ECE 5745 PARCv2 Accelerator Tutorial

The infrastructure for the ECE 5745 lab assignments and projects has support for implementing medium-grain accelerators. Fine-grain accelerators are tightly integrated within the processor pipeline (e.g., a specialized functional unit for bit-reversed addressing useful in implementing an FFT), while coarse-grain accelerators are loosely integrated with a processor through the memory hierarchy (e.g., a graphics rendering accelerator sharing the last-level cache with a general-purpose processor). Medium-grain accelerators are often integrated as co-processors: the processor can directly send/receive messages to/from the accelerator with special instructions, but the co-processor is relatively decoupled from the main processor pipeline and can also independently interact with memory.

This tutorial will use the vector-vector-add (vadd) microbenchmark as an example. We will explore the area and timing of a baseline PARCv2 pipelined processor and the energy and performance when this processor is used to execute a pure-software version of the vadd microbenchmark. We will then implement a vadd accelerator, integrate it with the PARCv2 pipelined processor and determine the potential benefit of hardware acceleration for this simple microbenchmark. This tutorial assumes you have already completed the "basic" ECE 5745 tutorials on Linux, Git, PyMTL, and Verilog, as well as the new ECE 5745 ASIC tutorial.

The first step is to source the setup script, clone this repository from GitHub, and define an environment variable to keep track of the top directory for the project.

```bash
% source setup-ece5745.sh
% mkdir $HOME/ece5745
% cd $HOME/ece5745
% git clone git@github.com:cornell-ece5745/ece5745-tut-xcel
% cd ece5745-tut-xcel
% TOPDIR=$PWD
```

Baseline PARCv2 Processor FL and RTL Models

The following figure illustrates the overall system we will be using with our PARCv2 processors. The processor includes eight latency insensitive val/rdy interfaces. The mng2proc/proc2mng interfaces are used for the test harness to send data to the processor and for the processor to send data back to the test harness. The imemreq/imemresp interfaces are used for instruction fetch, and the dmemreq/dmemresp interfaces are used for implementing load/store instructions. The xcelreq/xcelresp interfaces are used for the processor to send messages to the accelerator. The mng2proc/proc2mng and memreq/memresp interfaces were all introduced in ECE 4750. For now we will largely ignore the accelerator, and we will defer discussion of the xcelreq/xcelresp interfaces to later in this tutorial.
We provide two implementations of the PARCv2 processor. The FL model in ProcFL is essentially an instruction-set-architecture (ISA) simulator; it simulates only the instruction semantics and makes no attempt to model any timing behavior. As a reminder, the PARCv2 instruction set is defined here:

- http://www.csl.cornell.edu/courses/ece5745/handouts/ece5745-parc-isa.txt

The RTL model in ProcAltRTL is almost identical to the alternative design for lab 2 in ECE 4750. It is a five-stage pipelined processor that implements the PARCv2 instruction set and includes full bypassing/forwarding to resolve data hazards and a variable-latency iterative multiplier. The datapath diagram for the processor is shown below.

We can start by running all of the unit tests on both the FL and RTL processor models:

```bash
% mkdir -p $TOPDIR/pymtl/build
% cd $TOPDIR/pymtl/build
% py.test ../proc
```

See the handout for lab 2 from ECE 4750 for more information about how we use py.test and the mgr2proc/proc2mng interfaces to test the PARCv2 processor.
Cross-Compiling and Executing PARCv2 Microbenchmarks

We will write our microbenchmarks in C. Take a closer look at the vvadd microbenchmark which is located here:

```c
% cd $TOPDIR/app/ubmark
% more ubmark-vvadd.c
<snip>
__attribute__((noinline))
void vvadd_scalar( int *dest, int *src0, int *src1, int size )
{
    for ( int i = 0; i < size; i++ )
        dest[i] = src0[i] + src1[i];
}
<snip>
int main( int argc, char* argv[] )
{
    int dest[size];
    for ( int i = 0; i < size; i++ )
        dest[i] = 0;

    test_stats_on();
    vvadd_scalar( dest, src0, src1, size );
    test_stats_off();

    verify_results( dest, ref, size );
    return 0;
}
```

The src0, src, and ref arrays are all defined in the ubmark-vvadd.dat file. The microbenchmark first initializes the destination array to be all zeros, turns stats on, does the actual vvadd computation, turns stats off, and finally verifies that the results are as expected. We need the test_stats_on() and test_stats_off() functions to make sure we can keep track of various statistics (e.g., the number of cycles) only during the important part of the microbenchmark. We do not want to count time spent in initialization or verification when comparing the performance of our various microbenchmarks. These two functions are defined in app/common/common-misc.h as follows:

```c
inline void test_stats_on()
{
    int status = 1;
    asm( "mtc0 %0, $21;" :: "r" (status) )
}

inline void test_stats_off()
{
    int status = 0;
    asm( "mtc0 %0, $21;" :: "r" (status) )
}
```

We are using the GCC inline assembly extensions to enable us to directly insert a specific assembly instruction into our C code. You can find out more about inline assembly syntax here:


At a high level, %0 acts as a place holder for whatever register specifier the compiler ends up allocating for the status variable. The PARCv2 instruction set defines the co-processor 0 register number 21 as the stats_en register, which is why we use $21 in the inline assembly. The idea is that the microarchitecture and/or simulator can monitor for writes to the stats_en register to determine when to start and stop keeping statistics. For more on writing microbenchmarks, please review the handout for lab 5 from ECE 4750.
We have a build system that can compile these microbenchmarks natively for x86 and can also cross-compile these microbenchmarks for PARCv2 so they can be executed on our simulators. When developing and testing microbenchmarks, we should always try to compile them natively to ensure the microbenchmark is functionally correct before we attempt to cross-compile the microbenchmark for PARCv2. Debugging a microbenchmark natively is much easier compared to debugging a microbenchmark on our simulators. Here is how we compile and execute the pure-software vadd microbenchmark natively:

```
% cd $TOPDIR/app
% mkdir build-native
% cd build-native
% ../configure
% make ubmark-vvadd
% ./ubmark-vvadd
```

