6.175 Final Project Report

Lasya Balachandran, Sanjay Seshan
lasyab@mit.edu, seshan@mit.edu

May 15, 2023

1 Introduction

In this paper, we present our process and implementation of 6.175 design project. We chose to implement a working dual core processor with a functioning cache hierarchy and parent protocol processor. For the second part, we decided to run our code on an AWS FPGA.

2 Cache and Processor

In labs 4 and 5 we implemented a single core 32-bit RISC-V processor and a single level cache. The first step of our project was to create a single core processor with a L1 cache. However, to merge our code we had a few shortcomings that had to be addressed.

(1) The original cache only stores lines, but the processor stores words

(2) The processor has two types of data in DRAM – instructions and data – that was handled by a 2 port BRAM, but our cache has to use a single ported main memory

(3) Our `mem.vmh` stored words, but we needed lines (16 words per line)

(4) Our cache interface was incompatible with the processor's request system

(5) The processor expects half words and byte storing

To start our conversion to words, we made the system handle four offset bits, rather than returning or writing a whole line. Requests to and fro main memory remain lines.

We used the following definition of an address to index into the cache.

```
function CacheReqWorking extract_bits(CacheLineAddr addr, CacheReq e);
  let tag = addr[31:13];
  IndexAddr index = addr[12:6];
  let offset = addr[5:2];
  return CacheReqWorking{tag:tag,idx:index,offset:offset,memReq:e};
endfunction
```

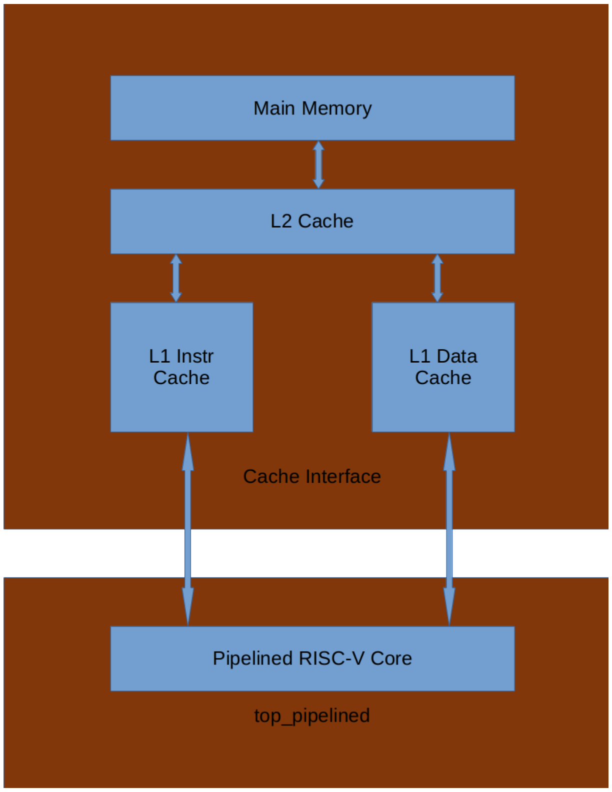
Furthermore, we created two BRAMs. This allows us to use the byte enable feature on the data while storing tags and miss/hit/valid bits separately.

```
BRAM1Port#(IndexAddr, CacheReqLine) bram1 <- mkBRAM1Server(cfg);
BRAM1PortBE#(IndexAddr, Vector#(16, Word), 64) bram2 <- mkBRAM1ServerBE(cfg2);
```

We chose to create the data bram as a vector of 16 words to easily index into a line using `line[offset]`. Furthermore, we treat the requested `word_byte` from the processor and shift it by the $offset * 4$ to write to the desired entry in the bram on a hit. The miss/store/load logic remains the same as lab 5. We also created a new Cache Interface that acts as a wrapper for cache requests from the processor.

```
interface CacheInterface;
    method Action sendReqData(CacheReq req);
    method ActionValue#(Word) getRespData();
    method Action sendReqInstr(CacheReq req);
    method ActionValue#(Word) getRespInstr();
endinterface
```

We created two caches – one for data and one for instructions. This interface converts to putFromProc, getToProc, etc. requests. To handle the getToMem request for each to a single memory, we created a FIFO queue to enqueue requests with a label (0 or 1) for type of data and return it to the appropriate cache. Our L2 cache is twice the size of the L1 ones. Main Memory is considered delayed by 20 cycles. We decided to keep our cache abstracted for both instructions and data to make instantiation and changes easier.



