Superscalar and Superpipelined Microprocessor Design and Simulation: A Senior Project

Victor Lee, Nghia Lam, Feng Xiao and Arun K. Somani, *Senior Member, IEEE*

***Abstract*— An undergraduate senior project to design and simulate a modern Central Processing Unit (CPU) with a mix of simple and complex instruction set using a systematic design method is presented. The main objectives of the project are to accustom the students with modern design methods as well as to help the students gain practical experience in designing digital computers. Findings and suggestions on the use of modern work and design concepts in a group project environment are discussed.**

***Index Terms*— Processor design, systematic design methods, work and design concepts, complex instruction set, superscalar, superpipeline, simulation of systems, class project.**

I. Introduction

THIS paper presents a senior study project in the Department of Electrical Engineering at the University of Washington. This project is the work of three senior undergraduate students in the department. The goals of the project were twofold. First, it allowed the students to exercise their knowledge in computer architecture and hardware design. Second, it provided them with practical work experience involving team work and project management. One of the two main objectives of this paper is to present the findings of how well these concepts work in the group project environment. The second objective of this paper is to present the modern design approach that was used in the project.

The scope of the project was to design and simulate a realistic and modern Central Processing Unit (CPU) including both simple and complex instructions for a microcomputer system in a ten-week quarter time frame. This setting provides several challenges. First, a ten-week period is a very short time to design and simulate a complete processor. Therefore, decisions had to be made to simplify the design to keep the task achievable. Second, attempts to provide realistic and modern architecture add extra burdens to the heavy workload of the project. Third, division of work among team members and following an agreed-upon aggressive time table was a real struggle as well.

After consulting with the project advisor, Dr. Arun K. Somani, the design team decided that the only way to complete the project successfully was to follow a modern design approach. After defining the architectural goals, the team developed a very aggressive and tight project schedule (see Appendix I). In addition, coding and documentation standards were introduced to facilitate communication among team members. Moreover, many design and work concepts taken from previous projects and/or work experience were applied in this project. Some of these concepts worked well in this environment and some of them did not.

As the project progressed, the team encountered many additional problems. For example, one team member had to leave for a job in the middle of the quarter. The team had to be restructured and the project schedule had to be redefined. The restructured schedule turned out to be even more aggressive. Fortunately, the flexible nature of the modern design approach helped the team recover from the drawbacks and enabled the team to finish the project successfully on time.

Our goal as the authors of this paper is to summarize our experiences in applying many modern design and work concepts as well as to discuss the steps we took to resolve problems in a team-work environment.

We used several modern design and work concepts while doing this project. The main design and work concepts used included the following.

1) Top-down architecture design and bottom-up implementation process.
2) Modularized approach for complicated system design.
3) Coding and documentation standard.
4) Version control.
5) Cooperative learning environment to exchange ideas and facilitate development.

The following hardware and software topics were covered in the project are of interest in this paper.

1) Pipelined parallel computer architecture.
2) Cache design.
3) Data hazard detection and forwarding concepts.
4) Verilog XL programming techniques.

We hope that this paper will provide useful suggestions to other students who are interested in attempting similar projects. Our overall design approach is described in Section II and project management techniques used are listed in Section III. We outline the design and work concepts applied during the project in Section IV and main features of our architecture are presented in Section V. Following that, we explain the hardware partition we arrived at in Section VI. We did some measurements on the simulated systems and the results are

Manuscript received July 21, 1994; revised March 15, 1995.

V. Lee, N. Lam, and F. Xia are with Department of Electrical Engineering, University of Washington, Seattle, WA 98195 USA

A. K. Somani is with the Department of Electrical Engineering and the Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195 USA.

Publisher Item Identifier S 0018-9359(97)01546-X.

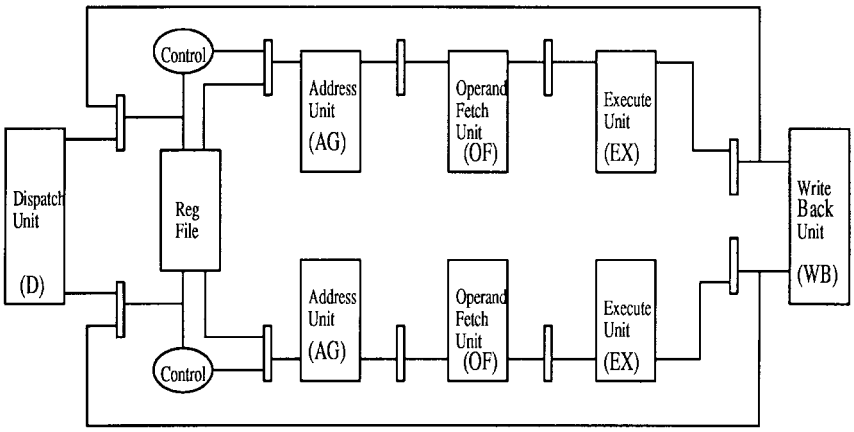

Fig. 1. Internal core of the superpipeline and superscalar processor.

presented in Section VII followed by our conclusions in Section VIII.