The microbenchmark should display `passed`. Once you are sure your microbenchmark is working correctly natively, you can cross-compile the microbenchmark for PARCv2.

```
% cd $TOPDIR/app
% mkdir build
% cd build
% ../configure --host=maven
% make ubmark-vvadd
```

This will create a `ubmark-vvadd` binary which contains PARCv2 instructions and data. You can disassemble a PARCv2 binary (i.e., turn a compiled binary back into an assembly text representation) with the `maven-objdump` command like this:

```
% maven-objdump -dC ubmark-vvadd | less
0000168 <vvadd_scalar(int*, int*, int*, int)*>
  1168:  18e0000c  blez a3, 119c
  116c:  00001021  move v0, zero
  1170:  00001821  move v1, zero
  1174:  00c24021  addu a5, a2, v0  # <-
  1178:  00a24021  addu a4, a1, v0  # |
  117c:  8d2a0000  lw  a6, @a5  # |
  1180:  8d090000  lw  a5, @a4  # |
  1184:  24630001  addiu v1, v1, 1  # |
  1188:  00824021  addu a4, a0, v0  # |
  118c:  01494021  addu a5, a6, a5  # |
  1190:  ad090000  sw  a5, @a4  # |
  1194:  24420004  addiu v6, v6, 4  # |
  1198:  1467ffe6  bne v1, a3, 1174  # --'
  119c:  03e00008  jr  ra

00001210 <main>:

<snip>
  1248:  00001021  move v0, zero
  124c:  24420001  addiu v0, v0, 1
  1250:  ac000000  sw  zero, @v1
  1254:  24630004  addiu v1, v1, 4
  1258:  1447ffe6  bne v0, a3, 124c
  125c:  24020001  li  v0, 1
  1260:  4082a800  mtc0  v0, c0_staten
  1278:  3c050002  lui  a1, 0x2
  127c:  3c060002  lui  a2, 0x2
  1280:  2a45eb50  addiu a1, a1, -5296
  1284:  24c6ec0  addiu a2, a2, -4896
  1288:  82002021  move a0, s0
  128c:  8c00045a  jal  1168 <vvadd_scalar(int*, int*, int*, int)>
  1290:  00001021  move v0, zero
  1294:  4082a800  mtc0  v0, c0_staten
  12ac:  8f860004  lw  a2, -32764(gp)
```
Recall that you can search with `less` by simply pressing the forward slash key and typing in your search term. So you can find the assembly code for `vadd_scalar` or `main` functions by using `less` to search for either `vadd_scalar` or `main`. You can also redirect the output from `maven-objdump` to a text file for viewing with your favorite text editor. The disassembly shows the address, bytes, and assembly text for each instruction in the binary.

You can see the MTC0 instructions to set and clear the `stats_en` bit have been inserted in the `main` function around the call to `vadd_scalar`. The assembly code for the actual `vadd_scalar` function should look familiar to ECE 4750 students. I have added some comments to show the backwards branch for the `vadd` loop. The loop has 10 instructions. Four instructions do useful work (i.e., two LW instructions, the actual ADDU instruction, one SW instruction), three ADDU instructions generate the array addresses, one ADDIU instruction bumps the array offset, one ADDIU instruction increments the loop counter, and the BNE instruction implements the loop control flow.

We have provided you with a simulator that composes a processor, memory, and accelerator and is capable of executing PARLv2 binaries. The simulator enables flexibly choosing the processor implementation (FL vs. RTL) and the type and implementation of the accelerator. By default, the simulator uses the processor FL model and a null accelerator which we will discuss later. So let’s execute the `vadd` PARLv2 binary on the instruction-set simulator:

```
% cd $TOPDIR/pymlt/build
% ../pmx-sim ../app/build/ubmark-vvadd
```

After a few seconds the simulator should display `passed` which means the microbenchmark successfully executed on the ISA simulator. The `--trace` command line option will display each instruction as it is executed on the ISA simulator.

```
% cd $TOPDIR/pymlt/build
% ../pmx-sim --trace ./app/build/ubmark-vvadd > ubmark-vvadd-fl.trace
```

When dumping out large line traces, it is usually much faster to save them to a file and then open the file in your favorite text editor. You can search in the line trace for the MTC0 instruction to quickly jump to where the actual `vadd_scalar` function starts executing.

Here is what the line trace looks like for one iteration of the `vadd` loop:

<table>
<thead>
<tr>
<th>PC</th>
<th>instruction</th>
<th>xcel</th>
<th>imemreq</th>
<th>imemresp</th>
<th>dmemreq</th>
<th>dmemresp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1541:</td>
<td>00001174 addu r09, r06, r02</td>
<td>()</td>
<td>()</td>
<td>()</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1542:</td>
<td></td>
<td>()</td>
<td>rd:00:0001178()</td>
<td>()</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1543:</td>
<td></td>
<td>()</td>
<td>()</td>
<td>rd:00:00a24021</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1544:</td>
<td>00001178 addu r08, r05, r02</td>
<td>()</td>
<td>()</td>
<td>()</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1545:</td>
<td></td>
<td>()</td>
<td>rd:00:000117c()</td>
<td>()</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1546:</td>
<td></td>
<td>()</td>
<td>()</td>
<td>rd:00:08d2a000</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1547:</td>
<td>#</td>
<td>()</td>
<td>()</td>
<td>rd:00:001ece4</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1548:</td>
<td>#</td>
<td>()</td>
<td>()</td>
<td>rd:00:000000e</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1549:</td>
<td>0000117c lw  r10, 0000(r09)</td>
<td>()</td>
<td>()</td>
<td>()</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1550:</td>
<td></td>
<td>()</td>
<td>rd:00:0001180()</td>
<td>()</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1551:</td>
<td></td>
<td>()</td>
<td>()</td>
<td>rd:00:08d90000</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1552:</td>
<td>#</td>
<td>()</td>
<td>()</td>
<td>rd:00:001eb54</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1553:</td>
<td>#</td>
<td>()</td>
<td>()</td>
<td>rd:00:000000e</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1554:</td>
<td>00001180 lw  r09, 0000(r08)</td>
<td>()</td>
<td>()</td>
<td>()</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1555:</td>
<td></td>
<td>()</td>
<td>rd:00:0001184()</td>
<td>()</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1556:</td>
<td></td>
<td>()</td>
<td>()</td>
<td>rd:00:02463001</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1557:</td>
<td>00001184 addiu r03, r03, 0001</td>
<td>()</td>
<td>()</td>
<td>()</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1558:</td>
<td></td>
<td>()</td>
<td>rd:00:0001188()</td>
<td>()</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1559:</td>
<td></td>
<td>()</td>
<td>()</td>
<td>rd:00:000824021</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1560:</td>
<td>00001188 addu r08, r04, r02</td>
<td>()</td>
<td>()</td>
<td>()</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1561:</td>
<td></td>
<td>()</td>
<td>rd:00:000118c()</td>
<td>()</td>
<td>()</td>
<td>()</td>
</tr>
</tbody>
</table>
Since this is an ISA simulator, instructions can functionally execute in a single cycle, although technically they take multiple cycles to interact with the memory system. These cycles are not really modeling any kind of realistic timing, but can instead be thought of as the "steps" required for functional simulation.

