corresponding circuits in the standard pipeline. Pipelining a memory by using this method causes some limitation that we consider them in the following. List of parameter used in the analysis are listed in Table 7.

Conventional synchronizer circuits must consider the timing limitations for longest path and the race in the network. Race limitations necessitate the data to be able to pass through a synchronizer operator, be propagated in the network and then enter the next synchronizer operator in a single clock cycle. Thus the lowest time needed for the propagation of data from one synchronizer operator to the next synchronizer operator, must be less than the starting time elapsed from the output edge till the locking time in the same cycle. Thus the result from actual input data will not have any interference with the result from the previous data. The limitations imposed by the longest path, necessitate that the results of the inputs of this cycle, be valid till the next cycle for the next synchronizer operator. Thus the time needed for the propagation of data from a synchronizer operator in the combinational network to the next synchronizer operator must be less than the time interval between the starting edge of the actual clock cycle to the locking edge of the next clock cycle.

Table 7
List of parameters in the equations

T_{max}	Maximum propagation delay in a combination network
T_{min}	Minimum propagation delay in a combination network
RF_{min}	Minimum rise time and fall time of shortest path in a network
RF_{max}	Maximum rise time and fall time of longest path in a network
ΔC	Maximum clock skew
T_s	Minimum setup time
T_h	Minimum holding time

In addition to these two common limitations found in pipelining systems, the different data waves need not to conflict with each other in any point of the combinational circuit in wave pipelining. The following equation expresses these limitations.

$$T_{clk} \geq T_{max} - T_{min} + 2\Delta c + T_5 + T_h + \frac{RF_{min} + RF_{max}}{2} \quad (8)$$

In addition to the output limitation, wave conflict will not happen in the whole logical network. Here is the equation representing that:

$$T_{clk} \geq T_{max} - T_{min} + \Delta c + T_{ms} + \frac{RF_{min} + RF_{max}}{2} \quad (9)$$

Where T_{max} is the smallest time that a voltage must be stable for the logical levels to be able to become stable reliably and correctly. More details concerning the limitations of wave pipelining can be found in [20]

Table 8 illustrates the memory access time after simulation, by using the wave pipelining method and the possible states being specified (the optimized values are stated in bold).

Table 8
Access times for instruction cache with wave pipeline (optimized cases with wave pipeline are highlighted)

	32 Byte		64 Byte		128 Byte		256 Byte	
Bit Rate	DM	2-Way	DM	2-Way	DM	2-Way	DM	2-Way
8 KB	0.681	**0.478**	0.685	0.936	0.708	0.956	0.752	1.075
16 KB	0.762	**0.509**	0.762	0.998	0.784	1.002	0.828	1.127
32 KB	0.933	**0.583**	0.933	1.114	0.966	1.128	0.962	**0.612**
64 KB	1.136	1.310	1.136	1.294	1.119	**0.672**	1.171	**0.731**
128 KB	1.567	1.585	1.365	1.519	1.347	1.577	1.430	**0.900**
256 KB	1.767	1.919	1.786	1.886	1.789	1.991	1.927	2.230
512 KB	2.241	2.530	2.255	2.474	2.255	2.580	2.397	2.823

Using this method, various desirable states are generated having an appropriate access time and low miss rate. Our final choice will be one of these configurations.

7.1.3. Performance Evaluation of the Instruction Cache

We have calculated the miss rate and the access time to the stated configurations in section 4. In the following section, we will calculate and discuss about the average access time to these configurations.

The penalty of the cache miss rate is the delay that a processor suffers so that data transfer from the lower level memory to the cache is needed. If we consider the lower level of the memory to be ideal, the number of the penalty cycles can be calculated by using the following equations [14]:

$$Miss\ Penalty\ Cycle = Addr.Transfer\ Cycle \times \frac{l_2 Bus\ Cycle\ Time}{CPU\ Cycle\ Time} + L_2\ Cache\ Access\ Cycle + \frac{l_1 Block\ Size}{L_2\ Bus\ Width} \times \frac{l_2 Bus\ Cycle\ Time}{CPU\ Cycle\ Time} \quad (10)$$

This equation computes the miss penalty for the on-chip and off-chip L2 cache. The delay due to the off-chip L2 cache can be calculated using the expression *L_2 Bus Cycle Time / CPU Cycle Time*. The off-chip bus has a much lower frequency and data is transferred with the bus frequency. The first expression shows the number of the cycles needed for the transmission of data to the lower memory. The second expression, illustrates the time needed for a successful access to L_2 cache. Finally the third expression shows the number of cycles needed for the transmission of data within a block. This last expression is expressed in the

bus width and the ratio of its frequency to the processor frequency.

