ECE 5745 PyMTL CL Modeling

Most students are quite familiar with functional-level (FL) and register-transfer-level (RTL) modeling from ECE 4750, but students are often less familiar with cycle-level (CL) modeling. Here is a reminder of the differences between these modeling levels:

• **Functional-Level**: FL models implement the functionality but not the timing of the hardware target. FL models are useful for exploring algorithms, performing fast emulation of hardware targets, and creating golden models for verification of CL and RTL models. FL models can also be used for building sophisticated test harnesses. FL models are usually the easiest to construct, but also the least accurate with respect to the target hardware.

• **Cycle-Level**: CL models capture the cycle-approximate behavior of a hardware target. CL models will often augment the functional behavior with an additional timing model to track the performance of the hardware target in cycles. CL models are usually specifically designed to enable rapid design-space exploration of cycle-level performance across a range of microarchitectural design parameters. CL models attempt to strike a balance between accuracy, performance, and flexibility.

• **Register-Transfer-Level**: RTL models are cycle-accurate, resource-accurate, and bit-accurate representations of hardware. RTL models are built for the purpose of verification and synthesis of specific hardware implementations. RTL models can be used to drive EDA toolflows for estimating area, energy, and timing. RTL models are usually the most tedious to construct, but also the most accurate with respect to the target hardware.

The tutorial on PyMTL introduced all three levels of modeling, but we will review the CL models for the sort unit and GCD unit here. We will also explore a new CL model for a simple cache, and we will discuss a sophisticated CL modeling framework used to model complex superscalar out-of-order processors. Please keep in mind that developing a CL model is required for lab 1 and may be required for lab 2. While CL modeling is not required for the project, students may find that CL modeling can facilitate a much broader initial design-space exploration, and thus enable students to focus on the most promising designs for RTL implementation.

The first step is to source the setup script, clone this repository from GitHub, and make a build directory to work in.

```bash
% source setup-ece5745.sh
% mkdir $HOME/ece5745
% cd $HOME/ece5745
% git clone git@github.com:cornell-ece5745/ece5745-sec-pyml1-cl
% mkdir -p ece5745-sec-pyml1-cl/pyml1/build
% cd ece5745-sec-pyml1-cl/pyml1/build
% BUILDDIR=$PWD
```

**Sort Unit CL Model**

We will start by reviewing the sort unit CL model first introduced in the PyMTL tutorial and shown in the following figure. We immediately sort the input values in a single cycle, and then we delay sending the response by several cycles to model the pipeline latency. The key to CL modeling is to capture the cycle-level behavior (i.e., how long a transaction takes in cycles) without worrying about the bit-level details or the exact hardware structures required to implement the desired cycle-level behavior.
Here is the corresponding code for the sort unit CL model.

```python
class SortUnitCL( Model ):
    def __init__( s, nbits=8, nstages=3 ):
        s.in_val = InPort(1)
        s.in_ = [ InPort(nbits) for _ in range(4) ]
        s.out_val = OutPort(1)
        s.out = [ OutPort(nbits) for _ in range(4) ]
        s.pipe = deque( [[0,0,0,0]]*(nstages-1) )

    @s.tick_cl
    def block():
        s.pipe.append( deepcopy( [s.in_val] + sorted(s.in_) ) )
        data = s.pipe.popleft()
        s.out_val.next = data[0]
        for i, v in enumerate( data[1:] ):
            s.out[i].next = v
```

We can use any Python code we want to implement a CL model. Here we use Python's built-in `sorted` function to do the actual sorting operation on the input values. Notice how we use a `deque` to model the pipeline behavior. Each entry in the `deque` has a valid bit and then the four sorted values. Also notice how we need to use a `deepcopy` to ensure that we really do have a copy of the sorted values in the `deque` and not just a reference. This is very common CL modeling pattern: we do all of the real work in the first cycle and then delay the response to model the desired latency. Let's run the sort unit CL tests and then produce a line trace:

```
% cd $BUILDDIR
% py.test ../tut3_pymtl/sortUnitCL_test.py
% py.test ../tut3_pymtl/sortUnitCL_test.py -k test[2stage_stream] -s
```

```
2: {04,03,02,01}
3: {09,06,07,01}
4: {04,08,00,09} | {01,02,03,04}
5: | {01,06,07,09}
6: | {08,04,08,09}
```

We can see that the first transaction goes into the sort unit on cycle 2 and then comes out of the sort unit on cycle 4. The sort unit CL model is modeling a two-stage pipeline. Since CL modeling is a higher level modeling abstraction compared to RTL modeling, it is much easier to experiment with different parameters. For example, we can easily experiment with longer latency sort units and analyze the impact the sort latency has on the full system cycle-level performance.