II. Overall Design Approach

In this project, we followed the top-down architecture design and bottom-up implementation process. The steps we took included:

1) defining our goals clearly;
2) specification of the processor under design;
3) lay down the overall architecture for the processor;
4) define and development a schedule;
5) break down individual parts of the CPU to separate modules (modular design approach) and assignment of design works of modules to each member of the design team;
6) build individual modules;
7) test the low-level functionalities of each module;
8) integrate the modules;
9) test the overall CPU functionality; and
10) document the design and the design process.

In the following subsections, we will briefly describe each step in the design process.

A. Defining Goals

At the beginning of the project, we defined the architectural goals of the processor. The starting point for the design team was the book we followed in a previous computer design and organization course [1]. To give us a better idea of what some of the realistic goals were, we studied the architecture of several modern CPU's: Intel Pentium^{TM} [2], DEC Alpha^{TM} [3], MIPS R4000^{TM} [4], Motorola 68040^{TM} [5], HP's Precision^{TM} [6], and IBM/Motorola PowerPC 601^{TM} [7]. We compared the design tradeoffs of the different architectures and narrowed down our goals to implement a CPU with mostly simple instructions and some minor complicated instruction component. The motivation here was to understand the difficulties presented by direct memory instructions present in some microprocessors such as Intel x86 series and Motorola 680x0 series. Subsequently, we have also learned of the implementation of the P6 architecture [8].

B. Specification of the Processor

Our CPU included a simple instruction set which can be easily implemented as a superscalar, superpipelined machine. We borrowed the instruction set from MIPS R2000 [1]. We enhanced the MIPS R2000 instruction set with direct memory operand instructions for all R-type instructions. The processor was also to include separate data and instruction caches, each of 8 kB. More specification details are provided in Appendix II.

C. Layout of Architecture

Having defined the specifications, we began the top-down architecture design. First, we defined the instruction set as well as the addressing modes we wanted to support (see Appendix II for details). Next, we started to design the internal structure of the CPU using superscalar and superpipeline concepts [9]. Based on this, we divided the CPU pipeline operation into the following stages: Instruction Fetch (IF), Instruction Dispatch (ID), Instruction Decode (D), Address Generation (AG), Operand Fetch (OF), Execution (EX), and Write Back (WB). Each stage was studied further for the possibility of breaking it down more. Fig. 1 shows the structure of the CPU core.

To improve the performance of the CPU, we included a Branch Prediction Unit and two on-chip caches, one for instructions and one for data. A Bus Interface Unit (BIU) and a memory system that supported byte addressing were also provided to complete the design. Fig. 2 depicts the overall CPU functional block diagram.

Interested readers should turn to Section V for a more detailed description of each unit.

D. Schedule Development

Next, we began to lay down the development schedule. As stated earlier, the whole project was to be completed in a ten-week term, so the schedule had to be aggressive but achievable. Also, since all students were in their final quarter, we kept margins for one of them being not available for some time due to an interview call or other similar reasons. Our project schedule is shown in Appendix I.

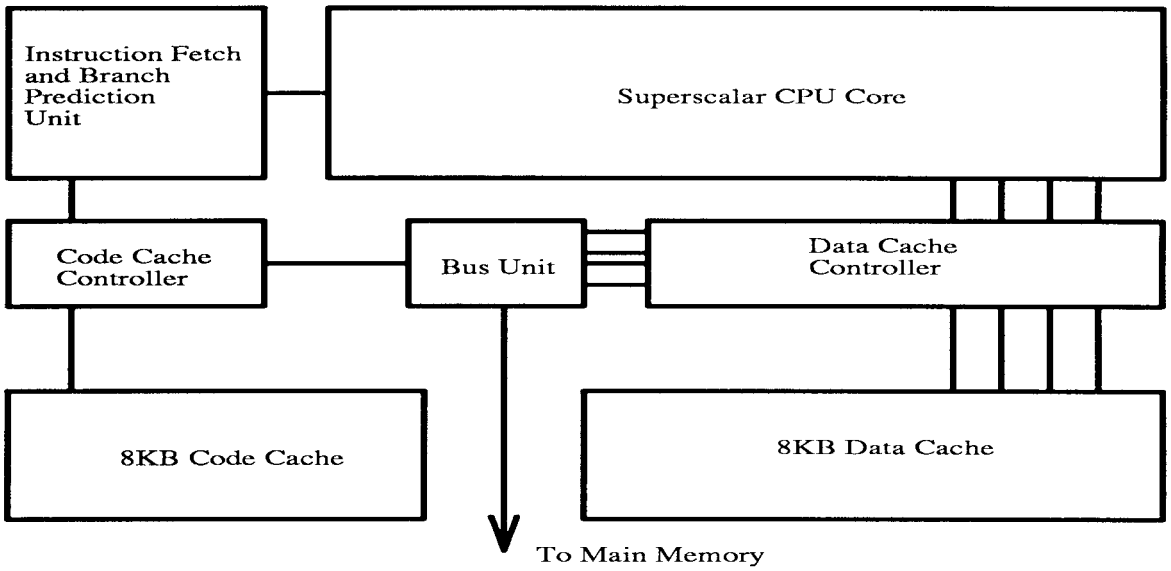

Fig. 2. CPU functional block diagram.

