CacheMeOutside

Armando Moncada, Yeabsira Hawaz, Orion Li

Intro:

For this project, we aimed to supercharge the performance of the task we were given as the final assignment in 6.192. For reference, our goal was to implement a neural network that receives an black-and-white, 28x28 image of a handwritten decimal digit, and outputs a prediction for what the number drawn was. In order to perform this as quickly as possible, we chose a few optimizations we learned in class, along with moving it to actual hardware. We first added in Superscalar functionality, then Branch Prediction, and finally moved it onto an FPGA. Throughout the process, we ensured correct Processor functionality by constantly using the lab 3's automated test bench, along with 6.004's provided test bench.

Diagram:

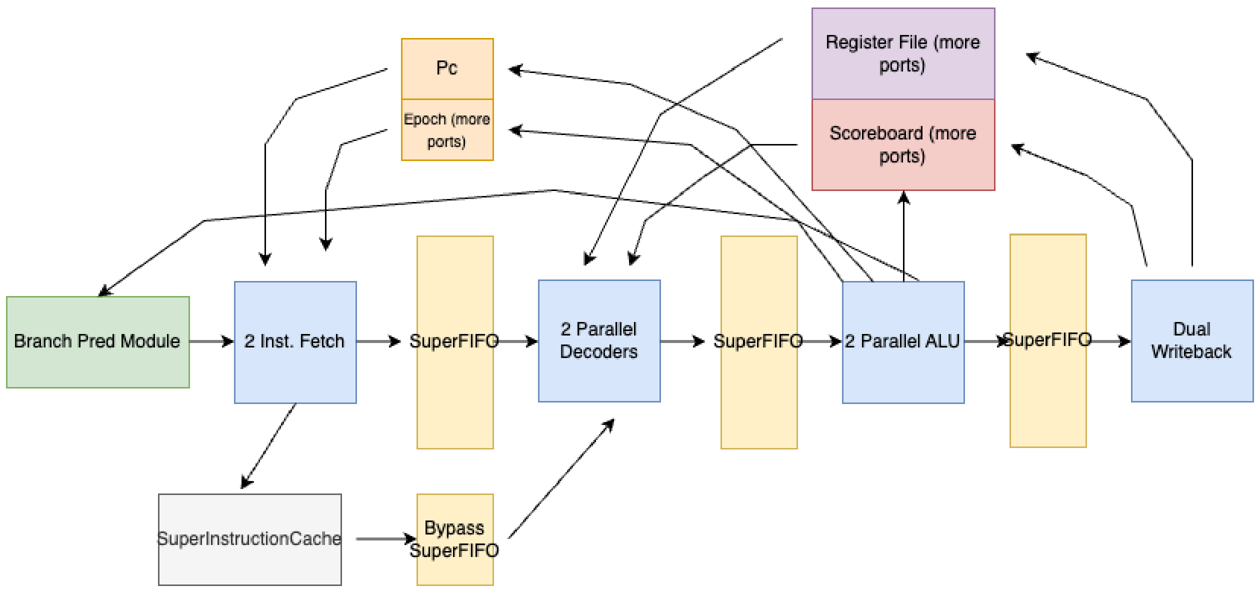
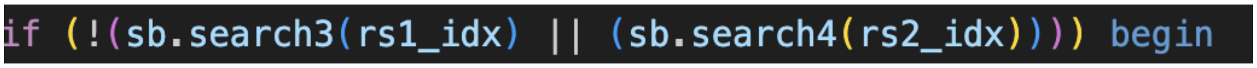


Figure 1. Main Memory + Data Caches not included for clarity.

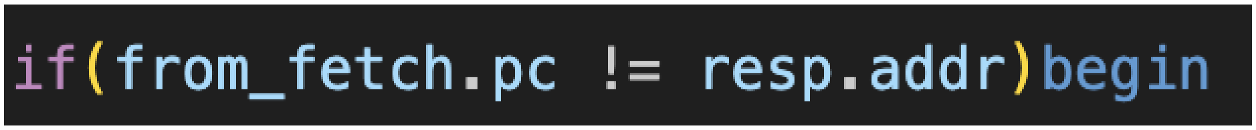
Superscalar:

For our Superscalar processor we decided to follow a lot of the outline given in the superscalar lecture. We'll break it down at a high level and the implementations first, then move onto some bugs and how we got through them. So let's take it stage by stage. Before that let's make all the queues between stages Super-fifo's so we can enq and deq twice in one cycle. Let's also make it so our cache can spit out two instructions at once if we want it to(just have a flag where it grabs two insts, and have that flag be set in fetch). Now in fetch we just check if we're at a line boundary, if we aren't we fetch two instructions and we set the flag to the cache that we want two instructions, and we're done.