Now that we have verified the microbenchmark works correctly on the ISA simulator, we can run the microbenchmark on the baseline PARCv2 pipelined processor RTL model:

```bash
% cd $TOPDIR/pymtl/build
% ../pmx/pmx-sim --proc-impl rtl --stats ../..app/build/ubmark-vvadd
num_cycles = 1217
```

We use the --proc-impl command line option to choose the processor RTL model. The reported number of cycles is only when stats are enabled. You can use the --trace command line option to understand how the processor is performing in more detail.

```bash
% cd $TOPDIR/pymtl/build
% ../pmx/pmx-sim --proc-impl rtl --trace ../..app/build/ubmark-vvadd > ubmark-vvadd-rtl.trace
```

This is the line trace for one iteration of the vvadd loop (without the memory system):

```
F stage  D stage          M   X   W
-----------------------------
00001178|addu    r09, r06, r02  | | |  bne   |
0000117c|addu    r08, r05, r02  |addu| | |   |
00001180|lw      r10, 0000(r09) |addu|addu| |   |
00001184|lw      r09, 0000(r08) |lw |addu|addu| |
00001188|addiu   r03, r03, 0001 |lw |lw |addu|   |
0000118c|addu    r08, r04, r02  |addiu|lw |lw |   |
00001190|addu    r09, r10, r09  |addu|addiu|lw | |
00001194|sw      r09, 0000(r08) |addu|addu|addiu|
00001198|addiu   r02, r02, 0004 |sw |addu|addu| |
0000119c|bne     r03, r07, fff6 |addiu|sw |addu| |
~ | | |   |bne |addiu|sw | |
00001174|       |   |bne |addiu|
```

Each iteration takes 12 cycles and has a CPI of 1.2 (10 instructions, two bubbles for the branch resolution latency). The microbenchmark is for arrays with 100 elements, so 12*100 = 1200 which is about what we see in terms of the overall cycle count for this microbenchmark.

**PARCv2 Processor ASIC**
We can now push the baseline PARCv2 pipelined processor through the ASIC flow to better characterize the area and timing of the processor as well as the energy when running the vpadd microbenchmark. We can use the pmx-sim simulator to translate the processor into Verilog and to generate the VCD file we need for energy analysis.

% cd $TOPDIR/pyrtl/build
% ../pmx/pmx-sim --proc-impl rtl --translate --dump-vcd ../..//app/build/ubmark-vpadd
% ls
<br>ProcMemXcel_null_rtl.v
pmx-sim-null-rtl-ubmark-vpadd.verification1.vcd

Note that the generated Verilog does not include a suffix hash. This is because by the ProcMemXcel wrapper model which instantiates the processor, cache, and accelerator takes the types of these components as parameters. Due to PyMTL implementation details, this causes PyMTL to generate a different suffix hash every time we run the simulator which can be frustrating since it means we need to constantly change our Makefrag to push the right version through the ASIC flow.

We explicitly tell PyMTL what to name the corresponding top-level Verilog file by setting the special explicit_modulename member variable. For example, this is what ProcMemXcel looks like:

class ProcMemXcel ( Model ):
def __init__ ( self, modulename, ProcModel, MemModel, XcelModel ):
    self.explicit_modulename = modulename
...

Then we can set the explicit module name in the simulator based on which accelerator the user has chosen. We are now ready to push the baseline PARCv2 pipelined processor through the ASIC flow. First, let’s check that the Makefrag is setup correctly:

% cd $TOPDIR/asic
% grep "design = " Makefrag
design = proc-alt
% grep -A8 "proc-alt" Makefrag
ifeq ($(design),proc-alt)
    flow = pyrtl
clock_period = 2.0
sim_build_dir = pyrtl/build
vsrcl = ProcMemXcel_null_rtl.v
vmmame = ProcMemXcel_null_rtl
vname = TOP/v
vcd = pmx-sim-null-rtl-ubmark-vpadd.verification1.vcd
endif

Now we can go through the standard synthesis, place-and-route, and power analysis steps:

% cd $TOPDIR/asic/dc-syn && make
% cd $TOPDIR/asic/icc-par && make
% cd $TOPDIR/asic/pt-pwr && make

vsrcl = ProcMemXcel_null_rtl.v
input = pmx-sim-null-rtl-ubmark-vpadd
area = 128861 # um^2
constraint = 2.0 # ns
slack = -0.06 # ns
cycle_time = 2.06 # ns
exec_time = 3862 # cycles
power = 19.4 # mW
energy = 122.369768 # nJ
Note that you cannot use the value for `exec_time` from this final summary. The summary is reporting 3000 cycles because that includes the time for initialization and verification, while the cycle count from `pmx-sim` (i.e., 1217) is only when stats are enabled. Unfortunately, this also means the energy estimate is not correct since the summary is reporting the energy for initialization, the `vavdd` kernel, and verification. To determine how much energy is actually consumed by just the `vavdd` kernel, we can estimate the energy for a dummy microbenchmark that just does the initialization and verification steps but does not do any real work. Here is how we can modify `ubmark-vavdd.c` to become a dummy microbenchmark.

```c
int main( int argc, char* argv[] )
{
    int dest[ size ];

    int i;
    for ( i = 0; i < size; i++ )
        dest[ i ] = ref[ i ];

    test_stats_on();
    // vavdd_scalar( dest, src0, src1, size );
    test_stats_off();

    verify_results( dest, ref, size );

    return 0;
}
```

