UMOC: Unified Modular Ordering Constraints to Unify Cycle- and Register-Transfer-Level Modeling

Shunning Jiang, Yanghui Ou, Peitian Pan, Christopher Batten
Computer Systems Laboratory
Cornell University

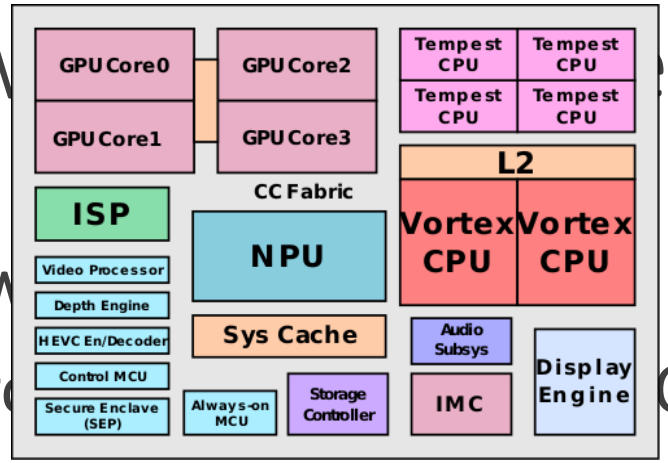
DEC 5 - 9, 2021 ◆ SAN FRANCISCO, CALIFORNIA

Hardware Design Trend

- Hardware Specialization!
- Heterogeneous System-on-Chips (SoC)

Hardv end

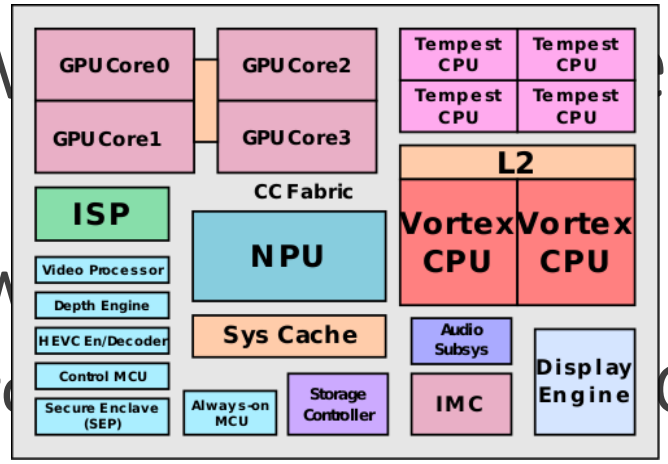
- Hardw
- Heter Chips (SoC)