As another example, let's change our sort unit CL model to sort the four input values in the reverse order. We can do this
quite easily, by just setting the reverse flag when calling Python’s built-in sorted function. Here is the updated concurrent block:

```python
@s.tick_cl
def block():
    s.pipe.append( deepcopy([s.in_val] + sorted(s.in_, reverse=True))
    data = s.pipe.popleft()
    s.out_val.next = data[0]
    for i, v in enumerate(data[1:]):
        s.out[i].next = v
```

Of course we also need to update the test script to generate reference outputs where the values are in descending order. Once we make the changes to the test script, we can see the result as follows:

```
% cd $BUILDDIR
% py.test ./tut3_pymtl/sort/SortUnitCL_test.py -k test[2stage_stream] -s

2: {04,03,02,01}
3: {09,06,07,01}
4: {04,08,00,09} | {04,03,02,01}
5: | {09,07,06,01}
6: | {09,08,04,00}
```

As expected, the output values are now in descending order. The key idea is that experimenting with different sort orders in an RTL model would require significantly more work than the quick change shown above. CL modeling can enable more productive design-space exploration of cycle-level performance.

## GCD Unit CL Model

We now review the GCD unit CL model also introduced in the PyMTL tutorial and shown in the following figure. Unlike the sort unit CL model, here we want to model an iterative microarchitecture. In this case we can simply do the GCD immediately in a single cycle and then use a counter to wait a certain number of cycles. The number of cycles to wait should roughly approximate how long the real hardware will require to calculate the GCD.

![GCD Unit CL Model Diagram]

Here is the code used to estimate the number of cycles:

```python
def gcd( a, b ):
    ncycles = 0
    while True:
        ncycles += 1
        if a < b:
            a, b = b, a
        elif b != 0:
            a = a - b
        else:
            return (a, ncycles)
```
Take a look at how the CL model uses adapters to simplify interfacing with the latency insensitive val/rdy interface. Instead of having to directly read and write the val/rdy ports, a CL model can call \texttt{deq} (dequeue) and \texttt{enq} (enqueue) methods on the adapters to receive or send messages on val/rdy interfaces. Let’s run the GCD unit CL tests and then produce a line trace:

\begin{verbatim}
% cd $BUILDDIR
% py.test ../tut3_pymtl/gcd/GcdUnitCL_test.py
% py.test ../tut3_pymtl/gcd/GcdUnitCL_test.py -k test[basic_0x0] -s

2:   >   (\textcolor{red}{\texttt{.}})   >
3:  000F:0005 > 000F:0005 (\textcolor{red}{\texttt{.}})
4:  0003:0009 > 0003:0009 (\textcolor{red}{\texttt{.}})
5: #  > #  (\textcolor{red}{\texttt{.}})
6: #  > #  (\textcolor{red}{\texttt{.}})
7: #  > #  (\textcolor{red}{\texttt{.}})
8: #  > #  (\textcolor{red}{\texttt{.}})
9: #  > #  (\textcolor{red}{\texttt{.}})(0005 > 0005
10: 0000:0000 > 0000:0000 (\textcolor{red}{\texttt{.}})
11: #  > #  (\textcolor{red}{\texttt{.}})
12: #  > #  (\textcolor{red}{\texttt{.}})
13: #  > #  (\textcolor{red}{\texttt{.}})
14: #  > #  (\textcolor{red}{\texttt{.}})
15: #  > #  (\textcolor{red}{\texttt{.}})
16: #  > #  (\textcolor{red}{\texttt{.}})(0003 > 0003
17: 001b:000f > 001b:000f (\textcolor{red}{\texttt{.}})
18: #  > #  (\textcolor{red}{\texttt{.}})(0000 > 0000
19: 0015:0031 > 0015:0031 (\textcolor{red}{\texttt{.}})
20: #  > #  (\textcolor{red}{\texttt{.}})
21: #  > #  (\textcolor{red}{\texttt{.}})
22: #  > #  (\textcolor{red}{\texttt{.}})
23: #  > #  (\textcolor{red}{\texttt{.}})
24: #  > #  (\textcolor{red}{\texttt{.}})
25: #  > #  (\textcolor{red}{\texttt{.}})
26: #  > #  (\textcolor{red}{\texttt{.}})
27: #  > #  (\textcolor{red}{\texttt{.}})
28: #  > #  (\textcolor{red}{\texttt{.}})
29: #  > #  (\textcolor{red}{\texttt{.}})(0003 > 0003
\end{verbatim}

