A Superscalar and Branch Prediction Processor Implementation

by

Ana Camba Gomes

Fabiana Gonzalez

Course: 6.192 Constructive Computer Architecture

Project Advisor: Martin Chan

Department of Electrical Engineering and Computer Science

Massachusetts Institute of Technology

May 14, 2024

Contents

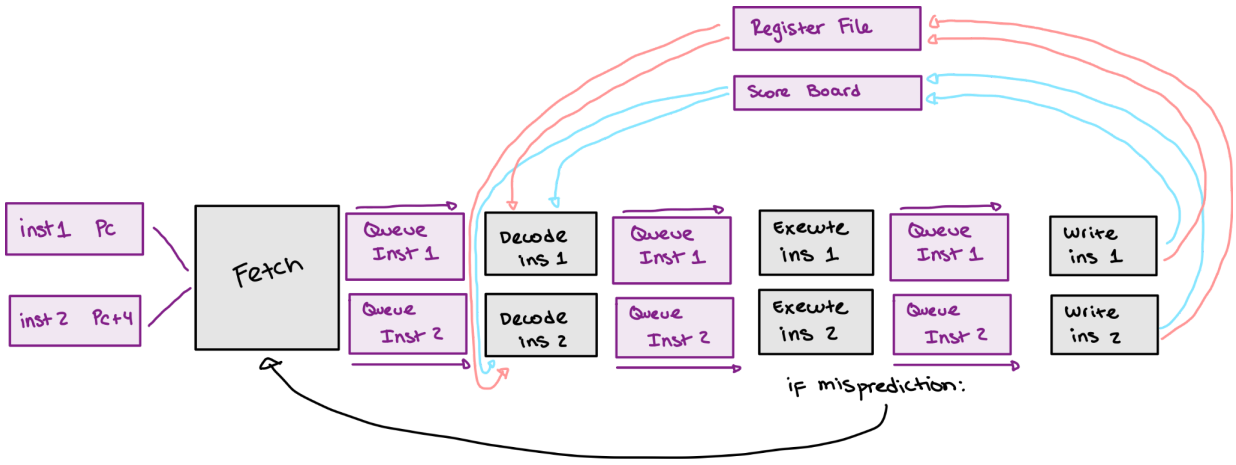
1. Introduction/Motivation:

This final project details the design and implementation of a superscalar processor and a branch prediction unit, aiming to enhance program execution efficiency by reducing the number of cycles required for task completion. Our primary objective was to implement a superscalar architecture capable of fetching and executing two instructions simultaneously, thereby accelerating program execution. Additionally, the team started the integration of branch prediction units to improve the handling of conditional branches, further optimizing program flow.

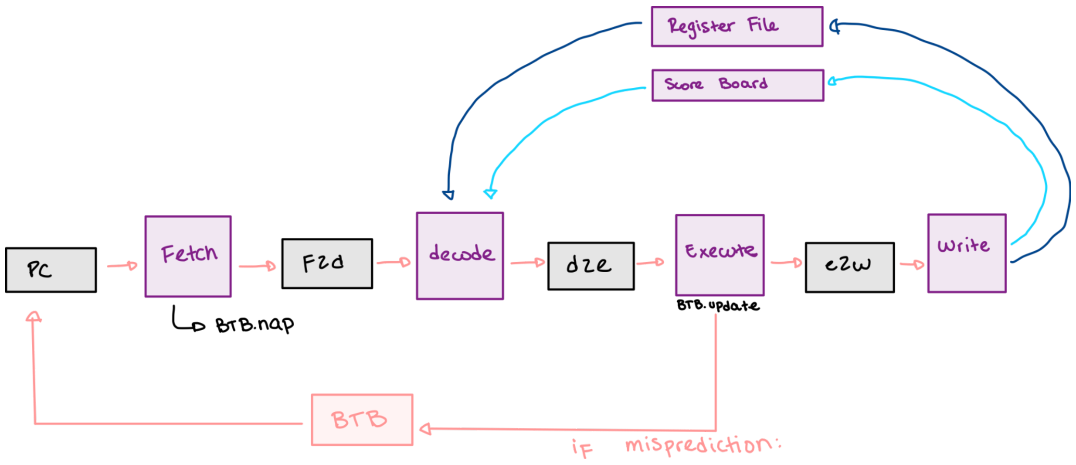
The work presented here is part of a final project led by the course 6.1920 in collaboration with our direct advisor Martin Chan. The aim of the project is to design, implement and test our superscalar and branch prediction processor's capacities.

