Superscalar Processor with Branch Prediction

Ziyad Hassan and Olivia Stoner | 6.192 Final Project

Introduction

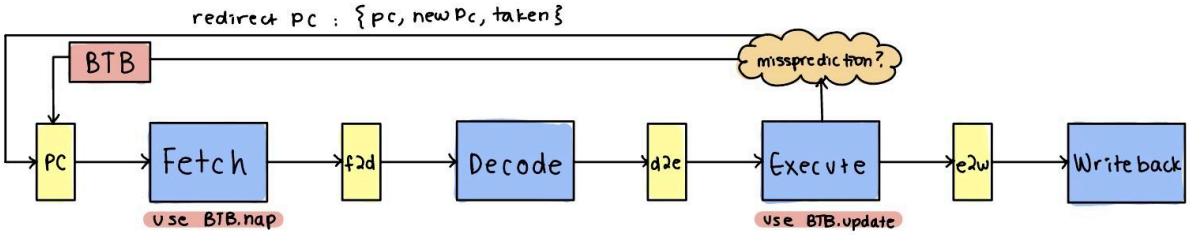
In this project, we implement a superscalar processor with branch prediction. We achieve significant performance gains through each of these optimizations individually, and combined, we achieve a notably more efficient processor, as justified in the “Results” section below.

Motivation

The motivation behind these two architectures comes from our background in 6.1910/6.004. We always heard about lowering our Instructions Per Cycle (IPC) as close to 1 as possible along with how long squashing instructions that propagate throughout the pipeline take. This is most apparent in the MatMul32 test case which involves a lot of branches, thus a lot of squashed instructions. So we wanted to achieve a processor that not only performs basic instructions ‘quickly’ but we wanted a way to ‘predict’ branches and not have to waste time squashing too many instructions. So we decided on a superscalar process with branch prediction.