Now let’s assume we want to experiment with the impact of a hardware optimization that combines the swap and subtract operation into a single step. We simply need to change the code that estimates the number of cycles to something like this:

\begin{verbatim}
def gcd( a, b ):
    ncycles = 0
    while True:
        ncycles += 1
        if a < b:
            a, b = b-a, a
        elif b != 0:
            a = a - b
        else:
            return (a,ncycles)
\end{verbatim}

Notice how we now also do a subtraction at the same time we swap $a$ and $b$. The resulting line trace is now:

\begin{verbatim}
2:   >   (\textcolor{red}{\texttt{.}})   >
3:  000F:0005 > 000F:0005 (\textcolor{red}{\texttt{.}})
4:  0003:0009 > 0003:0009 (\textcolor{red}{\texttt{.}})
\end{verbatim}
Our model now suggests the GCD unit will finish four transactions on cycle 26 instead of finishing on cycle 29. Conducting a similar experiment in an RTL model would require significant changes to the datapath and control unit, while experimenting with such cycle-level behavior in a CL model is much easier.

**Simple Cache CL Model**

We now examine a more involved CL model for a specific target cache design. The target cache includes `cachereq / cacheresp` interfaces for the unit that wants to use the cache, and `memreq / memresp` interfaces for connecting the cache to main memory. We will be modeling a relatively simple cache that is direct-mapped, write-through, and no-write-allocate. The cache will have eight single-word cache lines and will only support full-word cache requests. You can run all of the tests for the cache CL model like this:

```
% cd $BUILD_DIR
% py.test ../cache_cl
```

The sequential concurrent block for modeling the cycle-level behavior of the cache is shown below.

```python
@s.tick_cl
def tick():
    # Tick adapters
    s.cachereq_q.xtick()
    s.cacheresp_q.xtick()
    s.memreq_q.xtick()
    s.memresp_q.xtick()

    # Some constants to simplify the code
    s.READ = s.cache_ifc_dtypes.req.TYPE_READ
    s.WRITE = s.cache_ifc_dtypes.req.TYPE_WRITE

    # If we are not waiting for a memory response ...
```
if not s.wait_bit and not s.cachereq_q.empty() \
    and not s.memreq_q.full() \
    and not s.cacheresp_q.full():

cachereq = s.cachereq_q.deq()
    if cachereq.type_ == s.WRITE  : s.process_cachereq_write( cachereq )
elif cachereq.type_ == s.READ : s.process_cachereq_read ( cachereq )
else : assert False

# If we are waiting for a memory response ...
if s.wait_bit and not s.memresp_q.empty() \
    and not s.cacheresp_q.full():

    memresp = s.memresp_q.deq()
    if memresp.type_ == s.WRITE : s.process_memresp_write( memresp )
elif memresp.type_ == s.READ : s.process_memresp_read ( memresp )
else : assert False

