ECSE 425 : Final Project Report

Simon Krafft, Ringgold Lin, Shawn Liu, and Matthew Beirouti

***Abstract*—Branch prediction was the optimization chosen for our final project. This paper describes the design methodology used to build a pipelined processor with dynamic branch prediction. First, we give a macroscopic view of the processor, with a general description of the design entities and how they interface with one another. A detailed microscopic description of each entity is then given, along a list of design parameters and the reasoning behind particular decisions. Evaluation guidelines are listed for the design modules and results of the testing procedure are presented and analyzed. Finally, the challenges faced and improvements for future iterations are discussed.**

Keywords*—*pipelined processor, branch prediction, VHDL, design

I. Introduction

Processors can be pipelined to increase efficiency. Splitting execution of an instruction into multiple stages allows processors to work on multiple instructions at once. The typical 5-stage pipeline processor can be optimized to improve their efficiency. Caching, branch prediction, multiple issue and single-instruction-multiple-data (SIMD) processor optimizations (among others) are all accessible optimizations that can allow a processor to increase throughput.

For our project, we chose to implement *branch prediction* into our pipelined processor to improve the processor's efficiency. Branch instructions in an unoptimized processor are a source of wasted clock cycles. This is because a the processor does not know which instruction a branch will point to until the execution stage. Without prediction, proceeding instructions must stall until the branch can be resolved. Since most programs have branches an opportunity exists for potentially large gains in efficiency from a technique that is relatively simple to implement.

Many kinds of branch prediction techniques exist, each of which have their own strengths and shortcomings. This is where a computer architects design ability comes into play. The choice of branch prediction technique depends heavily on the type of code the processor is expected to run in practice. Given that we are not (yet) designing commercial processors, we chose to design our branch predictor to work well with code that has a relatively high number of branches. Since static prediction doesn't work well in these scenarios, dynamic (1-bit and 2-bit) prediction was chosen instead. The following is a description of the processor designed.

II. Brief Background

As mentioned previously, there are many types of branch prediction each with their own strengths and weaknesses. For our optimized processor, 1-bit and 2-bit prediction was chosen as they are both relatively simple to understand and implement. A brief overview of how they work follows. 1-bit prediction works by storing the outcome of a branch instruction in memory, then predicting the next outcome to be the same as the last. This is the simplest kind of prediction and works reasonably well for code with loops that have many iterations. Moreover, this prediction technique works well for when branch outcomes are consistent or when one type of outcome vastly outnumbers the other (taken is much greater than not taken for example). However, when these assumptions dont hold, 1-bit prediction isnt very effective. This technique has 2 mispredictions per anomaly (or branch whose outcome is not consistent with the majority of the branches in the code).

A 2-bit predictor improves over the 1-bit prediction technique by using a finite state machine to maintain a rough measure of dominant behavior of branch outcomes of the code. There are four possible states the 2-bit predictor can be in, which are strongly not taken, weakly not taken, weakly taken and strongly taken. As shown in figure 1, this technique is effectively a 2 bit counter which is incremented when a branch is taken, and decremented when a branch is not taken. Taken states predict future branches to be taken and not taken states predict future branches to not be taken. Having these 4 states gives the system a little more weight, meaning it is slightly more difficult to transition from one prediction state to another. This gives a 2-bit predictor 1 misprediction per anomaly (an improvement over the 1-bit predictor) and 2 mispredictions per dominant behavior change.

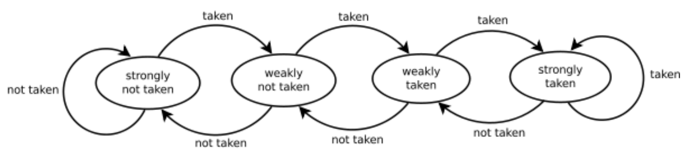


Fig. 1. The Finite State Machine for a 2-bit predictor

III. Macroscopic Design View

Figure 3 (shown in the Appendices section) shows a simplified block diagram of the designed processor. Many control signals have been abstracted away to ensure readability. Note: What follows is a general macroscopic description of the processor, for a more detailed description, skip on to section X. All components are connected to the master clock and a reset signal. All signals that are sent forward in the pipeline from one component to another need to be forwarded through each register in between (this is what the dashed lines represent) to synchronize data flow to avoid timing issues. The design in this figure shows the internals of our processor component, which is put into a testbench to simulate. This test bench provides the instruction memory.