Next is Decode, we'll basically have 2 decode stages, the first decode stage will stay exactly the same. The second decode stage won't fire if its rs1 or rs2 from the second Decode rely on the result register from decode 1. Besides that it mainly stays the same, you also have to double the EHRs on for the scoreboard in-order to access it that many times though. We also check in Decode if the PC of the instruction is the actual pc of the instruction? Since we have this weird bug where sometimes it gets the pc mixed up, if we end up being wrong we just don't do the second decode at all and stall like a regular pipelined proccessor. So it really is just copy pasting decode but with these two if statement before the logic



```
if (!(sb.search3(rs1_idx) || (sb.search4(rs2_idx)))) begin
```

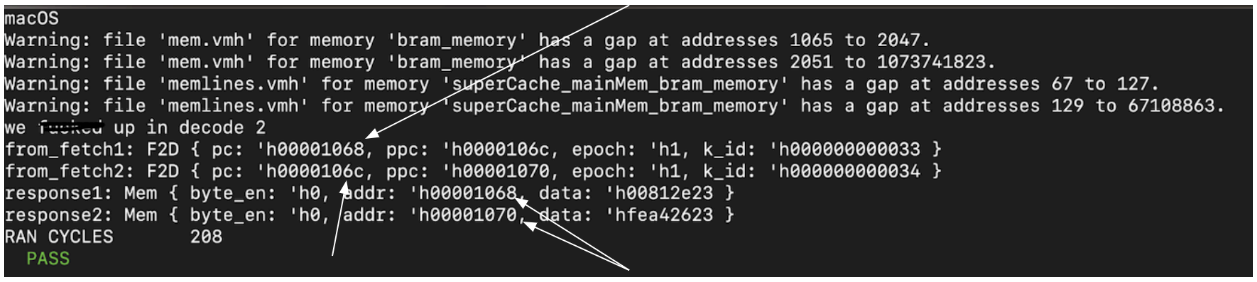


```
if(from_fetch.pc != resp.addr)begin
```

Notice how we don't have to explicitly check if there's a dependency between the two, since it's already implicitly checked by seeing if either rs1 or rs2 for instruction two is in the scoreboard.

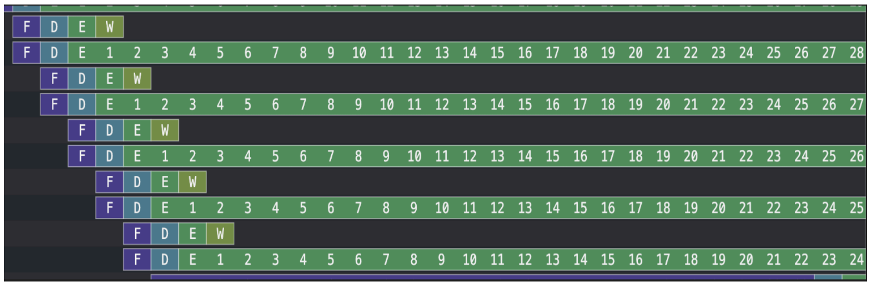
Next is Execute, the Execute 1 is the same from the pipelined processor, except now we set an EHR var(called isExec1Alu, if you're curious) equal to true if we're processing a Alu func in Execute 1. Now in Execute 2 we only handle the instruction and do deq2, if both the instructions in Execute 1 and Execute2 are Alu Funcs otherwise we just don't process the second instruction and stall it (so we get the same amount of work as a regular pipelined execute). Oh yeah also if we squash something in Execute 1 it always gets squashed in Execute 2. Also, we make sure that flax isExec1Alu, is always set to false when Execute2 is Executed(so its default is false).

Next is Writeback, we had to increase the EHR on the RF, and make sure Writeback2 writes after Writeback1(incase there's a conflict we need to make sure it happens in the right order).The Writeback 1 stays the same, Writeback2 keeps the same functionality as Writeback1 unless it's a Memory Instruction, in which case we just stall and don't do anything. And those are all the stages! Some bugs we ran into was the one I mentioned in Decode where the PC's just wouldn't match which was really annoying.