As with the GCD unit CL model, we use adapters on all four interfaces to simplify interacting with the latency insensitive val/rdy microprotocol. There are two primary parts to this concurrent block, and which part we execute depends on the wait_bit. The wait_bit is true when we are waiting for a memory response, and the wait_bit is false otherwise. If we are not waiting for a memory response, then we check to see if there is a new cache request (i.e., is the adapter for the cachereq interface not empty?) but we also make sure that the memory request and cache response interfaces are available. Note that this is somewhat conservative; not all transactions need both the memory request and the cache response (e.g., cache read hits do not need to send a memory request), but we simplify our logic ensuring that all potentially required resources are available before starting to process a cache request. Based on the type of the cache request we dispatch to one of two member functions. If we are waiting for a memory response, we again make sure that all potentially required resources (i.e., the cache response interface) are available before starting to process the memory response. Based on the type of the memory response we dispatch to one of two member functions. Take a closer look at how these four member functions are implemented. Note how CL modeling is more of a software engineering effort to model hardware as compared to RTL modeling which is more of a hardware engineering effort to implement hardware. We can run a test and view a line trace for the cache CL model like this:

% cd $BUILD_DIR
% py.test ..cache_cl -k test_lport[basic] -s

<table>
<thead>
<tr>
<th>#</th>
<th>cachereq</th>
<th>set 0</th>
<th>set 7</th>
<th>memreq</th>
<th>memresp</th>
<th>cacheresp</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>(</td>
<td>...</td>
<td></td>
<td></td>
<td>()</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>wr:00:00001000:deadbeef (</td>
<td>...</td>
<td></td>
<td></td>
<td>()</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>rd:01:00001000: (</td>
<td>...</td>
<td></td>
<td>wr:00:00001000:deadbeef()</td>
<td>()</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>.</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>wr:00:0:</td>
</tr>
<tr>
<td>6</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td>()</td>
<td>wr:00:0:</td>
</tr>
<tr>
<td>7</td>
<td>(</td>
<td></td>
<td></td>
<td>rd:01:00001000:</td>
<td>()</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>.</td>
<td></td>
<td></td>
<td>rd:01:0:deadbeef</td>
<td>()</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>(0000000000</td>
<td></td>
<td></td>
<td></td>
<td>()</td>
<td>rd:01:0:deadbeef</td>
</tr>
</tbody>
</table>

We have replaced some of the line trace with ... to simplify the line trace for this document. We have also added some extra annotation to label the various columns. The line trace shows cache request/response, memory request/response, as well as the tags for all eight cache lines. The first write cache request is passed through to main memory because this is a write-through cache. Since the cache is no-write-allocate, the write cache request does not update the cache state, and the read cache request misses. The read cache request eventually will update the cache state with the data returned from main memory (notice how the tag 0000080 is in set 0 at the beginning of cycle 9.

The basic_hit test includes a series of reads to the same cache line and thus illustrates cache hits:
Notice how this test uses a test memory with a latency of five cycles. The later reads hit in the cache. They do not require memory requests and are returned with a single-cycle hit latency. After looking at the code, it should hopefully be clear how it would be possible to modify the cache CL model to flexibly model a variety of caches with different cache line sizes, cache capacities, write policies, and replacement policies. Creating these kinds of highly parameterized models is much easier when working with CL modeling as compared to RTL modeling.

Let's modify our cache CL model to use a write-allocate policy. We simply need to change the process_cachereq/write member function to update the cache state on a write miss. The required changes are shown below:

```python
def process_cachereq_write( s, cachereq ):
    # Check to see if we hit or miss in cache
    tag = cachereq.addr[5:32]
    idx = cachereq.addr[2:5]
    hit = s.cache[idx].valid and (s.cache[idx].tag == tag)

    # On a cache hit, update the corresponding data
    if hit:
        s.cache[idx].data = cachereq.data

    # On a cache miss, update the valid bit, tag, and data
    if not hit:
```
cache_line = Cacheline()
cache_line.valid = True
cache_line.tag = tag
cache_line.data = cachereq.data

s.cache[idx] = cache_line

# Always send write request to main memory
s.memreq_q.enq( cachereq )

# Set wait bit so cache knows to wait for response message
s.wait_bit = True

If we rerun the basic_hit test we should see the data being allocated in the cache on the very first write cache request.

% cd $BUILD_DIR
% py.test ../cache_cl -k test_iport[basic_hit] -s

Compare this line trace to the previous line trace and notice how the first read cache request now hits in the cache.
Experimenting with changing the write policy in an RTL model would likely require significant changes to the datapath and the control unit, while this experiment is relatively simple to implement in a CL model. If we were evaluating a full system where this cache was just one component, we could use our CL model to help understand the system-level performance impact of no-write-allocate vs. write-allocate, and this would guide us to a promising design before we even write one line of RTL code.