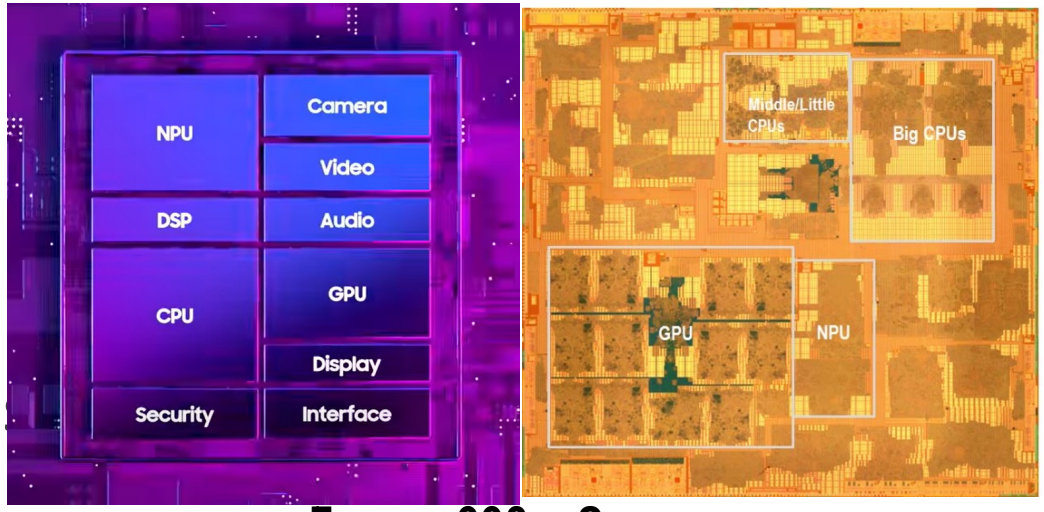
A12 Bionic – Apple

Hardv end

- Hardw
- Heter Chip



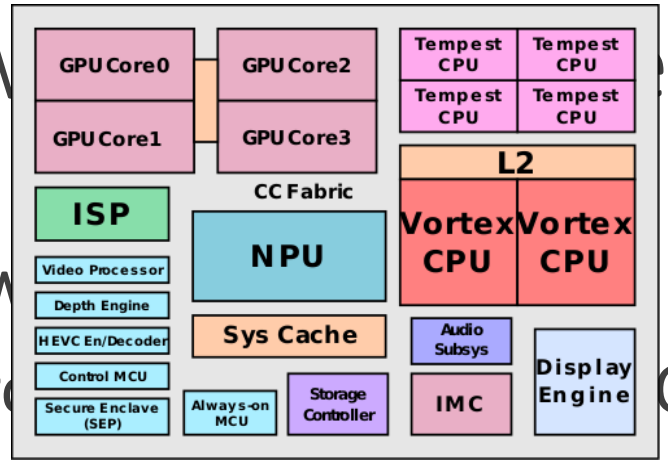
A12 Bionic – Apple



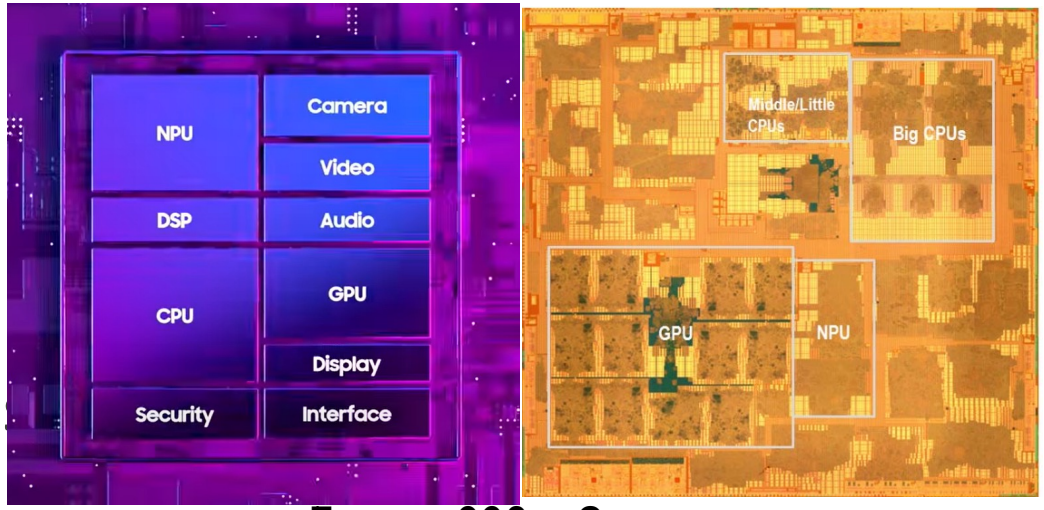
Exynos 990 – Samsung

Hardv end

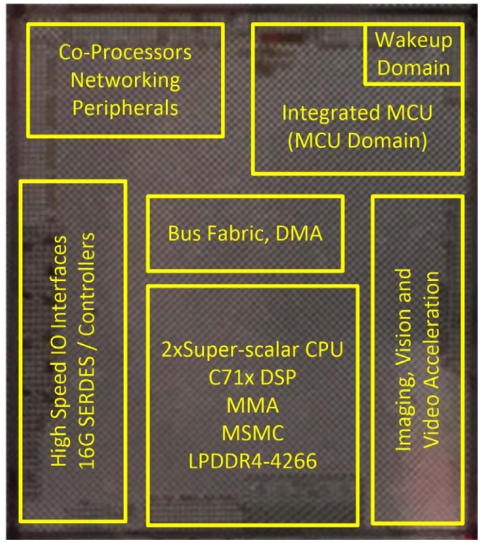
- Hardw
- Heter Chip



A12 Bionic – Apple



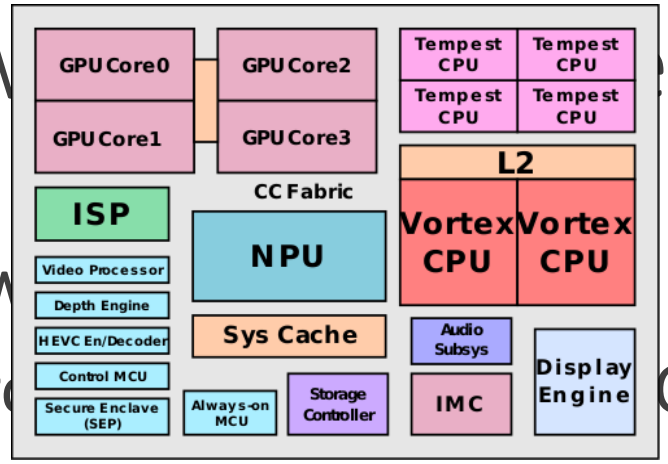
Exynos 990 – Samsung



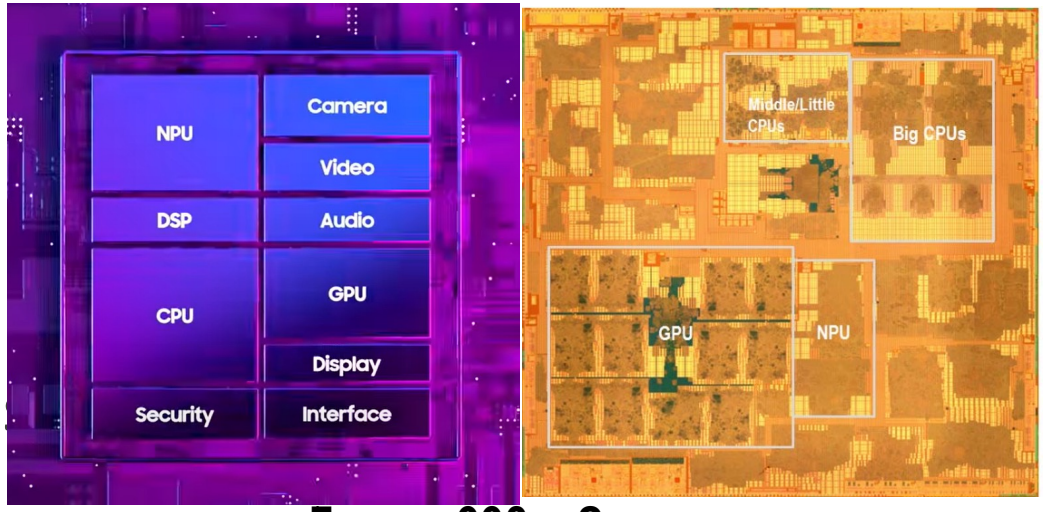
Jacinto – Texas Instrument

Hardv end

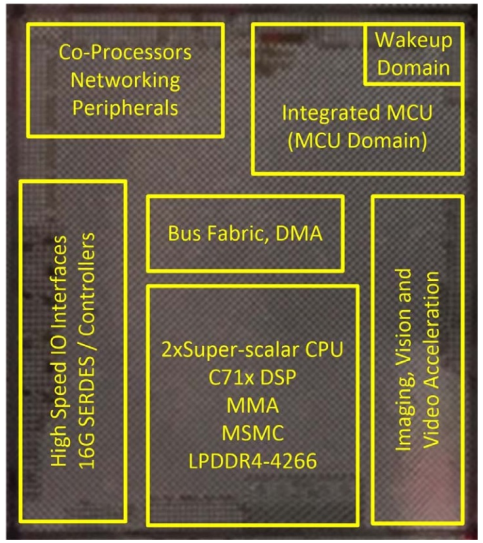
- Hardw
- Heter Chip



A12 Bionic – Apple



Exynos 990 – Samsung



Jacinto – Texas Instrument

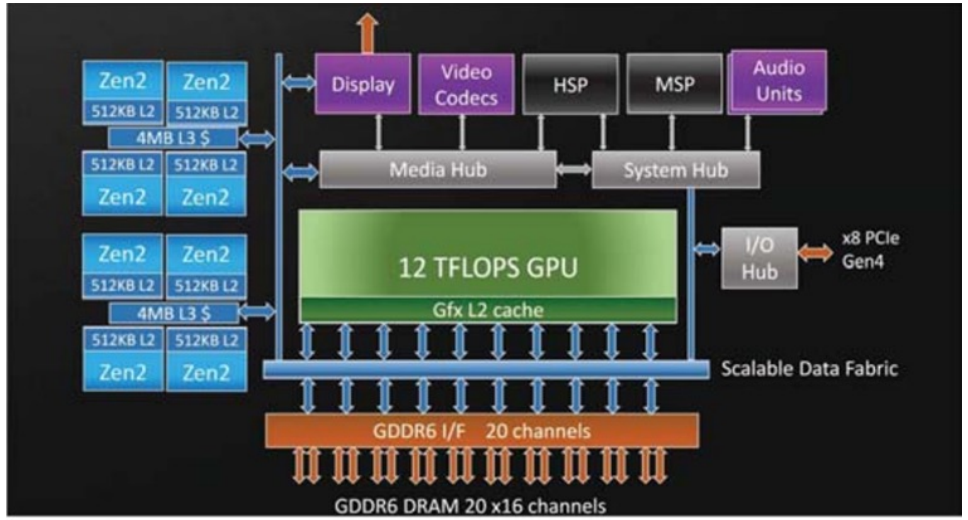
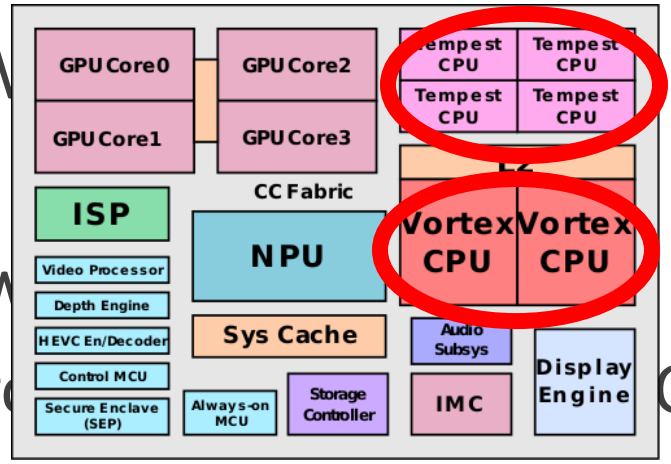


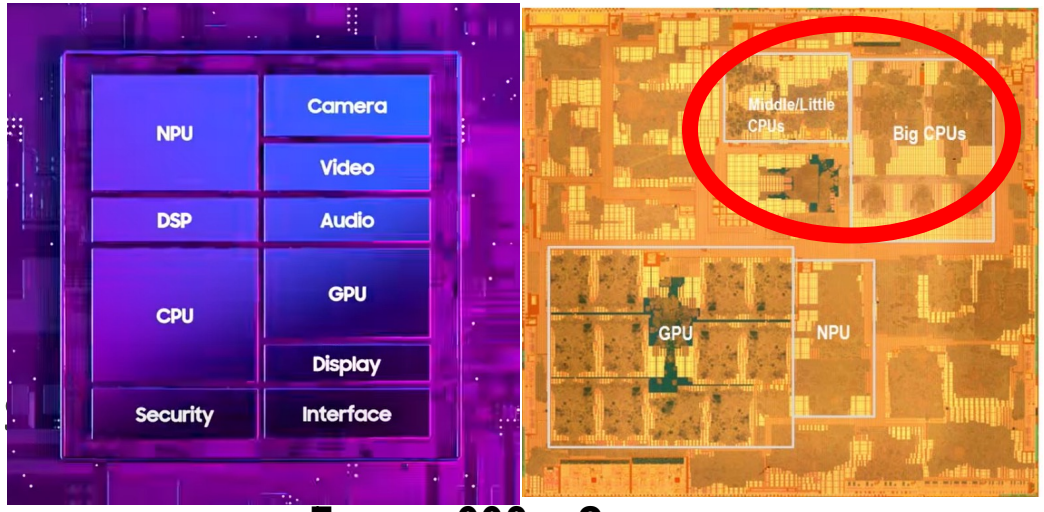
Figure 3.1.2: Xbox Series X SoC architecture block diagram.

Hardv nd

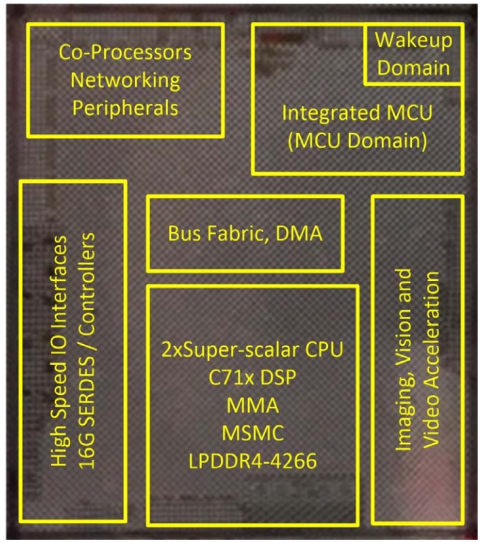
- Hardw
- Heter Chip



A12 Bionic – Apple



Exynos 990 – Samsung



Jacinto – Texas Instrument

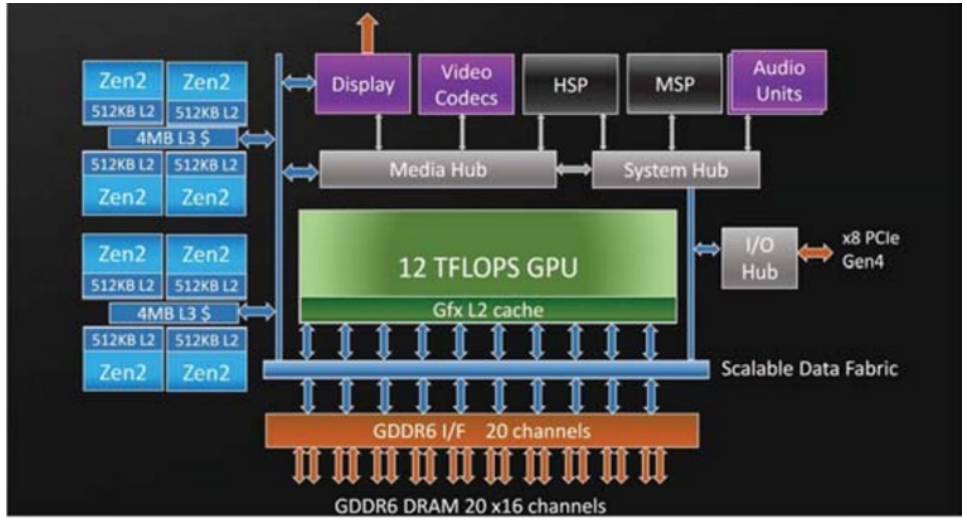
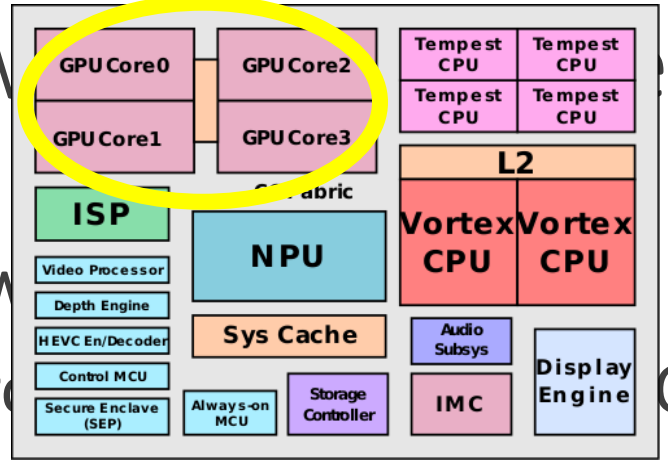


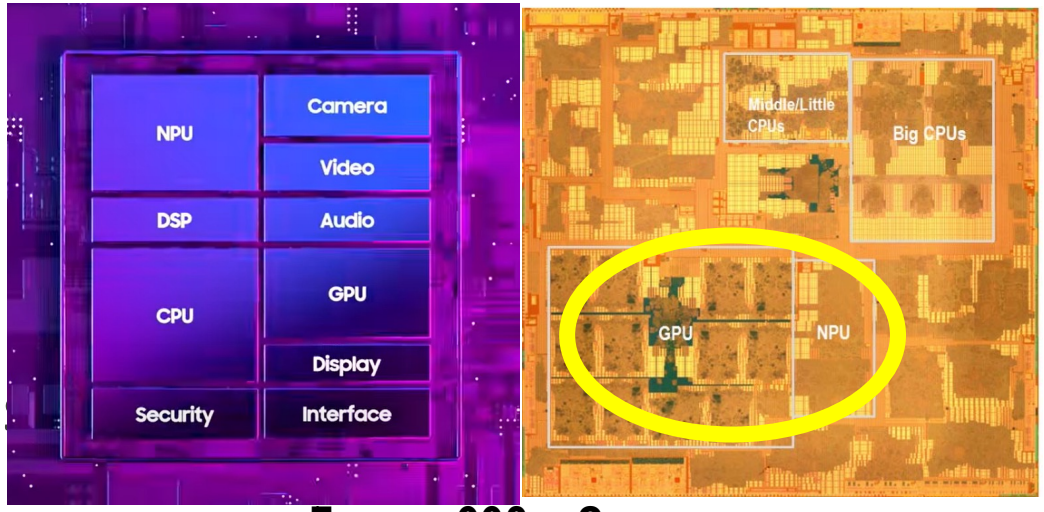
Figure 3.1.2: Xbox Series X SoC architecture block diagram.