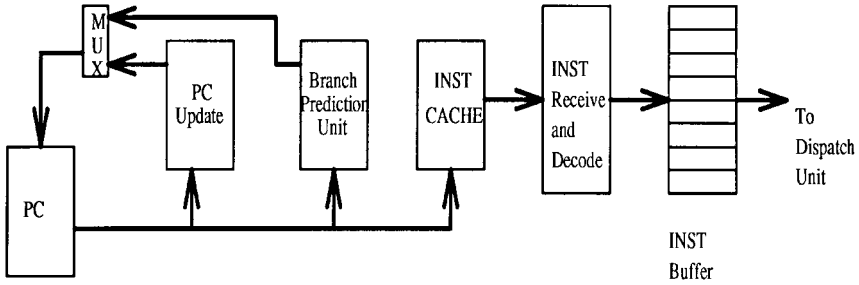

Fig. 3. Breakdown of IFU.

E. Breaking Down Individual Parts of the CPU

Once we had the development schedule, work was assigned to each individual team member. Each stage in a pipeline was a natural part to design. However, some of the units were further broken down into multiple modules. For example, Instruction Fetch Unit (IFU) has many parts. PC update by incrementing PC or from Branch Prediction Unit, instruction fetching from Instruction Cache, Instruction Receive and Decode Unit, and instruction storage in Instruction Buffer and Management Unit were modules of IFU as shown in Fig. 3. Each module in the pipeline design was assigned to an individual. The functional specification of each stage were clearly laid out. The interface between all pairs of interacting stages (neighbors in the pipeline) were clearly defined including the signal names and naming conventions. The meaning of each interface signal and the signal conventions (i.e., negative- or positive-edge trigger or synchronous clock latch) were also defined. Then the design team along with the project coordinator estimated the time it would take to design each module. This along with the expertise of each member became the basis for work design Then, we broke up the design to begin individual work.

F. Building Individual Modules

Each member was given complete freedom in performing his design. The only obvious restriction was that all interface signals must communicate correctly with the peripheral stages in accordance with the overall specification and the design must adhere to the functional and timing specifications. The whole CPU design team met several times each week to discuss the project progress as well as to "negotiate" the changes, if any, to be made to the specifications. The team leader monitored the changes to make sure that these changes did not adversely affect the overall design.

G. Testing the Low-Level Functionality

To reduce the extent of overall testing, we used an incremental testing approach. Each stage was built and tested thoroughly before it was integrated with other stages. Furthermore, interface tests were designed and testing was also performed after combining two neighboring stages to ensure that all designs met the interface specifications. This allowed us to isolate any potential design flaws in individual modules and to fix the problems quickly. It should be noted that there were many such flaws detected and fixed before testing the whole system.

H. Integrating Individual Modules

Once all the module design and testing was completed and the interface testing was completed between individual pairs, we began to put the whole processor unit together. At this

step, the master control unit, the forwarding unit, and the pipeline registers were tested thoroughly. To our surprise, it did not take us long to get the whole processor working. We can attribute this to the design of complete specification of individual modules, interface design, module testing, and interface testing performed earlier.

I. Testing Overall Functionality

Once the CPU integration was completed, we began overall testing of the CPU. This step was to test the functionality of the CPU to determine how well it met our design specifications. If bugs were found during this step, the individual who designed the faulty stage was responsible for fixing the problem and retesting his stage from the lowest level up. After fixing all the known bugs, we ran a test program on our machine as well as on some other simulated processors to do a benchmark comparison. For the bugs found during the benchmarking process, we repeated the same process of fixing them.

J. Documentation

The final stage of the design process was to organize and generate documentation for the processor. The documentation includes the following.

1) Overall functional specifications for the processor.
2) Detailed design specifications for each stage and each unit.
3) All the software simulation codes.
4) Hardware implementation diagrams.
5) Test programs and scripts that we used.

III. Project Management

In this project, we used the concepts of version control to streamline the design process. Version control is a concept that employs a systematic way to keep track of the project development history. It allows easy reference to any level in the project development, and provides a good way to isolate and debug the design.

Version control can be used or implemented in many different ways. In this project, we chose to implement version control using the time stamps method. The rules we used to enforce version control with time stamp were as follows.

1) Different versions of a file were distinguished by the time stamps attached to the filename.
2) Time stamps included only month and day in *.mm.dd* format.
3) The most current version of a file did not have a time stamp attached to its filename.
4) Only one copy of the most current version of any module could be modified at any time by anyone.
5) Tests were run by test scripts that utilized time stamp information.
6) Files were promoted to a new version when a particular goal was achieved. To promote a file, we would copy the most current file to a file with the present day attached to the filename as a time stamp. All the current test scripts were changed to reflect these changes. For example, if we were working on a file called *dispatch.v* and we decided to promote it to the next level on March 2nd, we would copy the file *dispatch.v* to *dispatch.v.03.02* and modify all the test script references of *dispatch.v* to *dispatch.v.03.02*.