We modify the initialization code to copy the values from the reference to the destination and then we comment out the call to `vavdd_scalar`. This is not exactly that the initialization phase in the `vavdd` microbenchmark does, but it is close enough for now. After modifying `ubmark-vavdd.c` we need to re-cross-compile the microbenchmark:

```
% cd $TOPDIR/app/build
% make ubmark-vavdd
% maven-objdump -D ubmark-vavdd | grep -A40 "<main>:
00001210 <main>:
   1210: 8f88004 lw a4, -32764(gp)
   1214: 27bdf0f addiu sp, sp, -8
   1218: afb0004 sw s8, 4(sp)
   121c: 00028b0 s11 a1, a4, 0x2
   1220: 2a2000e addiu v0, a1, 14
   1224: 000210c2 srl v0, v0, 0x3
   1228: 000210c0 s11 v0, v0, 0x3
   122c: 03a0f0d1 move s8, sp
   1230: 03a2e823 subu sp, sp, v0
   1234: 03a300d1 move a2, sp
   1238: 19000009 blez a4, 1260
   123c: 3c0700e2 lui a3, 0x2
   1240: 24e7ee59 addiu a3, a3, -4528
   1244: 00001021 move v0, zero
   1248: 00e21821 addu v1, a3, v0    # \n   124c: 8c640000 lw a0, 0(v1)    # |
   1250: 00c21821 addu v1, a2, v0    # | initialize dest array
   1254: 24420004 addiu v0, v0, 4    # |
   1258: ac640000 sw a0, 0(v1)    # |
   125c: 1445fffa bne v0, a1, 1248    # /
   1260: 2482e001 li v0, 1
   1264: 4082a000 mtc0 v0, c0_staten    # \n   1268: 00001021 move v0, zero    # | nothing in stats region
   1270: 4082a000 mtc0 v0, c0_staten    # /
   1278: 19000014 blez a4, 12ec    # \n   127c: 3c030002 lui v1, 0x2    # |
   1280: 3c070002 lui a3, 0x2    # | verification code
   1284: 2463ee50 addiu v1, v1, -4528    # |
   1288: 34e70001 ori a3, a3, 0x1    # |
   1292: 3c070002 lui a3, 0x2    # |
   129c: 3c030002 lui v1, 0x2    # |
   12a0: 8c40000 lw a0, 0(a2)    # |
   12a8: 8c650000 lw a1, 0(v1)    # |
   12b0: 8c650000 lw a1, 0(v1)    # |
```
We can use `maven-objdump` again to confirm that the dummy microbenchmark still does the initialization and verification. Now we can run this dummy microbenchmark on the processor RTL model and re-estimate the energy.

% cd $TOPDIR/pyrtl/build
% ../pmx/pmx-sim --proc-impl rtl --stats ../..//app/build/ubmark-vmadd
num_cycles = 7
% ../pmx/pmx-sim --proc-impl rtl --translate --dump-vcd ../..//app/build/ubmark-vmadd
% cd $TOPDIR/asic/pt-pwr
% make

vsrc = ProcMemXcel_null_rtl.v
input = pmx-sim-null-rtl-ubmark-vmadd
area = 128861 # um^2
constraint = 2.0 # ns
slack = -0.06 # ns
cycle_time = 2.06 # ns
eexec_time = 2849 # cycles
power = 28.0 # mW
energy = 84.4188 # nJ

Not surprisingly there are only a few cycles in the timing loop since there is basically no work done in the timing loop. So based on this rough analysis it looks like 85nJ are spent in initialization and verification. Recall that the original estimate for the entire microbenchmark including initialization, the vadd kernel, and verification was 122nJ, so we can estimate the energy required for just the vadd kernel to be 122-85 = 37nJ.

Next, we might want to take a closer look at the critical path. Remember, that we need to look at the QOR report first to find out which path group has the critical path.

% cd $TOPDIR/asic/icc-par/current-icc/reports
% more chip_finish_icc.qor.rpt
# we note critical path is in the REGOUT path group
% less -PREGOUT chip_finish_icc.timing.rpt

