Behavioral Modeling in HW/SW Co-design Using C++ Coroutines

JEFFREY ERICKSON & SEBASTIAN SCHOENBERG

Cppcon
The C++ Conference

2023

October 01 - 06

cppcon 2023

Behavioral Modeling in HW/SW Co-design using C++ coroutines

Jeffrey E Erickson, Ph.D.
Sebastian Schönberg, Ph.D.

intel.

A guide to this presentation

First: A story and a problem

Then: Address the problem

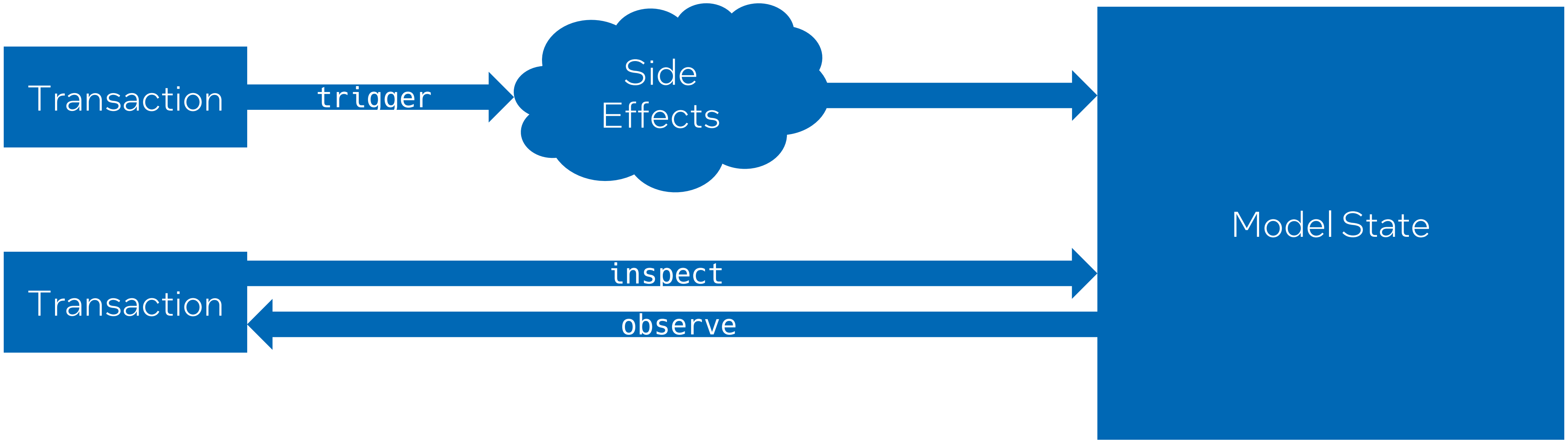
Finally: Integrate with production

A Story and a Problem

- You’re an embedded software developer working on a new HW peripheral that is in development.
- Timelines are tight and you want to be able to work on the SW that interacts with this HW early – so-called “shift left”
- You could:
  - Wait for RTL to be developed and run your code on a simulator or emulator
  - Use the transactional/behavioral/functional definition to model the HW using SW elements

The Pace of RTL and C++

- Often there is a goal to have SW ready when HW is ready
- How do we shift the production of the SW earlier?
  - Answer: We need a model!



