A Two-tiered Modeling Framework for Undergraduate Computer Architecture Courses

Jason Loew
Department of Computer Science
State University of New York at Binghamton

Dmitry Ponomarev
Department of Computer Science
State University of New York at Binghamton

Abstract

We describe a new methodology for modelling key microarchitectural features in an advanced undergraduate computer architecture course and demonstrate the specific application of this methodology to branch predictors. The proposed approach consists of two separate, but synergistic programming assignments. In the first part, students implement the branch prediction logic as an independent software module, but the interfaces are defined in such a way that the developed code can be easily integrated into a cycle-accurate simulator. In the second part, such integration into the M-Sim simulator takes place. Such decoupling allows the students to focus on the features of the branch predictor in isolation from rather complex simulator code. In addition, the two-stage process also better aligns itself with the class schedule, because the students are only exposed to the simulator code at the end of the semester, after they learned most of the key design concepts that are supported in the simulator.

In this paper, we present the details of both assignments and also describe the modifications introduced to the M-Sim simulator to support such modelling capabilities. All assignments and the modified simulator are available online and the framework has already been used in the undergraduate computer architecture course at SUNY Binghamton. Finally, the proposed framework can be easily extended for modelling other key architectural paradigms, such as register renaming and caches.

1 Introduction and Motivation

Several key microarchitectural concepts used in modern processor design (branch prediction, register renaming, cache memories) are fairly easy to understand conceptually, but the real appreciation of the internal operations of these mechanisms can only be achieved if one tries to model and implement these subsystems using various simulation and design automation tools. This is why it is extremely important to augment the theoretical underpinnings with a solid experimentation methodology to provide students with sufficient hands-on experience. The key challenge for using such modelling in the undergraduate courses is that the students often do not have sufficient expertise and time to use the tools typically designed for research. In this paper, we introduce a novel way to approach this problem and we illustrate our approach using dynamic branch prediction logic as an example of a subsystem to be modelled.

Dynamic branch prediction is among the most fundamental concepts in modern processor design. It is also an integral part of any advanced undergraduate computer architecture course. While the fundamental concepts and ideas of basic dynamic branch prediction logic are straightforward and can be fairly easily explained to an undergraduate class using a few powerpoint slides, the main challenges lie in supporting these basic concepts with adequate hands-on exercises that would allow the students to reinforce the theoretical fundamentals. This reinforcement is especially important in the field of computer architecture, because the real performance impact of any design can only be gauged by observing the processor's behavior (with the hardware additions being evaluated) on a set of standard benchmarking programs. The easiest way to accomplish this is by using cycle-accurate simulators.

Cycle-accurate simulations represent the primary vehicle of early-stage evaluations of key architectural ideas, both in academia and in the industry. A large number of open-source simulators, some of them specifically designed for the use in academia, are widely available [1, 3, 6, 8, 7]. Unfortunately, most of these simulators are designed for research activities and are of limited use in education, especially at the undergraduate level. The main reason is that it takes a significant amount of time and familiarity with the discipline to understand how these simulators work, and in some cases even simply to be able to run them. For example, the simulators almost never employ "execute-at-execute" model, where instructions would actually be executed out-of-order, as they would on a real hardware. In-

stead, "execute-at-decode" model is used, where the instructions are actually executed at the same time they are decoded (in-order), and the out-of-order effects are modelled through rather complex manipulations with auxiliary structures, such as the Register Update Unit (RUU) used in Simplescalar simulator [1]. While such tools are certainly suitable for advanced PhD students involved in research projects (who can afford the learning curve involved in using the simulators), and even for some simple projects in graduate architecture courses, they are too complex for an average undergraduate student. Indeed, it is unreasonable to spend a significant amount of time learning about the simulator's intricacies just to carry out one or two assignments. Even though the branch prediction code is usually somewhat isolated from the rest of the simulator code, the interface is still rather non-trivial for the undergraduate students and the students still need to understand large chunks of highly-optimized C code to complete even the simplest assignments. In summary, the main problem stems from the absence of an intermediate step, where the students could easily understand the branch predictor interface and internal organization and functionality, and then seamlessly incorporate it into the full-fledged simulator. In this paper, we describe such a two-tiered framework, which we successfully integrated into the CS325 (Advanced Computer Architecture) course at SUNY Binghamton.