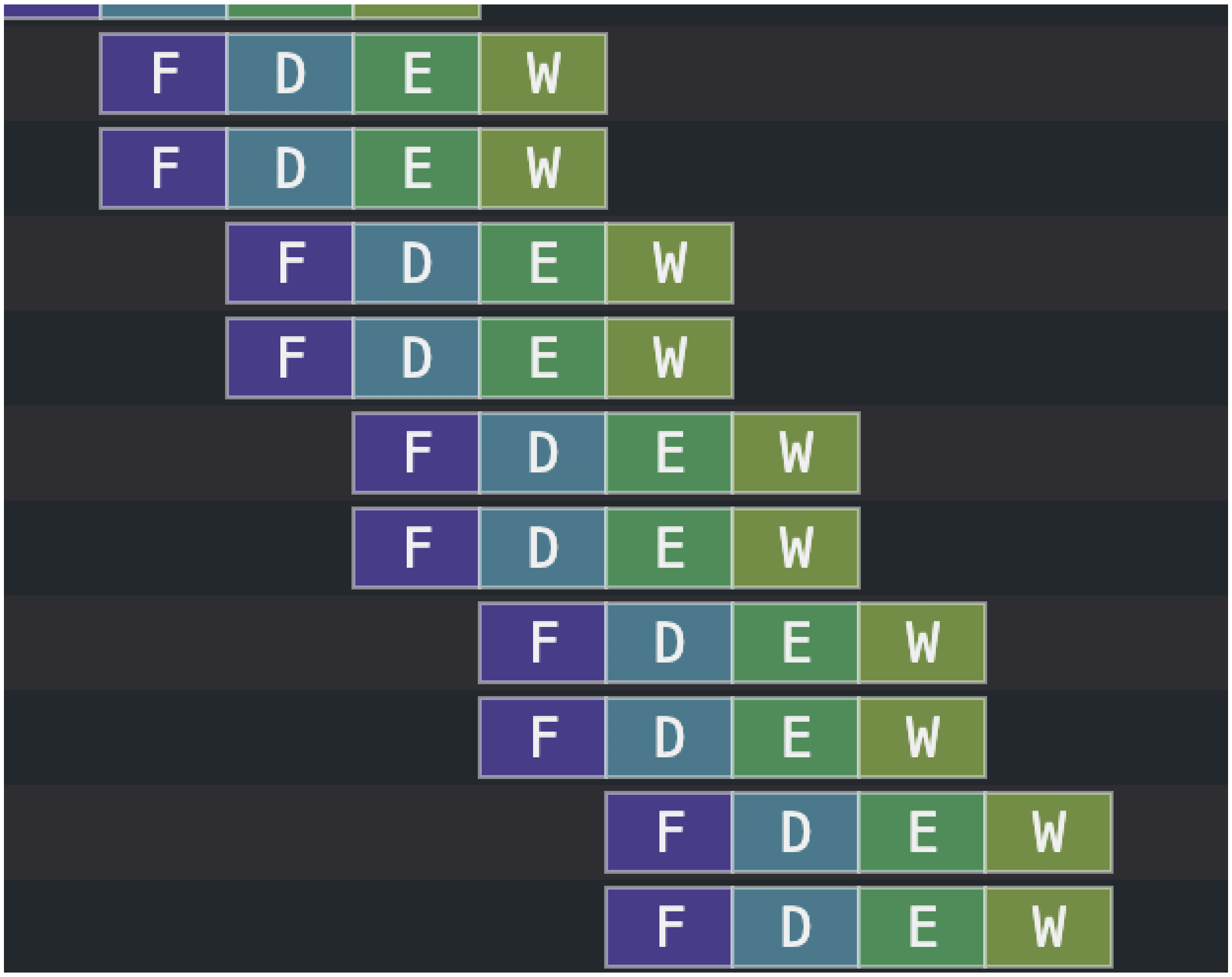


```
macOS
Warning: file 'mem.vmh' for memory 'bram_memory' has a gap at addresses 1065 to 2047.
Warning: file 'mem.vmh' for memory 'bram_memory' has a gap at addresses 2051 to 1073741823.
Warning: file 'memlines.vmh' for memory 'superCache_mainMem_bram_memory' has a gap at addresses 67 to 127.
Warning: file 'memlines.vmh' for memory 'superCache_mainMem_bram_memory' has a gap at addresses 129 to 67108863.
we [redacted] up in decode 2
from_fetch1: F2D { pc: 'h00001068, ppc: 'h0000106c, epoch: 'h1, k_id: 'h000000000033 }
from_fetch2: F2D { pc: 'h0000106c, ppc: 'h00001070, epoch: 'h1, k_id: 'h000000000034 }
response1: Mem { byte_en: 'h0, addr: 'h00001068, data: 'h00812e23 }
response2: Mem { byte_en: 'h0, addr: 'h00001070, data: 'hfea42623 }
RAN CYCLES       208
  PASS
```

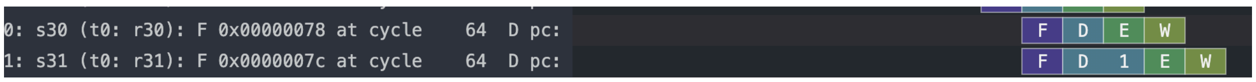
Another bug was the Konata fifo's(squashed and retired) since in a superscalar processor, we could be retiring twice in a cycle, but you can't write to a fifo twice in a cycle, and even if you switch it to a super-fifo you get weird results like this.



The fix was just having two fifo's for each of those and doing some slight tweaks to the Konata Helper Functions to account for two fifo's and two iids. Thats a high level rundown of our SuperScalar Processor and how we implemented it. Here are some screenshots proving its actually doing what we want it to do.

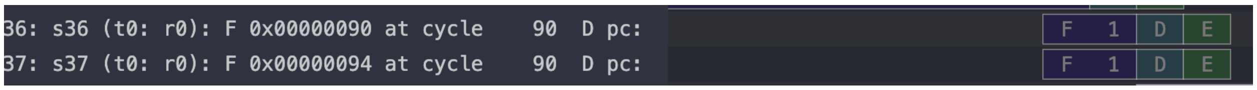


There's it processing two instructions at once when it gets a bunch of back to back doable ones with no dependencies.



```
0: s30 (t0: r30): F 0x00000078 at cycle    64  D pc:
1: s31 (t0: r31): F 0x0000007c at cycle    64  D pc:
```

Theres it stalling in the 2nd decode when there's a dependency.