Table 9 demonstrates the miss penalty for the two states where we have an off-chip and on-chip caches. The values are illustrated for four caches L1 which have the size of 32 bytes to 256 bytes. For the off-chip cache, the clock cycle time of a bus is considered to be 4 times greater than the clock cycle of the processor and the access time to the cache L2 is also assumed to be equal to two cycles. In this section we have considered the cache L2 to be an ideal on-chip cache.

Table 9
Miss rate in the caches

Block size (Byte)	A. T. Cycle	Bus width (Byte)	L2 Access Time (cycle)	Miss penalty (L2 on-chip) (cycle)	Miss penalty (L2 off-chip) (cycle)
32	1	32	2	4	10
64	1	32	2	5	14
128	1	32	2	7	22
256	1	32	2	11	38

Figure 12 illustrates the diagram of the average access time for the states where we have an on-chip and an off-chip cache. The configurations have an average access time of less than two cycles. As you can see, even if we use the wave pipelining technique, small caches (tough having a small access time) do not have an appropriate performance. The reason can be found in the high miss rate of these configurations. The best situation occurs for the two cache types of (64KB, 12KB, 2) and (128KB, 256K, 2). The values for their AATs are respectively equal to 1.006552 and 1.003762. As you can see, the performances of these two configurations are very close to each other.

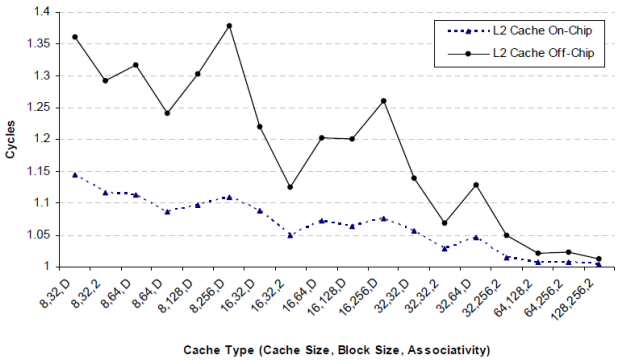

Fig. 12. Performance of the caches.

Since the AAT values for these configurations are very close to each other, we used the BIPS (Billion Instruction Per Second) metric as a second parameter to select the most appropriate configuration. AAT presents the average access time for the various configurations in a processor with a constant cycle time, where the effect of the access time is not considered in the processor cycle. This parameter reveals its effects properly, when computing BIPS. In other words, BIPS specifies the best configuration for the cache. The processor can only reach this performance when the instructions are executed in the expressed clock cycle. But AAT computes the performance of each configuration based on a specified clock cycle.

We will evaluate these parameters using Table 10. In this table, 4 best configurations are stated regarding their performance. Besides the values for AAT and BIPS, the consumed power and area are computed respectively by using a CACTI 2.0 simulator and Cache Design Tools. As specified in the table, the second option is more suitable regarding the cost per performance ratio. Even though this configuration does not have the best AAT, but it is a very close approximation to the optimized state. It also has a cost equivalent to the half cost of the first state regarding the power and the needed area. There is also a big gap between the value of its BIPS and the value of the first configuration's BIPS. The third and fourth options have nothing superior regarding BIPS, AAT and other parameters as the consumed power. Even though the fourth option has a lower area cost, but since there is a big difference between its AAT value and the value of the optimized state, it cannot be considered an appropriate option.

Table 10
Evaluation of selected configurations

Cache Config.	AAT Off Chip	AAT On Chip	BIPS	Access Time (ns)	Miss Rate (%)	Power (nj)	Area (mm^2)
128KB,256B,2	1.0130	1.0038	2.062	0.900	0.034	8.257	36.262
64KB,128B,2	1.0206	1.0066	2.662	0.672	0.074	4.091	18.499
64KB,256B,2	1.0227	1.0066	2.523	0.731	0.060	6.428	18.666
32KB,256B,2	1.0502	1.0145	2.646	0.612	0.132	5.269	9.865

7.2. Branch Prediction Timing

The second critical operation is the branch prediction which is performed by using the alloyed method. Table 11 demonstrates the access time for various sizes of PHT and BHT tables. This table shows that if we use a 64Kbits predictor and consider that the tables are accessed sequentially, the total time needed for a branch prediction is approximately equal to (0.53 + 0.53)=1.06ns (refer to the third row of Table 11). This amount of time is higher than the time needed for one cycle. This problem does not occur for the configurations where we use 2Kbits and 8Kbits predictors. The solution is to partition the PHT into multiple physical tables and access all of them in parallel.