We found that version control provided a lot of help to the design process. It was particularly useful for isolating the design flaws as well as facilitating repetitive testing. Combining version control and incremental testing, we estimate there was a minimum of 20% reduction in our debugging and development time.

IV. Application of Design and Work Concepts

In addition to the structured design approach and the version control used in project management as described above, other design and work concepts were also employed including enforcing coding and documentation standards, allowing flexible work hours, and using "release date" notation as described below.

A. Coding and Documentation Standards

Adapting coding and documentation standards provided a means for effective communications among the team members. It facilitated the design, because the functions and features of each module were clearly defined and gave the project a neat and well-organized structure.

B. Flexible Work Hours

One of the valuable contributions to the project from our the Co-Op experience of the design team members was adopting flexible work hours in the design schedule. Flexible working hours not only made project scheduling simpler, but also gave the team members a sense of trust which motivated them to work harder. The down side to this approach was that the team leader had to keep track of the progress of each member, a very time-consuming task.

C. Release Day

The other concept that we found very useful in a time constrained project was the use of the "Release Days." In our schedule, we defined target days for "Alpha" and "Beta" releases. The team was only allowed to change the architectural specifications before the "Alpha" release day. After the "Alpha" release day, only substantial changes could be made to the design. The time between the "Alpha" and "Beta" releases was used to develop the individual modules. All modules completed on or before the "Beta" release day were not allowed to change unless major flaws were found. This forced the uncompleted modules to design around the finished modules.

One problem we found in the design of a modern computer was that there are many ways to do one thing. Thus constant changes can be made to improve the design. These changes are good in general but they are definitely bad for a time-constrained project. Changes tend to throw the project off schedule. The use of the "release day" concept allowed the

team to have better control of time. However, it also made the final design somewhat less comprehensive due to time constraints.

V. Architecture and Design Details

In this section, we discuss some special architecture and design aspects of our CPU. Our CPU included the following specific architectural features:

1) Superscalar architecture, which allowed two independent instructions to be executed simultaneously.
2) Superpipeline architecture that consisted of nine pipeline stages; three external and six internal.
3) Separate Instruction and Data Caches.
4) Data Cache, which supported multiple accesses.
5) Modularized and scalable components, which allowed easy upgradability.

Each of these advanced architectural features required a great deal of effort in designing and implementing (simulating) and gave students a much deeper insight into and understanding of modern microprocessor architecture. The material presented in this section requires the reader to have some background knowledge about computer architecture. Please refer to the reference book for further readings [1].

A. Instruction Set

In this section, we discuss the chosen instruction set for the processor. An instruction set provides a complete description of the capabilities of the processor. It is the first thing a design team has to define. A good instruction set should provide programmers all operations readily available in one or a combination of a few instructions which are critical for software development. With that in mind, we have designed our enhanced instruction set. See Tables I–IV in Appendix II for the complete instruction set.

For our instruction set, we chose the instruction set of the MIPS R2000 as a model. The MIPS R2000 instruction set contains three different types of instructions: R-type (register), I-type (immediate), and J-type (jump). To enhance the versatility of the instruction set, we added an additional type: the E-type, for extended instructions. This type of instruction is just an extension of the R-type with the additional capability of direct access memory for one of the operands. The E-type instruction supports seven memory addressing modes and these modes can be used with either the destination operand of an instruction or one of the two source operands. The other source operand is always in a register in E-type instruction. All instructions of this type are no longer restricted to the 32-b scheme of the typical RISC architecture. They are extended to 64 b. The only change is in the opcode field of the corresponding R-type instruction of the instruction. The second 32 b of the instructions contain the necessary information to access the memory for one of the two source operands or the destination operand. Fig. 4 depicts the instruction formats. The enhanced instruction set provides 100% compatibility to existing MIPS R2000 programs, while offering increased versatility.

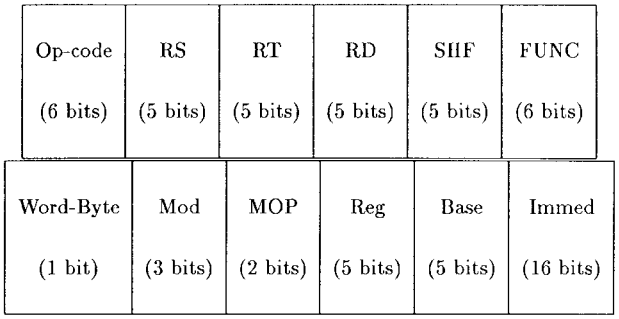

Fig. 4. Instruction format: Basic MIPS instruction format and format for the second word in the extended instruction set.

After generating the instruction set, we were able to generate the appropriate datapath and control for the CPU as described in the following section.

VI. Building Blocks (internal Stages) of the CPU

We partitioned the entire hardware for pipeline implementation into several building blocks. Each building block is roughly a stage in the pipeline or some sort of control unit. In this section, we describe the internal stages and peripheral units for the processor.