Finally, the team aimed to evaluate and investigate the capabilities and constraints of Bluespec as both a hardware synthesis language and, more broadly, as a tool for digital systems design. This endeavor will delve into the functionality of Bluespec tools, encompassing not only their individual features but also their integration into higher-level system design.

2. Superscalar Diagram:



3. Branch Prediction Diagram:



4. Implementation Superscalar:

The SuperScalar tries to execute multiple instructions during a single clock cycle if possible. In this case we will take two instructions per cycle. The benefit is that it will potentially increase the performance of the processor without increasing its clock speed, so it will take less cycles to finish the program

4.2 Fetch:

To begin our implementation, we created one data cache and one instruction cache. In the instruction cache, we modified the logic so that it can now fetch two instructions at a time. We also established a separate interface for each cache.

The new feature of the instruction cache is that it now retrieves two instructions from the cache instead of just one, as previously implemented. We achieve this by using the OneorTwoWords struct, which includes a word for the first instruction and a potentially valid word type for the second instruction, provided it falls within the cache's boundary.

Boundary Function:

We created a function called notLineBoundary, which checks the program counter (pc) of the first instruction to determine if it is on the edge of the cache line or if it can proceed to the next instruction. This Boolean function returns false if the address offset consists entirely of ones, and true otherwise.

We made changes to the other documents so that every function has the correct type, including modifying the getToProc method type for the instruction cache interface. It now accepts the struct instead of a single word. After all these changes, we can proceed to the rule fetch, where

we submit the information of the first instruction (the pc). It will then be returned to us as a struct containing two pieces of information: pc and pc+4

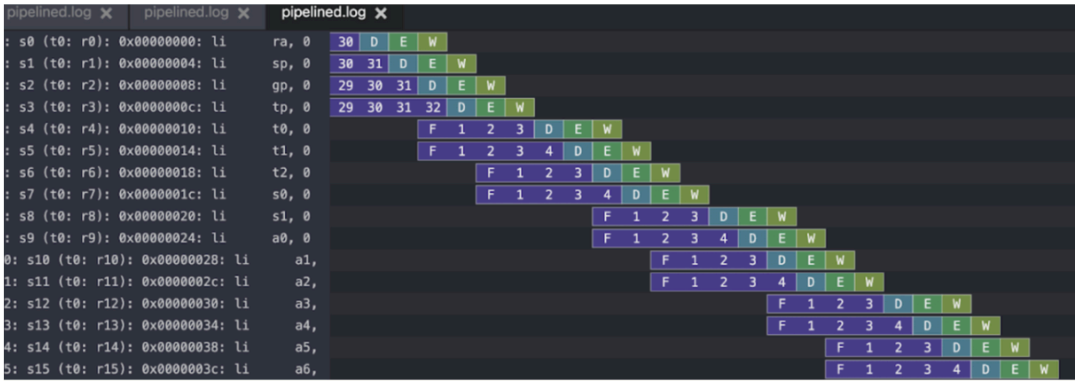
4.2.1 Fetch Bugs:

In our boundary function, we had the logic incorrect because we were checking the index to determine if it was the last space in the cache line, rather than checking the instruction offset. The index actually indicates which line of the cache we are on, not the position within the line.

Another issue we encountered was related to the second instruction in the Konada. To address this, we added another ID specifically for the second instruction, allowing us to track which stage it was in.

We created a numFetched variable which will be set to 2 if instruction 1 was not at the end of the line, and 1 if it was at the end of the line. We will add that amount to the iid depending on the condition. This is defined using the nfetchKonata function from the Konata files, which returns the first ID of the consecutive k IDs allocated.

After making the changes, we can see that the fetch stage is passing two instructions at a time, while the other stages are passing one per cycle:



4.3 Decode:

For the decode stage, we created two different rules: one for decode 1 and another for decode 2. We will retrieve the first value from each of the two FIFOs that contain information from the Fetch stage.

We will check in getIresp if we are receiving a one-word or two-word type. Enqueue the first word and check if we can enqueue the second word (if it is not empty). We will enqueue both by using SuperFIFOs.

To proceed to decode 2:

- We need the condition that has already passed to decode 1.
- The second instruction must not be empty (we checked this by using .deqReadyN(2), which was defined inside the SuperFIFO interface). This deqReadyN() will return true if we have two functions to dequeue and false otherwise.

It is important to discuss the definition of SuperFIFOS, SuperFIFO contains the same characteristics as the FIFO but allows the enqueuing and dequeuing of two different FIFOs in the same cycle by including new features. We use this to transport the information of the two instructions within the same cycle

4.3.1 Decode Bugs:

We have revised the implementation of our scoreboard mechanism to accommodate scenarios where two instructions may target the same register. Consequently, instead of statically assigning values of 0 or 1 to the scoreboard, we now dynamically adjust its count. Specifically, we increment the count each time a register is utilized and decrement it upon completion of its use.