Hardv end

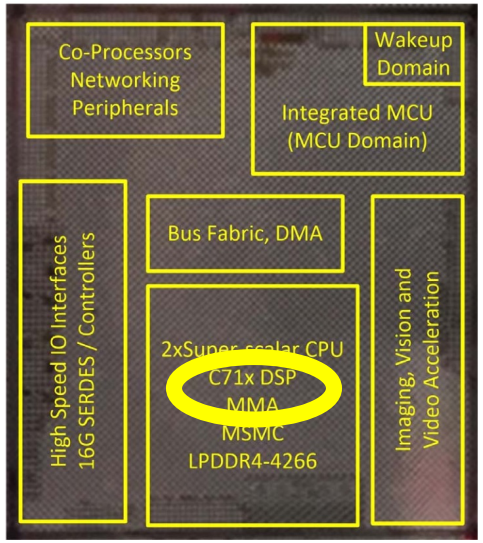
- Hardw
- Heter Chip



A12 Bionic – Apple



Exynos 990 – Samsung



Jacinto – Texas Instrument

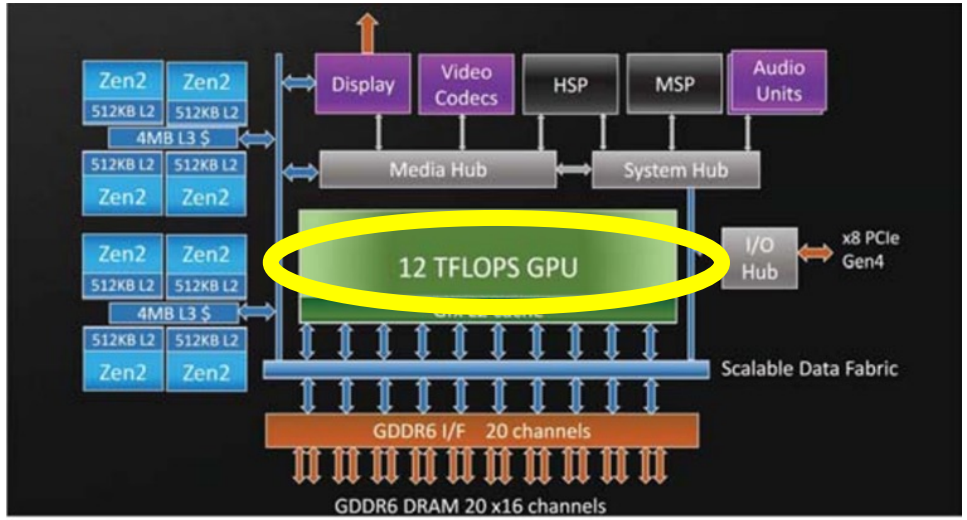
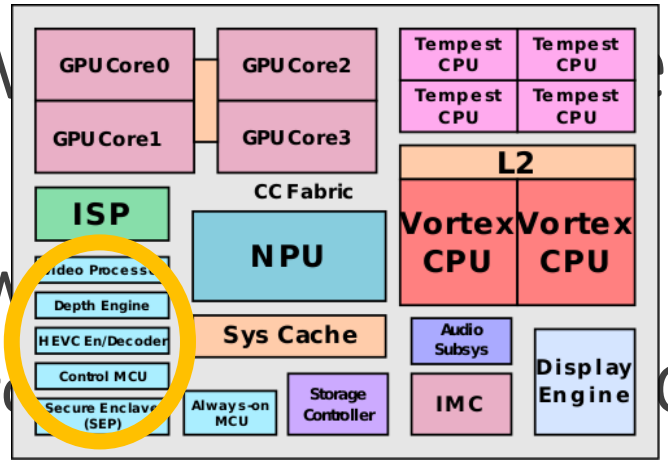


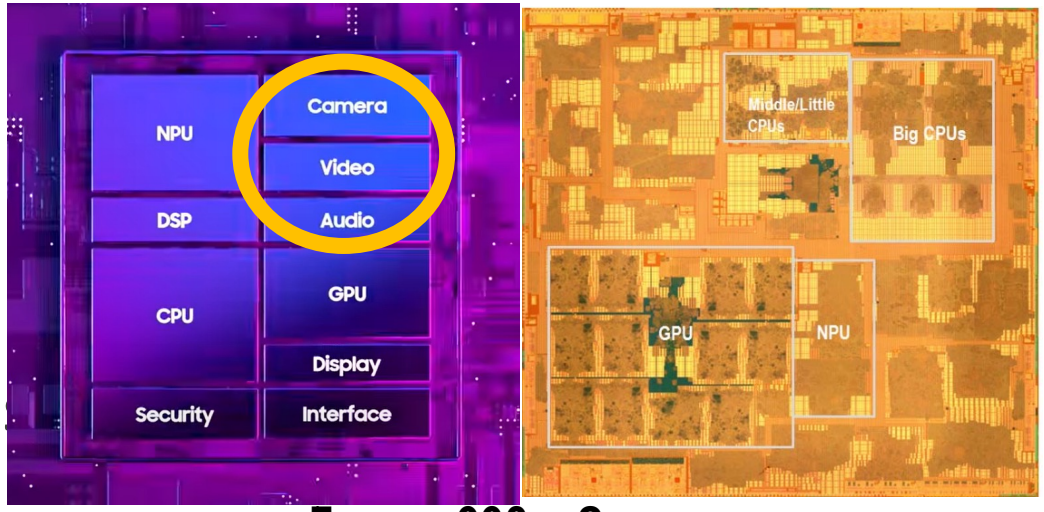
Figure 3.1.2: Xbox Series X SoC architecture block diagram.

Hardv end

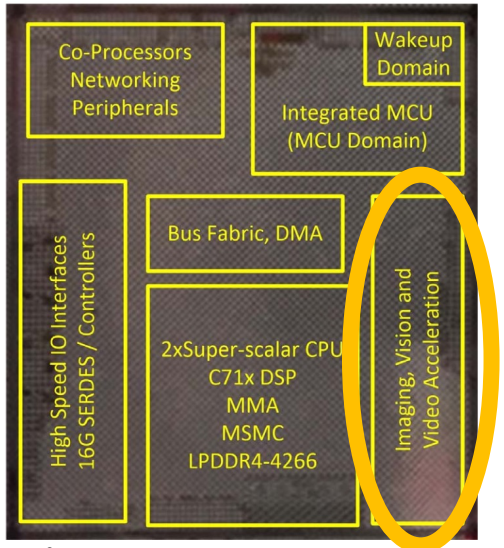
- Hardw
- Heter Chip



A12 Bionic – Apple



Exynos 990 – Samsung



Jacinto – Texas Instrument

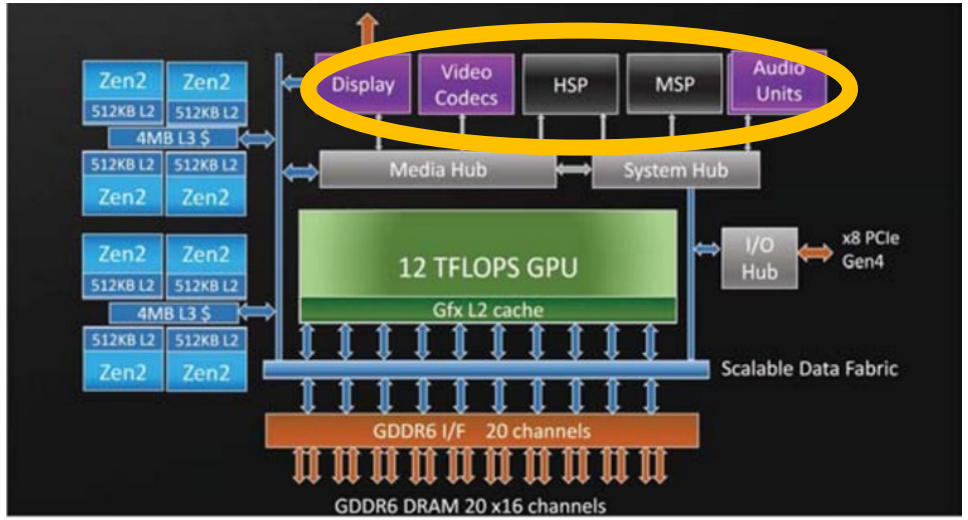
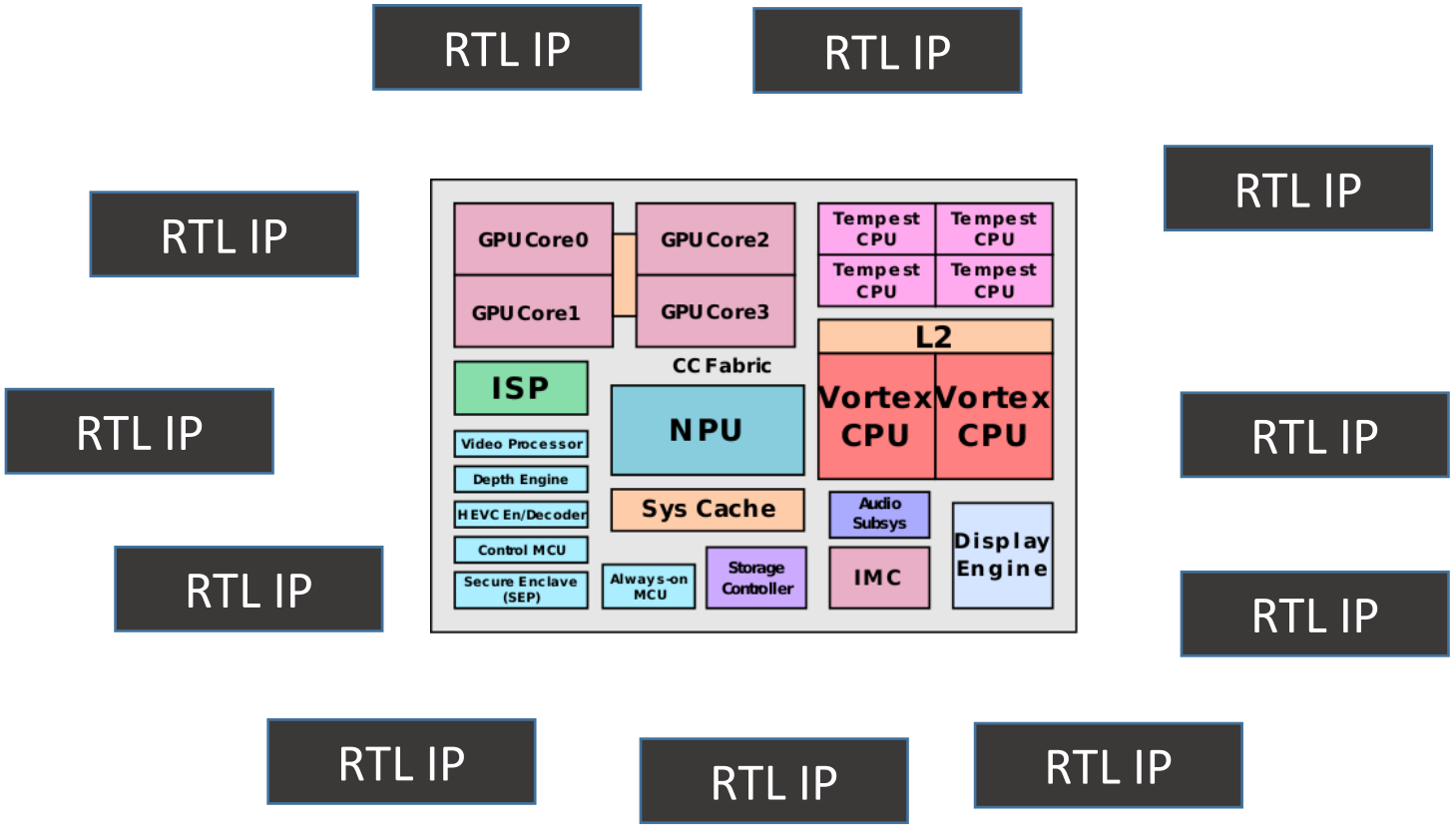


Figure 3.1.2: Xbox Series X SoC architecture block diagram.