Instruction Fetch Unit (IFU): The instruction fetch unit consists of a program counter (PC), a 64-b two-word buffer to hold two instructions coming from the instruction cache every clock cycle, a 32-word instruction queue which gets the accepted instructions from the two-word buffer, and a 256-b instruction buffer which provides four instructions to the dispatch unit every clock cycle. It is superpipelined into the following three stages.

1) A program counter generation unit to generate address for instruction fetch from memory. It sends the PC to the instruction cache to read instructions and to the branch prediction unit to check if a branch history/condition exists in branch prediction table.
2) The second stage is for data decoding and storage. It latches the 64-b information coming back from the instruction cache. It then decodes the instructions and extracts the relevant information and separates 32-b and 64-b instructions. This step is necessary because our processors supports multiword instructions. The decoded instructions are stored on a 32-word instruction queue.
3) Instruction fetching and buffer management is performed at the third stage. This unit is called the instruction prefetch unit (IPF). It feeds the dispatch unit with four executable instructions in the extended format (256 b) in every clock cycle.

The instruction fetch unit runs independently from the CPU core pipeline to provide maximum performance.

Branch Prediction Unit (BPU): The branch prediction unit (BPU) which works in parallel with the IFU, consists of a 32-entry lookup table for the address and the target address of the most recent branch taken at that address.

When an address is generated by the IFU for the instruction cache, this address is also forwarded to the BPU. The branch prediction unit then performs a 32-entry lookup in its table.

If the address is in the table, then it returns the target address if the probability bit associated with the entry has been set by previous branch executions. The IFU then fetches the instructions starting at the addressed location.

Branch prediction has proven to effectively decrease the number of flushes required by the processor while executing any program, and thus improves the performance.

Dispatch Unit (DU): The dispatch unit selects two independent instructions from the four instructions fed to it by the IPF to be executed simultaneously in the two pipelines. The dispatch unit is the key component to superscalar architecture. In selecting the two independent instructions the dispatch unit must check data dependencies among the four instructions. Two independent subunits are responsible for doing this. The first unit compares the second, third, and fourth instructions with the first instruction. The second unit checks data dependencies among instruction two, three, and four to find whether they are absolutely independent of one another or not. The dependency codes generated by these two units allow the dispatch unit to choose the best two independent instructions to be executed. In any case, the first instruction is always executed. If there is no second independent instruction available, then the second pipeline is fed with a no-operation instruction.

Control Unit (CU): The main purpose of the control unit is to generate the appropriate control signals for all the pipeline stages. These signals are used to control the multiplexors for selecting the correct data or function to be executed in each pipeline stage.

Register File (Reg-File): The register file contains the standard 32 registers of the MIPS R2000. To prevent possible data conflicts, we implemented the Write-before-Read and the priority write algorithms in the register file. The Write-before-Read algorithm ensures that the most up-to-date information is read from the register file every time. The priority write algorithm handles data collisions which could result from multiple write requests. This register file allows for eight register reads and four register writes every clock cycle.

Address Generation Unit (AGU): This stage of the pipeline is designed for the E-type instruction. Here, the operand address used by the current E-type instruction is generated. This stage contains two additional adders in additions to ones located in the execution stage. These two adders compute the desired address and send it to the data cache.

Operand Fetch Unit (OFU): The operand fetch stage interfaces with the data cache to provide data to the instructions being executed. Once the data are ready, they are latched into this unit and pipelined to the execution stage. However, if a data cache miss occurs, a CPU stall signal is generated to the master control unit until the data are returned from the data cache.

Execution Unit (EU): The execution unit contains a 32-b ALU and a 32-b barrel shifter. The 32-b ALU provides the following logical operations to the two 32-b inputs: AND, OR, ADD, SUB, SLT, BEQ. There is only one 32-b shifter between the two execution units which can perform either a right shift or a left shift to one of the two 32-b inputs. Only one of the two ALU can be use the shifter at any time.

Write-Back Unit (WB): The function of the write-back unit is to write the results of the two execution units back to the appropriate memory or register locations.

Hazard Detection and Forwarding Unit (HDFU): The forwarding unit provides data hazard detection and forwarding functions for the CPU. Because of the pipeline nature of the processor, data hazards can occur between different stages of the pipeline. One of the two functions of the forwarding unit is to detect the occurrence of these hazards. The second function is to provide forwarding signals to correct the data hazards. In our CPU design, there are 66 forwarding signals to check eight different possible hazard spots. Two stall signals are also generated to stall the pipeline operation in the event of a hazard which cannot be resolved without stalling.

Master Control Unit (MCU): The master control unit provides system level control signals to the pipeline registers, the instruction fetch unit, and the branch prediction unit. It also handles system-level errors and initial/reset conditions.

Instruction and Data Cache: In our design, we have two separate caches for instruction and data as described below.

Instruction Cache (IC): The code cache is made up of 8-kB static memory, and it uses a four-way set-associative architecture. Each set is 2 kB which consists of 64 entries and each entry is 256 b in length. A Tag, a Valid bit, 2 LRU bits, and a Modify bit are associated with each entry. The data bus is 64-b wide. Normal read access time for the code cache is 8 ns and the memory access is handled by the cache controller and the bus interface unit.