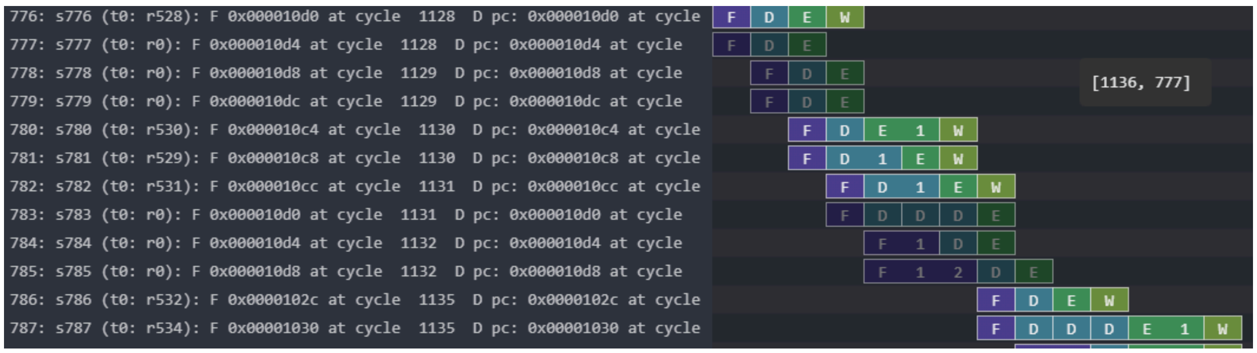
```
36: s36 (t0: r0): F 0x00000090 at cycle    90  D pc:
37: s37 (t0: r0): F 0x00000094 at cycle    90  D pc:
```

There's the squashing working properly(i.e. there's no world where the second instruction stays but the first one gets squashed). I’m not gonna show every single squash, but you can just look at this one and believe me.

Branch Prediction

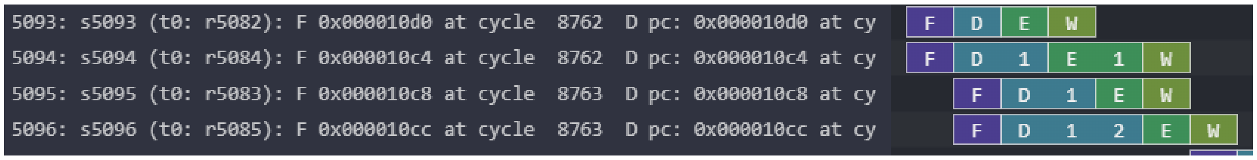
Branch Prediction, as one of the default upgrades one would want to add to their processor and a source of major, immediate performance boost (according to the TAs and other teams, at least), has been part of our plan since the beginning. However, due to the fact that we have to design our branch predictor and how it interacts with the cache based on whether we have a four-stage or seven-stage superscalar, we decided to wait until we finish the superscalar to start the branch prediction part.

With the seven-stage superscalar design mentioned above, featuring two decodes, two executes, and two writebacks, we decide that the best we can do is to add one BTB to facilitate both processing queues of the superscalar processor. Instead of superscalar-ing when pc and pc+4 are on the same cache line, the BTB enhanced processor does this when pc and the BTB-predicted next pc are on the same cache line. This does require us to change our instruction cache so that it can return two instructions on the same cache line when required to do so.



```
776: s776 (t0: r528): F 0x000010d0 at cycle  1128  D pc: 0x000010d0 at cycle
777: s777 (t0: r0): F 0x000010d4 at cycle  1128  D pc: 0x000010d4 at cycle
778: s778 (t0: r0): F 0x000010d8 at cycle  1129  D pc: 0x000010d8 at cycle
779: s779 (t0: r0): F 0x000010dc at cycle  1129  D pc: 0x000010dc at cycle
780: s780 (t0: r530): F 0x000010c4 at cycle  1130  D pc: 0x000010c4 at cycle
781: s781 (t0: r529): F 0x000010c8 at cycle  1130  D pc: 0x000010c8 at cycle
782: s782 (t0: r531): F 0x000010cc at cycle  1131  D pc: 0x000010cc at cycle
783: s783 (t0: r0): F 0x000010d0 at cycle  1131  D pc: 0x000010d0 at cycle
784: s784 (t0: r0): F 0x000010d4 at cycle  1132  D pc: 0x000010d4 at cycle
785: s785 (t0: r0): F 0x000010d8 at cycle  1132  D pc: 0x000010d8 at cycle
786: s786 (t0: r532): F 0x0000102c at cycle  1135  D pc: 0x0000102c at cycle
787: s787 (t0: r534): F 0x00001030 at cycle  1135  D pc: 0x00001030 at cycle
```

A few jumps and branches without BTB. THe processor, although superscalar-activated, struggles with short loops in particular due to the lack of correct next pc predictions. Notice that there are two jumps involved: 10d0 to 10c4 and 10cc to 102c.



```
5093: s5093 (t0: r5082): F 0x000010d0 at cycle  8762  D pc: 0x000010d0 at cy
5094: s5094 (t0: r5084): F 0x000010c4 at cycle  8762  D pc: 0x000010c4 at cy
5095: s5095 (t0: r5083): F 0x000010c8 at cycle  8763  D pc: 0x000010c8 at cy
5096: s5096 (t0: r5085): F 0x000010cc at cycle  8763  D pc: 0x000010cc at cy
```

A few jumps and branches with BTB. Notice how the processor superscalar-es 10d0 and 10c4 together since it knows that this branch will jump back to the previous instruction on the same line. Also notice (although poorly screenshot) that the next instruction, 102c, that 10cc should be jumping to, is directly pushed into the queue instead of wasting a few more instructions like in the example above.