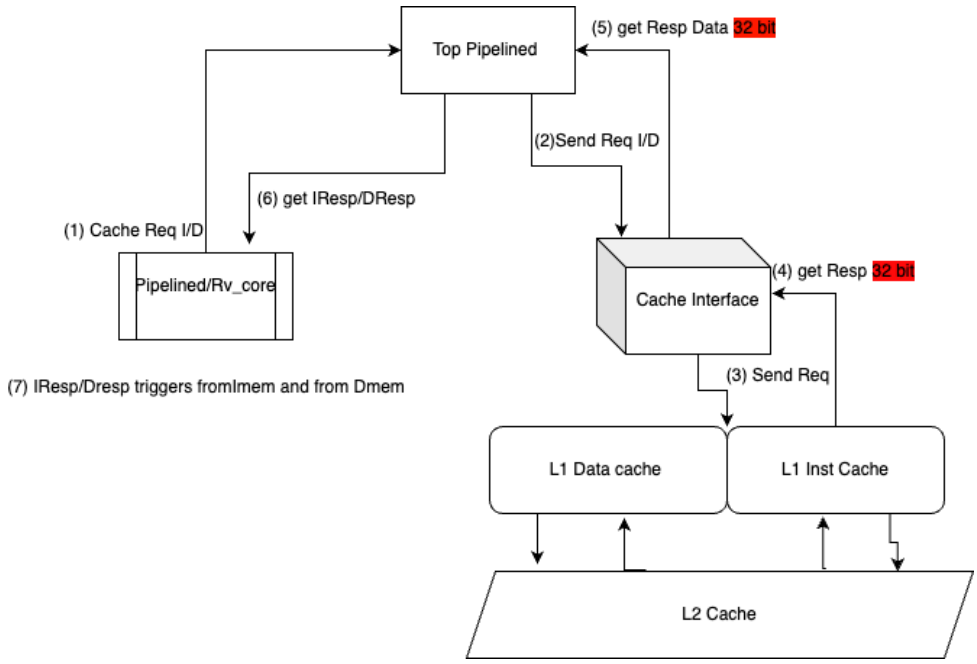
System Diagrams

System Diagram 1: Branch Prediction



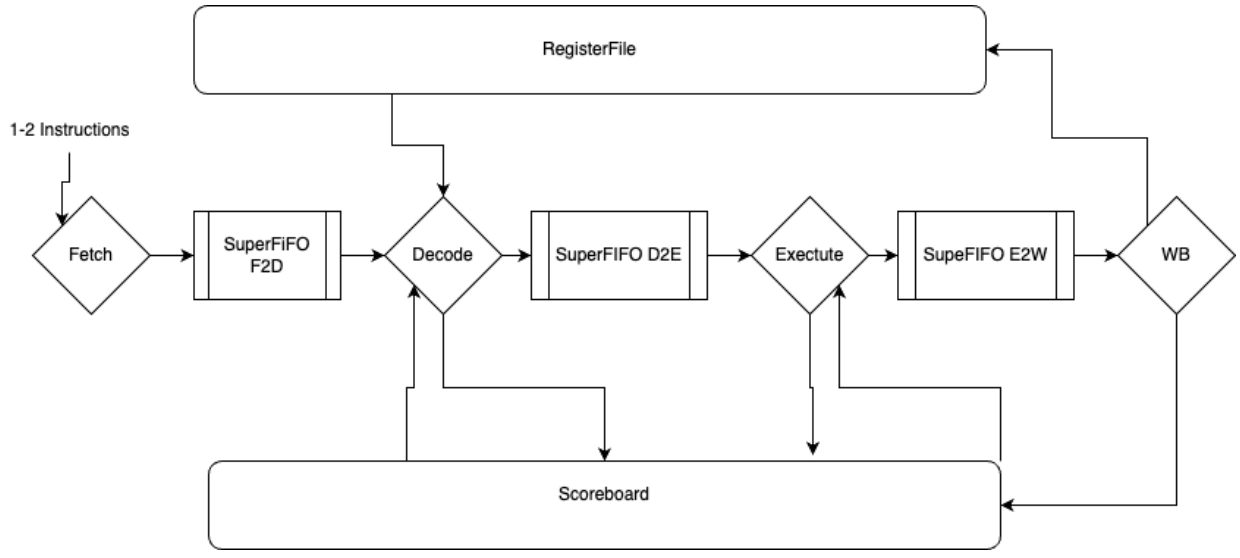
Branch prediction via a BTB follows the diagram flow above. The BTB is checked in the fetch stage to determine the predicted next PC of a given instruction. It is then updated in the execute stage.

System Diagram 2: Module Interaction



This diagram describes the high level flow of information between the different modules. It was important to understand this because it allowed us to pinpoint places where we needed to change 32 bit information to “super responses” (32 bit + maybe 32 bit).

System Diagram 3: Superscalar Architecture



Here we have the diagram for a normal pipelined processor except we changed 2 things: fetch takes in 1-2 instructions(based on what we ask it / is able to fetch from the cache) and in between each stage there’s a superFIFO instead of a FIFO. This allows us to dequeue and enqueue 1-2 instructions at a time, which achieves our superscalar architecture.

Weekly Updates

Week 1
- Exploring the system, understanding how it works

Week 2
- Implementing SuperResponses(32 bit + maybe 32 bit type) and SuperMem.
- Fixing Konata labeling
- Adjust Cache to give 1-2 responses
- Implement Fetch to take in 2 instructions at a time
- Implement Decode to take in 2 instructions at a time
- Debug Fetch and Decode
- BTB implementation
- Optimal BTB size testing

Week 3:
- Implement Execute to take ALU instructions two at time
- Make WB Superscalar
- Work on Slides for final presentation
- Work on getting superscalar integration with branch prediction
- Gathering data, graphing performance gains