<table>
<thead>
<tr>
<th>Point</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock ideal_clock1 (rise edge)</td>
<td>0.0000</td>
</tr>
<tr>
<td>clock network delay (propagated)</td>
<td>0.3342</td>
</tr>
<tr>
<td>proc/dpath/op1_reg_X/out_reg_2/CLK (DFX1)</td>
<td>0.3342 r</td>
</tr>
<tr>
<td>proc/dpath/op1_reg_X/out_reg_2/Q (DFX1)</td>
<td>0.5948 f</td>
</tr>
<tr>
<td>proc/dpath/op1_reg_X/out[2]</td>
<td>0.5948 f</td>
</tr>
<tr>
<td>proc/dpath/xcelreq_msg_data[2]</td>
<td>0.5948 f</td>
</tr>
<tr>
<td>proc/dpath/alu_X/in[2]</td>
<td>0.5948 f</td>
</tr>
<tr>
<td>proc/dpath/alu_X/out[10]</td>
<td>1.5314 f</td>
</tr>
<tr>
<td>proc/dpath/ex_result_sel_mux_X/in__000[10]</td>
<td>1.5314 f</td>
</tr>
<tr>
<td>proc/dpath/ex_result_sel_mux_X/out[10]</td>
<td>1.6555 f</td>
</tr>
<tr>
<td>proc/dpath/ex_result_sel_mux_X_out[10]</td>
<td>1.6555 f</td>
</tr>
<tr>
<td>proc/dpath/op0_byp_mux_D/in__001[10]</td>
<td>1.6555 f</td>
</tr>
<tr>
<td>proc/dpath/op0_byp_mux_D/n49</td>
<td>1.6926 r</td>
</tr>
<tr>
<td>proc/dpath/op0_byp_mux_D/n183</td>
<td>1.7394 f</td>
</tr>
<tr>
<td>proc/dpath/op0_byp_mux_D_out[10]</td>
<td>1.7394 f</td>
</tr>
<tr>
<td>proc/dpath/pc_sel_mux_F/in__003[10]</td>
<td>1.8213 f</td>
</tr>
<tr>
<td>proc/dpath/pc_sel_mux_F/n44</td>
<td>1.8561 r</td>
</tr>
<tr>
<td>proc/dpath/pc_sel_mux_F/out[10]</td>
<td>1.8847 f</td>
</tr>
<tr>
<td>proc/dpath/imemreq_msg[44]</td>
<td>1.8847 f</td>
</tr>
<tr>
<td>proc/dpath_imemreq_msg[44]</td>
<td>1.8847 f</td>
</tr>
<tr>
<td>proc/imemreq_queue/eqn_msg[44]</td>
<td>1.8847 f</td>
</tr>
<tr>
<td>proc/imemreq_queue/queue0/enq_msg[44]</td>
<td>1.8847 f</td>
</tr>
<tr>
<td>proc/imemreq_queue/queue0/dpath/enq_bits[44]</td>
<td>1.8847 f</td>
</tr>
<tr>
<td>proc/imemreq_queue/queue0/dpath/bypass_mux/in__001[44]</td>
<td>1.8847 f</td>
</tr>
</tbody>
</table>
I have cleaned the path a bit but removing some columns and mostly only showing the nets. We can see that the critical path starts at the operand 0 pipeline register in between the D and X stages, goes through the ALU, through the mux at the end of the X stage, around through the bypass paths to the bypass mux at the end of the D stage, then through the PC select mux, out to the imemreq queue, and since this queue is a bypass queue, the path continues out to the instruction fetch interface. This path is highlighted on the processor datapath diagram below.

The following instruction sequence would use the bypass path which is the critical path:

```
addu r1, r2, r3   F D (X) M W
jr                  F (D) X M W
nop                 F - - - -
```
So from the end of the X stage of the ADDU instruction, to the end of the D stage of the JR instruction, to fetching opA. Bypass paths are often on the critical path; especially bypass paths which then feed into instruction fetch. We could eliminate this critical path by moving the operand 0 bypass mux to the other side of the operand 0 mux. We would need to add a new stalling condition to ensure correct execution, and this would result in the following pipeline diagram:

```
addu   r1, r2, r3   F   D   X   M   W
  jr     F   D   D   D   X   M   W
   nop   F   F   F   -   -   -   -
     opA   F   D   X   M   W
```

This is a classic trade-off between cycle time and CPI. To improve performance a bit, we could maybe add back in the bypass path from W. The bypass path from the end of M would probably be just as bad as the bypass path from the end of X.

The following figure shows an amoeba plot of the baseline processor. The large cyan region on the left is the register file; the green region in the lower right corner is the ALU; and the purple region in the middle is the iterative multiplier. It is not too surprising that the register file dominates the overall area since flip-flops are relatively large and a 32 element register file with 32-bit values requires 1024 flip-flops.

![RFIELD, IMUL, ALU](image)

**VVADD Accelerator FL, CL, and RTL Models**

We will take an incremental approach when designing, implementing, testing, and evaluating accelerators. We can use test sources, sinks, and memories to create a test harness that will enable us to explore the accelerator cycle-level performance
and the ASIC area, energy, and timing in isolation. Only after we are sure that we have a reasonable design-point should we consider integrating the accelerator with the processor.

All accelerators have an xcelreq/xcelresp interfaces along with a standard memreq/memresp interfaces. The messages sent over the xcelreq/xcelresp interfaces allow the test harness or processor to read and write accelerator registers. These accelerator registers can be real registers that hold configuration information and/or results, or these accelerator registers can just be used to trigger certain actions. The messages sent over the xcelreq interface from the test harness or processor to the accelerator have the following format:

```
  1b   5b   32b
+--------------------------+  
| type | raddr | data |  
+--------------------------+
```

The 1-bit type field indicates if this messages if for reading (0) or writing (1) an accelerator register, the 5-bit raddr field specifies which accelerator register to read or write, and the 32-bit data field is the data to be written. For every accelerator request, the accelerator must send back a corresponding accelerator response over the xcelresp interface. These response messages have the following format:

```
  1b   32b
+--------------------------+  
| type | data |  
+--------------------------+
```

The 1-bit type field gain indicates if this response is from if for reading (0) or writing (1) an accelerator register, and the 32-bit data field is the data read from the corresponding accelerator register. Every accelerator is free to design its own accelerator protocol by defining the meaning of reading/writing the 32 accelerator registers.

We have implemented a null accelerator which we can use when we don’t want to integrate a "real" accelerator, but this null accelerator is also useful in illustrating the basic accelerator interface. The null accelerator has a single accelerator register (xR0) which can be read and written. You can find the NullXcel here:

```
% cd $TOPDIR/pyml/proc
% more NullXcel.py
```

Here is a unit test which writes a value to the null accelerator's xR0 register and then reads it back:

```
% cd $TOPDIR/pyml/build
% py.test ..//proc/NullXcel_test.py -k basic -s
```

```
  2:  >     ()()                      > 
  3:  wr:00:00000000 > wr:00:00000000a() >  
  4:  rd:00:     > rd:00:     ()wr:       > wr:  
  5:  >     ()rd:000000a > rd:000000a
```

From the line trace, you can see the write request message (with write data 0x0a) going into the accelerator, and then the write response being returned. You can also see the read request message going into the accelerator, and then the read response being returned (with read data 0x0a).