Our proposed framework for supporting experiments with dynamic branch predictors in undergraduate computer architecture course consists of two assignments (both of which are presented in detail in the subsequent sections). The first part is a simple programming assignment that provides the students with all the interfaces to the dynamic branch predictor, including the input trace of branch instruction outcomes to be fed into the predictor. Students are asked to fill in the body of the functions (bpred_update and bpred_lookup) to implement a particular prediction mechanism (such as gshare [4]). Evaluation is performed using a trace of supplied branch instruction PC addresses and corresponding branch outcomes. This is implemented as a standalone piece. The second part of our framework is the integration of this code into a modified M-Sim simulator [3]. M-Sim is a redesigned version of Simplescalar simulator, that supports simulation of multithreaded and multicore architectures, and also explicitly models register renaming, load-hit speculation, replays and a variety of other features. For this assignment, the branch prediction implementation of M-Sim has been completely rewritten to seamlessly support the interface provided to the students in the first assignment. In essence, the students can simply "drop" the code that they developed during the first assignment into the modified M-Sim, and end up with the branch predictor implementation that can be driven by the actual outcomes of the branches executed within SPEC programs. The main benefit of the proposed approach is that it allows the students to abstract the details of the simulator and focus instead on the logic pertaining to the operation of the branch predictor. The modified M-Sim code supporting this framework is available at the following URL: **http://www.cs.binghamton.edu/∼msim/branch**

Another advantage of using such a multi-step approach is that it aligns more naturally with the course schedule in terms of giving the simulator code to the students only at the time when most of the advanced concepts implemented in the simulator are already covered during the lectures. For example, consider a typical Fall semester schedule for an undergraduate architecture course taught based on Patterson and Hennessy's book [5]. Typically, the first several weeks of the course are spent on the topics such as performance metrics, ISA design, advanced arithmetic and non-pipelined datapath and control logic design. Pipelined execution, forwarding and branch prediction are the first advanced topics covered, and this usually happens sometime around midway point of the semester (mid-October). At this time, it is too early to introduce the students to the simulator, because most of the advanced concepts implemented in the simulator (register renaming, out-of-order execution, cache hierarchies) are not yet explained during the lectures. In this case, the students would have a double burden of trying to implement a new branch prediction design in a simulator that supports extensive functionality, much of it they do not yet understand. In contrast, if a simple self-contained programming assignment (as detailed in Section 2) is given at this point, the students would already know enough information to implement the prediction and update logic without having to get concerned with the details of the rest of the pipeline stages. If this assignment takes 2-3 weeks to complete, then sometime around the middle of November, the students can be introduced to the simulator to complete the second part of the branch prediction modelling assignment. By that time, all the key concepts implemented within the simulator would already be covered and it would be much easier for students to work with the simulator.

The rest of the paper is organized as follows. Section 2 describes the first programming assignment for modelling branch prediction logic without the use of full-fledged simulators. Section 3 provides a short overview of the M-Sim-3.0 simulator. Section 4 describes the second assignment, where the code designed in the first assignment is seamlessly integrated into M-Sim simulator. Finally, we conclude in Section 5.

2 First Programming Assignment

The first portion of the modelling framework, described in this section, encompasses all of the programming work, but allows the branch predictor code to be implemented out-

side of the simulator and in the form of a standalone programming assignment. In this assignment, which can be given out right after the branch prediction fundamentals are covered in class, the students are given the base class branch predictor code and a set of other files that are required for implementation and testing. The students are also provided with examples in the form of the two stateless predictors - "always taken" and "always not taken". A driver (implemented through main.c file) feeds the trace file to the predictors and collects the prediction results. In the following subsections, we describe the details of the trace file and the interfaces that the students are asked to work with. The goal of this assignment is to implement a specific prediction logic in the framework of the defined interfaces.

2.1 A Generic Branch Predictor Model

A generic dynamic branch predictor model would have the following components:

- Counters: Various counters keep track of the statistics of the predictor. Students will generally need to keep track of the number of lookups and hits/misses.
- reset(): Resets the counters after fast-forwarding. Students do not need to implement this unless they add their own counters.
- retstack: The return address stack. If left unimplemented, it does nothing.
- Constructors: These create the branch predictor. Students should generally see an example of how these work.
- bpred_lookup: The lookup abstraction that students are taught.
- bpred_update: The update abstraction that students are taught.
- bpred_reg_stats: Adds the counters to the simulator's statistics database. Can be ignored unless the students need to track their own counters.