The BTB we used here follows a 128-line design since more lines will generally cause the compiler to stall till infinity. This is caused by the fact that the BTBs are registers enacted in the main processor and are thus hugely inefficiency in terms of area, critical path length, and energy consumption. Nevertheless, lower line design does lead to a more inferior BTB due to the lack of capability to record more instructions within the same line (since more overwrite and thus BTB misses happen). Surprisingly, higher line designs (tested after often 10-20 minute compilations) lead to virtually no gain in performance. We anticipate that this is due to our cache geometry not matching the increasing line number.

Another behavior we observed is that, when we do not default to enqueuing a single instruction when the BTB superscalar check (pc and next pc on the same cache line) fails, but rather default to pc/pc+4 superscalar, and then to single instruction, the performance actually degrades compared to the simpler two-fold strategy (BTB superscalar or single instruction). Many times, this leads to even worse performance than a superscalar processor without the BTB. We anticipate that this is due to, once again, our I cache and BTB combination being unable to hold as many instructions as possible between the start and end of loops, compounded with the generally loop-less test suite (which is especially the case fo our MNIST example).

Cycle Performance Results:

	Pipelined	Superscalar	Superscalar + BTB
Add32	226	208	224
The Lie	52445	46690	43779
Morse	19487	19202	21269
Mat Mul	1235074	1084655	1061193

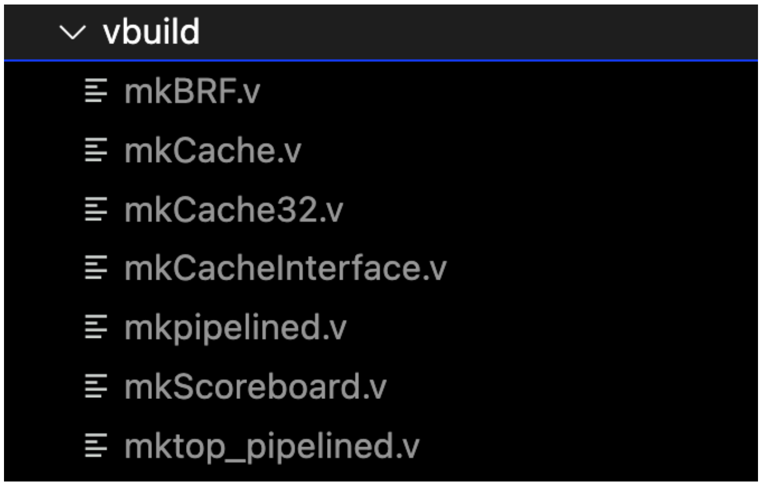
MNIST	646925	630117	648158

The spreadsheet shown above is the number of cycles the three types of processors need to complete four of the 6.1920 test suite tasks as well as our MNIST test, with the best performer highlighted, thus manifesting the performance gain we managed to record compared to the raw pipelined processor. These numbers are comparable against each other, but do not represent the most capable cycle performance of our processor since we kept many non-necessary yet processor-independent instructions like Kotana in there.

We can see that the superscalar managed to squeeze another 10-20% performance out compared to the pipelined processor. The improvement is especially notable in compute-intensive test cases such as Mat Mul, but not as much when memory/MMIO instructions are often sandwiched within (such as in MNIST and Morse.) BTB maintains a considerable edge when the program has a high number of short loops (such as in The Lie, which is mainly printing characters, and Mat Mul), but may lead to a performance drop in certain cases. This suggest space for improvement on the way our BTB conducts its prediction, as well as a potentially required redesign of the Superscalar framework to facilitate better cache coherence.

FPGA:

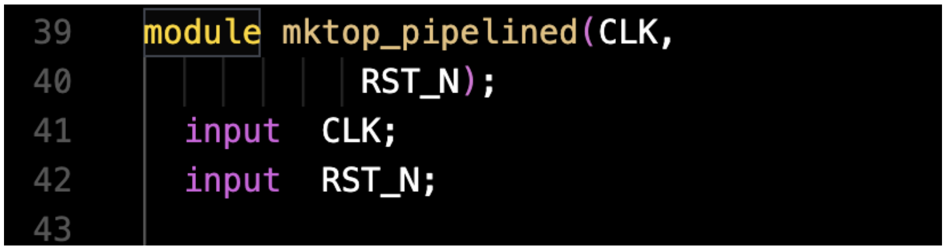
To get a sense of calculation time in real hardware, we wanted to synthesize our design onto an FPGA and compare the results to the simulation time. The first step was to take advantage of the Bluespec compiler's ability to generate Verilog files as the output, instead of the usual C++ files. Doing so resulted in many output and seemingly random modules, as follows:



Some examples of the output Verilog files after adding the -verilog flag

We eventually found that these were the modules that had the Bluespec compiler flag (*synthesize*) attached to it. In the future it would probably be helpful to add this flag to all modules but we left it as is for the time being.