The IF stage has instruction memory control signals that request instructions from the instruction memory. These instructions are then fed to the IF/ID register first before being directed to the ID stage. The IF stage also provides program count (PC) and contains the branch predictor that optimizes the pipeline.

The ID stage compares the resolved branch with the predicted outcome from IF, if these match it allows operation to flow normally. If the predicted branch outcome does not match the actual outcome or if a hazard which is detected in the ID stage occurs, the ID stage stalls by forwarding an add zero instruction to the EX stage and indicating this to all other components through the stall line. The ID stage contains the processor registers and therefore sends the correct register arguments to the EX stage and controls the EX operation.

The EX stage does calculations forwarding the results and/or addresses to the MEM stage and/or WB stage. For simplicity, the data memory was contained within the MEM stage and so it manages the interface to the data. Finally the WB stage receives any data resulting from an ALU operation or a MEM stage load and forwards it to the registers in the ID stage.

IV. Microscopic Design View

A. *Instruction Fetch*

The instruction fetch stage (IF stage) is the stage in which input instructions are loaded into the processor then passed forward to the next stage for identification, while the instruction address is recorded simultaneously. The designed processor takes in MIPS assembly instructions of 32 bits. When the falling edge of the clock signal is detected, a single line of instruction will be fetched from the instruction cache. A different signal named program counter will be incremented by 1 (one line of 32-bit MIPS instruction is actually 4 bytes long, but this value is incremented by 1 for simplicity) to make a quick and cheap prediction of the next instructions address and store it at the same time. This continuously incremented value can be inaccurate for any branching cases (like jump) and any scenarios where stalling is needed to address issues such as hazards. In the case of a branch, the next instruction decode stage (ID stage) will pass the destination address of a branch instruction into this fetch stage. The current prediction based value program counter will be updated as this value thus to have an accurate record of the actual address a program is in at this moment. As for stalling, the goal of IF stage is to not overflow the instructions when the processor is already stalling, and at the same time, keep a record of the instruction right before the processor stalls. This stalling signal is calculated by and passed from ID stage into the IF stage. To determine if the stalling process is over, a cancelling stall signal will be checked at the rising edge of the clock signal, so that during the falling edge, the instruction after stalling is done can be passed into the ID stage on time. An extra feature is the branch prediction. Our pipeline processor can use dynamic 1-bit or 2-bit prediction. The lower bit addresses of branching instructions are saved inside an array memory as branch prediction buffer. If a currently fetched instructions address have a corresponding address value of 1 inside the predict buffer, the processor will consider this predict taken (not taken if the corresponding value is 0), and pass the signal of this predict taken value into the ID stage. Each time when the actual destined branching address is received from the ID stage, the corresponding buffered value in the prediction array will be updated for a more accurate prediction in the next time as reference.

B. *Instruction Decode*

The instruction decode stage (ID stage) is to identify the information of each instruction then handle them by passing calculated and more organized signals to other stages. This stage handles most of the special cases and situations of an assembly program (including taking/updating predicted branch, detect/handle data hazards, and output the stalling signal when needed), and is the most complicated stage (component) of this pipeline processor. The basic function of this stage is to decode the type and instruction of each 32-bit long number line with MIPS table. The flowing through of this function is to first determine the instruction type of the code (R, I, or J) then decode the operations, destination register addresses, or the immediate value. After these parameters are successfully analyzed and cropped from the code, they are passed to the execute stage (EX stage) for calculation. Also, if there are storing or loading actions required, 2 signals which corresponding to each action respectively will be passed into the memory stage (MEM stage) so that the memory stage can carry out the correct action right after the EX stage. A similar signal which represents the existence of action accessing to the memory will be also passed to the write back stage (WB stage) so that the WB stage can pass the result directly to the register block if needed inside the ID stage. During this process of instruction decoding, if a branching case is found, certain actions will be carried out. The processor will check the branching value and update the current results of both of predicting result and branching operation address to the IF stage to update the parameters inside IF stage. If the branching prediction is wrong, when the correct branching address is sent to the IF stage, the IF stage will be also forced to stall for 1 clock cycle to wait for the new branching instruction. The last but not the least feature will be the hazard detecting function. Since the ID stage have the ability to decode any existing instruction and historical instructions records, our processor are designed to save 2 previously done instructions inside 2 instructions buffers. This hazard detecting function uses 2 buffers to record the instruction currently executed in the next 2 stages which are EX stage and MEM stage. The reason why the instruction in the WB stage is not saved in an extra buffer is that our ID stage are executing in a falling edge clock signal, and the WB stage is executing in rising edge. Thus the write back action can be always finished at the same time the ID stage just want to start to use that just updated register value. As a result, the ID stage save totally 3 instructions: The current one just passed from the IF stage, and 2 buffered instructions. With each new fetched instructions are passed into the ID stage, the decoded instruction will check if the registers it wants to use are currently updated already. If any parameter result in a conflict, which implies that at least one register in the ID stage now are still required to be updated or are not passed to WB stage yet, a data hazard will be detected, otherwise the processor will continue its routine work and push the new instruction into the instruction buffer. While a data hazard is detected, a stalling signal will be passed to both EX stage and IF stage so that to halt the processor from using outdated register value and prevent the IF stage pass new instruction into the ID stage and result in loss instruction.