As another example, let’s assume we wish to model a cache with a two cycle instead of single cycle hit latency. All we need to do is buffer the cache response for one extra cycle. First, we add this buffer in the constructor of our cache CL model:

# Extra buffering to model two-cycle hit latency
	s.pipebuf = None
Now we need to modify the `process_cachereq_read` member function as follows to place the cache response in this new buffer instead of immediately sending the cache response out the `cacheresp` interface.

```python
def process_cachereq_read( s, cachereq ):
    # Check to see if we hit or miss in cache
    tag = cachereq.addr[5:32]
    idx = cachereq.addr[2:5]
    hit = s.cache[idx].valid and (s.cache[idx].tag == tag)

    # On a cache hit, return data from cache
    if hit:
        cacheresp = s.cache_ifc_dtypes.resp
        cacheresp.type_ = s.cache_ifc_dtypes.resp.TYPE_READ
        cacherespopaque = cachereq.opaque
        cacheresp.len = cachereq.len
        cacheresp.data = s.cache[idx].data
        s.pipebuf = cacheresp

    # On a cache miss, send out refill request to main memory
    if not hit:
        s.memreq_q.enq( cachereq )

        # Set wait bit so cache knows to wait for response message.
        s.wait_bit = True
        s.wait_addr = cachereq.addr
```

Finally, in the `tick_cl` function we need to copy the cache response from the `pipebuf` and send it out the `cacheresp` interface. Try adding this code at the end of the `tick_cl` function:

```python
if s.pipebuf:
    s.cacheresp_q.enq( s.pipebuf )
    s.pipebuf = None
```

Every cycle we check to see if there is anything in the `s.pipebuf` (i.e., is the `s.pipebuf` not `None`) and if there is indeed something waiting in the `s.pipebuf` we send it out the `cacheresp` interface. We clear the `s.pipebuf` so we know that we are done with this response. If you rerun the `basic_hit` test, the end of the line trace should look like this:

```
% cd $BUILDDIR
% py.test ../cache_cl -k test_simport[basic_hit] -s
```

```
    # cachereq       set 0      memreq       memresp       cacheresp
 21: rd:04:00001000: (00000000 ... ) ( )           ( )         rd:03:0:deadbeef
 22: rd:05:00001000: (00000000 ... ) ( )           ( )         rd:04:0:deadbeef
 23: rd:06:00001000: (00000000 ... ) ( )           ( )         rd:05:0:deadbeef
 24: (00000000 ... ) ( )           ( )         rd:06:0:deadbeef
```

Nothing has changed! Read hits still take a single cycle. The problem is that the `process_cachereq_read` member function is called first in the `s.tick_cl`, we write the `s.pipebuf`, and then at the end of the `s.tick_cl` we send the cache response out the `cacheresp` interface. We need to place the code that sends the cache response out the `cacheresp` interface at the
Beginning of the s.tick_cl (well after the calls to xtick but before we process any cache requests or memory responses). With this change you should see the following line trace:

% cd $BUILDDIR
% py.test ../cache_cl -k test_iport[basic_hit] -s

<table>
<thead>
<tr>
<th>#</th>
<th>cachereq</th>
<th>set 0</th>
<th>memreq</th>
<th>memresp</th>
<th>cacheresp</th>
</tr>
</thead>
<tbody>
<tr>
<td>22:</td>
<td>rd:04:00001000:</td>
<td>(00000000 ... )</td>
<td>( )</td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>23:</td>
<td>#</td>
<td>(00000000 ... )</td>
<td>( )</td>
<td>rd:00:0:deadbeef</td>
<td></td>
</tr>
<tr>
<td>24:</td>
<td>rd:05:00001000:</td>
<td>(00000000 ... )</td>
<td>( )</td>
<td>rd:04:0:deadbeef</td>
<td></td>
</tr>
<tr>
<td>25:</td>
<td>#</td>
<td>(00000000 ... )</td>
<td>( )</td>
<td>rd:05:0:deadbeef</td>
<td></td>
</tr>
<tr>
<td>26:</td>
<td>rd:06:00001000:</td>
<td>(00000000 ... )</td>
<td>( )</td>
<td>rd:06:0:deadbeef</td>
<td></td>
</tr>
</tbody>
</table>