Looking at mktop_pipelined.v, it was quickly apparent that some work would need to be done in order to make use of this processor. The processor only had two input signals, the clock and reset lines, and no outputs. This means that we would have no way of telling whether or not our processor was doing anything, nor if it was successful. To accommodate this, we had to make two major changes: make yet another top level file, and then change the IO of the processor module.



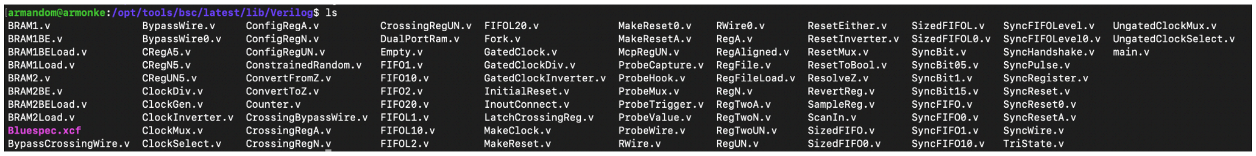
```
module mktop_pipelined(CLK,
                       RST_N);
  input  CLK;
  input  RST_N;

```

The module signature of our processor, fresh from Bluespec generation.

To make the first change, we made a new SystemVerilog file called top_level.sv that would be the front-facing module towards the Vivado compiler so that it could play nice with the XDC constraints and all the other requirements of the system. In theory we could have changed the mktop_pipelined structure to also work well with Vivado, but it would have required much more compiler massaging and this method was familiar from 6.205.

To make the second change, we did a bit of guessing and changed the interface of the mktop_pipelined module. It turns out Actions, ActionValues, and regular methods just translate quite well to the standard input and output lines of Verilog modules. By creating an Action method, we changed the signature to support input lines that would affect the state of internal registers. Once those were created, top_level.sv could simply wire the clock, reset, and IO lines into the processor, just like any other module.

Once the Processor what set up to control the FPGA's output, we tried to compile the design again, but Vivado was complaining about dependency issues. It turns out the Bluespec compiler expects the user to already support definitions of several modules, such as BRAM and FIFO modules. To resolve this, we eventually found that Bluespec has defined these already, and it is the programmer's job to import them into the project. Copying over the necessary modules, we finally got our first successful compile. At this point, the processor was solely capable of reporting the number of cycles executed.



```
armandom@armonke:/opt/tools/bsc/latest/lib/Verilog$ ls
BRAM1.v               BypassWire.v     ConfigRegA.v          CrossingRegUN.v  FIFOL20.v             MakeReset0.v    RWire0.v       ResetEither.v    SizedFIFOL.v   SyncFIFOLevel.v   UngatedClockMux.v
BRAM1BE.v             BypassWire0.v    ConfigRegN.v          DualPortRam.v    Fork.v                MakeResetA.v    RegA.v         ResetInverter.v  SizedFIFOL0.v  SyncFIFOLevel0.v  UngatedClockSelect.v
BRAM1BELoad.v         CRegA5.v         ConfigRegUN.v         Empty.v          GatedClock.v          McpRegUN.v      RegAligned.v   ResetMux.v       SyncBit.v      SyncHandshake.v   main.v
BRAM1Load.v           CRegN5.v         ConstrainedRandom.v   FIFO1.v          GatedClockDiv.v       ProbeCapture.v  RegFile.v      ResetToBool.v    SyncBit05.v    SyncPulse.v
BRAM2.v               CRegUN5.v        ConvertFromZ.v        FIFO10.v         GatedClockInverter.v  ProbeHook.v     RegFileLoad.v  ResolveZ.v       SyncBit1.v     SyncRegister.v
BRAM2BE.v             ClockDiv.v       ConvertToZ.v          FIFO2.v          InitialReset.v        ProbeMux.v      RegN.v         RevertReg.v      SyncBit15.v    SyncReset.v
BRAM2BELoad.v         ClockGen.v       Counter.v             FIFO20.v         InoutConnect.v        ProbeTrigger.v  RegTwoA.v      SampleReg.v      SyncFIFO.v     SyncReset0.v
BRAM2Load.v           ClockInverter.v  CrossingBypassWire.v  FIFOL1.v         LatchCrossingReg.v    ProbeValue.v    RegTwoN.v      ScanIn.v         SyncFIFO0.v    SyncResetA.v
Bluespec.xcf          ClockMux.v       CrossingRegA.v        FIFOL10.v        MakeClock.v           ProbeWire.v     RegTwoUN.v     SizedFIFO.v      SyncFIFO1.v    SyncWire.v
BypassCrossingWire.v  ClockSelect.v    CrossingRegN.v        FIFOL2.v         MakeReset.v           RWire.v         RegUN.v        SizedFIFO0.v     SyncFIFO10.v   TriState.v
```

The bluespec-implemented Verilog modules to be imported.