Level of Detail – Time quanta and synchronization

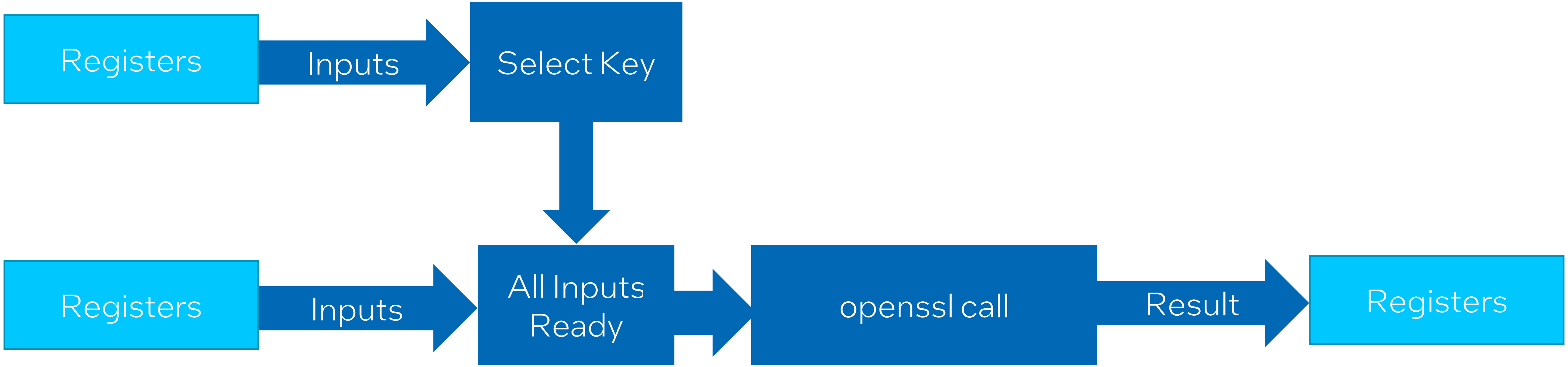
- How often do we really care to know the state of the model?
- Consider a Public Key Cryptography Accelerator

What happens in HW:

What we need to see:

Parallelism concerns as complexity rises

- Consider a Public Key Cryptography Accelerator, plus a Key Manager, plus some additional HW, all of which need to interact



Coroutines are an enabling language feature

- To date, coroutines have been about execution and workloads
- In the modeling context, coroutines…
  - Provide syntactic sugar for things like state machines
  - Look more like RTL
  - Handle parallelism in a way that isn’t clunky
  - Scale across large and complex HW interactions

But, coroutines lack comprehensive standard library support

- We used concurrencpp

Implementing Coroutines

Keywords

- `co_await <arg>` or `lhs = co_await <arg>`
  - Pause execution of the coroutine until the arg becomes ready
  - lhs applies only when waiting for a coroutine that yields
- `co_yield <arg>`
  - Pause execution and provides a value to the awaiter
- `co_return <arg>`
  - Ends execution of the coroutine and optionally returns a value

Understanding the pattern

- Coroutines are about ‘promises’…
  - Eventually the promise delivers a result, which may be void
- As an implementer, you specify a function-like object
- That object is scheduled by a runtime
- Coroutines use a time-slicing model
  - One coroutine runs at a time until it cannot execute any longer
  - Some other coroutine then runs until it cannot execute any longer
  - Eventually the first coroutine is able to continue and the pattern goes on

Syntactic breakdown

- It looks like a function but it’s not
- You’re providing a callable-like definition of a coroutine

```
concurrencpp::result<uint32_t> foo(){
      co_return 0x0000FFFF;
}
```

- Assignment gets an awaitable, not the result

```
auto bar = foo(); // bar is an awaitable, not uint32_t
```

How do we get the value back?

- In another coroutine

```
uint32_t val = co_await foo();
```

- From outside a coroutine

```
auto bar = foo(); // starts foo()
… //do other things
uint32_t val = bar.get(); // returns value
// Or if you just want to wait until the value is ready:
uint32_t val = foo().get();
```

C++ Implementation patterns

Threads and coroutines

Threads vs Coroutines

How to wait for a condition to become ready?

- Thread A (which sets the condition ready) needs to **gain a lock**
- Thread B (which waits for the condition) needs to **reach the wait condition, then attempt to gain the lock**
- Thread A releases the lock when 'ready'

- Coroutine A defines the logic that indicates 'ready'
- Coroutine B co_awaits coroutine A and becomes un-ready
- When coroutine A's logic is met, it becomes ready and co_returns
- Coroutine B's co_await is then satisfied and Coroutine B continues

Threads vs Coroutines

How to wait for a condition?