2.2 Trace File Format

The trace file contains the set of branch predictor lookup requests and the set of branch predictor update requests (generated based on the actual branch outcomes during program execution) that are generated by a hypothetical program. The format of each of these requests is shown in the table below. For this assignment, we ignore the issues associated with managing return address stack.

- Lookup: *lookup* <*branch_PC*> <*branch_target if known*> <*opcode*> <*is_call*> <*is_return*>
- Update: *update* <*branch_PC*> <*branch_target*> <*opcode*>

The contents of this trace file are used as an input to the predictor that the students are asked to design.

The easiest way to generate realistic trace files is to produce them from M-Sim directly (although, it is also possible to create them in some other manner). This can be accomplished by adding code before bpred_lookup and bpred_update are called. For bpred_lookup, we output "lookup" followed by the *branch address* and *branch target address* (both in hex, prefixed with 0x) then we output the *opcode* and the flags *is_call* and *is_return*. These can be copied straight out of the call to bpred_lookup (or done once in bpred_lookup). For bpred_update, we output "update" followed by the *branch address* and *branch target address* (both in hex, prefixed with 0x) then we output the *opcode*. The driver provided in main.c file determines the values of *taken*, *pred_taken* and *correct* and supplies those to the predictors.

Students generally find it difficult to craft the code to test their implementations. When using trace files, a "boilerplate" is provided that includes:

- A makefile that compiles all of the required files.
- A driver (such as main.cpp) that runs their code.

2.3 Data Structures and Predictor Interfaces

2.3.1 Data Structures

Appropriate data structures need to be designed to represent the predictor state. This can be managed in any reasonable way within the constructs of the class. Data structures can be created within the class constructor which must call the base constructor with the predictor's name. This is shown at the top of Figure 1 using a standard initializer list approach. Any size requirements (such as power of 2) should be enforced here. Initialization of the predictor with some starting values is also done here. A destructor is only needed if the students use their own memory management. It is suitable to allow them the use of existing STL structures such as vector.

2.3.2 The Lookup Interface

Branch predictor lookup procedure takes the address of the branch instruction (md_addr_t *baddr*) and the target address (md_addr_t *btarget*) in order to make a prediction. Students must additionally handle *dir_update_ptr* which is cleared

during lookup and has a pointer (*pdir1*) that points to the entry used by the branch predictor. The other arguments (*op, is_call, is_return, stack_recover_idx*) can be ignored by students at the discretion of the instructor. The number of lookups is maintained here.

2.3.3 The Update Interface

Branch predictor update procedure takes the address of the branch instruction (md_addr_t *baddr*), the target address (md_addr_t *btarget*) and dir_update_ptr (*dir_update_ptr->pdir1*) to update the branch predictor state. The result of the branch comes from the remaining parameters (*op* can be ignored). *taken* is a boolean that tells us what the branch actually did. *pred_taken* tells us the prediction that was made. *correct* indicates if address prediction was correct (this is optional for students). Misses/hits and most other performance statistics are maintained here.

2.4 Statistics

Each branch predictor maintains statistics related to its own performance and these stats are printed to the screen at the end of simulation. The relevant statistical counters are contained in the parent branch predictor class, which registers the statistics with the database, identical to the one maintained in the M-Sim simulator. The database was included explicitly as part of this assignment in order to make future transition to M-Sim (as described in Section 4) easier. The statistics handling is already built in and students can simply increment/adjust the counters that are provided by the parent class. Students can add their own statistics, but this is neither required, nor necessary for this assignment.

2.5 Examples for Students

Students will often find it helpful to have an example to start with. The stateless predictors (always taken, never taken) can be provided to students for this purpose. The stateless predictors show how minimal the interface can be and provide an example of how to use the existing components that are inherited from the base branch predictor class. See Figure 1.

2.6 The Assignment

The complete assignment package, including trace files, is available at the following URL: **http://www.cs.binghamton.edu/∼msim/branch**