Next, we replaced all main memory requests in the processor to use this interface. Then, we made our system have a multi layer cache (L1 and L2). To do this, we used the lab5 code as is for the 2nd layer. L2 was connected to main memory. We converted the Beveren test to evaluate words to ensure coherency. The single core processor tests all ran at this point.

2.1 Word to Lines

To handle loading programs into main memory as lines, we used Martin's script from Piazza (`arrange_mem.py`) with a few notable bug fixes.

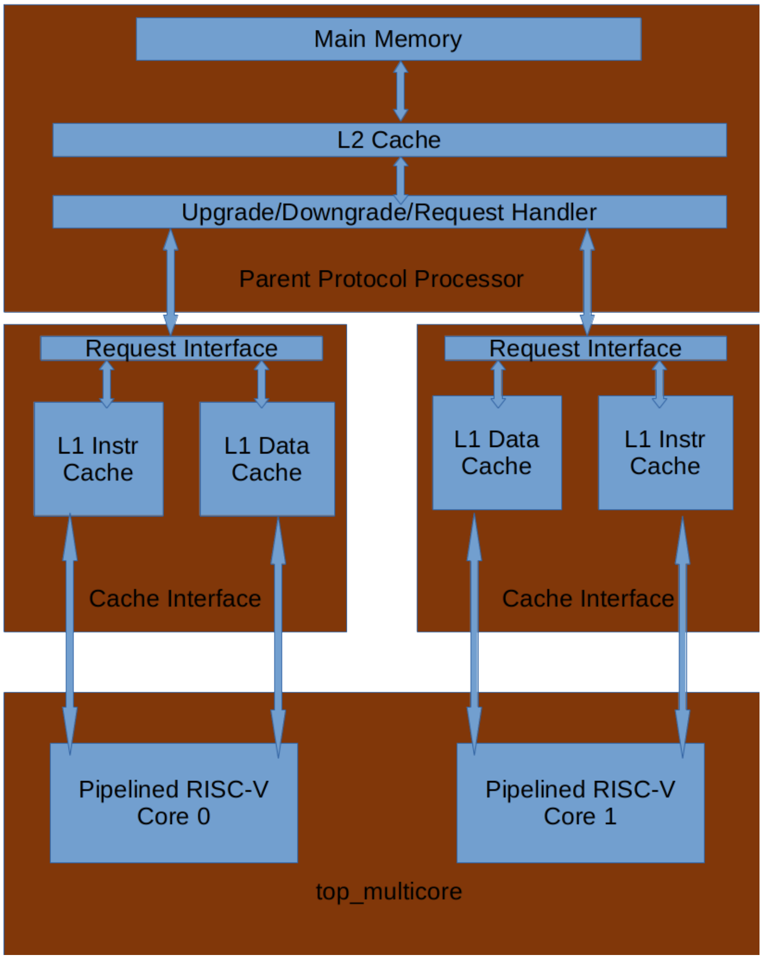
(1) We pad everything with zeroes to fix some issues with addressing in bluespec when there is only a single 0 on a line. We therefore pad with A's and 0's as appropriate

(2) We fix the handling of explicit memory addresses (starting with @) to handle hex values

```
def to_string(list_of_words):
  list_of_words.reverse()
  output = "".join(list_of_words)
  if not output:
      output = "0"
  list_of_words.clear()
  return "a"*(128-len(output)) + output + "\n"
.....
    for line in input:
    if ’@’ in line:
        if current_line:
            output.write(to_string(current_line))
        current_word = 0
        num = line[1:-2]
        output.write("@" + str(num) + "\n")
.....
```

3 Multi-core Processor

This section covers our implementation of a dual-core processor, its parent protocol processor, and its cache hierarchy.



3.1 Processor Requests

The processor aspect changed very little. Here, we simply instantiate two cores, but set one core to start at $pc = 0$ and the other to start at $pc = 4096$. Requests to cache/memory are handled separately for each. This setup allows us to run different code on each core and potentially add more cores.