The vwadd accelerator is obviously more sophisticated. Accelerator protocols are usually defined as a comment at the top of the FL model:

```
% cd $TOPDIR/pyml/vwadd_xcel
% more VwaddXcelFl.py
```
The vvadd accelerator protocol defines the accelerator registers as follows:

- \( x0 \): go/done
- \( x1 \): base address of the array src0
- \( x2 \): base address of the array src1
- \( x3 \): base address of the array dest
- \( x4 \): size of the array

The actual protocol involves the following steps:

1. Write the base address of src0 to \( x1 \)
2. Write the base address of src1 to \( x2 \)
3. Write the base address of dest to \( x3 \)
4. Write the number of elements in the array to \( x4 \)
5. Tell accelerator to go by writing \( x0 \)
6. Wait for accelerator to finish by reading \( x0 \), result will be 1

A close look at the vvadd accelerator FL model in `vvaddXcelFL` shows that most of the work is really in managing this accelerator protocol. The accelerator waits for accelerator requests, updates its internal state registers, and when it receives a write to \( x0 \) it starts doing the actual vvadd computation. The FL model makes use of `ListMemPortAdapters` to simplify interacting with the memory system. Let’s run the unit tests on the FL model first:

```bash
% cd $TOPDIR/pymtl/build
% py.test ../vvadd_xcel/VvaddXcelFL_test.py -v
```

The vvadd accelerator CL model is actually very close to the RTL implementation largely due to the need to carefully interact with the latency insensitive memory interface. CL modeling may or may not be useful in this context. The vvadd accelerator RTL model implements an FSM, and a simplified version of this FSM is shown below.

![FSM Diagram]

While the accelerator is in the XCFG state, it will update its internal registers when it receives accelerator requests. When the accelerator receives a write to \( x0 \) it moves into the M_RD state. While in the M_RD state, the accelerator will send out two memory read requests to read the current element from each source array. In the ADD state, the accelerator will do the actual addition, and in the M_WR state, the accelerator will send out the memory write request to write the result to the destination array. The accelerator will wait in the final WAIT state until it receives the memory write response, and then will either move back into the M_RD state if there is another element to be processed, or move into the XCFG state if we have processed all elements in the array.

The accelerator is not implemented with a control/datapath split because the accelerator is almost entirely control logic; it was just simpler to implement the accelerator as a single model. When a model is almost all control logic or almost all datapath logic, then a control/datapath split may be more trouble than its worth.

Let’s run the unit tests for all of the vvadd accelerator models:
We have also included a simulator for just the vvadd accelerator in isolation which can be used to evaluate its performance.

We could use the simulator to help evaluate the cycle-level performance of the accelerator on various different datasets as we try out various optimizations.

**VVADD** Accelerator **ASIC**

We can now push the vvadd accelerator through the ASIC flow in isolation to get a feel for the area, timing, and energy. We use the simulator to translate the accelerator into Verilog and to generate the VCD file we need for energy analysis.

We need to make sure the `Makefrag` is setup to push the vvadd accelerator through the flow:

Note that we use a rather conservative clock constraint of 2ns since that is what we used for the baseline processor. Our goal in designing these accelerators is just to ensure the accelerator doesn't increase the critical path of the processor; there is no benefit in pushing the accelerator to have a cycle time which is less than the processor since this will not help the overall cycle time of the processor, memory, accelerator composition. We push the accelerator through the flow as follows:
The accelerator has no trouble meeting timing. If the accelerator did have trouble meeting the 2ns clock constraint, then we could do timing optimizations on the accelerator in isolation before moving onto the processor, memory, accelerator composition. We can also do energy optimization on the accelerator in isolation.

Notice that the area of the vadd accelerator is 26,612 um^2, while the area of the baseline processor is 128,861 um^2. Adding the vadd accelerator to the baseline processor would result in an area overhead of 20%, mostly due to the area consumed by the various queues in the design.

We can also generate the obligatory amoeba plot. Notice how the tools have distributed the accelerator on either side of the register file and how the iterative multiplier is now spread apart into two different clusters. The tools use all kinds of heuristics for placement, and it is not always obvious why they have decided to place a certain module in a specific place.

## Integrating the PARCv2 Processor and an Accelerator

Now that we have unit tested and evaluated both the baseline PARCv2 pipelined processor and the vadd accelerator in isolation, we are finally ready to compose them. The processor will send messages to the accelerator using the MTX and MFX instructions defined here:

- [http://www.csl.cornell.edu/courses/ece5745/handouts/ece5745-xcel-isa.txt](http://www.csl.cornell.edu/courses/ece5745/handouts/ece5745-xcel-isa.txt)

When the processor executes an MTX instruction it reads the general-purpose register file and creates a new accelerator request message which is then sent to the processor through the xcelreq interface. The processor waits for the response message to be returned through the xcelresp interface. The processor executes an MFX instruction in the same way, except that when the response message is returned, the data from the accelerator is written into the general-purpose register file.

You can run a simple test of the MTX/MFX instructions with the null accelerator like this:
I have cleaned up the line trace a bit to annotate the columns and make it more compact. You can see the processor executing MTX/MFX instructions and sending accelerator requests to the null accelerator, and then the accelerator sending the corresponding accelerator responses back to the processor.

We have not added similar assembly tests for the composition of the processor and vadd accelerator, but this is certainly something one could add in the future.

**Accelerating a PARCv2 Microbenchmark**

To use an accelerator from a C microbenchmark, we can use the same GCC inline assembly extensions we used to insert MTC0 instructions earlier in the tutorial. Take a closer look at the `ubmark-null-xcel.c` example:

```c
% cd $TOPDIR/app/ubmark
% more ubmark-null-xcel.c
<snip>
__attribute__((noinline))
unsigned int null_xcel( unsigned int in )
{
    unsigned int result;
```
asm volatile (\n    "mtx %[in], $0, 0 \n"\n    "mfx %[result], $0, 0 \n"

    // Outputs from the inline assembly block
    : [result] "=r"(result)
    // Inputs to the inline assembly block
    : [in] "r"(in)
);\nreturn result;
}

We are inserting an MTX instruction to copy the value passed to this function through the \n argument, and then we are using an MFX instruction to retrieve the same value from the null accelerator. Notice that unlike the inline assembly for MTC0, here we also need to handle outputs from the assembly block. Again, you can find out more about inline assembly syntax here:


Let's cross-compiler this microbenchmark. Note that you cannot natively compile a microbenchmark that makes use of an accelerator, since x86 does not have any accelerators!

```bash
% cd $TOPDIR/app/build
% make ubmark-null-xcel
% maven-objdump -dC ubmark-null-xcel | less -p"<null_xcel"
00001168 <null_xcel(unsigned int>):
 1168: 48040000 mtx a0, zero, 0x0
 116c: 48020800 mfx v0, zero, 0x800
 1170: 03e00000 jr ra
```

Always a good idea to use `maven-objdump` so you can verify your C code is compiling as expected. Here we can see that the `null_xcel` function compiles into an MTX, MFX, and JR instruction as expected. You can ignore the fact that the MFX instruction has an odd looking final immediate field; this is just due to an artifact in our version of `maven-objdump`.