Week 4
- Finalize code
- Finalize Slides

Implementation

Branch Table Buffer

We implement the BTB as a vector of registers, where the size of the vector corresponds to the number of lines of the BTB. $2^{\#\text{ idx bits}}$ = size of BTB, where the index bits are evaluated from the value of the last (# idx bits) elements of the pc. The remaining upper bits of the pc, pc[31:(# idx bits)], is used to generate the “tag” of each BTB line. Each BTB line is formatted as {tag, ppc, valid bit}. We implemented this address predictor as an AddrPred interface with two methods:

```
interface AddrPred;
   method Bit#(32) nap(Bit#(32) pc);
   method Action update(Bit#(32) pc, Bit#(32) nextPC, Bool taken);
endinterface
```

The nap method is used to either return the ppc if the given pc is stored in the BTB or pc+4 if not. The update method is used to update the BTB with a different ppc if a branch has been taken or delete the PC entry if not.

Superscalar

SuperResponse + SuperMem

- `typedef struct { Bit#(4) byte_en; Bit#(32) addr; SuperResponse data; Bit#(1) request;} SuperMem deriving (Eq, FShow, Bits);`
- `typedef struct { Word data1 ; Maybe#(Word) data2; } SuperResponse deriving (Eq, FShow, Bits);`
- `typedef struct { Bit#(4) word_byte; Bit#(32) addr; Bit#(32) data; Bit#(1) request; } CacheReq deriving (Eq, FShow, Bits, Bounded);`
- These types were used to indicate to the cache how many lines it should bring based on the predictedPC(for branches) and line offset(for end of the cache line)