Cycle-Level Simulators/Models for Design Space Exploration



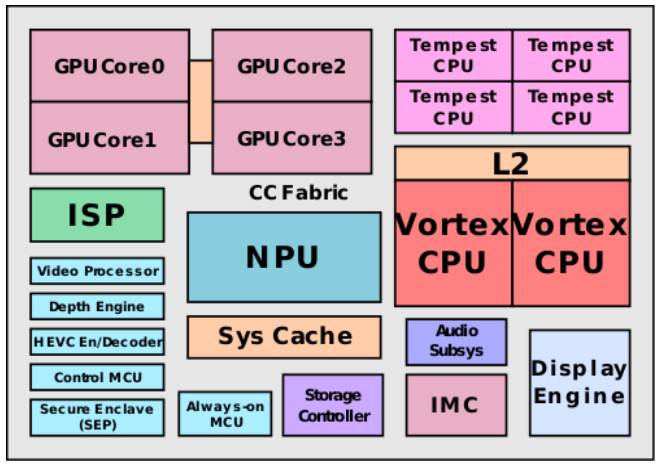
Cycle-Level Simulators/Models for Design Space Exploration

Choice#1: $80,000
Choice#2: $90,000

RTL IP

Choice#1: $.....
Choice#2: $.....

Choice#1: $.....
Choice#2: $.....



RTL IP

RTL IP

RTL IP

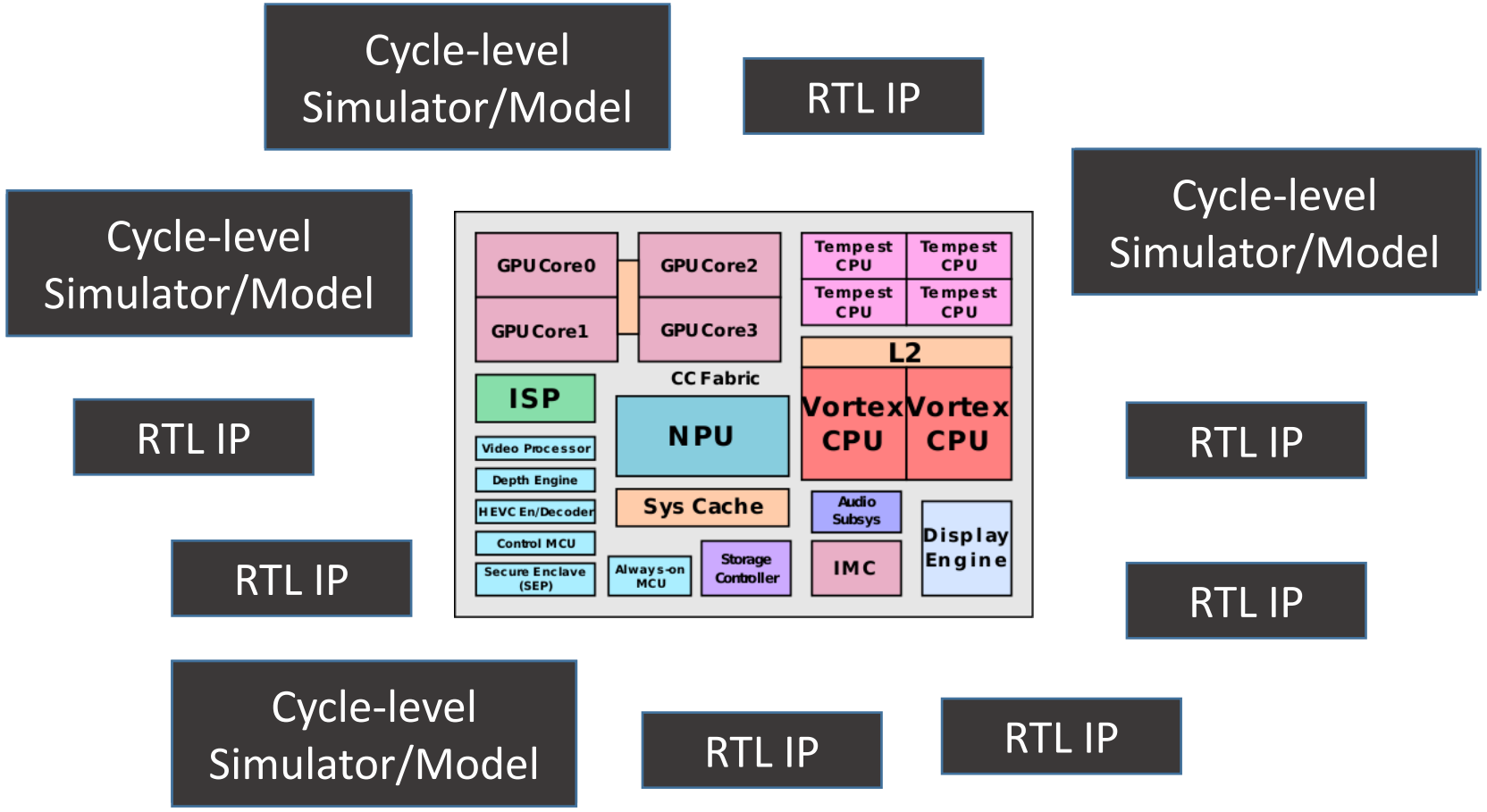
RTL IP

Choice#1: $.....
Choice#2: $.....

RTL IP

RTL IP

Cycle-Level Simulators/Models for Design Space Exploration



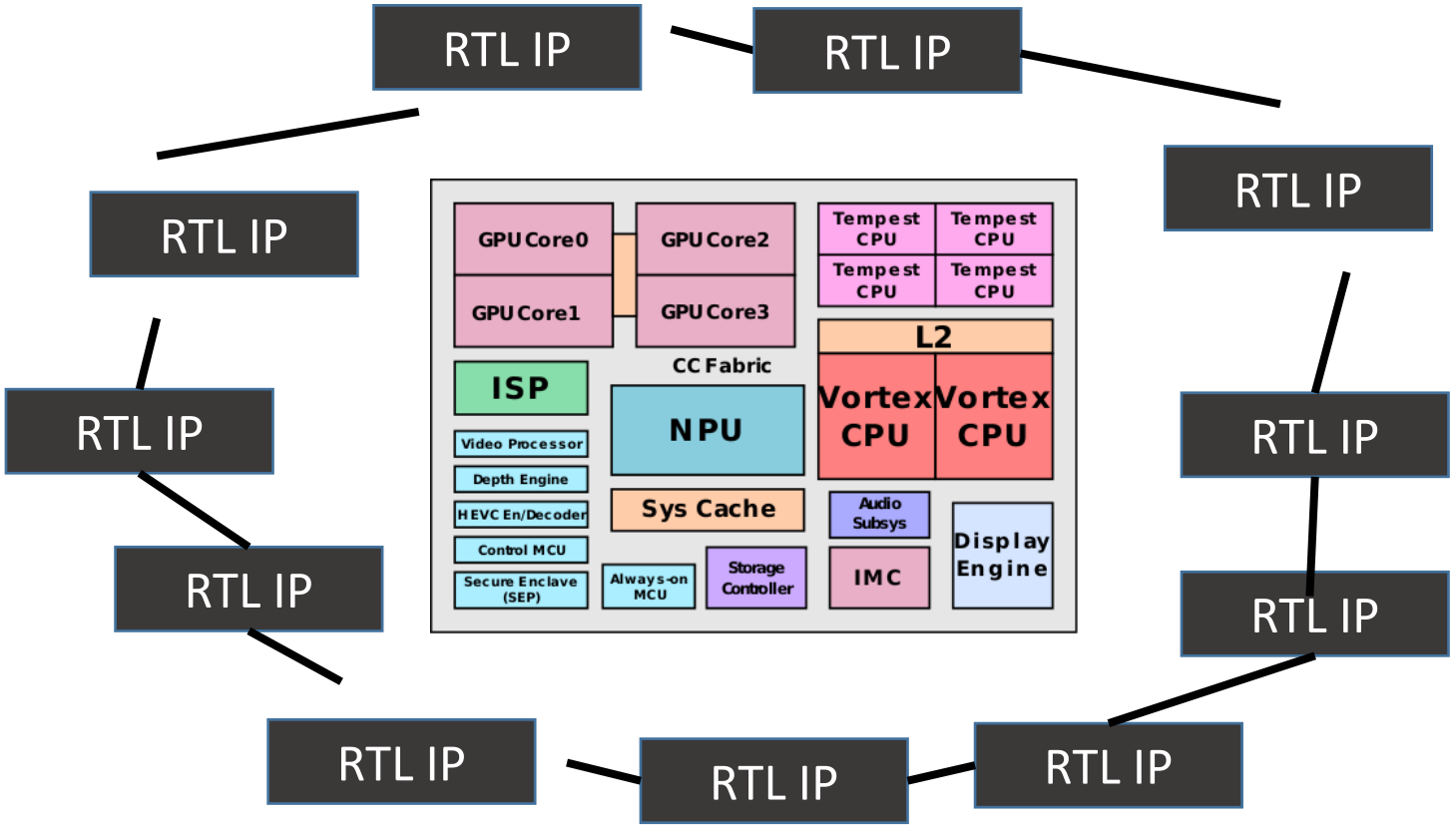
Cycle-level (CL) modeling:

- Approximate timing behaviors
- Analytical area, energy, timing models

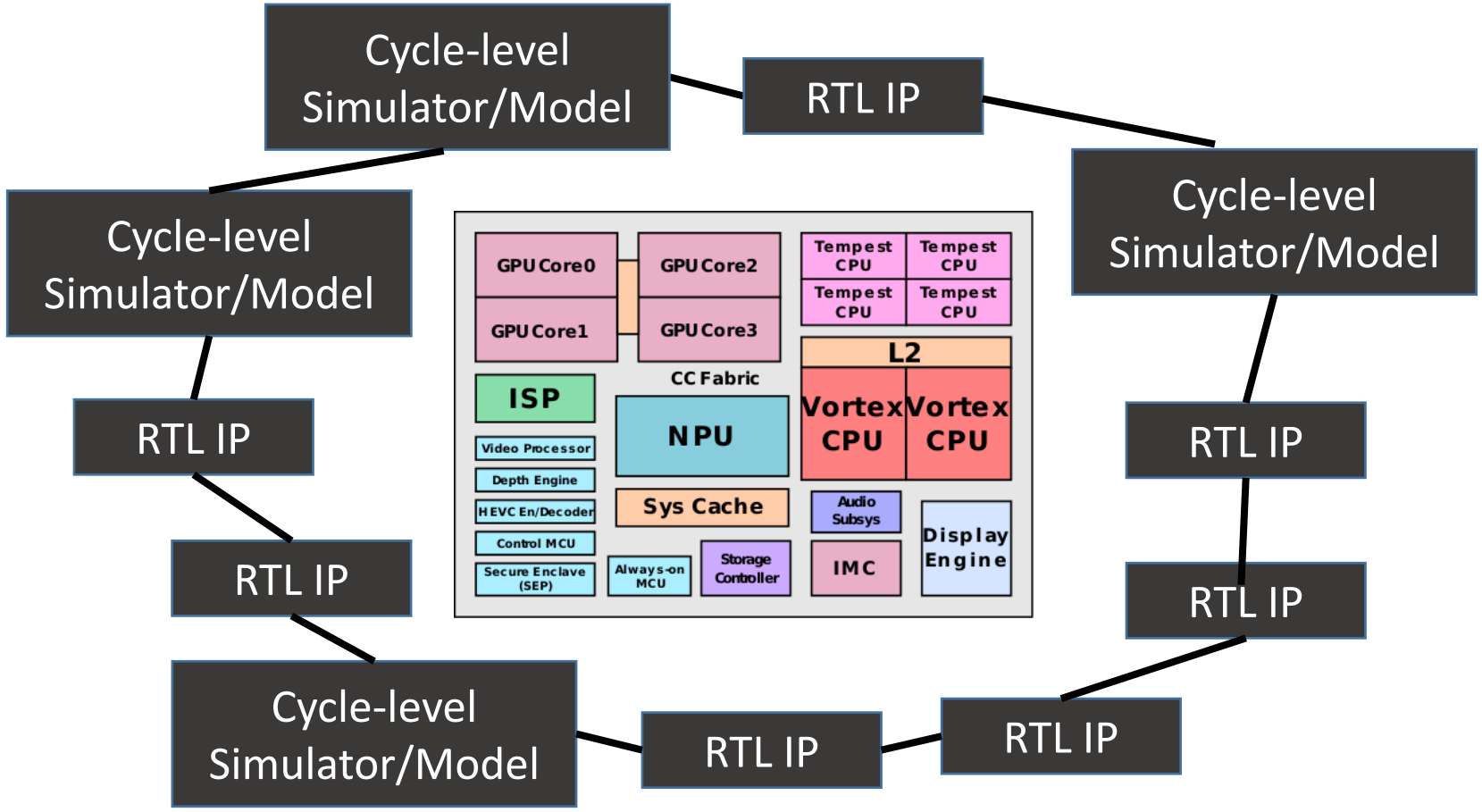
CL models provide valuable insights to help make first-order design decisions

(e.g., cycle-level "N-cycle hit-latency cache")

Composing Cycle-Level/RTL Models for Design Space Exploration



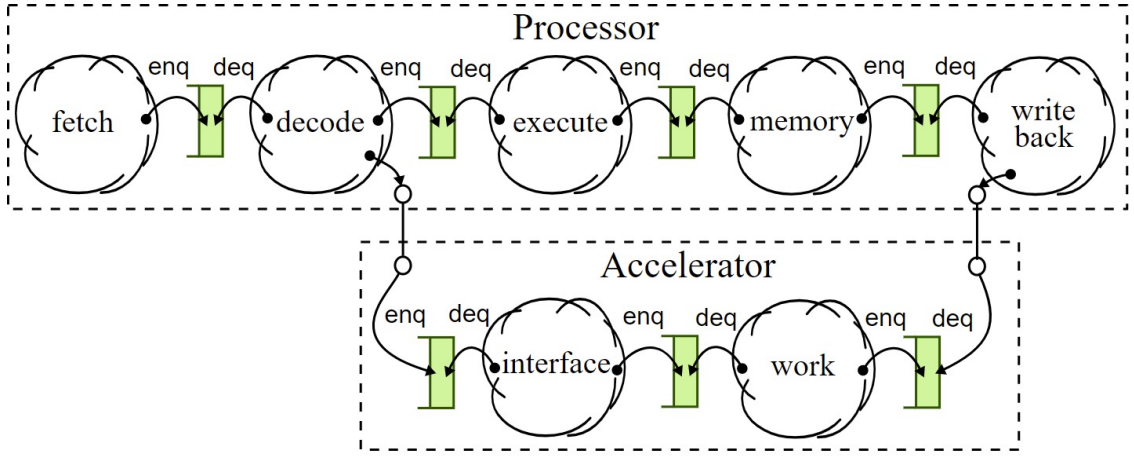
Composing Cycle-Level/RTL Models for Design Space Exploration



CL/RTL Composition:

- Use some CL models for faster overall simulation
- Gradually replacing CL models with RTL models

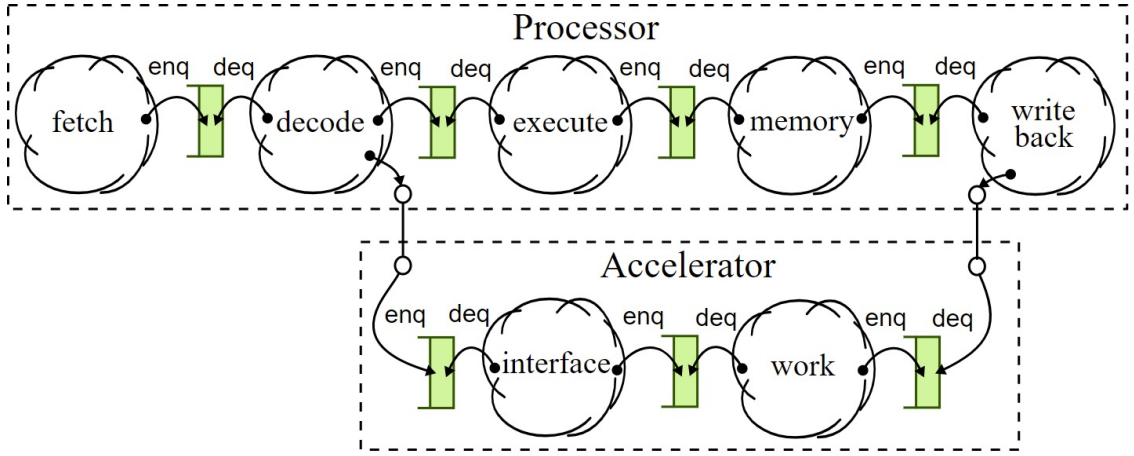
Challenge #1: Trade-off between model fidelity and scheduling modularity in cycle-level modeling



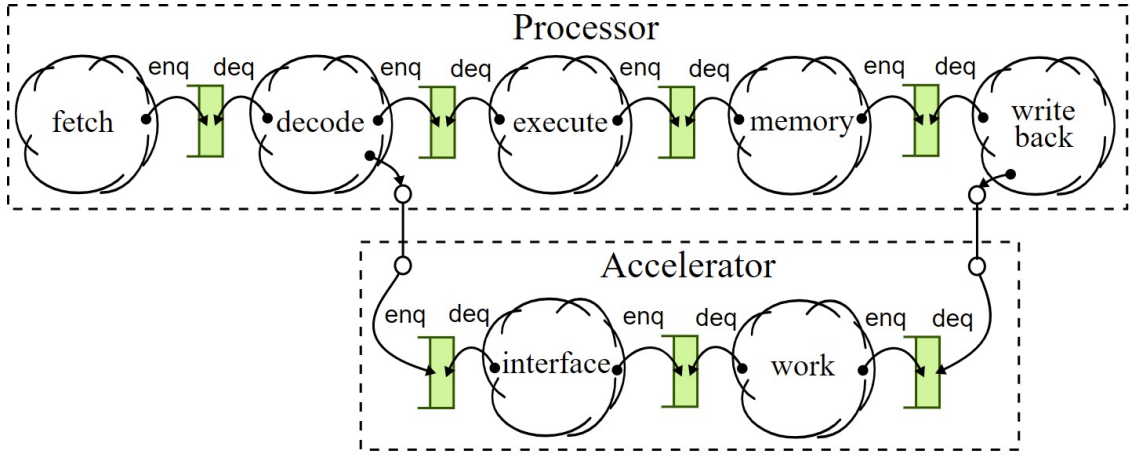
Challenge #1: Trade-off between model fidelity and scheduling modularity in cycle-level modeling

```
void Proc::tick()
{
  writeback();
  mem();
  execute();
  decode();
  fetch();
}

void Accel::tick()
{
  work();
  interface();
}
```



Challenge #1: Trade-off between model fidelity and scheduling modularity in cycle-level modeling



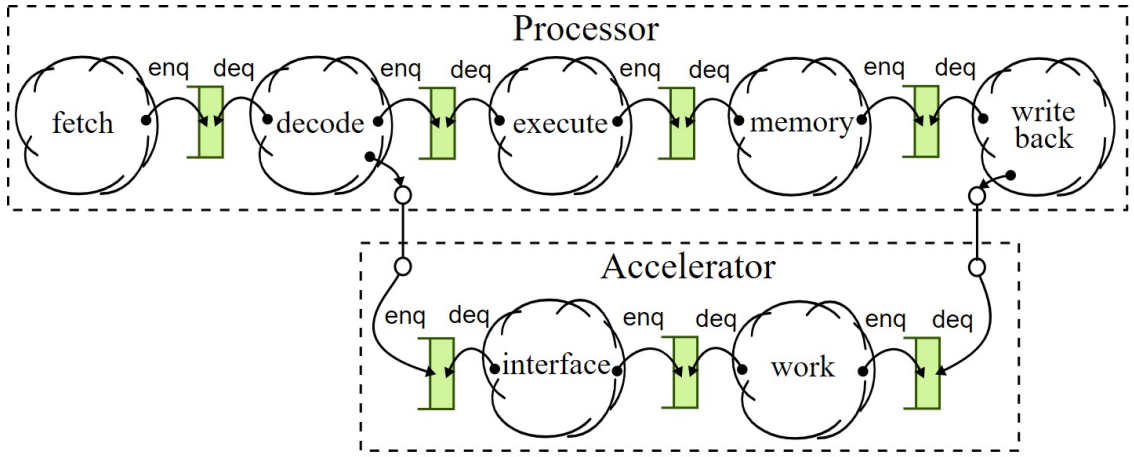

```
void Proc::tick()
{
  writeback();
  mem();
  execute();
  decode();
  fetch();
}
```

```
void Accel::tick()
{
  work();
  interface();
}
```

```
void Tile::tick()
{
  // modular
  accel.tick();
  proc.tick();
}
```

modular
but inaccurate

Challenge #1: Trade-off between model fidelity and scheduling modularity in cycle-level modeling




```
void Proc::tick()
{
  writeback();
  mem();
  execute();
  decode();
  fetch();
}
```

```
void Accel::tick()
{
  work();
  interface();
}
```

```
void Tile::tick()
{
  // modular
  accel.tick();
  proc.tick();
}
```

modular but inaccurate

```
void Tile::tick()
{
  // flattened
  proc.writeback();
  accel.work();
  proc.memory();
  accel.interface();
  proc.execute();
  proc.decode();
  proc.fetch();
}
```