Students need to download the code package provided by the instructor, unpack the code and execute the ”make” command. The code is executed as follows: ”./main.exe gcc.trace” where gcc.trace can be replaced with any other

```
#include"bpred_taken.h"
#include<cassert>

bpred_bpred_taken::bpred_bpred_taken()
:
bpred_t("taken")
{}

md_addr_t
bpred_bpred_taken::bpred_lookup(
        md_addr_t,
        md_addr_t btarget,                  //branch target, if taken
        md_opcode op,                       //opcode of instruction
        bool, bool,
        bpred_update_t *dir_update_ptr,     //pred state pointer
        int*)
{
        assert(dir_update_ptr);

        if(!(MD_OP_FLAGS(op) & F_CTRL))
        {
            return 0;                       //If not a control inst
        }
        lookups++;
        dir_update_ptr->dir.ras = false;    //Clear dir_update_ptr
        dir_update_ptr->pdir1 = NULL;
        dir_update_ptr->pdir2 = NULL;
        dir_update_ptr->pmeta = NULL;

        return btarget;
}

void
bpred_bpred_taken::bpred_update(
        md_addr_t,
        md_addr_t,
        bool taken,                         //non-zero if branch was taken
        bool pred_taken,                    //non-zero if branch was pred taken
        bool correct,                       //was earlier prediction correct?
        md_opcode op,                       //opcode of instruction
        bpred_update_t *dir_update_ptr)     //pred state pointer
{
        if(!(MD_OP_FLAGS(op) & F_CTRL))
        {
            return;                         //If not a control inst
        }

        addr_hits += correct;               //Update stats
        dir_hits += (pred_taken == taken);
        misses += (pred_taken != taken);

        if(dir_update_ptr->dir.ras)         //If return address stack used
        {
            used_ras++;
            ras_hits += correct;
        }

        if(MD_IS_INDIR(op))                 //If indirect jump
        {
            jr_seen++;
            jr_hits += correct;

            if(!dir_update_ptr->dir.ras)
            {
                jr_non_ras_seen++;
                jr_non_ras_hits += correct;
            }
            else
            {
                //used return address stack, done
                return;
            }
        }
}
```

Figure 1. Always Taken Predictor

.trace file. The first time this is executed, the program will run and provide results for the "always taken" branch predictor. The students need to provide their own <GIVEN TYPE> predictor in the form of the files *student.c* and *student.h*. These should be modelled after the predictors provided (the skeleton provided in the package can be given to students). No other files need to be modified. Figure 2 shows the complete formulation of the assignment, as used in CS325 at SUNY Binghamton in the Fall 2009 semester.

The next step in the modelling framework is the seamless integration of the code developed in this assignment into M-Sim simulator. Before showing the details of that assignment, we briefly outline the key features of M-Sim simulator itself, show how it is different from the Simplescalar simulator (from which it was derived), and explain why it is a suitable tool for performing modelling of architectural components in a manner described in this paper.

3 Overview of M-Sim Simulator

M-Sim [3] is a multi-threaded and multi-core extension to Simplescalar 3.0d simulator [1] that provides explicit modeling of various datapath components such as the issue queue, register file and reorder buffer. Version 3.0 of M-Sim is a significant re-write of the prior code using built-in data structures and algorithms where possible and restructuring the code to provide some encapsulation of the various architectural abstractions that are simulated. This encapsulation and relative isolation of various subsystems makes it easier to experiment with one given subsystem (such as the branch prediction logic), while largely abstracting out the details of other activities within the processor's datapath. While M-Sim (just like most other simulators) has been originally designed for research purposes, a separate version of it has been created for performing assignments, such as the one described in this paper, in undergraduate courses. The most recent research release of M-Sim is available here: **http://www.cs.binghamton.edu/∼msim/**. The modified code suitable for supporting this assignment is available here: **http://www.cs.binghamton.edu/∼msim/branch**. In the next section, we describe the mechanics for integrating the code developed in the assignment described in Section 2 into M-Sim simulator.

4 Step 2: Integrating the Code into M-Sim Simulator

The second portion of the assignment involves incorporating the selected files developed during the first assignment (described in Section 2) into a complete simulator package. To support such a capability, we provide a slightly modified version of M-Sim simulator that allows the student code to be copied in directly. Specifically, all branch predictors are declared in bpreds.h file. Conditional compilation (using the include guards for the predictors) show where the predictors need to be added in sim-outorder.c file so they can be used during simulation. The new predictor can be added to the Makefile using the existing branch predictors as examples. Of course, all of this can be provided in the simulation framework that is given to the students. (See Figure 2).

4.1 Implementation

The following four steps are needed to implement a seamless migration of the developed code into the simulator:

- The student.h and student.c files need to be copied into the same folder as the simulator.
- The Makefile needs to be modified to include these files in compilation.
- The file bpreds.h now needs to include student.h in order to add the student predictor to the set of possible predictors.
- The sim-outorder.c file needs to be augmented with the code to allow the student predictor to be used in the command line with "-bpred student".