This is still not correct. Now a cache read hit is taking three cycles and it looks like there is some extra occupancy instead of the full-throughput pipeline behavior we were hoping for. The problem now is that our s.tick_cl function is still checking to see if the cacheresp interface is ready; we should really be checking to see if the s.pipebuf is empty or not. The final correct s.tick_cl implementation should look like this:

```python
@s.tick_cl
def tick():
    # Tick adapters
    s.cachereq_q_xtick()
    s.cacheresp_q_xtick()
    s.memreq_q_xtick()
    s.memresp_q_xtick()

    # Some constants to simplify the code
    s.READ = s.cache_ifc_dtypes.req.TYPE_READ
    s.WRITE = s.cache_ifc_dtypes.req.TYPE_WRITE

    # Model an extra cycle of latency for read hits
    if s.pipebuf:
        s.cacheresp_q_enq(s.pipebuf)
        s.pipebuf = None

    # If we are not waiting for a memory response ...
    # Notice that we check s.pipebuf not s.cacheresp_q_full()
    if not s.wait_bit and not s.cachereq_q_empty() \
        and not s.memreq_q_full() \
        and not s.pipebuf:
        cachereq = s.cachereq_q_deq()

        if cachereq.type_ == s.WRITE: s.process_cachereq_write(cachereq)
        elif cachereq.type_ == s.READ: s.process_cachereq_read(cachereq)
        else: assert False

    # If we are waiting for a memory response ...
    if s.wait_bit and not s.memresp_q_empty() \
        and not s.cacheresp_q_full():
```
memresp = s.memresp_q.deq()

if memresp.type_ == s.WRITE : s.process_memresp_write( memresp )
elif memresp.type_ == s.READ  : s.process_memresp_read ( memresp )
else : assert False

And the resulting line trace should look like this:

% cd $BUILDDIR
% py.test ../cache_cl -k test_iport[basic_hit] -s

<table>
<thead>
<tr>
<th># cachereq</th>
<th>set θ</th>
<th>memreq</th>
<th>memresp</th>
<th>cacheresp</th>
</tr>
</thead>
<tbody>
<tr>
<td>21: rd:04:00001000: (00000000000000000000000000000000)</td>
<td>( )</td>
<td>( )</td>
<td>rd:03:0:deadbeef</td>
<td></td>
</tr>
<tr>
<td>22: rd:05:00001000: (00000000000000000000000000000000)</td>
<td>( )</td>
<td>( )</td>
<td>rd:04:0:deadbeef</td>
<td></td>
</tr>
<tr>
<td>23: rd:06:00001000: (00000000000000000000000000000000)</td>
<td>( )</td>
<td>( )</td>
<td>rd:05:0:deadbeef</td>
<td></td>
</tr>
<tr>
<td>24: (00000000000000000000000000000000)</td>
<td>( )</td>
<td>( )</td>
<td>rd:06:0:deadbeef</td>
<td></td>
</tr>
</tbody>
</table>

This is our desired cycle-level behavior. Cache read hits have a two cycle latency, yet we can still sustain full throughput.
One of the key take-aways is that developing a CL model which does a reasonable job of modeling the cycle-level behavior can be challenging and often requires iteratively experimenting with small performance-focused test cases.

**gem5 CL Simulator**

Most computer architects in industry and academia spend far more time developing CL models instead of RTL models. Computer architects will often start with a previously developed CL model and then experiment with new computer architecture ideas by modifying this CL model. A great example is the gem5 CL simulator. This simulator is written in C++ and includes detailed CL models for processors, memories, and networks. It includes complex processor CL models which model superscalar execution, out-of-order execution, register renaming, memory disambiguation, branch prediction, and speculative execution.