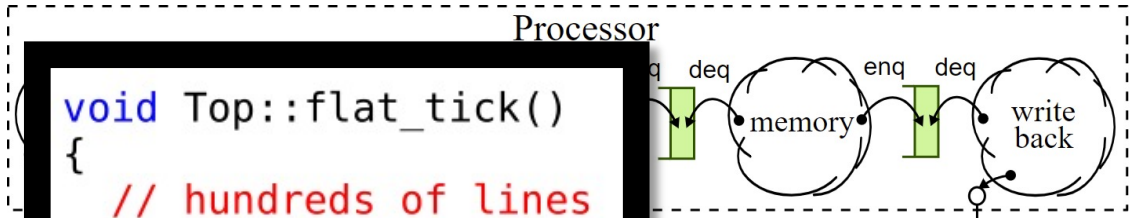
accurate but flat

Challenge #1: Trade-off between model fidelity and scheduling modularity in cycle-level modeling

```
void Proc::tick()
{
  writeback();
  mem();
  execute();
  decode();
  fetch();
}
```

```
void Accel::tick()
{
  work();
  interface();
}
```

```
void Top::flat_tick()
{
  // hundreds of lines
  mem.array.advance();
  mem.ctrl.work();
  ...
  tile[0].l2.access()
  ...
  tile[5].accel.work()
  tile[5].proc.memory()
  ...
  tile[7].proc.decode()
  ...
  tile[1].proc.fetch()
  ...
}
```




```
void Til
{
  // mod
  accel.
  proc.t
}
```

```
 Tile::tick()

 flattened
oc.writeback();
cel.work();
oc.memory();
cel.interface();
oc.execute();
oc.decode();
oc.fetch();
```

mod
but ina

accurate but flat

Challenge #2: Seamless General-Purpose CL/RTL Composition Methodologies

- No seamless CL/RTL compositions

Challenge #2: Seamless General-Purpose CL/RTL Composition Methodologies

- No seamless CL/RTL compositions
- PyMTL: manually CL ordering mixed with event-driven RTL

```
@s.tick_cl
def block():
  # TODO: we might want to see if this ticking order makes sense

  if s.l0_enabled:
    s.icache_mem_req_adapter.xtick()
    s.icache_mem_resp_adapter.xtick()

  if s.dmem_funnels is not None:
    for req_adapter, resp_adapter in zip( s.dcache_mem_req_adapters,
                                          s.dcache_mem_resp_adapters ):

      req_adapter.xtick()
      resp_adapter.xtick()

  if s.tmu:
    s.xcelreq_adapter.xtick()
    s.xcelresp_adapter.xtick()
    s.tmu.xtick()
```

Challenge #2: Seamless General-Purpose CL/RTL Composition Methodologies

- No seamless CL/RTL compositions
- PyMTL: manually CL ordering mixed with event-driven RTL
- SystemC: RTL/CL communication need to go through a clock edge

```
@s.tick_cl
def block():
  # TODO: we might want to see if this ticking order makes sense

  if s.l0_enabled:
    s.icache_mem_req_adapter.xtick()
    s.icache_mem_resp_adapter.xtick()

  if s.dmem_funnels is not None:
    for req_adapter, resp_adapter in zip( s.dcache_mem_req_adapters,
                                          s.dcache_mem_resp_adapters ):

      req_adapter.xtick()
      resp_adapter.xtick()

  if s.tmu:
    s.xcelreq_adapter.xtick()
    s.xcelresp_adapter.xtick()
    s.tmu.xtick()
```

Challenge #2: Seamless General-Purpose CL/RTL Composition Methodologies

- No seamless CL/RTL compositions
- PyMTL: manually CL ordering mixed with event-driven RTL
- SystemC: RTL/CL communication need to go through a clock edge
- … other ad-hoc approaches

```
@s.tick_cl
def block():
  # TODO: we might want to see if this ticking order makes sense

  if s.l0_enabled:
    s.icache_mem_req_adapter.xtick()
    s.icache_mem_resp_adapter.xtick()

  if s.dmem_funnels is not None:
    for req_adapter, resp_adapter in zip( s.dcache_mem_req_adapters,
                                          s.dcache_mem_resp_adapters ):

      req_adapter.xtick()
      resp_adapter.xtick()

  if s.tmu:
    s.xcelreq_adapter.xtick()
    s.xcelresp_adapter.xtick()
    s.tmu.xtick()
```

Challenge #2: Seamless General-Purpose CL/RTL Composition Methodologies

- No seamless CL/RTL compositions
- PyMTL: manually CL ordering mixed with event-driven RTL
- SystemC: RTL/CL communication need to go through a clock edge
- … other ad-hoc approaches

```
@s.tick_cl
def block():
  # TODO: we might want to see if this ticking order makes sense

  if s.l0_enabled:
    s.icache_mem_req_adapter.xtick()
    s.icache_mem_resp_adapter.xtick()

  if s.dmem_funnels is not None:
    for req_adapter, resp_adapter in zip( s.dcache_mem_req_adapters,
                                          s.dcache_mem_resp_adapters ):

      req_adapter.xtick()
      resp_adapter.xtick()

  if s.tmu:
    s.xcelreq_adapter.xtick()
    s.xcelresp_adapter.xtick()
    s.tmu.xtick()
```

Unified Modular Ordering Constraints (UMOC)

Unified abstraction for signal-based RTL modeling and method-based CL modeling

Unified Modular Ordering Constraints (UMOC)

Unified abstraction for signal-based RTL modeling and method-based CL modeling

$$\left.\begin{array}{r} x \text{ is a combinational wire} \\ A \text{ writes signal } x \\ B \text{ reads signal } x \end{array}\right\} \Longrightarrow \begin{array}{c} A \text{ precedes } B \\ (A < B) \end{array}$$

Unified Modular Ordering Constraints (UMOC)

Unified abstraction for signal-based RTL modeling and method-based CL modeling

$$\left.\begin{array}{r} x \text{ is a combinational wire} \\ A \text{ writes signal } x \\ B \text{ reads signal } x \end{array}\right\} \Longrightarrow \begin{array}{c} A \text{ precedes } B \\ (A < B) \end{array}$$

```
subcomponent subcomponent_instance_name (
  .clk      ( clk_sub   ), // input
  .rst_n    ( rst_n     ), // input
  .data_rx  ( data_rx_1 ), // input  [9:0]
  .data_tx  ( data_tx   )  // output [9:0]
);
```

Unified Modular Ordering Constraints (UMOC)

Unified abstraction for signal-based RTL modeling and method-based CL modeling

$$\left.\begin{array}{l} x \text{ is a combinational wire} \\ A \text{ writes signal } x \\ B \text{ reads signal } x \end{array}\right\} \Longrightarrow \begin{array}{l} A \text{ precedes } B \\ (A < B) \end{array}$$

```
subcomponent subcomponent_instance_name (
  .clk      ( clk_sub   ), // input
  .rst_n    ( rst_n     ), // input
  .data_rx  ( data_rx_1 ), // input  [9:0]
  .data_tx  ( data_tx   )  // output [9:0]
);
```

```
void Proc::tick()
{
  writeback();
  mem();
  execute();
  decode();
  fetch();
}
```

```
void Proc::decode() {
  auto i = FD_q.dequeue();
  ...
  if (i.is_accel_inst)
    Accel_q.enqueue(...);
  ...
  DX_q.enqueue(...);
}

void Proc::execute() {
  auto i = DX_q.dequeue();
  switch (i.type) {
    ...
  }
  ...
  XM_q.enqueue(...);
}
```

Unified Modular Ordering Constraints (UMOC)

Unified abstraction for signal-based RTL modeling and method-based CL modeling

$$\left.\begin{array}{l} x \text{ is a combinational wire} \\ A \text{ writes signal } x \\ B \text{ reads signal } x \end{array}\right\} \Longrightarrow \begin{array}{l} A \text{ precedes } B \\ (A < B) \end{array}$$

```
subcomponent subcomponent_instance_name (
  .clk      ( clk_sub   ), // input
  .rst_n    ( rst_n     ), // input
  .data_rx  ( data_rx_1 ), // input  [9:0]
  .data_tx  ( data_tx   )  // output [9:0]
);
```

```
void Proc::tick()
{
  writeback();
  mem();
  execute();
  decode();
  fetch();
}
```

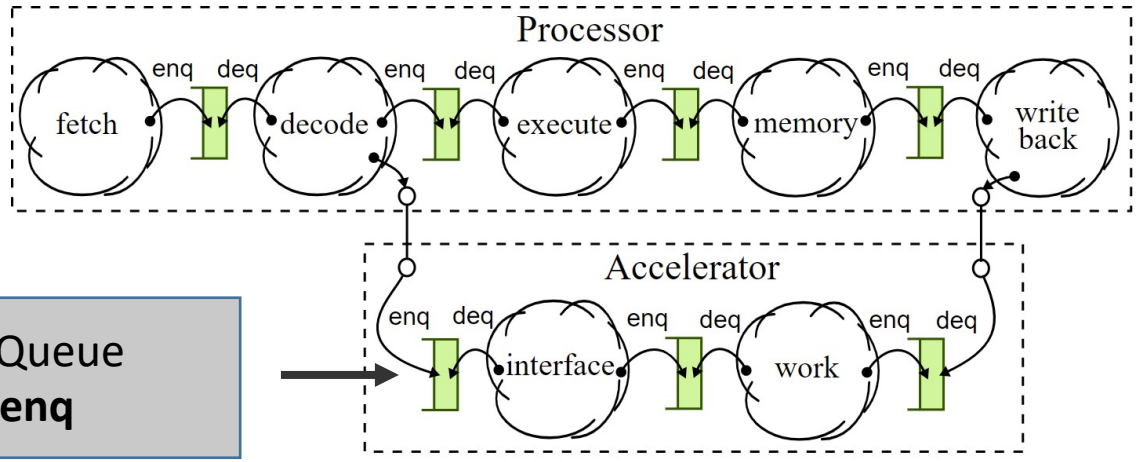
```
void Proc::decode() {
  auto i = FD_q.dequeue();
  ...
  if (i.is_accel_inst)
    Accel_q.enqueue(...);
  ...
  DX_q.enqueue(...);
}

void Proc::execute() {
  auto i = DX_q.dequeue();
  switch (i.type) {
    ...
  }
  ...
  XM_q.enqueue(...);
}
```

$$\left.\begin{array}{l} \texttt{q.dequeue} \textbf{ precedes } \texttt{q.enqueue} \\ A \text{ calls } \texttt{q.dequeue} \\ B \text{ calls } \texttt{q.enqueue} \end{array}\right\} \Longrightarrow \begin{array}{l} A \text{ precedes } B \\ (A < B) \end{array}$$