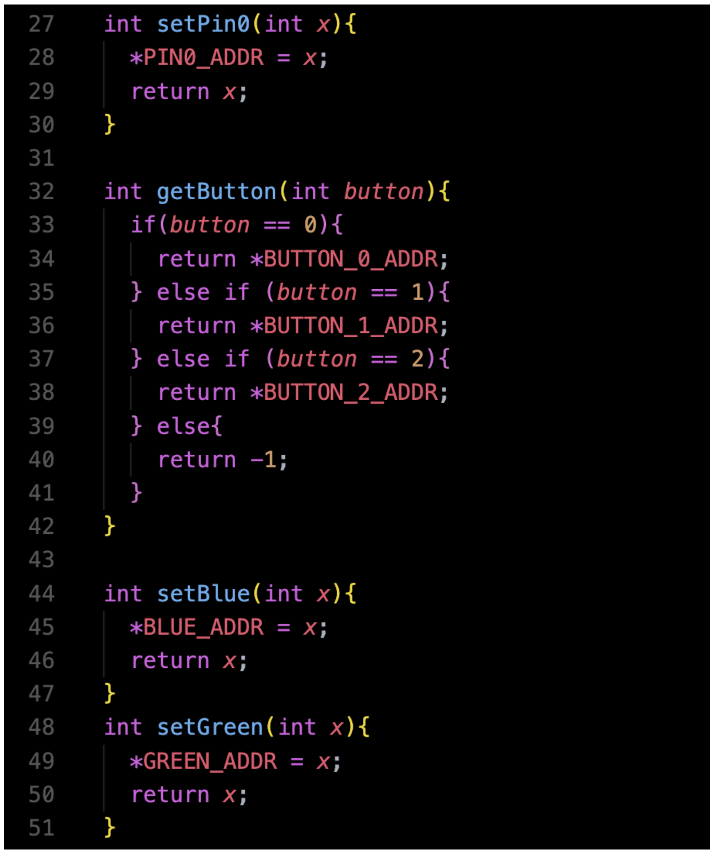
Still, however, the processor was reporting 0 as the number of cycles it could complete. It seemed unable to start moving for some unknown reason. Without any interface to understand what was going wrong with the FPGA, we resorted back to simulation. This time, however, we used the verilog files, not bluespec, and xsim instead of bluesim.

Xsim was unfortunately unable to parse the ‘@’ directives to point to memory, so we shrunk down the size of MainMemory and generated entire .vmh files that would be used to initialize memory. No 0s were left behind. Once we could display with simulation again, we found the reset lines were constantly resetting all register values. The bluespec modules had active low reset lines, despite the system being active high. After making the switch, we successfully performed our first simulation of our Processor.

The celebration was short lived as Vivado could not find a way to make our modules work with the timing constraints. We did not meet timing, resulting in faulty behavior. The solution was to use a slower clock, 20 MHz instead of the built in 100Mz. Generating a .v file from Vivado’s clock wizard, we succeeded in meeting timing.