This approach necessitates stalling operations when any register on the scoreboard reflects a count exceeding zero, indicating ongoing usage.

In the writeback phase, we decrement the scoreboard count following the release of the register. Similarly, during the execution phase, if an instruction is squashed, we adjust the scoreboard to reflect that the register is no longer required

After making the changes, we can see that the fetch and decode stage is passing two instructions at a time, while the other stages are passing one per cycle:

4.4 Execute:

For the “Execute” stage we also duplicated our logic. In this case, we have two execution rules. Execute 1 and Execute 2. Some restrictions that we took into account for the instructions to pass through execute 1 and execute 2 where:

- **Control instruction** (one at a time)
- **Memory instruction** (one at a time)
- **Alu instruction** (both can pass at same time)

For this we created a vector ehr of 2 spaces initialized to 1 meaning that both instructions were alu instructions and then in the decode stage we checked whether the instruction was a control instruction or memory instruction.

Vector instantiation:

```
Vector#(2,Ehr#(2, Bit#(1))) instruction_bool <- replicateM(mkEhr(1));
```

Decode 1 Check Example:

```
if(isMemoryInst(decodedInst)||isControlInst(decodedInst)) begin
                instruction_bool[0][0] <= 0;
 end
```

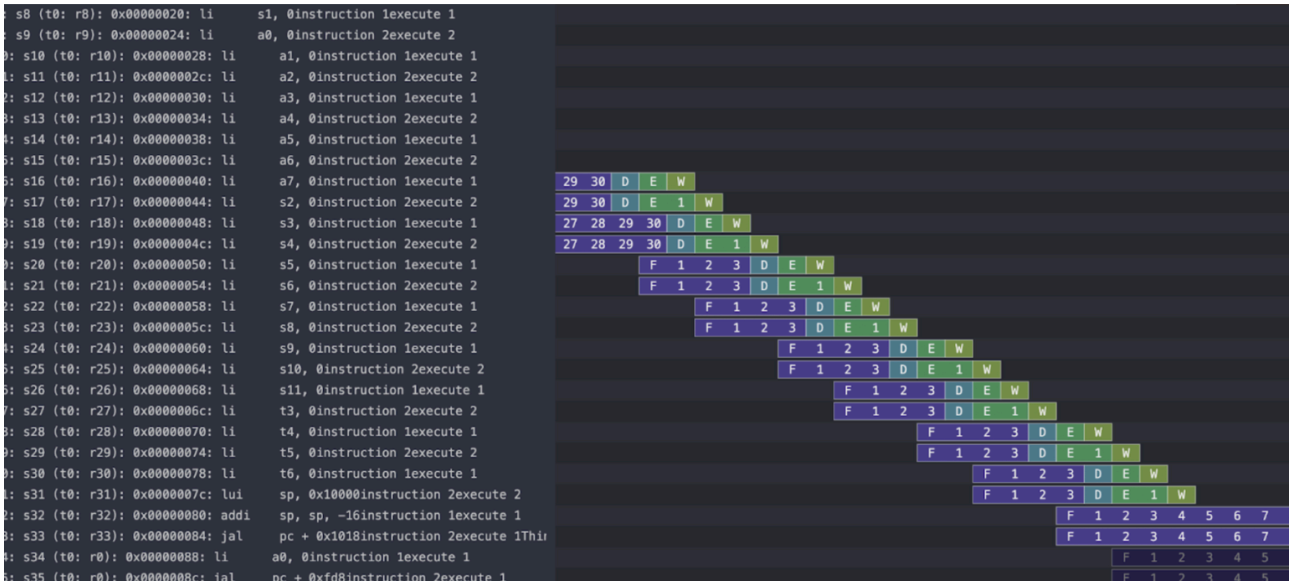
If so, we changed the values in the vectors to 0. Then in execution we checked the values of this ehr to see if they had changed from our previous prediction that the instructions were both alu instructions. If they are alu instructions we can use both execute sections but if one of them is a control or memory we only allow one to pass and the other instruction waits for the next cycle.

Execute 2 check:

```
    rule execute2 if (!starting  && (instruction_bool[1][0]==1)
&& (instruction_bool[1][1]==1));
```

4.4.1 Execute Bugs:

1. None of the instructions were passing through execute 2 so that made us know that something was happening that was not permitting us to enter. We started looking a possibilities like FIFOS and EHR by commenting every line of code and see if they were blocking each other (we checked this on the mkpipelined.sched file which give us every rule with its predicate and the blocking rules)
2. Blocking rules: execute 1 was blocking execute 2. We needed to make a superfifo for toMMIO and toDmem which were FIFO before. So execute 1 can enq one port and execute 2 can enq the other port on the same cycle.



