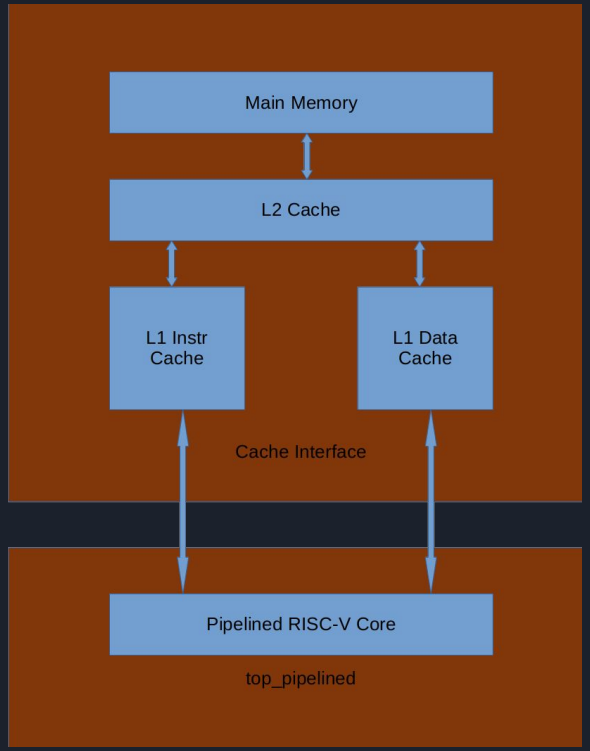
# Multi-core Processor on FPGA

Lasya Balachandran and Sanjay Seshan

6.1920[6.175] Final Project

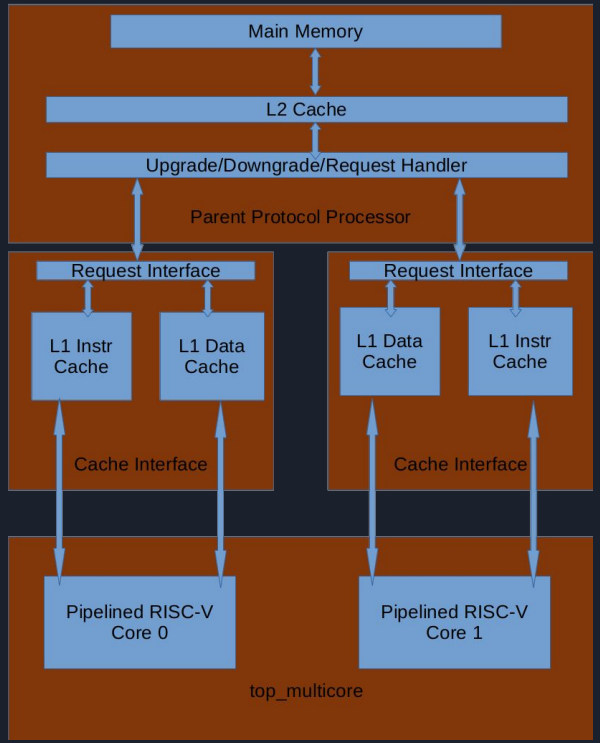
# Connecting Cache to the Processor

- Converted cache to store words
  - Used 4 offset bits to correspond to 16 words per line
- Used two BRAMs  for the cache
  - One with tag and valid
  - One byte enabled for data
- Created two caches (data and instruction)
- Requests for instructions and data are tagged in a queue when sending to L2 to return to the cache in FIFO order
- 2 level cache hierarchy connects to main memory
  - Built an interface between processor and cache
- Main memory and L2 cache store lines
- Evaluated using modified Beveren tests