C. Execute

Execution stage is the place where the processor executes the decoded instructions using the ALU component. The stage mainly consists of an ALU controller, an ALU operation unit and a branch handler. The execution stage would take the separated MIPS instruction as inputs from the previous decode stage and pass the processed data to the next two stages. There are some detail steps of the implementation: After receiving RS, RT and Immediate values, a multiplexer would decide which data should be the target source to send to the ALU between RT and Immediate value, selected by whether the instruction is R format or I format. Simultaneously, the six bits opcode(or function) would be mapped to a four bits vector that recognized by ALU operation unit in the ALU controller. Normally, the ALU component would calculate the result according to the given opcode, RS and RT(or Immediate). If the instruction is branch, two input data would be compared and shows the result in Zeros, which is the second output of ALU operation unit. Branch handler would take that Zeros result and the type of branch(beq or bne) as inputs to decide whether the branch will be taken or not, and calculate the branch address by immediate and current PC value. The result of ALU unit will be sent to memory and writeback stage and the output of branch unit will be fed back to decode stage.

D. Memory

Memory stage only contains one component, which is data memory, and its responsibility is to put the data, which is RT inside the instruction, into the proper memory address that generated from ALU unit if the instruction is store, or read data out from that address and pass to the next stage if the instruction is load. Otherwise, just forward the ALU result to the writeback stage with one cycle latency.

E. Writeback

Writeback stage will write the corresponding value back to the register storage. It contains a multiplexer to choose whether it would write back the result calculated by ALU unit or the output from data memory based on the instruction. For example, if the instruction is load, the input data from memory stage would be stored back to register file, otherwise, the result from execution stage would be written back.

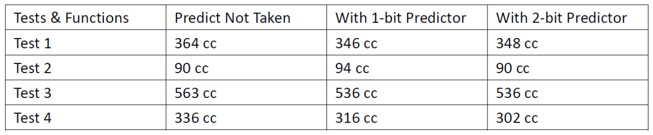
V. Design Parameters and Decisions

How data hazards, control and structural hazards were handled. One was directly eliminated by having separate instruction vs data memory (Data_memory, I_memory components) (control hazards). Data hazards were dealt with by stalling using an add \$0, \$0, \$0 instruction. We didnt implement forwarding. Structural hazards (ie. resource contention) are non-existent because all stages are at most 1 clock cycle long, and the first half of the clock cycle in ID stage writes back values into registers while the second half retrieves register values from ID stage.

The processor was built progressively. The first step was getting one instruction programs to work. Once this was functional, we could work on adding other operations, and longer programs. Each group member is assigned with one or more components function implementation, and each components are submitted together for final assembling after tested in a basic way by using each components testbench created by the corresponding implementer. When the final assembling of the processor is done, a more systematic test as to perform a basic MIPS instruction (without branching, jumping, or data hazard existing) is executed to examine the processor. With such basic processor finished, then the extra features including branching and jumping are added into the processor. For data hazard detection, our first method is to create a separated component called the hazard detecting unit, then use this particular unit to send out the stalling signal. However, there is always a time gap between sending the stalling signal and halting the processor, and there will also be one clock cycle wasted to simply send the signal to the other components. With such limitation of a separated hazard detecting unit, we shifted our plan from adding extra component to adding a hazard detecting function inside the ID stage because the instruction decoding functions are already implemented in the ID stage, and reuse some of those functions can faster the implementation of the hazard detection. For processor optimization, we chose the 1-bit and 2-bit prediction for that the performance of a dynamic prediction instead of a static. The predicting buffer is implemented inside the IF stage mostly to insure a direct effect on the next stage (ID stage). Since we wish not to create extra glitches to other components, besides the prediction buffer values are updated through the ID stage, the optimization and data hazard detection are implemented in IF stage and ID stage respectively and individually.