CL Model Fidelity & Schedule Modularity

- We insert a queue with deq<enq to accelerator

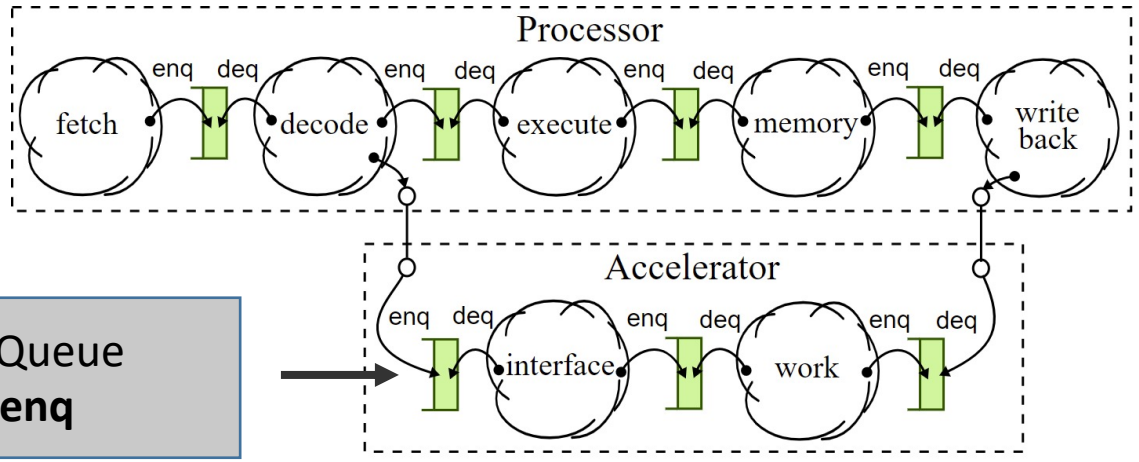


Pipeline Queue
deq < enq

$$\left.\begin{array}{l} \texttt{q.dequeue} \ \textbf{precedes} \ \texttt{q.enqueue} \\ A \text{ calls } \texttt{q.dequeue} \\ B \text{ calls } \texttt{q.enqueue} \end{array}\right\} \Longrightarrow \begin{array}{l} A \text{ precedes } B \\ (A < B) \end{array}$$

CL Model Fidelity & Schedule Modularity

- We insert a queue with deq<enq to accelerator
  - The interface process invokes deq



Pipeline Queue
deq < enq

`q.dequeue` **precedes** `q.enqueue`

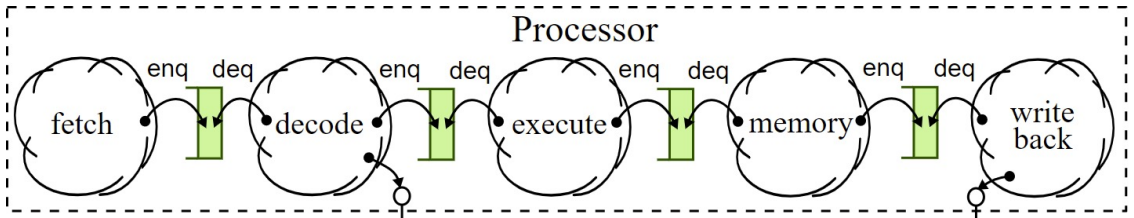
A calls `q.dequeue`

B calls `q.enqueue`

$\Longrightarrow$ *A* precedes *B* $(A < B)$

CL Model Fidelity & Schedule Modularity

- We insert a queue with deq<enq to accelerator
  - The interface process invokes deq
  - Expose its enq method to the parent tile

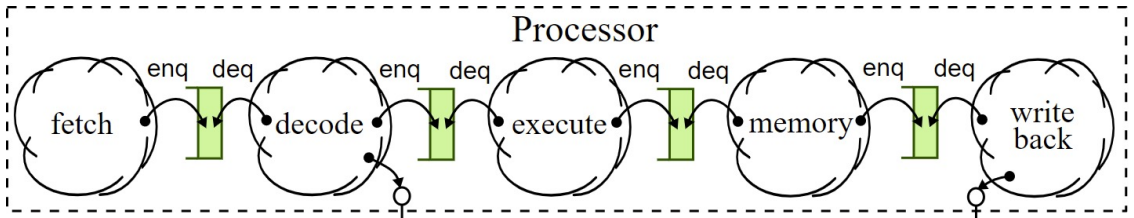


Pipeline Queue
deq < enq

$$\left.\begin{array}{l} \texttt{q.dequeue}\ \textbf{precedes}\ \texttt{q.enqueue} \\ A\ \text{calls}\ \texttt{q.dequeue} \\ B\ \text{calls}\ \texttt{q.enqueue} \end{array}\right\} \Longrightarrow \begin{array}{l} A\ \text{precedes}\ B \\ (A < B) \end{array}$$

CL Model Fidelity & Schedule Modularity

- We insert a queue with deq<enq to accelerator
  - The interface process invokes deq
  - Expose its enq method to the parent tile
  - Pass the enq method to processor

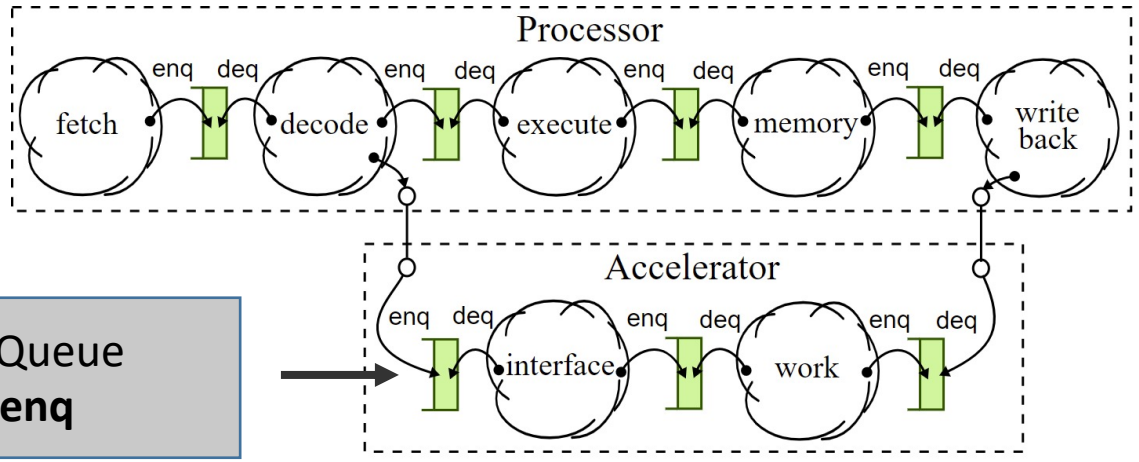


Pipeline Queue
deq < enq

$$\left.\begin{array}{l} \texttt{q.dequeue} \ \textbf{precedes} \ \texttt{q.enqueue} \\ A \text{ calls } \texttt{q.dequeue} \\ B \text{ calls } \texttt{q.enqueue} \end{array}\right\} \Longrightarrow \begin{array}{l} A \text{ precedes } B \\ (A < B) \end{array}$$

CL Model Fidelity & Schedule Modularity

- We insert a queue with deq<enq to accelerator
  - The interface process invokes deq
  - Expose its enq method to the parent tile
  - Pass the enq method to processor
  - The decode process invokes enq

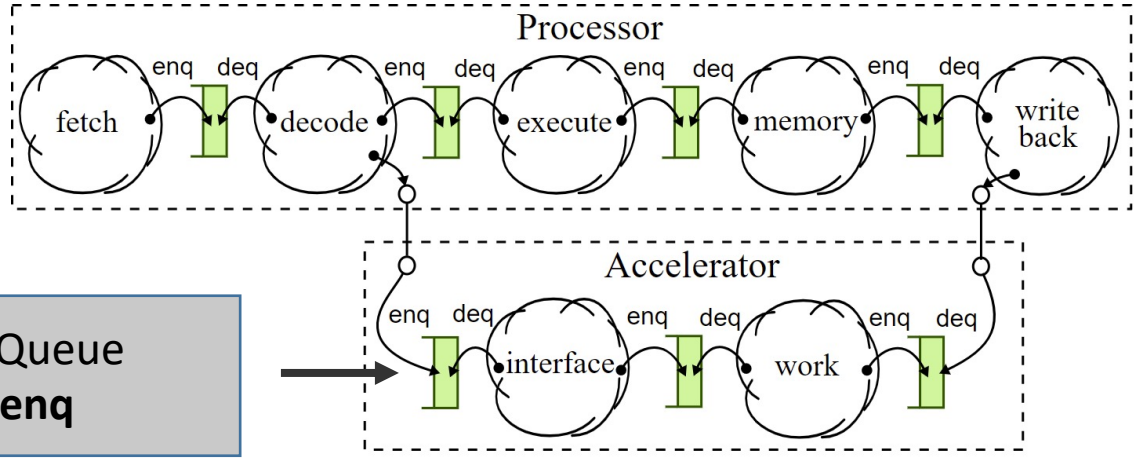


Pipeline Queue
deq < enq

$$\left.\begin{array}{l} \texttt{q.dequeue} \textbf{ precedes } \texttt{q.enqueue} \\ A \text{ calls } \texttt{q.dequeue} \\ B \text{ calls } \texttt{q.enqueue} \end{array}\right\} \Longrightarrow \begin{array}{l} A \text{ precedes } B \\ (A < B) \end{array}$$

CL Model Fidelity & Schedule Modularity

- We insert a queue with deq<enq to accelerator
  - The interface process invokes deq
  - Expose its enq method to the parent tile
  - Pass the enq method to processor
  - The decode process invokes enq
- Global scheduler: interface before decode