Data Cache (DC): The data cache is also a four-way set-associative cache, but it allows a total of four simultaneous accesses (two reads and two writes) to the cache. This is achieved by allowing multiple decoding to happen at the same time. The read is always done before the write. There is no hazard detection or forwarding done in the data cache. All of these are handled by the cache controller, which also contains two write buffers. These buffers are used for writing data back to the memory in case of a write-back operation when a dirty cache line is replaced or some data are to be directly written to the main memory.

Bus Interface Unit (BIU): The function of the bus interface unit (BIU) is to provide control for the bus access. It also arbitrates between different caches if more than one need to access the main memory.

Main Memory: A memory module was developed to completely perform the testing of the new CPU. This memory is assumed to have a two-clock-cycle access time. It also supports normal burst-mode access in which the first read takes two clock cycles to access while subsequent reads in the same block take only one clock cycle. The memory is byte-addressable and supports transfers of 8-b (byte), 16-b (half-word), 32-b (word), and 64-b (double word). Only the bus interface unit is directly connected to this test memory module.

VII. Results

We have benchmarked our developmental CPU with a simple single five-stage pipeline RISC-based CPU without Branch

Prediction as described in [1]. The benchmarking program was a factorial computation program written in regular MIPS R2000 instructions. The program consisted of a main loop and a subroutine to perform multiplication. We benchmarked three CPU's—the basic pipeline of [1] implementing the MIPS R2000 instruction set, the developmental CPU with the Branch Prediction feature, and the developmental CPU without the Branch Prediction feature. See Appendix III for details of the program and Appendix IV for detailed time calculation. From the test results, we concluded that the branch prediction feature provides approximately 35% improvement in execution time, while the superscalar architecture provides only a modest 4% improvement in execution time. From the above results, it can be seen that the advanced features help in improving the performance a lot. In particular, the branch prediction is very effective for this program. In general, any program with a loop will benefit from the branch prediction feature. Although the results demonstrated that the branch prediction feature had a bigger performance improvement than the superscalar architecture, we believe that the superscalar architecture would excel in a larger test program.

In this project, we learned that both branch prediction and superscalar architecture features are valuable in modern processor design. The effectiveness of E-type instructions is yet to be judged but for architectures such as the Intel x86 series and Motorola 68040, such instructions are required for compatibility reasons.

VIII. Conclusions

A senior project in designing a modern Central Processing Unit has been described. The project used many modern design methods and concepts such as the systematic design process and the version control. The project familiarized the students with a modern design approach and effective project management methods. The project also provided a cooperative learning environment and leadership training. The students involved in the project strongly believe that it is a good model for other senior level design projects.

Appendix I
Project Schedule

The project schedule for the project was as follows.

1st week:	Define the basic CPU function blocks and instruction set. Set up the team, and talk about coding standard.
2nd–3rd weeks:	Each member starts working on his part and starts to determine the requirements for his part. Revise the overall architecture.
4th week: ("Alpha" Release)	The purpose of the "Alpha" Release is to have each individual part of the CPU working by this time. Each person should be testing his part.
5th–7th weeks:	Start putting the whole CPU together and debug. Also start to develop the testing program.
8th week: ("Beta" Release)	The "Beta" Release is to have most of the CPU working together. Final adjustments and changes will be made at this time.
9th week:	Work on the final testing and the final version of the CPU.
10th week: (Final Release)	At the end of the 10th week we should have the final version of a working CPU. Testing is done during this week to determine the performance.

Documentation is done along the way, and any time remaining after the final release will be devoted to documentation.

Appendix II

The instruction set includes the following instructions:.

TABLE I
R-Type Instructions for the Processor

Inst	op(6)	rs(5)	rt(5)	rd(5)	shf(5)	func(6)		Example
Add	000000	00010	00011	00001	00000	100000	Add	$1,$2,$3
Addu	000000	00010	00011	00001	00000	100001	Addu	$1,$2,$3
Sub	000000	00010	00011	00001	00000	100010	Sub	$1,$2,$3
Subu	000000	00010	00011	00001	00000	100011	Subu	$1,$2,$3
And	000000	00010	00011	00001	00000	100100	And	$1,$2,$3
Or	000000	00010	00011	00001	00000	100101	Or	$1,$2,$3
Xor	000000	00010	00011	00001	00000	100110	Xor	$1,$2,$3
Slt	000000	00010	00011	00001	00000	101010	Slt	$1,$2,$3
Sltu	000000	00010	00011	00001	00000	101011	Sltu	$1,$2,$3
Sll	000000	00010	00011	00001	00000	000000	Sll	$1,$2,$3
Srl	000000	00010	00011	00001	00000	000010	Srl	$1,$2,$3