VI. Evaluation Methodology and Results

For the evaluation of the 1-bit and 2-bit predictors, we used the 4 assembly testing programs shown in figures 4, 5, 6 and 7 in the appendices section. Test 1 is a simple for-loop program, which should be more suitable optimized by a 1-bit predictor. Test 2 contains 4 branching instructions, and are expected to be executed faster in the predict not taken or 2-bit predictor since in this case, the 1-bit predictor could have cause more wrong prediction. Test 3 is to make a mixing version of Test 1 and 2 so that there will be a branching condition inside the loops and make the program jump out loop in the middle of the iteration. Test 4 is designed to test the function of 2-bit predictor on purpose, it is implemented in a method that one particular line of instruction (code) will be branched into after a loop (with this line as a start line) is finished. With the consideration of the ability of 2-bit predictor, the 2-bit predictor is going to have a better predicting process then the 1-bit prediction and prediction-not-taken processes.



Tests & Functions	Predict Not Taken	With 1-bit Predictor	With 2-bit Predictor
Test 1	364 cc	346 cc	348 cc
Test 2	90 cc	94 cc	90 cc
Test 3	563 cc	536 cc	536 cc
Test 4	336 cc	316 cc	302 cc

Fig. 2. Evaluation Results

As it is shown on the table above, the 1-bit predictor dominates the for loop concentrated program (Test 1) for its simple and direct response to the branch prediction. And the 2-bit predictor is just 2 cc longer for the stage transferring. For Test 2, the 1-bit predictor has a worst performance overall. This is because of the branching case will result in more wrong predictions in the 1-bit predicting process. Since there are 4 extra branching instructions in total, the 4 cc latency of Test 2 for 1-bit predictor is as expected. Test 3 is a more complex example, since the overall iteration of the for loop is long, and is mixed with few branching condition, the disadvantage of the non-predict-taken is expanded overall. Both 1-bit predictor and 2-bit predictor have the same clock cycles on executing this program, and this illustrates that the overall performance of the processor will based on the longest execution time based lines, which are the for loop section of Test 3. Test 4 is designed for 2-bit predictor, and the result is as expected. Since there are loop sections in the program, 1-bit predictor and 2-bit predictor are overall better than non-prediction-taken. Since the program branches back to the starting line at the end of the loop for multiple times, the 1-bit predictor shows its disadvantages without having a stage transfer deciding mechanism like the 2-bit predictor.

VII. Challenges

As with any major collaborative effort, many difficulties were faced during the implementation of the optimized pipelined processor. To begin with, all team members had a full understanding of the general theory behind pipelined processors and the purpose and functionality of individual stages and components. Despite this and despite the fact that designs were created and discussed during team meetings, implementation proved to be challenging. While on the macroscale concepts are relatively graspable, microscale focus brings forward many practical issues that are glossed over in theoretical books and online guides. This created a constant need to go back to the drawing board to revise and retest existing modules to accommodate for newly discovered by rethinking interfaces between components or adding new control signals.

Macroscopically defined components meant that there was also more freedom to make design choices within specific components. While this can be beneficial, it also creates a great deal of uncertainty throughout the implementation process. Moreover, this freedom meant each team member needed to decide how to balance component efficiency vs complexity. More efficient components while faster are harder to design, requiring iterative improvements which can be quite time consuming. To overcome this, a decision was made to implement the components in the most straightforward way and to only optimize later on if time permitted such as with the data hazard control and branching control modules.

Design cohesion across team members also proved somewhat challenging. The fact that there were 5 stages meant that tasks could be cut up cleanly (perhaps even too much so). Since each team member was focused on their individual task (or stage), implementations often wouldnt fit as smoothly as desired with other team members components. Addressing this issue is discussed in the improvements section below.

Finally, efficient testing was difficult to achieve. With component designs evolving constantly throughout the development process, it was very time consuming to have to manually adjust and rewrite testbenches for the individual components.