- Thread A

```
std:binary_semaphore data_rdy{1};
…
data_rdy.release();
…
data_rdy.acquire();
```

- Thread B

```
…
data_rdy.acquire(); //blocks
…
data_rdy.release();
```

- Coroutines

For coroutine A, define "`data_ready`" to `co_return` when ready

In coroutine B:

```
co_await data_ready();
```

How threads become untenable

- Thread locking mechanisms are designed for shared resources
- They don’t handle waiting for combinatorial conditions well
- Could use a scoped lock or latch/barrier, but need application of DeMorgan’s theorem
- Example:

```
std::mutex lock[100]; // Needs to be an ‘inverted’ mutex
…
std::scoped_lock joint_lock(lock[1], lock[2]); // Simple enough
…
// Now let’s lock on 10 conditions
std::scoped_lock wide_lock(lock[0], lock[1], lock[2], lock[3],
lock[4], lock[5], lock[6], lock[7], lock[8], lock[9]);
```

Coroutines avoid the thread problems

- Because co_await is just awaiting for a ready condition, the syntax for multiple condition waits is cleaner

```
List<result> conditions = {cond0, … , condN};
…
// Variatic Argument Approach
co_await concurrencpp::when_all(cond0, cond1, cond2);
…
// Iterator approach
co_await concurrencpp::when_all(conditions.begin, conditions.end);
```

Execution Differences when using Coroutines

- Yes, you’re still spawning threads, but a backend runtime is doing it
- Yes, deadlocks can occur, but they are generally easier to find
- Following some safe patterns will avoid most of this:
  - Don’t loop without delay
  - Don’t create simple cycles of co_awaits between coroutines without delay
  - Follow best practices design patterns, like switch/case state machine

Coroutine Patterns

Common Implementations

- Waiting
- Waiting for multiple conditions
- Delaying
- Generators
- Starting a coroutine with a callable

How to Wait

- Single Condition

```
co_await coro(a,b,c);
```

- Multiple condition ‘AND’

```
co_await concurrencpp::when_all(coro_a(), coro_b());
```

- Multiple condition ‘OR’

```
co_await concurrencpp::when_any(coro_a(), coro_b());
```

How to Delay and when to Delay

- Concurrencpp has a built in timer queue that handles delays

```
make_delay_object(std::chrono::milliseconds,concurrencpp::executor);
co_await tq->make_delay_object(10ms, runtime.thread_pool_executor());
```

- As mentioned, delay within loops.
- Multiple short delays are generally helpful to allow the runtime to check for ‘ready’ coroutines
- Realize that this time is virtual/fungible (e.g. 10ms in the runtime could be anything -- 10us, 10ns, 10ps)

State Machine Pattern

```
while(!reset) {
      switch(state) {
            case STATE_1:
                  state = co_await wait_condition_1();
                  break;
            case STATE_2:
                  co_await tq->make_delay_object(1ms, exec);
                  state = STATE_1;
                  break;
            default:
                  // error trap
                  break;
      }
}
```

Generators

- Previously mentioned co_yield
- Generator implementation

```
concurrencpp::result<uint32_t> gen_uint(){
      uint32_t n = 0;
      while(true){
            co_yield n;
            n = n+1;
      }
}
```

- Consumer implementation:

```
next_int = co_await gen_uint();
```

Callables and Coroutines

- A function can be scheduled in the coroutine context
- Similar to future/async
- Any callable can be used

```
runtime.thread_pool_executor()->submit(/* callable */);
```

Ex:

```
runtime.thread_pool_executor()->submit([&](){ foo().wait(); });
```

Do not neglect atomicity

- Atomicity within an object may be important…
- If the runtime is a thread or thread pool, then there is real parallelism
- Consider atomic operations in your coroutines
  - They generally won't hurt
  - They generally enable more scalability
- Alternatively, consider lock-less implementations

Practical Example

The Queue

- Defined Capacity (length)
- Elements of type T
- Push
  - If not full, complete immediately
  - If full, wait until not full
- Pop
  - If not empty, complete immediately
  - If empty, wait until not empty

The ‘push’ coroutine – First Order Concept

```
concurrencpp::result<void> push(const T& value)
{
     theQueue.push(value);
     co_return;
}
```

The ‘push’ coroutine – Second Order Concept

```
concurrencpp::result<void> push(const T& value)
   {
       bool success = false;
       while(!success){
           if(theQueue.size() < max_elements){
               theQueue.push(value);
               success = true;
               break;
           }
       }
       co_return;
   }
```