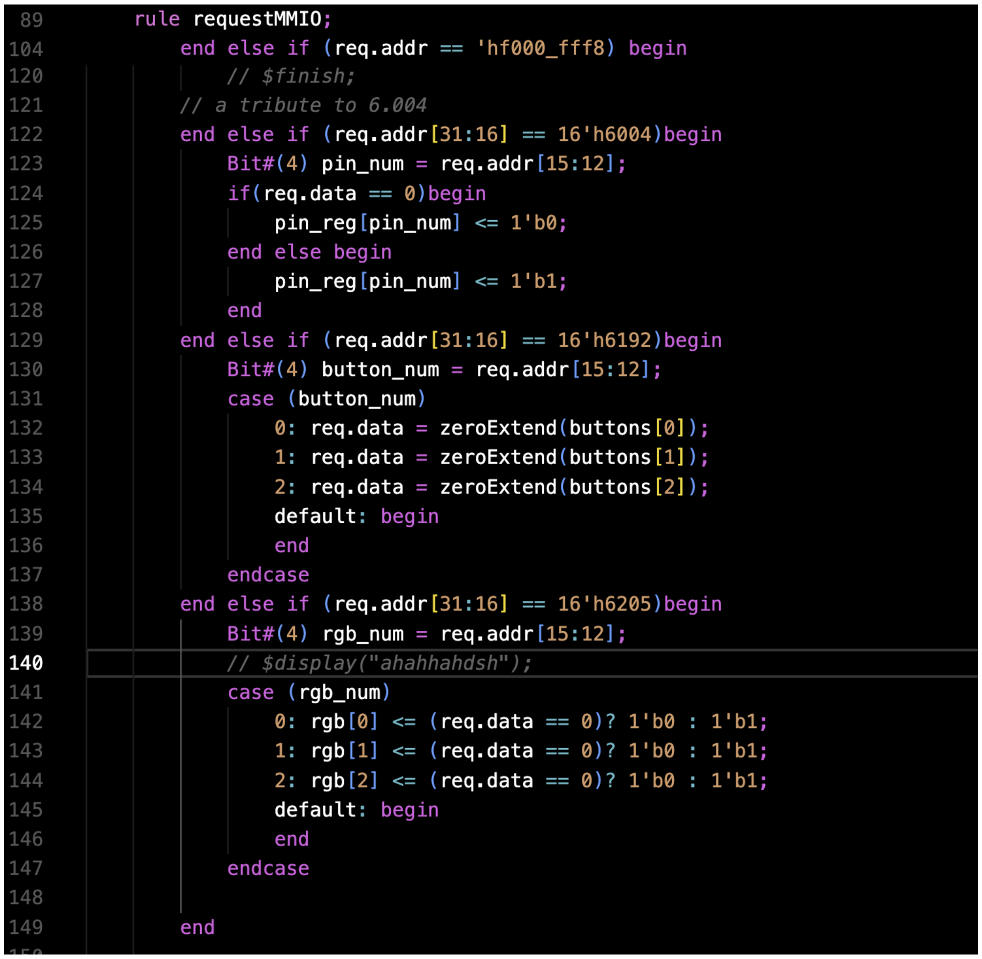
After a bit more wrestling with Vivado, we accomplished the ability to run any program on our FPGA. It was quite slow as we needed to rebuild the entire module every time a change to the C code was made. Regardless, we were able to take advantage of the current MMIO structure to allow our processor to use the GPIO, such as the LEDs on the seven segment display. The MNIST calculation was performed in a matter of milliseconds, much faster than we could ever dream of making the simulation. As a whole, the FPGA was a worthwhile experience, considering the amount of performance gain. However, for future reference, we would caution against choosing FPGA as an option for the project without a solid understanding of creating and debugging FPGA projects already.

All FPGA content was kept in a separate branch, as some fundamental changes, such as removal of all konata functionality, was required in order to allow for Vivado synthesis to complete. To view the version used for FPGA building, please see the fpga branch.



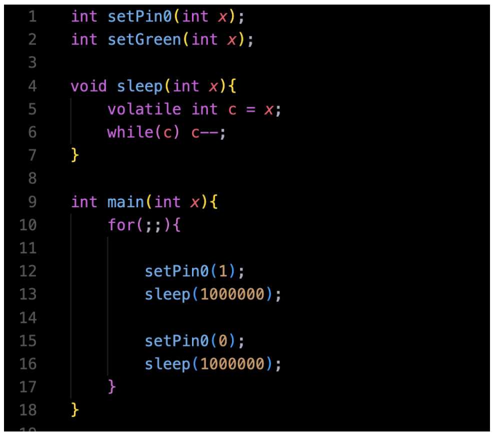
```c
int setPin0(int x){
  *PIN0_ADDR = x;
  return x;
}

int getButton(int button){
  if(button == 0){
    return *BUTTON_0_ADDR;
  } else if (button == 1){
    return *BUTTON_1_ADDR;
  } else if (button == 2){
    return *BUTTON_2_ADDR;
  } else{
    return -1;
  }
}

int setBlue(int x){
  *BLUE_ADDR = x;
  return x;
}
int setGreen(int x){
  *GREEN_ADDR = x;
  return x;
}
```

Mmio.c, following the structure the staff had already laid out for us



```
    rule requestMMIO;
        end else if (req.addr == 'hf000_fff8) begin
            // $finish;
        // a tribute to 6.004
        end else if (req.addr[31:16] == 16'h6004)begin
            Bit#(4) pin_num = req.addr[15:12];
            if(req.data == 0)begin
                pin_reg[pin_num] <= 1'b0;
            end else begin
                pin_reg[pin_num] <= 1'b1;
            end
        end else if (req.addr[31:16] == 16'h6192)begin
            Bit#(4) button_num = req.addr[15:12];
            case (button_num)
                0: req.data = zeroExtend(buttons[0]);
                1: req.data = zeroExtend(buttons[1]);
                2: req.data = zeroExtend(buttons[2]);
                default: begin
                end
            endcase
        end else if (req.addr[31:16] == 16'h6205)begin
            Bit#(4) rgb_num = req.addr[15:12];
            // $display("ahahhahdsh");
            case (rgb_num)
                0: rgb[0] <= (req.data == 0)? 1'b0 : 1'b1;
                1: rgb[1] <= (req.data == 0)? 1'b0 : 1'b1;
                2: rgb[2] <= (req.data == 0)? 1'b0 : 1'b1;
                default: begin
                end
            endcase

        end
```

Creating new MMIO addresses in the mktop_pipelined module.



```
int setPin0(int x);
int setGreen(int x);

void sleep(int x){
    volatile int c = x;
    while(c) c--;
}

int main(int x){
    for(;;){

        setPin0(1);
        sleep(1000000);

        setPin0(0);
        sleep(1000000);
    }
}
```

An example of a C program that blinks an LED on and off.