VIII. Improvements and Conclusion

For future iterations of this project, it would be wise to flesh out detailed designs that are agreed upon by and accessible to all team members. These design outlines would ensure design cohesion across team members since any implementation changes should be made on these accessible design plans before they are implemented in code. Moreover, it could be useful to dedicate a team member as the interfacer between component implementers. This interfacers job would be to list the uncertainties and ensure they are resolved between component implementers and entered into the design plan before they are implemented.

As discussed in the previous section, integration of separate components was a challenge. If given the chance to improve the design over a second iteration, clearer signal names should be used to allow for faster integration regardless of the time it takes to come up with an intuitive name. Furthermore, a design key table maintained in unison with code which provides signal names, signal descriptions, expected input sources and expected output destinations for each components signals could provide a useful reference as the signal number of the design grows and could help eliminate uncertainty between team members. More time should also be left for integration testing, which took a lot longer than expected. Building designs preliminarily through the use of block diagrams could also have been useful.

Optimizations such as forwarding (a functionality not implemented in our processor) or out of order execution (challenging but fun to implement) could be added to make the processor more efficient. Better branch prediction techniques such as tournament prediction which alternates between two branch prediction techniques can also be implemented to the same effect. All in all, the processor implementation was a challenging but rewarding endeavour which gave the team a good understanding of subtleties involved with designing, implementing, testing and optimizing computer architectures.

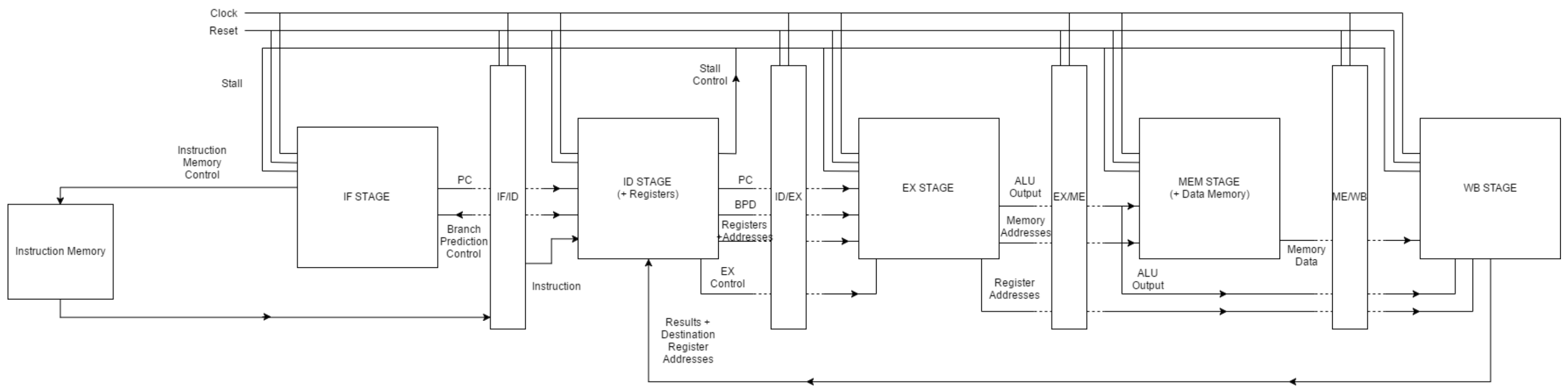
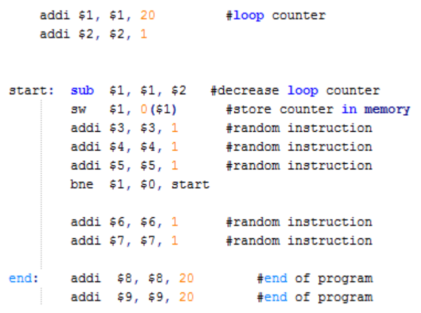
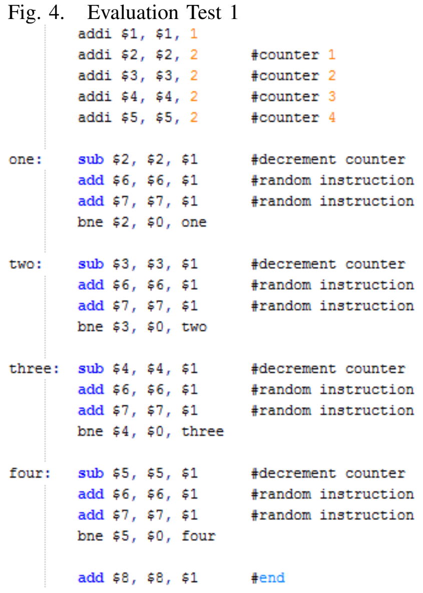