The ‘push’ coroutine – Third Order Concept

```
concurrencpp::result<void> push(
     std::shared_ptr<concurrencpp::thread_pool_executor> tpe,
     std::shared_ptr<concurrencpp::timer_queue> tq,
     const T& value)
   {
       bool success = false;
       while(!success){
           if(theQueue.size() < max_elements){
               theQueue.push(value);
               success = true;
               break;
           }
           co_await tq->make_delay_object(10ms, tpe);
       }
       co_return;
   }
```

The ‘pop’ coroutine – First Order Concept

```
concurrencpp::result<T> pop()
    {
        T return_value;
        return_value = theQueue.front();
        theQueue.pop();
        co_return return_value;
    }
```

The ‘pop’ coroutine – Second/Third Order Concept

```
concurrencpp::result<T> pop(
std::shared_ptr<concurrencpp::thread_pool_executor> tpe,
std::shared_ptr<concurrencpp::timer_queue> tq)
    {
        T return_value;
        while(theQueue.empty()){
            co_await tq->make_delay_object(1ms, tpe);
        }
        return_value = theQueue.front();
        theQueue.pop();
        co_return return_value;
    }
```

Interacting with the Coroutine Model

Register Hooking

- We have a coroutine model of some complex process. Great!
- Now, how do we interact with it?
- Create a `HookableRegister`
  - Ben Saks from cppcon22 calls this a `device_register`
- We offer two extensions
  - How to hook via preprocessor and typedef
  - How to handle cross-language use of the model

Hooking with the preprocessor

- Use the preprocessor to determine if you are using the models via a compiler flag, then:

```
#ifdef USE_MODELS
typedef iouint32_t HookableRegister;
#else
typedef iouint32_t uint32_t;
#endif
```

Why? Often there are embedded requirements that mean application code must be written in C, this gives an easy way to not modify the C code but use the C++ coroutine models.

Integrating the side effect handler with the coroutine

- Start the coroutine

```
HookableRegister& operator=(const uint32_t& rhs){
      auto push = test_queue.push(runtime, rhs).wait();
      return *this;
}
```

- Return by ‘getting’ the result<T> promise

```
operator uint32_t() const{
      auto pop = test_queue.pop(runtime, runtime.timer_queue());
      return pop.get();
}
```

Handling parallel effects

- How do we get coroutines to run in ‘parallel’? Start multiple coroutine contexts

```
auto first_op = runtime.thread_pool_executor()->submit(
       [&](){
              auto p = some_coroutine().wait();
       }
);
auto second_op = runtime.thread_pool_executor()->submit(
       [&](){
              auto p = some_other_coroutine().wait();
       }
);

first_op.wait();
second_op.wait();
// Results now ready
```

Testing

- After you've reached a condition because the 'right' combination of coroutines have completed, then assert as normal
- Could inspect through coroutines with `<coroutine>.get();`
- Might want to consider back-end inspection mechanisms
- Coroutines can also be used for a more 'classic' purpose: awaiting I/O to external HW or files.
  - Well designed coroutines for this case should be easily substitutable between the SW and HITL implementations.

The Takeaway Message

What I hope you can walk away with…

- Coroutines are a great modeling tool!
- Coroutines make complex code easier to write
- You can create complex 'parallel' models with relative ease
- Your more HW focused friends will probably find coroutine based models easier to read than thread-based ones
- This is a really cool language feature that you should try!
- Coroutine use cases are more than just execution and workloads. There are meaningful use cases in modeling and test.

intel®

References

- Concurrencpp
  - https://github.com/David-Haim/concurrencpp
- Cppcon2022 talks
  - An Introduction to Multithreading in C++20 by Anthony Williams
    - https://www.youtube.com/watch?v=A7sVFJLJM-A
  - Simulating Low Level Hardware Devices in C++ by Ben Saks
    - https://www.youtube.com/watch?v=zqHvN8xpuKY

Copyright/License Acknowledgements (1)

This presentation uses Hack as a code font.

Copyright/License Acknowledgements (2)

This presentation references concurrencpp.