In the modified method the local history bits are needed for the selection of a multiplexer which has an input for the output of each table. This method is specified in Figure 14. According to [16] the largest multiplexer needed in this state is a 16x1 multiplexer. The delay of this multiplexer is computed by using the Horowitz approximation method [10] (used in the CACTI simulation) and is approximately equal to 0.24ns.

Table 11
Access times for PHT and BHT

Cost of Predictor	PHT			BHT		
	Entreis	Access Time (ns)	Entry width (bit)	Entreis	Access Time (ns)	Entry width (bit)
2K	512	0.33	2	512	0.33	2
8K	2 K	0.40	2	2 K	0.40	2
64K	16 K	0.53	2	8 K	0.53	4

Since PHT table is divided into 16 segments, its access time is equivalent to the access time of a table with 1024 entries, and equals 0.37ns. Using this technique, the time needed for a branch prediction operation (used in a 64Kbits predictor) approximately equal (0.24 + 0.53) = 0.77ns. This value is considered to be an appropriate amount of time for the branch prediction. The maximum time needed for the operation of finding the next address to be fetched (which is performed by the NLS table) is 0.51ns, which does not cause any trouble for our cycle. According to Figure 14, the timing of the instruction fetch related to a 64 Kbytes 2-way set-associative cache, a 64Kbits predictor and an NLS table with 1024 entries are specified.

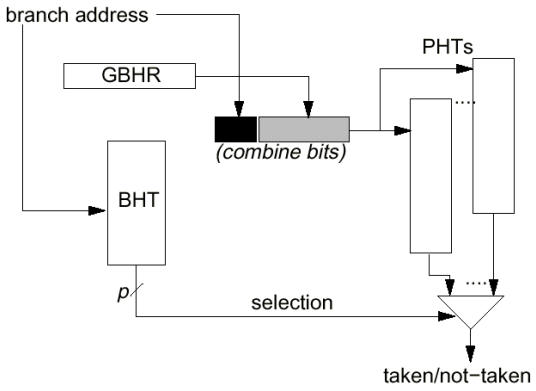

Fig. 13. Modified alloyed branch prediction to simultaneous access to BHT and PHT.

8. Conclusion

According to the widespread use of many core architectures, there has been an increased need for an optimized design and higher performance for each processing core. Instruction fetch is the most important factor having an effect on the performance of a RISC processing core. These processors need a high instruction width to use of parallelism available in their architectures. The first step for the execution of multiple simultaneous instructions is fetching them from the memory. In order to reach the maximum level of parallelism, we must have a maximum number of simultaneous instructions.

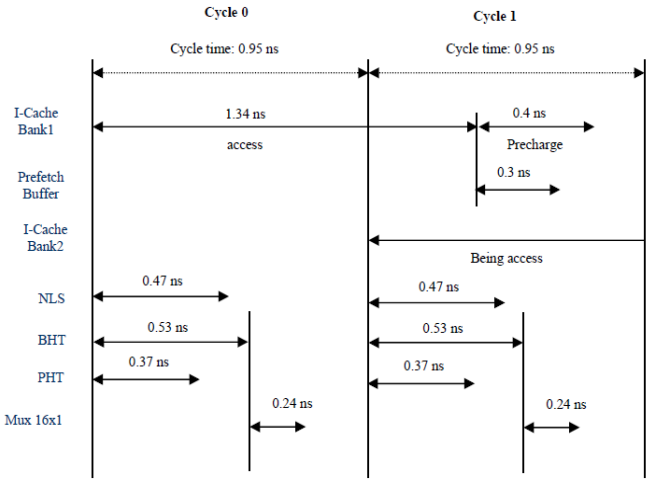

Fig. 14. Timing of instruction fetch stage.