Fig. 3. A simplified block diagram of our optimized pipelined processor



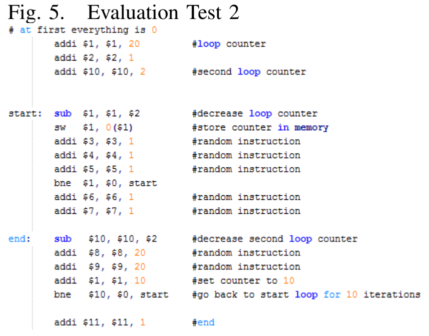
```
    addi $1, $1, 20         #loop counter
    addi $2, $2, 1


start:  sub  $1, $1, $2   #decrease loop counter
        sw   $1, 0($1)      #store counter in memory
        addi $3, $3, 1      #random instruction
        addi $4, $4, 1      #random instruction
        addi $5, $5, 1      #random instruction
        bne  $1, $0, start

        addi $6, $6, 1      #random instruction
        addi $7, $7, 1      #random instruction

end:    addi  $8, $8, 20        #end of program
        addi  $9, $9, 20        #end of program
```

Fig. 4. Evaluation Test 1



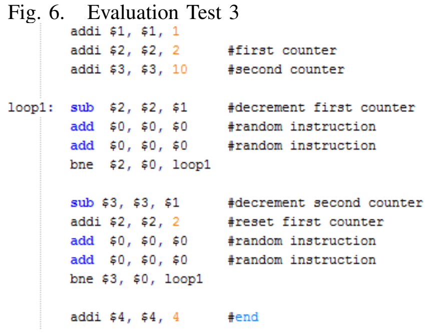
```
        addi $1, $1, 1
        addi $2, $2, 2      #counter 1
        addi $3, $3, 2      #counter 2
        addi $4, $4, 2      #counter 3
        addi $5, $5, 2      #counter 4

one:    sub $2, $2, $1      #decrement counter
        add $6, $6, $1      #random instruction
        add $7, $7, $1      #random instruction
        bne $2, $0, one

two:    sub $3, $3, $1      #decrement counter
        add $6, $6, $1      #random instruction
        add $7, $7, $1      #random instruction
        bne $3, $0, two

three:  sub $4, $4, $1      #decrement counter
        add $6, $6, $1      #random instruction
        add $7, $7, $1      #random instruction
        bne $4, $0, three

four:   sub $5, $5, $1      #decrement counter
        add $6, $6, $1      #random instruction
        add $7, $7, $1      #random instruction
        bne $5, $0, four

        add $8, $8, $1      #end
```

Fig. 5. Evaluation Test 2



```
# at first everything is 0
        addi $1, $1, 20         #loop counter
        addi $2, $2, 1
        addi $10, $10, 2        #second loop counter


start:  sub  $1, $1, $2         #decrease loop counter
        sw   $1, 0($1)          #store counter in memory
        addi $3, $3, 1          #random instruction
        addi $4, $4, 1          #random instruction
        addi $5, $5, 1          #random instruction
        bne  $1, $0, start
        addi $6, $6, 1          #random instruction
        addi $7, $7, 1          #random instruction

end:    sub   $10, $10, $2      #decrease second loop counter
        addi  $8, $8, 20        #random instruction
        addi  $9, $9, 20        #random instruction
        addi  $1, $1, 10        #set counter to 10
        bne   $10, $0, start    #go back to start loop for 10 iterations

        addi $11, $11, 1        #end
```

Fig. 6. Evaluation Test 3



```
        addi $1, $1, 1
        addi $2, $2, 2      #first counter
        addi $3, $3, 10     #second counter

loop1:  sub  $2, $2, $1     #decrement first counter
        add  $0, $0, $0     #random instruction
        add  $0, $0, $0     #random instruction
        bne  $2, $0, loop1

        sub $3, $3, $1      #decrement second counter
        addi $2, $2, 2      #reset first counter
        add  $0, $0, $0     #random instruction
        add  $0, $0, $0     #random instruction
        bne $3, $0, loop1

        addi $4, $4, 4      #end
```

Fig. 7. Evaluation Test 4