Fetch + Decode + Execute + WB

- Fetch was implemented by requesting 1- 2 pcs and then offloaded into the next superFIFO
  - ```
    let insNum=(offset==15||(ppcF != (pc_fetched +4)))?1:2;
    ```
  -
- Decode, decoded two instructions at a time whenever possible while adhering to scoreboard rules
  - `if(fromImem.deqReadyN(2)&&!sbDependency&&!sbDependency1&&!instDependency(instruction,instruction1))begin`
- Execute would do 2 instructions at once if they were ALU instructions ; 1 instruction otherwise
  - ```
    if(isAlu0&&isAlu1&&decodeObj2.epoch==epoch[0]&&d2e.deqReadyN(2))begin
    ```
  -
- Writeback would also do 2 instructions if they weren't memory instructions; 1 otherwise
  - ```
    if(secondWb&&!isMemoryInst(dInst)&&!isMemoryInst(temp_execObj2.dinst))begin
    ```
  -

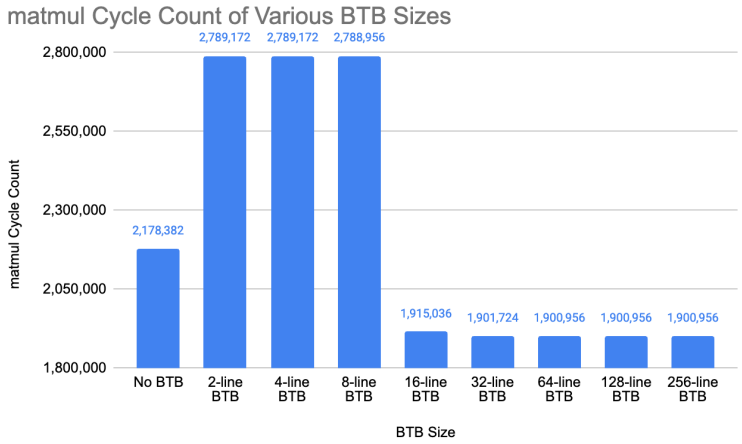
Superscalar + Branch Prediction

- We found that it was easier to work on the tw features separately and then integrate rather than try to do it all at once
- The integration was easy; we adjusted when we requested 2 lines from the cache to also include if the pc_fetched was the pc_fetched + 4 in the case that it was a predicted branch, to which we would only grab 1 line
    ```

Results

Branch Prediction (non-superscalar implementation)

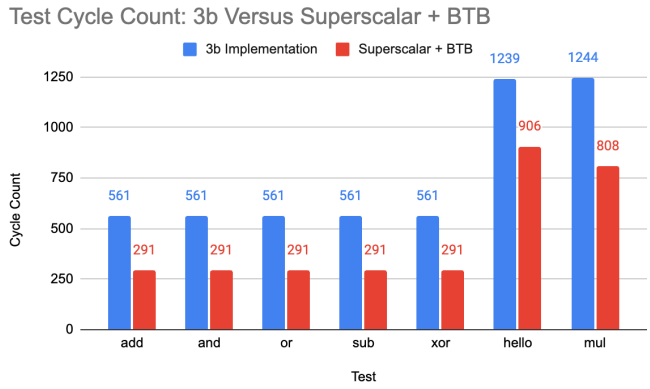
We first evaluate the branch prediction performance on a non-superscalar implementation with the matmul test. This specific test includes many branching instructions, and so implementing a BTB dramatically impacts our performance, even without the superscalar implementation, as shown below. These isolated branch prediction results (without the superscalar implementation) have been determined by running the matmul test in OliviasDir.

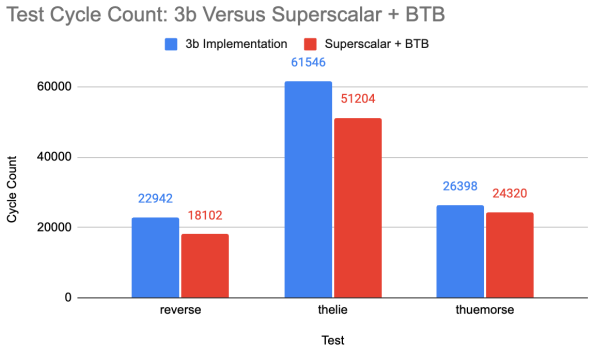


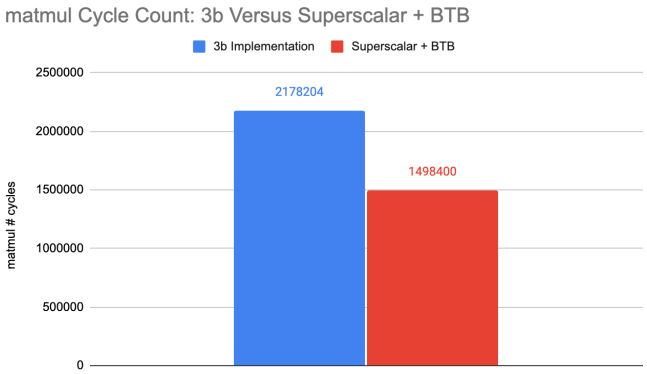
The optimal BTB size for this test is 64 lines; after 64 lines, the performance (number of cycles it takes to execute the instruction) plateaus. Notable values also include the decreased performance with a 2, 4, and 8-line BTB versus no BTB and the significantly-increased performance from an 8-line BTB to a 16-line BTB, from 2,788,956 to 1,915,036 cycles. This solidifies the performance gains one can achieve with even a small BTB, if scaled to the optimal size for the specific use case.

Superscalar + Branch Prediction

Below, we graph the performance gains that resulted from implementing both the superscalar architecture and the 64-line BTB. These optimizations saved us close to 700,000 cycles when performing matrix multiplication in the matmul test, and it also cut back our cycle count by 48% for smaller tests like add, and, or, sub, and xor.


Test Cycle Count: 3b Versus Superscalar + BTB
3b Implementation
Superscalar + BTB
Cycle Count
1250
1000
750
500
250
0
561
291
561
291
561
291
561
291
561
291
1239
906
1244
808
add
and
or
sub
xor
hello
mul
Test


Test Cycle Count: 3b Versus Superscalar + BTB
3b Implementation
Superscalar + BTB
Cycle Count
60000
40000
20000
0
22942
18102
61546
51204
26398
24320
reverse
thelie
thuemorse
Test


matmul Cycle Count: 3b Versus Superscalar + BTB
3b Implementation
Superscalar + BTB
matmul # cycles
2500000
2000000
1500000
1000000
500000
0
2178204
1498400