3.2 Cache Updates

We instantiate one cache interface (each has a data and instruction cache) for each core. The Cache Interface now only handles the immediate requests, with higher levels abstracted to the parent protocol processor. Here, we have a new interface as a result.

```
interface CacheInterface;
    method Action sendReqData(CacheReq req);
    method ActionValue#(Word) getRespData();
    method  Action sendReqInstr(CacheReq req);
    method ActionValue#(Word) getRespInstr();
    method ActionValue#(MainMemReq) sendReq();
    method ActionValue#(CacheReq) upgrade();
    method Action downgrade(CacheReq req);
    method Action connectL2L1Cache(MainMemResp resp);
```

```
endinterface
```

We implement a `sendReq` to handle any existing requests to the L2 cache. The `upgrade` function forwards any data write requests to the Parent Protocol Processor (PPP). Downgrade receives the upgrades from other caches and processes them.

Notably, downgrades are handled slightly differently than normal write. Here, we only handle hit logic, because we only care to update old versions of data, not lack thereof. Furthermore, all writes are immediately enqueued to the Parent Protocol Processor to make sure future reads from other cores are getting the latest values.

3.3 Parent Protocol Processor

The Parent Protocol Processor is the sole part where the two cores interact with each other. Therefore, all data/cache coherency must be handled by the PPP. Upgrades are immediately forwarded to other caches in the upgrade/downgrade handler. Writes are also enqueued to the L2 cache to ensure the latest data is available to all cores. Any other requests for lower level evictions or loads for instructions and data are sent directly to the L2 cache.

The PPP takes in an input of the two core's cache interfaces which are both created by the processor.

```
module mkParentProtocolProcessor#(CacheInterface core1, CacheInterface core2)(
    ParentProtocolProcessor);
```

To handle multiple cores, we identify each core as 0 or 1. Any requests from each of these cores that are sent to L2 cache are tagged in a FIFO queue with a label so that when receiving a response, we return the data to the core in FIFO order.

Note that instruction/data labelling is still handled within the L1 cache to PPP interface. The PPP uses `connectL2L1Cache` to return the data from L2 for processing.