4.2 Testing

Execution is slightly different than in the first part of the assignment since it actually implements a full-fledged simulation and provides realistic dynamically generated streams of predictor lookup requests and predictor update requests. Students can now test their implementations in the full simulation environment and see the overall performance using programs such as the SPEC benchmark suites [2].

In order to run the simulator, students have to execute the following command from the benchmark directory (provided in the simulator tarball): "../sim-outorder -fastfwd 1000000 -max:inst 1000000 -bpred student gccNS.1.arg". *-fastfwd* and *-max:inst* options determine the number of instructions to skip and the number of instructions to execute, respectively. *-bpred option* determines which branch predictor is used: "student" is for the student predictor, "taken", "nottaken", "bimod", "2lev" and "comb" are the built in predictor options (traditional options used in Simplescalar).

When running in this environment, the output now indicates the performance of the entire simulation which is more indicative than the simple prediction accuracies that the students collected in the first part of the assignment. At this point in the class, students should have the additional

understanding required to better appreciate the impact of a branch predictor on the overall processor performance. In fact, additional questions can be asked at this point, such as experimenting with the pipeline depth and width, the sizing of processor resources, including the sizing of the prediction tables themselves.

5 Conclusion

We presented a new experimental framework for teaching advanced concepts in processor architecture to undergraduate students. The key novelty of our approach is decoupling of the implementation and the integration of this implementation into the existing cycle-accurate simulator. Specifically, in the first part of the proposed framework, the students are asked to develop the implementation of a dynamic branch predictor (or other advanced concept, such as register renaming) as a standalone software module with the interfaces defined in exactly the same way as the interfaces of the real simulator. In the second step, the students can trivially incorporate their code into the simulator and measure the performance of the new designs using the actual dynamically generated inputs. All tools described in this paper are well documented and available online at **http://www.cs.binghamton.edu/∼msim/branch/** for easy adoption in undergraduate computer architecture courses. The unmodified M-Sim (and documentation) is available at **http://www.cs.binghamton.edu/∼msim/**.

6 Acknowledgements

This work was supported in part by NSF award CNS-0720811 and by the Graduate School at SUNY Binghamton.

References

[1] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, 1997.

[2] J. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *IEEE Computer*, pages 28–35, July 2000.

[3] M-sim: The multi-threaded simulator: Version 3.0, July 2009. Available online at: http://www.cs.binghamton.edu/∼msim.

[4] S. McFarling. Combining branch predictors. *DEC Western Research Laboratory Technical Note TN-36*, June 1993.

[5] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware / Software Interface*. Morgan Kaufmann, fourth edition edition, 2008.

[6] PTLSIM. PTLsim simulator, documentation, and source code. www.ptlsim.org.

[7] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[8] Virtutech simics, 2004-2009. http://www.virtutech.com/products.

Programming Assignment 1

The purpose of this assignment is to design a software model of a dynamic branch predictor as a standalone program. (In the later assignment, this model will be integrated into a full cycle-accurate processor simulator). The new code should be incorporated into existing package (distributed with this assignment) that provide the predictor interfaces to be supported. It will also include a testing package.

The new code to implement a history-based dynamic branch predictor should be added to student.c and student.h files. In particular, it is important to decide what data members need to be added to store the required information, and also implement a class constructor and predictor lookup and update functions.

The following interfaces to the branch predictor have to be maintained (to promote future integration with M-Sim simulator in a seamless manner):

Data types used (note that these are identical to what is used in M-Sim simulator, for the ease of subsequent integration):

md_addr_t	This refers to an address in memory
md_opcode	This refers to the opcode of an instruction
bpred_update_t	This data type is unnecessary for this portion of the assignment. It contains a pointer to the location where a predictor result is generated.

Constructor:
This must initialize the memory and any other variables that are used within the predictor.

bpred_lookup:
This function must return the predicted target address for the requested branch.
It takes the following parameters:

md_addr_t baddr:	Branch PC
md_addr_t btarget:	Target address (if taken)
md_opcode:	The operation code for the branch
bpred_update_t *dir_update_ptr	Can be ignored for this assignment

bpred_update:
This function takes the return of a branch and updates its own state to reflect this.
It takes the following parameters:

md_addr_t baddr:	Branch PC
md_addr_t btarget:	Target address (if taken)
bool taken:	Was the branch taken?
bool pred_taken:	Did we predict taken?
bool correct:	Was our prediction correct? Specifically, did we generate the correct target address?
md_opcode:	The operation code for the branch
bpred_update_t *dir_update_ptr	Can be ignored for this assignment

The included design of two stateless predictors (bpred_taken and bpred_not_taken) can be used as examples. However, neither of these contains any state and will only be of structural help.
Trace files:
The trace files (*.trace) contain the data that replicates the requests to bpred_lookup and bpred_update. These will be used to test your code.