TABLE II
I-Type Instructions for the Processor

Inst	op(6)	rs(5)	rt(5)	I(16)		Example
Addi	001000	00010	00001	0x003f	Addi	$1,$2,63
Addiu	001001	00010	00001	0x003f	Addiu	$1,$2,63
Andi	001100	00010	00001	0x003f	Andi	$1,$2,63
Ori	001101	00010	00001	0x003f	Ori	$1,$2,63
Xori	001110	00010	00001	0x003f	Xori	$1,$2,63
Slti	001010	00010	00001	0x003f	Slti	$1,$2,63
Sltiu	001011	00010	00001	0x003f	Sltiu	$1,$2,63
Lw	100011	00010	00001	0x003f	Lw	$1,63($2)
Lwi	001111	00000	00001	0x003f	Lwi	$1,63
Lb	100000	00010	00001	0x003f	Lb	$1,63($2)
Lbu	100100	00010	00001	0x003f	Lbu	$1,63($2)
Sw	101011	00010	00001	0x003f	Sw	$1,63($2)
Sb	101000	00010	00001	0x003f	Sb	$1,63($2)
Beq	000100	00010	00001	0x003f	Beq	$1,$2,63
Bne	000101	00010	00001	0x003f	Bne	$1,$2,63

TABLE III
J-Type Instructions for the Processor

Inst	op(6)	I(26)	Example	Note
j	000010	0x000003f	j 63	
jal	000011	0x000003f	jal 63	
jr	000000	0x3e00000	jr $31	This is really an R-Type

In addition to the above, our processor also includes a corresponding memory-reference instruction (E-type) for all instructions of R-type. In an E-type instruction, the first word

TABLE IV
E-TYPE INSTRUCTIONS FOR THE PROCESSOR

Inst	w/b(1)	mode(3)	mop(2)	reg(5)	regb(5)	I(16)	Example	
Add	0	rbc	10	00011	00100	vv	Add	$1,$2,vv($3+$4)
Addu	0	rbc	01	00001	00100	vv	Addu	vv($1+$4),$2,$3
Sub	0	rbc	10	00011	00100	vv	Sub	$1,$2,vv($3+$4)
Subu	0	rbc	01	00001	00100	vv	Subu	vv($1+$4),$2,$3
And	0	rbc	10	00011	00100	vv	And	$1,$2,vv($3+$4)
And	0	rbc	01	00001	00100	vv	And	vv($1+$4),$2,$3
Or	0	rbc	10	00011	00100	vv	Or	$1,$2,vv($3+$4)
Xor	0	rbc	01	00001	00100	vv	Xor	vv($1+$4),$2,$3
Slt	0	rbc	10	00011	00100	vv	Slt	$1,$2,vv($3+$4)
Sltu	0	rbc	01	00001	00100	vv	Sltu	vv($1+$4),$2,$3
Sll	0	rbc	10	00011	00100	vv	Sll	$1,$2,vv($3+$4)
Srl	0	rbc	01	00001	00100	vv	Srl	vv($1+$4),$2,$3

is exactly the same as what it is in the corresponding R-type instruction except that the op(6) field is 111111 instead of 000000. The E-type instruction carries a memory address for one operand (either destination or one source operand). The field, mop (2 b) decides if the memory operand is a destination or a source operand. The memory address is specified by a combination of the contents of a register (reg field), a base register (regb field), and a constant (I filed). Any one or more of these fields may not be used. Mode field (mode) includes 3 b to specify which one of these components in the address will be used. Depending on the value of rcb bits, the register, base register, and constant parts are used. The register can be any one of the 32 registers.

APPENDIX III
BENCHMARKING PROGRAM AND CALCULATION.

The following program was written to benchmark our developmental CPU against other modern CPU's running MIPS R2000 instructions. The program was intended to provide a comparison between CPU's with different features. In this test, we were particularly interested to see how well Branch Prediction works and how much performance improvement it provides. Our secondary interest was to see how much the superscalar architecture improves the CPU performance.

Three sets of results are presented here. The first is the execution time required by a simple five-stage single pipelined RISC-based CPU. This CPU is modeled after the MIPS R2000 in [1]. The second set of results was taken from the simulation of the developmental CPU with Branch Prediction feature turned on. The last set of results came from the simulation of the developmental CPU with the Branch Prediction feature turned off.

```
// Factorial Program:
// addi $6, $0, 2
001000_00000_00110_00000_00000_000010
// addi $1, $0, 7//Find the factorial of 7.
001000_00000_00001_00000_00000_000111
// add $2, $0, $1//Set up the counter.
000000_00000_00001_00010_00000_100000
// addi $3, $0, 1//Set $3=1. Useful for decrement.
001000_00000_00011_00000_00000_000001
// sub $2, $2, $3//Decrement counter.
000000_00010_00011_00010_00000_100010
// jal 28//Multiply function.
000011_00000_00000_00000_00000_001000
// bne $2, $6, (12. or -2) //Loop if $2 not equal to 1.
000101_00010_00110_11111_11111_110100 // j 52
00010_00000_00000_00000_00000_011000
// add $4, $0, $2//Set up separate counter.
000000_00000_00010_00100_00000_100000
// add $5, $0, $1//Set up common factor.
000000_00000_00001_00101_00000_100000
// add $1, $1, $5//Increment $1 by common factor
(here=5, nextloop=20)
000000_00001_00101_00001_00000_100000
// sub $4, $4, $3//Decrement counter.
000000_00100_00011_00100_00000_100010
// bne $4, $3, (36. or -2) //Loop until counter=1.
000101_00100_00011_11111_11111_110100
// jr $31//Return from procedure.
000000_11111_00000_00000_00000_001000
// sw $1,15($0)//Save the result into memory location 15.
101011_00000_00001_00000_01000_000000
// add $0, $0, $0//Increment $1 by common factor
(here=5, nextloop=20)
000000_00000_00000_00000_00000_100000
000000_00000_00000_00000_00000_100000
000000_00000_00000_00000_00000_100000
// Jump back and loop forever.
00010_11111_11111_11111_11111_111000
000000_00000_00000_00000_00000_100000
```