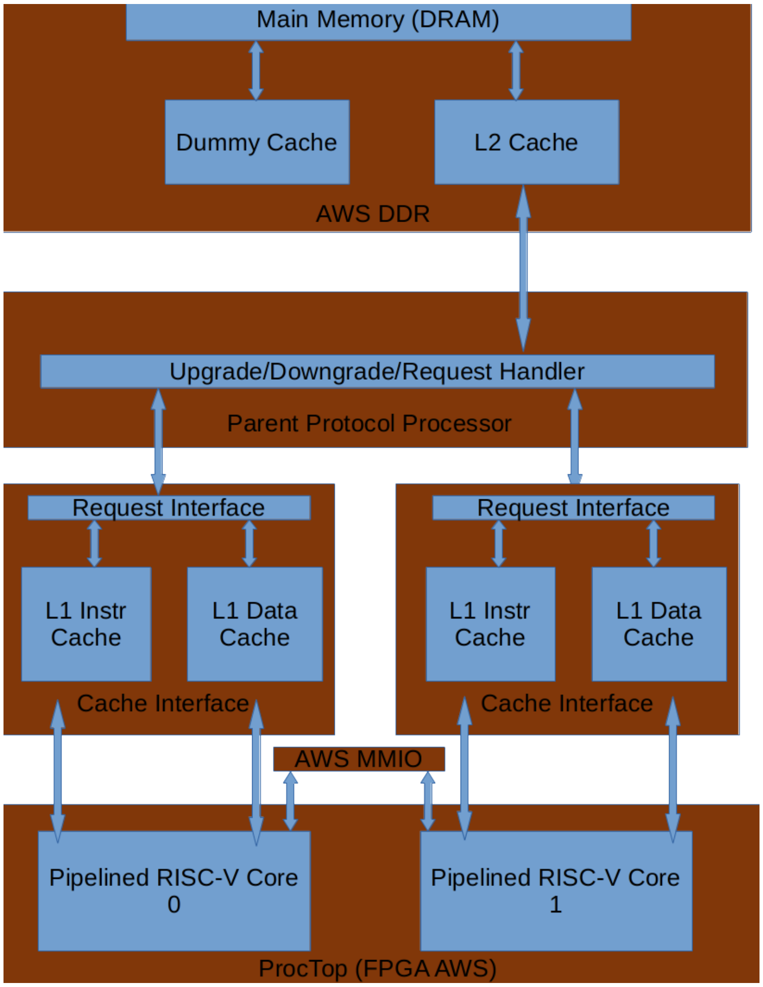
4 FPGA

This section covers our usage of the AWS FPGA server to run our program. Before we could run our code on the server, we had to make some changes to match the appropriate interfaces.

(1) The main memory worked through an AWS interface that expected an ICache and DCache

(2) The RISC-V cores needed a start method

(3) The MMIO was no longer emulated in the processor and was handled by the AWS interface



To fix these, we first abstracted the L2 cache out of the ParentProtocolProcessor and let it be part of the AWS DDR interface. We also used a single L2 cache for both cores, instructions and data, so we used only the L2 cache as part of the interface. We made a dummy cache for instructions. Then, we used FIFO ordering again to place MMIO requests and handle responses for each core. Finally, we instantiated both cores and their appropriate caches in the main file to run the programs.

```
let mmioIfc = (interface MMIOInput;
  interface ind = ind;
  method Bit#(32) getTick = tick._read;
endinterface);

let mmioHandler <- mkMMIOHandler(mmioIfc);

Cache cacheL2 <- mkDCache;
Cache cacheL2dummy <- mkDCache;

let memInput = (interface MemInput;
  interface ind = ind;
  method getTick = tick._read;
  interface dCache = cacheL2;
  interface iCache = cacheL2dummy;
  endinterface);
let memController <- mkMemController(memInput);

CacheInterface cache1 <- mkCacheInterface(0);
CacheInterface cache2 <- mkCacheInterface(1);
ParentProtocolProcessor ppp <- mkParentProtocolProcessor(cache1, cache2, ind, cacheL2);
```

```

RVIfc rv_core1 <- mkpipelined(0,0);
RVIfc rv_core2 <- mkpipelined(4096,1);
```

This initialization allows us to run our code in the AWS system. We successfully compiled our code on the server. However, we were not able to get it fully running in time, due to some memory addressing issues. We are in the process of debugging this with Miguel at the time of writing.

5 Tests & Evaluation

The testing of our code is broken up into modules, basic tests, and then our exhaustive evaluation.

5.1 Cache Evaluation

To test the cache, we modified Beveren to handle words and to run on two levels of caches. We made a nested cache as follows:

```
rule connectCacheL1L2;
  let lineReq <- cache.getToMem();
  cache2.putFromProc(lineReq);
endrule
rule connectL2L1Cache;
  let resp <- cache2.getToProc();
  cache.putFromMem(resp);
endrule
rule connectCacheDram;
  let lineReq <- cache2.getToMem();
  mainMem.put(lineReq);
endrule
rule connectDramCache;
  let resp <- mainMem.get;
  cache2.putFromMem(resp);
endrule
```

Note that our reference memory was made to load the program data and use Word size data blocks with 32-bit (really 30-bit by RISC-V specs) addressing.

```
// MAIN MEM FAST
BRAM_Configure cfg = defaultValue();
cfg.loadFormat = tagged Hex "mem.vmh";
BRAM1PortBE#(Bit#(30), Word, 4) bram <- mkBRAM1ServerBE(cfg);
DelayLine#(10, Word) dl <- mkDL(); // Delay by 10 cycles

//MAIN MEM
BRAM_Configure cfg = defaultValue();
cfg.loadFormat = tagged Hex "memlines.vmh";
BRAM1Port#(LineAddr, MainMemResp) bram <- mkBRAM1Server(cfg);
DelayLine#(20, MainMemResp) dl <- mkDL(); // Delay by 20 cycles
```

5.2 Simple Processor Tests

We began our tests by running the same code on both cores (basically the tests from lab4 with a new init.S provided by Thomas).

```
.section ".text.init0"
#  .text
  .globl _start
_start:
...
  li  x10,0
...
  li sp, 0xFFFFFF0 // Stack for Processor 0
  call main
  li x10, 0
  call exit
1:
  j 1b

  .align 6
.section ".text.init1"
#  .text
#  .align 6
...
  li  x10,1 // PUT 1, the core ID in register x10
...
  li sp, 0xF000000 // Stack for processor 1
  call main
  li x10, 0
  call exit
1:
  j 1b
```

This separates the stacks and calls `main(0)` on core0 and `main(1)` on core1.