4.5 Writeback:

For the “Writeback” stage we also duplicated our logic. In this case, we have two writeback rules. Writeback 1 and Writeback 2. Some restrictions that we took into account for the instructions to pass through Writeback 1 and Writeback 2 where:

- **Memory instruction** (one at a time)
- **Alu instruction** (both can pass at same time)

To make these stages behave as expected we first removed the logic for memory instructions for writeback 2, since if we have a second instruction it will be ALU instruction and wont need that additional logic. Second, we created a bit ehr variable “isMemOrControlIns” that allowed us to check in writeback 2 if our instruction was a memory or an alu instruction

```
Ehr#(2, Bit#(1)) isMemOrControlIns <- mkEhr(1);
```

Writeback 1 value setup:

```
if(!isMemoryInst(from_execute.dinst) &&
!isControlInst(from_execute.dinst)) begin
            isMemOrControlIns[0] <= 0;
 End
```

Writeback 2 Check:

```
rule writeback2 if (!starting && isMemOrControlIns[1] == 0);
```

Finally, created an independent rule that fires every cycle that allows us to set our ehr “isMemOrControlIns” back to invalid after one cycle.

```
rule settingMemory;
        isMemOrControlIns[1] <= 1;
Endrule
```

4.5.1 Writeback Bugs:

1. We had a problem with the logic of implementing isMemOrControlIns. Whenever we passed through writeback 1 and the instruction was either a control or a memory instruction it will be set to 0. But, we were setting the variable in writeback 1 and then after the end of writeback 2. This was incorrect because it was not resetting after every cycle. That is why the rule was necessary to make sure it is completely changed after the next cycle.

4.6 Superscalar Results

Test Name	Lab 3b processor	Superscalar Processor
Add32	291	275
And32	291	275
Hello32	1007	1094
Mul32	974	988
Or32	291	275
Reverse32	23933	26464
Sub32	291	275
Thelie32	65551	76974
Thuemorse32	26503	26097
Xor32	291	275

After conducting some tests, we observed that in some programs, such as Add32, And32, Or32, Sub32, and Xor32, the superscalar processor completes the program in fewer cycles than the regular processor implemented in 3b. However, there are other programs like Hello32, Mul32, and Reverse32 where the original processor finishes the program more quickly than the superscalar. We conclude that this may be due to only executing two instructions if the first instruction is within the boundary. By incorporating branch prediction, we believe we can accelerate the process. If we modify the logic to determine paths more efficiently, we could potentially complete the programs in fewer cycles.

5. Implementation Branch Prediction:

Dynamic Branch Prediction with a BTB is a good way to improve the cycle count in our programs. For example, the BTB will store the pc's that we visit in our program, and if we have already gone through that pc (maybe in the case of a loop) this will be beneficial because we have the information of the next pc stored in the BTB and we do not have to do the calculations all over again.

For the branch prediction we were planning on creating an interface that had two methods. The initial method, "nap," is designated for utilization during the fetch phase as a component of the Branch Target Buffer (BTB) operations. Subsequently, the "update" method will come into play during the execution phase, serving as another BTB function. In the event of a misprediction, this method will facilitate the transmission of the accurate information back to the program counter.

Following the establishment of these methods, the next step involves integrating the predictive logic into our superscalar pipeline. This integration is vital for ensuring the pipeline's ability to anticipate and execute the most precise branch transitions.

During the fetch phase, the system will examine the PC (program counter) and its associated target in the BTB. If the PC is not found, the pipeline will proceed by setting the predicted program counter (PPC) to PC + 4.

Subsequently, during the execute phase, the system will evaluate the prediction. In the event of an incorrect prediction, the pipeline will invalidate the instructions along the wrong path. Additionally, it will update the BTB to reflect the outcome of jumps and taken branches, thereby refining its predictive capabilities.

We started this implementation in the BTB.bsv file but due to time constraints we were not able to finish this branch prediction implementation.

References:

[1] BluespecTM SystemVerilog Reference Guide. Revision: 16 June 2010. Available at: http://csg.csail.mit.edu/6.S078/6_S078_2012_www/resources/reference-guide.pdf

[2] Bourgeat, Thomas (2024). Basics of Branch Prediction. Constructive Computer Architecture, Massachusetts Institute of Technology.

[3] Chan, Martin. (2024). Superscalar in-order Machines. Constructive Computer Architecture, Massachusetts Institute of Technology.