APPENDIX IV
TIMING CALCULATIONS

Set 1: Execution Time for a simple five stages single pipeline RISC based CPU.

CPU Time:

Setup time: 4 clock cycles.
The jump instructions assume a 4 clock cycles execution time. The branch instructions assume a 4 clock cycles execution time.

Write back: $(4+1) = 5$ clock cycles.

Subroutine: Called five times, each time with a decreasing number of loops to execute. Total of 20 loops. Each loop consists of four regular instructions and one branch instruction. The execution time is $(4 + 4) = 8$ clock cycles per loop. Each subroutine call requires a JR instruction for return which takes four clock cycles. Total subroutine execution time $=$ $20 \times 8 + 5 \times 4 = 180$ clock cycles.

Cache Time:

Instructions loadtime 24 instructions $= 3 \times 7$ clock cycles $= 21$ clock cycles.

Cache Miss penalty 3×1 clock cycles $= 3$ clock cycles.

Total execution time: To compute seven factorial from computation is 259 clock cycles.

Set 2: Execution Time for the developmental CPU with Branch Prediction turned on.

Total execution time: To compute seven factorial from measurement is 170 clock cycles.

Set 3: Execution Time for the developmental CPU with Branch Prediction turned off.

Total execution time: To compute seven factorial from measurement is 253 clock cycles.

ACKNOWLEDGMENT

The authors wish to thank the two reviewers who made extensive comments and suggestions to improve the quality of presentation.

REFERENCES

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design—The Hardware/Software Interface.* Los Altos, CA: Morgan Kaufmann, 1994.

[2] D. Alpert and D. Avnon, "Architecture of the Pentium Microprocessor," *IEEE Micro*, vol. 13, pp. 11–21, June 1993.

[3] E. McLellan, "The Alpha AXP architecture and 21064 processor," *IEEE Micro*, vol. 13, pp. 36–47, June 1993.

[4] *MIPS R-Series Architecture*, MIPS Computer Systems, Inc., 930 Arques Ave., Sunnyvale, CA 94086, Sept. 1990.

[5] *MC68040 32-Bit Microprocessor User's Manual.* Motorola Inc., 1989.

[6] T. Asprey, G. S. Averill, E. DeLano, R. Mason, B. Weiner, and J. Yetter, "Performance features of the PA7100 microprocessor," *IEEE Micro*, vol. 13, pp. 22–35, June 1993.

[7] M. C. Becker, M. S. Allen, C. R. Moore, J. S. Muhich, and D. P. Tuttle, "The PowerPC 601 microprocessor," *IEEE Micro*, vol. 13, pp. 54–67, Oct. 1993.

[8] *Intel P6 Architecture Manual.* Intel Corp., Santa Clara, CA, 1995.

[9] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," in *ASPLOS-III Proc. 3rd Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (IEEE, ACM, Boston, MA, Apr. 3–6, 1989), pp. 272–282.

Victor Lee received the B.S.E.E. degree in electrical engineering in 1994 from the University of Washington, Seattle, WA.

His research interests are in the area of computer architecture and parallel computer systems. He is currently pursuing the M.S.E.E. degree in computer engineering.

Nghia Lam received the B.S.E.E. degree in electrical engineering in 1994 from the University of Washington, Seattle, WA.

His research interests are in the area of computer engineering and signal processing.

Feng Xiao received the B.S.E.E. degree in electrical engineering in 1994 from the University of Washington, Seattle, WA.

His research interests are in the area of computer engineering and signal processing.

Arun K. Somani (S'83–M'85–SM'88) received the M.S.E.E. and Ph.D. degrees in electrical engineering from the McGill University, Montreal, Que., Canada, in 1983 and 1985, respectively.

He worked as Scientific Officer for the Government of India, New Delhi, from 1974 to 1982. During this period he designed and developed an anti-submarine warfare system for the Indian Navy. He is an Associate Professor of Electrical Engineering and Computer Science and Engineering at the University of Washington, Seattle. His research interests are in the area of fault-tolerant computing, interconnection networks, computer architecture, parallel computer systems, and parallel algorithms. Currently, he is involved in three major projects: i) high-integrity system design addressing the issues related to cache memory design in redundant computer systems and evaluation tools for such systems; ii) congestion control and fault tolerance in broadband networks; and iii) development of "Proteus" architecture, a multiprocessor system for automated classification of objects based on generalized enhanced hypercube reconfigurable interconnection network exploring coarse grain parallelism.