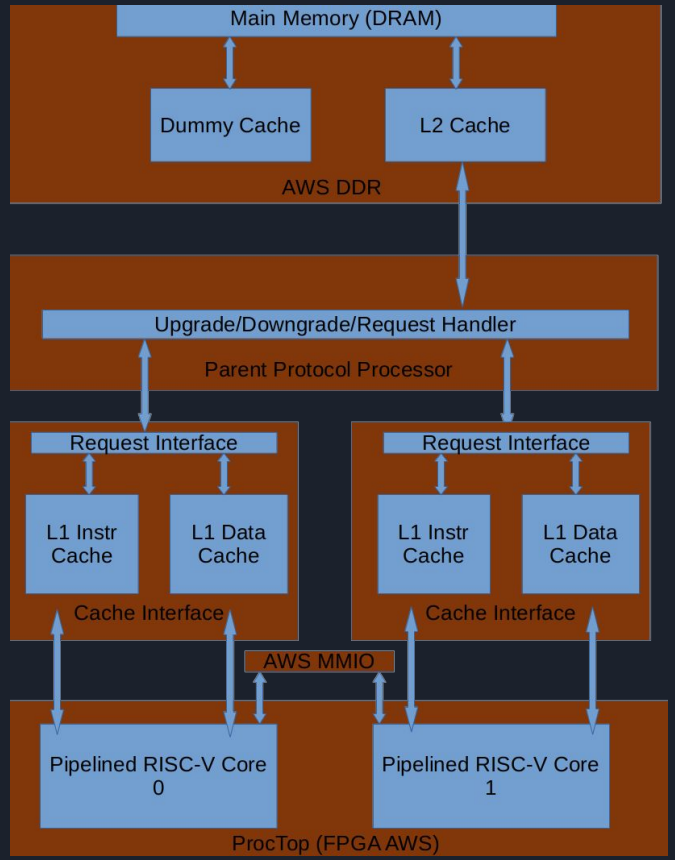
# Implementing Multi-Core Processor

- Created a Parent Protocol Processor
  - Routes upgrades and downgrades between cores
  - Abstracted the L2 cache and main memory into the PPP
  - PPP handles requests for evictions and loads into higher cache levels
  - Shared L2 cache stores writes for accessing by other cores
  - Tag core requests to memory to return FIFO order
- System is modular
- Two cores are instantiated
  - First starts at pc=0
  - Second starts at pc=4096
- Each core gets its own set of data and instruction caches



# Running on FPGA

- We modified our code to run within the AWS DDR system
- Restructured the cache hierarchy a bit to match
- Had to use a dummy cache to use a single L2 cache with no separate instruction L2 cache
- MMIO was also abstracted into the AWS module and was shared by the cores by using a FIFO tag on requests
- Only managed to get a few tests working in time but results are promising



# Results – Multicore

- Standard tests run same code on both cores which all work
- New tests ensures coherence between core's caches
- Created a distributed matrix multiply to display working shared data and speed up from two threads
- Each core produced half of the rows
- Thread1 notifies Thread0 when it finishes
- **Single core ran with 5.47 million cycles** (simulator 2m 43s)
- **Dual core ran with 2.67 million cycles** (simulator 2m 22s)
  - Roughly 2x cycle speed up
  - Simulator has overhead and does not really utilize multi core processing on the host

```
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 = l;
  while(1);
}
```

# Results – FPGA

- Many of the single core tests worked on the FPGA
- Remaining tests are a work in progress
- Clock speeds are shown at the right and are promising

```
------------------------------------------------------------
| Clock Summary
| -------------
------------------------------------------------------------

Clock                 Waveform(ns)          Period(ns)      Frequency(MHz)
-----                 ------------          ----------      --------------
CLK_300M_DIMM0_DP     {0.000 1.666}         3.332           300.120
  mmcm_clkout0        {0.000 1.874}         3.749           266.773
    pll_clk[0]        {0.000 0.234}         0.469           2134.187
      pll_clk[0]_DIV  {0.000 1.874}         3.749           266.773
    pll_clk[1]        {0.000 0.234}         0.469           2134.187
      pll_clk[1]_DIV  {0.000 1.874}         3.749           266.773
    pll_clk[2]        {0.000 0.234}         0.469           2134.187
      pll_clk[2]_DIV  {0.000 1.874}         3.749           266.773
  mmcm_clkout6        {0.000 3.749}         7.497           133.387
refclk_100            {0.000 5.000}         10.000          100.000
  qpll1outclk_out[3]  {0.000 0.100}         0.200           5000.001
    txoutclk_out[15]  {0.000 1.000}         2.000           500.000
      clk_core        {0.000 2.000}         4.000           250.000
        clk_main_a0   {0.000 4.000}         8.000           125.000
        tck           {0.000 16.000}        32.000          31.250
```

# Challenges

- Handling half words and bytes took some restructuring
- Many revisions and debugging to ensure PPP handles correct cache coherency
- Some issues turning words to lines in mem.vmh – needed padding of zeroes
- Unexpected loading values in the cache – had to zero out the valid/dirty bits

# Thanks for listening!