We should now run this microbenchmark on our ISA simulator to verify it works, and then we can run it on our RTL simulator.

```bash
% cd $TOPDIR/pymtl/build
% ../pmx/pmx-sim ../../app/build/ubmark-null-xcel
% ../pmx/pmx-sim --proc-impl rtl --xcel-impl null-rtl --trace \n  ../../../app/build/ubmark-null-xcel
```

Let's turn out attention to our vvadd accelerator. Take a closer look at the accelerated version of the vvadd microbenchmark here:

```bash
% cd $TOPDIR/app/ubmark
% more ubmark-vvadd-xcel.c
<snip>
__attribute__((noinline))
unsigned int null_xcel(unsigned int in)
{
    asm volatile (\n        ".set noat     \n"\n        "mtx %[src0], $1, 0 \n"
        "mtx %[src1], $2, 0 \n"
```
"mtx %[dest], $3, 0 \n"
"mtx %[size], $4, 0 \n"
"mtx $0, $0, 0 \n"
"mfx $0, $0, 0 \n"
".set at \n"

// Outputs from the inline assembly block

;

// Inputs to the inline assembly block

:[src0] "r"(src0),
[src1] "r"(src1),
[dest] "r"(dest),
[size] "r"(size)

// Tell the compiler this accelerator read/writes memory

:"memory"
);

Notice that our use of MTX/MFX corresponds exactly to the accelerator protocol described above. We first write the source base pointers, the destination base pointer, and the size before starting the accelerator by writing to xr0 and then waiting for the accelerator to finish by reading xr0. We need a final "memory" argument in our inline assembly block to tell the compiler that this accelerator reads and writes memory. Let's cross-compile the accelerated version of the vvadd microbenchmark:

% cd $TOPDIR/app/build
% make ubmark-vvadd-xcel
% maven-antbuild -Dc ubmark-vvadd-xcel | less -p<vvadd_xcel
0000168 <vvadd_xcel(int*, int*, int*, int*):
  1168: 48250000 mtx a1, at, 0x0
  116c: 48460000 mtx a2, v0, 0x0
  1170: 48640000 mtx a0, vi, 0x0
  1174: 48870000 mtx a3, a0, 0x0
  1178: 48000000 mtx zero, zero, 0x0
  117c: 48000000 mfx zero, zero, 0x800
  1180: 83e00000 Jr ra

Everything looks as expected, so we can now test our accelerated vvadd microbenchmark on the ISA simulator.

% cd $TOPDIR/pytll/build
% ./pmx/pmx-sim --xcel-impl vvadd-f1 ../app/build/ubmark-vvadd-xcel

Notice that we needed to specify the accelerator implementation as a command line option. If we forgot to include this option, then the simulator would use the null accelerator and clearly the accelerated vvadd microbenchmark does not work with the null accelerator! Finally, we can run the accelerated vvadd microbenchmark on the RTL implementation of the processor augmented with the RTL implementation of the vvadd accelerator:

% cd $TOPDIR/pytll/build
% ./pmx/pmx-sim --proc-impl rtl --xcel-impl vvadd-rtl --stats \n  ../app/build/ubmark-vvadd-xcel
num cycles = 830

Recall that the pure-software vvadd microbenchmark required 1217 cycles. So our accelerator results in a cycle-level speedup of almost 1.5x. We might ask, where did this speedup come from? Let's look at the line trace.

% cd $TOPDIR/pytll/build
Here is the line trace looks like for the initial configuration of the accelerator and the first two iterations of the vvadd loop:

<table>
<thead>
<tr>
<th>cyc</th>
<th>F-stage</th>
<th>D-stage</th>
<th>X</th>
<th>M</th>
<th>W</th>
<th>xcelreq</th>
<th>ST</th>
<th>xcel-&gt;memreq</th>
<th>xcel&lt;-#</th>
</tr>
</thead>
<tbody>
<tr>
<td>631</td>
<td>0000124</td>
<td>mtc0 r02, r21</td>
<td>addiu</td>
<td>bne</td>
<td>addiu</td>
<td>(X 0:0:00000000)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>632</td>
<td>00001254</td>
<td>nop</td>
<td>mtc0</td>
<td>addiu</td>
<td>bne</td>
<td>(X 0:0:00000000)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>633</td>
<td>00001250</td>
<td>nop</td>
<td>mtc0</td>
<td>addiu</td>
<td>bne</td>
<td>(X 0:0:00000000)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>634</td>
<td>00001254</td>
<td>nop</td>
<td>mtc0</td>
<td>bne</td>
<td>addiu</td>
<td>(X 0:0:00000000)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>635</td>
<td>00001258</td>
<td>nop</td>
<td>mtc0</td>
<td>bne</td>
<td>addiu</td>
<td>(X 0:0:00000000)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>636</td>
<td>00001250</td>
<td>nop</td>
<td>mtc0</td>
<td>bne</td>
<td>addiu</td>
<td>(X 0:0:00000000)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>637</td>
<td>00001260</td>
<td>lui r05, 0002</td>
<td>mtc0</td>
<td>bne</td>
<td>addiu</td>
<td>(X 0:0:00000000)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>638</td>
<td>00001264</td>
<td>lui r06, 0002</td>
<td>lui</td>
<td>bne</td>
<td>addiu</td>
<td>(X 0:0:00000000)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>639</td>
<td>00001268</td>
<td>lui r05, r05, eb30</td>
<td>lui</td>
<td>lui</td>
<td>addiu</td>
<td>(X 0:0:00000000)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>640</td>
<td>0000126c</td>
<td>lui r06, r06, ecc0</td>
<td>lui</td>
<td>lui</td>
<td>addiu</td>
<td>(X 0:0:00000000)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>641</td>
<td>00001270</td>
<td>lui r04, r16, r00</td>
<td>lui</td>
<td>lui</td>
<td>addiu</td>
<td>(X 0:0:00000000)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>642</td>
<td>~</td>
<td>jal 000045a</td>
<td>addu</td>
<td>addiu</td>
<td>addiu</td>
<td>(X 0:0:00000000)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>643</td>
<td>00001168</td>
<td>jal addu</td>
<td>addiu</td>
<td>addiu</td>
<td>addiu</td>
<td>(X 0:0:00000000)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>644</td>
<td>00001156</td>
<td>mtc0 r05, 0000</td>
<td>jal</td>
<td>addu</td>
<td>addiu</td>
<td>(X 0:0:00000000)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>645</td>
<td>00001170</td>
<td>mtc0 r02, 0000</td>
<td>mtc</td>
<td>jal</td>
<td>addiu</td>
<td>(X 0:0:00000000)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>646</td>
<td>00001174</td>
<td>mtc0 r04, 0000</td>
<td>mtc</td>
<td>mtc</td>
<td>addiu</td>
<td>(X 0:0:00000000)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>647</td>
<td>00001178</td>
<td>mtc0 r07, 0000</td>
<td>mtc</td>
<td>mtc</td>
<td>addiu</td>
<td>(X 0:0:00000000)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>648</td>
<td>0000117c</td>
<td>mtc0 r08, 0000</td>
<td>mtc</td>
<td>mtc</td>
<td>addiu</td>
<td>(X 0:0:00000000)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>649</td>
<td>00001180</td>
<td>mtc0 r09, 0000</td>
<td>mtc</td>
<td>mtc</td>
<td>addiu</td>
<td>(X 0:0:00000000)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