All tests from lab4 ran correctly with this updated system to run the same program on both cores.

5.3 Basic Multicore Test

The provided multicore test runs two threads to ensure data access works between the two's writes and reads. In essence, both must pull the latest written data.

```
static volatile int input_data[8] = {0,1,2,3,4,5,6,7};
static volatile int flag = 0;
static volatile int acc_thread0 = 0;
void program_thread0(){
  for (int i = 0; i < 4; i++) {
      acc_thread0 += input_data[i];
  }
  char *p;
  while (flag == 0); // Wait until thread1 produced the value
  if (flag + acc_thread0 == 28) {
      for (p = s; p < s + 8; p++) putchar(*p);
  } else {
      for (p = f; p < f + 8; p++) putchar(*p);
  }
}

void program_thread1(){
  int a = 0;
   for (int i = 0; i < 4; i++){
      a += input_data[4+i];
   }
  flag = a;
  while(1);
}
```

In this test, we see we have three variables instantiated in shared memory. These are edited by each thread for inter-thread communication. This test makes sure that a write in thread 1 can be read in thread 0 before continuing. This test works successfully.

5.4 Distributed Matrix Multiply

Our prime test of evaluation is our matrix multiply distributed across two cores. We started with the single core matrix multiply and split the calculations across two cores.

This is the original code of interest where the entire 16x16 matrix is calculated sequentially.

```c
int sum;
for (int i = 0; i < 16; i++)
{
    for (int j = 0; j < 16; j++)
    {
        sum = 0;
        for (int k = 0; k < 16; k++)
        {
            sum += multiply(a[i][k], b[k][j]);
        }
        putchar(sum);
        c[i][j] = sum;
    }
}
if (arrEquals(expected, c))
{
    exit(0);
}
else
{
    exit(1);
}
```

We converted this code to calculate the first and second 8 rows simultaneously.

```c
void program_thread0(){
  int sum;
  for (int i = 0; i < 8; i++)
  {
      for (int j = 0; j < 16; j++)
      {
          sum = 0;
          for (int k = 0; k < 16; k++)
          {
              sum += multiply(a[i][k], b[k][j]);
          }
          c[i][j] = sum;
          putchar(sum);
      }
  }

  while (flag == 0); // Wait until thread1 produced the value
  if (arrEquals(expected, c))
  {
      exit(0);
  }
  else
  {
      exit(1);
  }

}
```

```


void program_thread1(){
  int sum;
  for (int i = 8; i < 16; i++)
  {
      for (int j = 0; j < 16; j++)
      {
          sum = 0;
          for (int k = 0; k < 16; k++)
          {
              sum += multiply(a[i][k], b[k][j]);
          }
          c[i][j] = sum;
          putchar(sum);
      }
  }
  // mostly since the C compiler is too smart to do flag = 1 or something
  int l = 0;
  for (int i = 0; i < 4; i++){
      l += i;
  }
  flag = 1;
  while(1);
}
```

This code has thread0 compute the first 8 then wait for thread1 to finish the last 8 before checking that it is correct. The complicated flag code in thread1 is there since the compiler is smart and removes the unnecessary write when just setting to 1.

We compare the cycles it takes to run this program on the multicore processor to running to the original on the single core + cache processor.

On the multicore processor, it ran in 2min 22s with 2673485 cycles. The single core did the same calculations in 2min 43s with 5466137 cycles. Note that the time difference is not that different due to the simulator running everything sequentially in effect. However, the number of cycles is roughly half, as expected.

This would hold for any distributed program. Furthermore, running multiple, non-conflicting programs, can be done at the same time, which could only be done sequentially before.

6 Difficulties

TO DO

7 Sources

http://csg.csail.mit.edu/6.175/labs/project-part2.html
Course material from Canvas