In order to have a high performance fetch mechanism for a processing core with four issues, we need to fetch simultaneously eight instructions in each cycle and also use a prefetching queue and a self-aligned cache which has two ports for two simultaneous accesses.

Given the limitations due to the branch instructions in each fetching step, the alloyed branch prediction method has been proposed, which has an appropriate accuracy and the ability to be implemented with low costs. Because of the superiority of the NLS method (concerning the cost per performance ratio and also the lower delay compared to the BTB method) this method has been used for the instruction fetch prediction. Pipelining the unit of instruction fetch is performed by obtaining the delay of each component (assuming the clock cycle to be equal to 0.95ns) and by using the wave pipelining for the instruction cache and also dividing the PHT table into smaller tables.

The results show that in order to have a high performance processing core we need to make a lot of considerations and that the instruction fetch step is one of the most complicated units in the modern processors.

References

[1] D. Geer, Chip makers turn to multicore processors, Industry Trends, IEEE Computer Society, pp. 11-13, 2005.

[2] J. Held, J. Bautista and S. Koehl, From a few cores to many:a tera-scale computing research overview, Intel White Paper, 2006.

[3] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer and D. Shippy, Introduction to the cell multiprocessors, IBM Journal of Reserch and Developments, Vol.49, No.4/5, July/Sept. 2005.

[4] J. L. Hennessy and D. A. Patterson, Computer architecture: a quantitative approach, Morgan-Kaufmann, 4th edition, 2006.

[5] S.D. Wallace, Scalable Hardware Mechanism for superscalar processors, Ph.D. dissertation, University of California, Irvine, 1997.

[6] M.K. Akbari, M. Shojaei, O. Aghalatifi and B. Javadi, Design and simulation of cache controller unit for a risc processor, 7th Annual CSI Conf. (CSICC 2001), ITRC, Tehran, Iran, 2001.

[7] K. Skadron and P. S. Ahuja, HydraScalar: A multipath-capable simulator, Newsletter of the IEEE Technical Committee on Computer Architecture, Jan. 2001.

[8] D. Burger and T. M. Austin, The simplescalar tool set, Version 2.0, Technical Report #1342, University of Wisconsin-Madison Computer Sciences Department, June 1997.

[9] R. E. Kessler, The Alpha 21264 microprocessor, IEEE Micro, pp. 24-36, Apr.1999.

[10] G. Reinman and N. Jouppi, An integrated cache timing and power model, Compaq Corp., Western Research Lab., 1999.

[11] M.K. Akbari, B. Javadi, M. Shojaei and O. Aghalatifi, Design and simulation of fetch unit for a RISC processor, 7th Annual CSI Conf. (CSICC 2001), ITRC, Tehran, Iran, 2001.

[12] Standard Performance Evaluation Corp. SPEC CPU 2000 Benchmarks. http://www.specbench.org, 2000.

[13] Charles Price, MIPS IV instruction Set, revision 3.1. MIPS Technologies, Inc., M ountain View, CA, Jan. 1995.

[14] M. Shojaei, Design and Simulation of cashe system for a RISC processor with multi-processor capability, MS Thesis, Computre Engineering and Information Technology Department, Amirkabir University of Technology, 2001.

[15] S. Wallace and N. Bagherzadeh, Instruction fetching mechanism for superscalar microprocessors, Euro-Par'96, Aug. 1996.

[16] B. Javadi, Design and simulation of super-pipelined and super-scalar system for a RISC processor, MS Thesis, Computre Engineering and Information Technology Department, Amirkabir University of Technology, 2001.

[17] A. N. Eden and T. Mudge, The YAGS branch prediction scheme, Proc. of Micro-31, pp. 69-77, Dec. 1998.

[18] K. Skadron, M. Martonosi and D. W. Clark, Alloyed global and local branch history: a robust solution to wrong-history mispredictions, Technical Report TR-606-99, Princeton Dept. of Computer Science, 1999.

[19] K. Skadron, M. Martonosi and D. W. Clark, Alloying global and local branch history: taxonomy, performance, and analysis, Technical Report, Princeton Dept. of Computer Science, 1999.

[20] W. Burleson, M. Ciesielski, F. Klass and W. Liu, Wave-Pipelining: A tutorial and research survey, IEEE Trans. on VLSI vol.6, no. 3, pp. 464-474, September, 1998.