Testing:
To compile the code, type make. Run main.exe with any of the provided trace files (as an example: ./main.exe ammp.trace). The executable will run the trace file using both your code and a default stateless predictor. Results will follow after execution.

Results:
Not all results printed out at the end of the execution will be relevant. Below are the results of just the default always-taken predictor running the ammp tracefile.

default_pred.lookups	78248	# total number of bpred lookups
default_pred.updates	34941	# total number of updates
default_pred.addr_hits	12139	# total number of address-predicted hits
default_pred.dir_hits	12139	# total number of direction-predicted hits (includes addr-hits)
default_pred.misses	22802	# total number of misses
default_pred.jr_hits	0	# total number of address-predicted hits for JR's
default_pred.jr_seen	3245	# total number of JR's seen
default_pred.jr_non_ras_hits.PP	0	# total number of address-predicted hits for non-RAS JR's
default_pred.jr_non_ras_seen.PP	3245	# total number of non-RAS JR's seen
default_pred.bpred_addr_rate	0.3474	# branch address-prediction rate (i.e., addr-hits/updates)
default_pred.bpred_dir_rate	0.3474	# branch direction-prediction rate (i.e., all-hits/updates)
default_pred.bpred_jr_rate	0.0000	# JR address-prediction rate (i.e., JR addr-hits/JRs seen)
default_pred.bpred_jr_non_ras_rate.PP	0.0000	# non-RAS JR addr-pred rate (ie, non-RAS JR hits/JRs seen)
default_pred.used_ras.PP	0	# total number of RAS predictions used
default_pred.ras_hits.PP	0	# total number of RAS hits
default_pred.ras_rate.PP	<error: divide by zero>	# RAS prediction rate (i.e., RAS hits/used RAS)

The most relevant statistics are the number of lookups, the number of updates, addr_hits, dir_hits and misses. The other statistics are not relevant for this portion of the assignment.

Figure 2. First Assignment

Programming Assignment 2

The goal of this assignment is to incorporate the code developed during assignment 1 into the M-Sim simulator. The following four steps need to be completed:

- The student.h and student.c files need to be copied into the same folder as the simulator.
- The Makefile needs to be modified to include these files in compilation.
- The file bpreds.h now needs to include student.h in order to add the student predictor to the set of possible predictors.
- The sim-outorder.c file needs to be augmented with the code to allow the student predictor to be used in the command line with ”-bpred student”.

After completing these steps, evaluate the performance of your branch predictor using a set of SPEC benchmarks. (Note: if SPEC benchmark binaries or input data files are not available for the instructors, then any other programs compiled for Alpha AXP ISA can be used at this stage).
Students can also perform experiments with predictor table size, the impact of pipeline depth on the performance as a function of branch prediction accuracy, and other similar studies.

Figure 3. Second Assignment

model. A first course in computer architec-. ture might follow a bottom-up sequence: gates, circuits, buses, CPU, assembly lan-. guage. This approach lets students take small. steps as they build a conceptual model of a. computer component by component. UnforÂ points of the computer architecture curriculum. Computer architecture courses fall into six. broad historical eras: mainframe, minicom-. puter, early microprocessor, late microÂ modes and the set of computer operations. These two sets are orthogonal if the instruc-. tions are not related to the addressing modes. and an instruction can take any addressing. In many Computer Science programs, a course on computer architecture or computer organization is the only place in the curriculum where students are exposed to fundamental concepts that explain the structure of the computers they program. Unfortunately, most texts on com-puter architecture are written by hardware engineers and are aimed at students who are learning how to design hardware.Â Comer is a Distinguished Professor of Computer Science at Purdue University, where he develops and teaches courses and engages in research on computer organiza-tion, operating systems, networks, and Internets. In computer engineering, computer architecture is a set of rules and methods that describe the functionality, organization, and implementation of computer systems. Some definitions of architecture define it as describing the capabilities and programming model of a computer but not a particular implementation. In other definitions computer architecture involves instruction set architecture design, microarchitecture design, logic design, and implementation.