I have cleaned up the line trace a bit to annotate the columns and make it more compact. The ST column is the current state of the vvadd accelerator FSM. You can see the processor executing the MTX instructions, and these instructions then turn into messages over the xcelreq interface. The accelerator is in the XCFG state receiving these messages until it receives the write to x0 which causes the accelerator to move into the RD stage. The accelerator sends to memory read request requests into the memory system, does the addition when those memory responses come back, and finally sends a memory write request with the result into the memory system. Each iteration of the vvadd accelerator takes eight cycles, so 8*100 = 800 which is about what we see in terms of the overall cycle count for the accelerator. There is certainly room for improvement. We can probably remove some of the bubbles and improve the accelerator performance to three of four cycles per iteration. Three cycles per iteration is the best we can do, because we would then be completely memory bandwidth limited.

**PARCv2 Processor + VVADD Accelerator ASIC**

Now that we have some results suggesting our vvadd accelerator is able to improve the cycle-level performance by 1.5x, we can push the processor, memory, accelerator composition through the flow to see the overall impact on area, energy, and
timing. First, we need to run the pmx-sim simulator to translate the processor into Verilog and to generate the VCD file we need for energy analysis.

% cd $TOPDIR/pymtl/build
% ../../../pmx-sim --proc-impl rtl --xcel-impl vvadd-rtl \
   --translate --dump-vcd ../../../app/build/ubmark-vvadd-xcel
% ls
<snip>
ProcMemXcel_vvadd_rtl.v
pmx-sim-vvadd-rtl-ubmark-vvadd-xcel.verilator1.vcd

Let’s check that the Makefrag is setup correctly:

% cd $TOPDIR/asic
% grep "design = " Makefrag
design = proc-vvadd-xcel

% grep -A8 "proc-vvadd-xcel)" Makefrag
ifeq ($(design),proc-vvadd-xcel)
flow       = pymtl
clock_period = 2.0
sim_build_dir = pymtl/build
vsnc       = ProcMemXcel_vvadd_rtl.v
vmmname    = ProcMemXcel_vvadd_rtl
viname     = TOP/v
vcd        = pmx-sim-vvadd-rtl-ubmark-vvadd-xcel.verilator1.vcd
endif

And now we can push the full design through the flow.

% cd $TOPDIR/asic/dc-syn   && make
% cd $TOPDIR/asic/icc-par  && make
% cd $TOPDIR/asic/pt-pwr   && make

vsnc       = ProcMemXcel_vvadd_rtl.v
input      = pmx-sim-vvadd-rtl-ubmark-vvadd-xcel
area       = 146114 # um^2
constraint = 2.0 # ns
slack      = -0.17 # ns
cycle_time = 2.17 # ns
exec_time  = 2675 # cycles
power      = 18.3 # mW
energy     = 186.226925 # nJ

We can use our earlier results for the energy overhead of initialization and verification to estimate the energy of using the vvadd accelerator to be 106-85 = 21kJ. A close look at the critical path shows it to be similar to what we saw in the baseline processor, but we do see the actual cycle time increased a bit from 2.06ns to 2.17ns. This is likely due to the fact that the accelerator simply forced various processor of the design to be further part resulting in increased interconnect delay. We could potentially add this issue with better floorplanning. A ~5% increase in the cycle time is not too bad, so we are probably fine comparing the processor with accelerator to the baseline processor, but we do need to be careful if adding the accelerator increases the timing by quite a bit.

Here is the final summary of the performance, area, energy, and timing comparison between the pure-software vvadd microbenchmark and the accelerated vvadd microbenchmark.

<table>
<thead>
<tr>
<th>metric</th>
<th>pure-software</th>
<th>accelerated</th>
<th>relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycles</td>
<td>1217</td>
<td>830</td>
<td>1.47x</td>
</tr>
<tr>
<td></td>
<td>Cycle Time</td>
<td>2.06 ns</td>
<td>2.17 ns</td>
</tr>
<tr>
<td>------------------</td>
<td>------------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>Area</td>
<td>128,861 um^2</td>
<td>146,114 um^2</td>
<td>0.88x</td>
</tr>
<tr>
<td>Energy</td>
<td>37 nJ</td>
<td>21 nJ</td>
<td>1.76x</td>
</tr>
</tbody>
</table>