Here is an example of an assembly version of the vector-vector add microbenchmark running on the gem5 quad-issue out-of-order processor. The output from the simulator is post-processed to create a kind of pipeline view.

```
[fdn.ic.........................r.........................] 0x00000020.0 lw r6, θ(r2)
[fdn.pic.........................r.........................] 0x00000024.0 lw r7, θ(r3)
[fdn.p.........................ic.r.........................] 0x00000028.0 addu r8, r6, r7
[fdn.p.........................ic.r.........................] 0x0000002c.0 sw r8, θ(r4)
[....fdn.ic.........................r.........................] 0x00000030.0 addiu r2, r2, 4
[....fdn.ic.........................r.........................] 0x00000034.0 addiu r3, r3, 4
[....fdn.ic.........................r.........................] 0x00000038.0 addiu r4, r4, 4
[....fdn.ic.........................r.........................] 0x0000003c.0 addiu r5, r5, 1
[....fdn.ic.........................r.........................] 0x00000040.0 bne r5, r1, 0x808020
[....fdn.ic.........................r.........................] 0x00000020.0 lw r6, θ(r2)
[....fdn.pic.........................r.........................] 0x00000024.0 lw r7, θ(r3)
[....fdn.p.........................ic.r.........................] 0x00000028.0 addu r8, r6, r7
[....fdn.p.........................ic.r.........................] 0x0000002c.0 sw r8, θ(r4)
[.........fdn.ic.........................r.........................] 0x00000030.0 addiu r2, r2, 4
[.........fdn.ic.........................r.........................] 0x00000034.0 addiu r3, r3, 4
[.........fdn.ic.........................r.........................] 0x00000038.0 addiu r4, r4, 4
[.........fdn.ic.........................r.........................] 0x0000003c.0 addiu r5, r5, 1
[.........fdn.pic.........................r.........................] 0x00000040.0 bne r5, r1, 0x808020
```

Here is the legend:

- f = fetch stage, d = decode stage, n = rename stage
Here is the above pipeline view carefully transformed into a pipeline diagram similar in spirit to what we learned in ECE 4750.

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
-----------------------------------------------
lw r6, 0(r2)  F D N R I X q . . . . . . . . . . . . M W r . C lw -> LSQ, D$ miss, resp on cyc 17 in M
lw r7, 0(r3)  F D D N R I X q . . . . . . . . . . . M W r r . C lw -> LSQ, D$ miss, resp on cyc 17 in M
addu r8, r6, r7 F D N R I  . . . . . . . . . . . . . . . I X W r . C RAW hazard, addu waits in IQ for lw insts
sw r8, 0(r4)  F D N R I X q . . . . . . . . . . . . . . . I X W r C RAW hazard, sw waits in IQ for addu
addiu r2, r2, 4 F D N R I X W r . . . . . . . . . . . . . . . . . C
addiu r3, r3, 4 F D N R I X W r . . . . . . . . . . . . . . . . . C
addiu r4, r4, 4 F D N R I X W r . . . . . . . . . . . . . . . . . C
addiu r5, r5, 1 F D N R I I X W r . . . . . . . . . . . . . . . . . C struct hazard, only 4 issue ports
bne r5, r1, loop F D N R i I X W r . . . . . . . . . . . . . . . . . C RAW hazard, bne waits in IQ for addiu r5
-----------------------------------------------
```

And here is the legend:

- F: fetch
- D: decode
- N: rename
- R: ROB allocation
- I: issue
- X: execute for arithmetic operations
- M: memory access completion for ld/st operations
- W: writeback
- C: commit
- i: waiting in issue queue
- r: waiting in ROB
- q: waiting in LSQ for memory response to come back

From the pipeline diagram we can see how the CL model is capturing many of the complex features we learned in ECE 4750. It is faithfully tracking RAW hazards and structural hazards, and it is modeling cache effects, aggressive branch prediction, and fully out-of-order load/store issue. Implementing this kind of processor in RTL would require a tremendous design team and would never be as flexible and extensible as the gem5 CL simulator, which is why architects usually prefer CL models when doing early design-space exploration. Of course, the challenge with a CL model is it is possible to model completely unrealistic hardware, and it is very challenging to estimate the area, energy, and timing of a specific design. So
in this course, we argue for a balanced approach that uses a mix of FL, CL, and RTL modeling.