Pipeline Queue
deq < enq

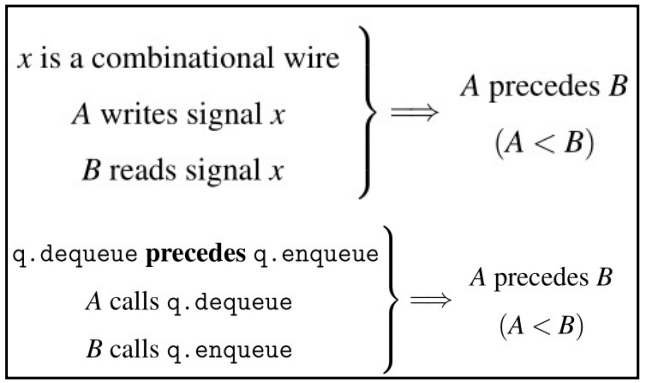
q.dequeue **precedes** q.enqueue

A calls q.dequeue

B calls q.enqueue

$\Longrightarrow$ *A* precedes *B* $(A < B)$

Seamless CL/RTL Composition



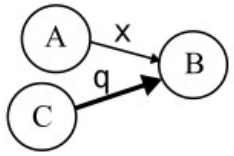

```
x is signal
q.dequeue < q.enqueue
```

```
A:  x = y + 1
```

```
B:  q.enqueue(x * 2)
```

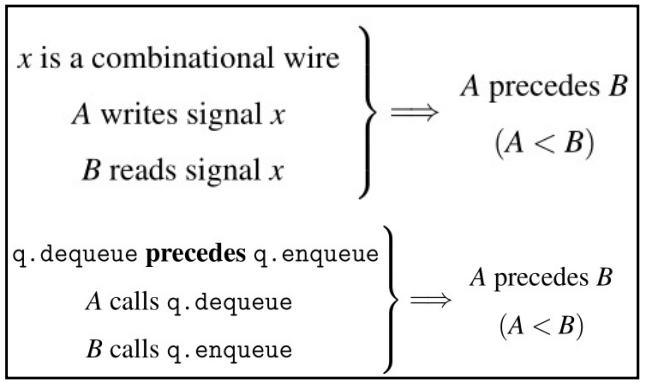
```
C:  z = q.dequeue()
```

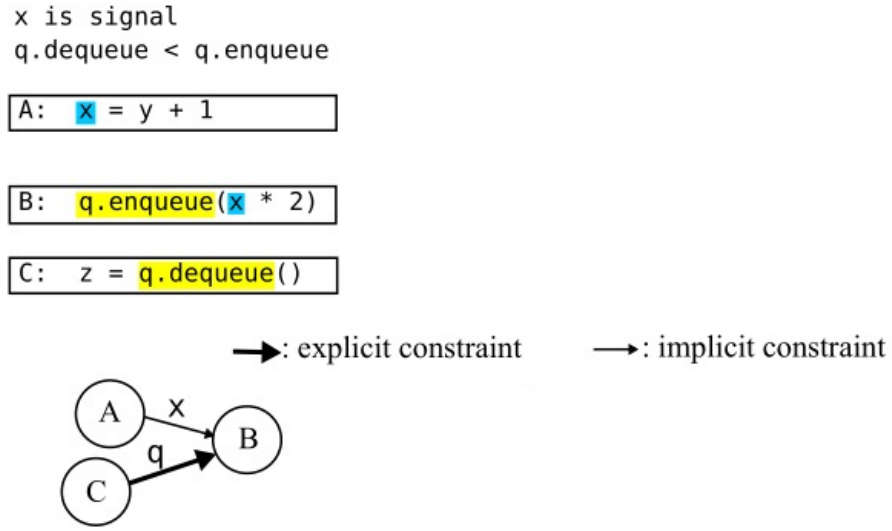
→: explicit constraint ⟶: implicit constraint



Seamless CL/RTL Composition

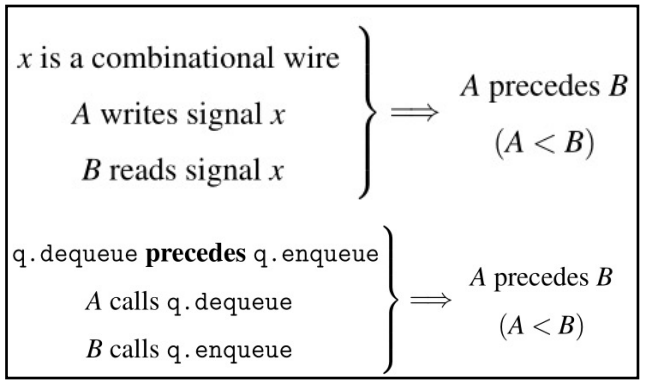
- Creating the Unified Directed Graph (UDG)
  - Edges include implicit and explicit ordering constraints

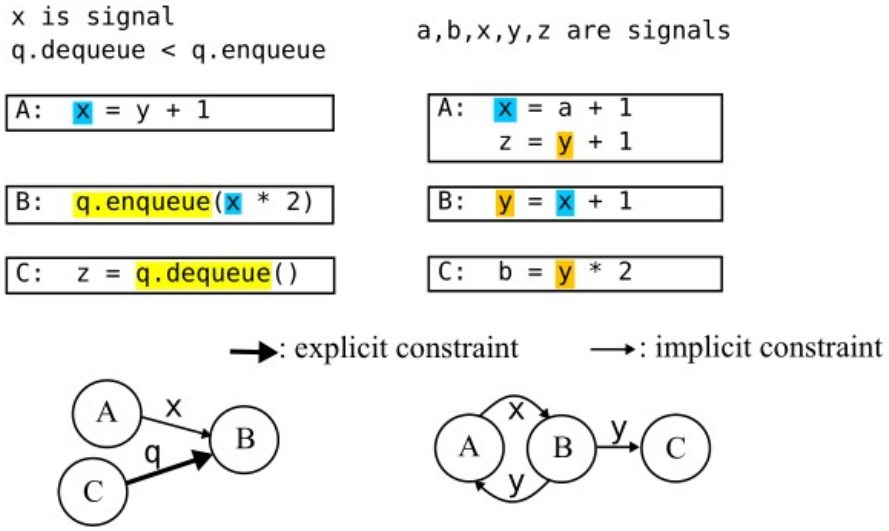




Seamless CL/RTL Composition

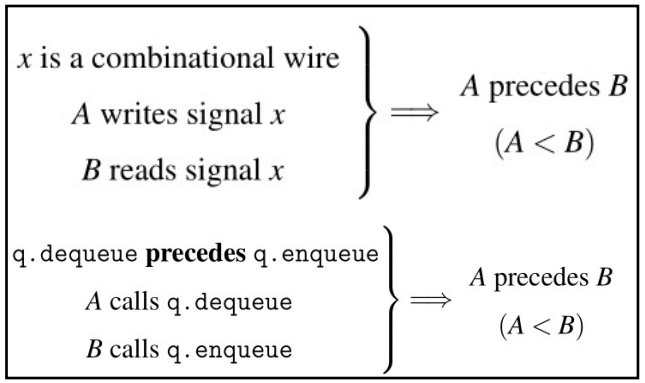
- Creating the Unified Directed Graph (UDG)
  - Edges include implicit and explicit ordering constraints
  - Loops between RTL processes are allowed

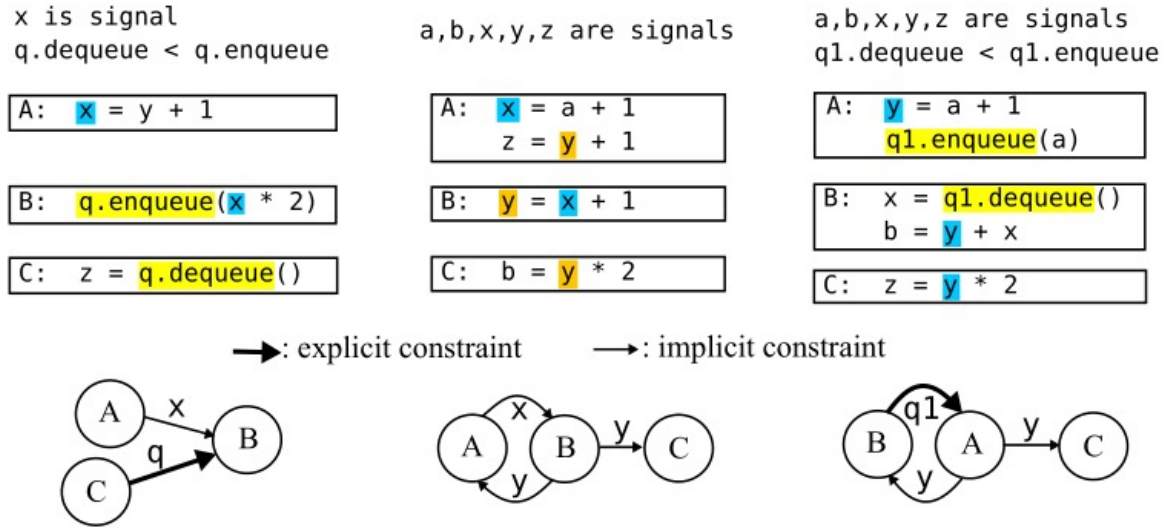




Seamless CL/RTL Composition

- Creating the Unified Directed Graph (UDG)
  - Edges include implicit and explicit ordering constraints
  - Loops between RTL processes are allowed
  - CL processes are not allowed to appear in any loop





Scheduling the Unified Directed Graph

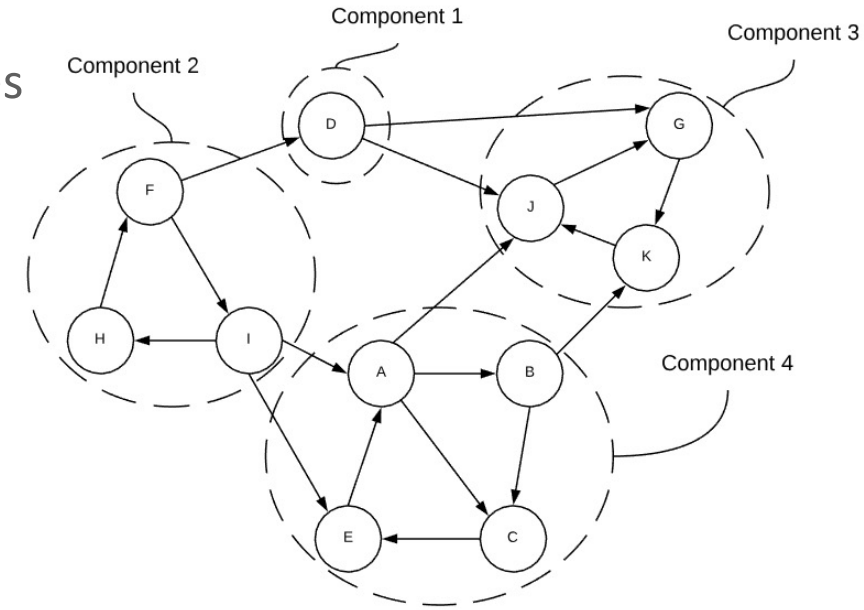
- Some properties of the UDG:
  - CL processes execute exactly once per cycle
  - RTL processes need to execute until value stabilize

Scheduling the Unified Directed Graph

- Some properties of the UDG:
  - CL processes execute exactly once per cycle
  - RTL processes need to execute until value stabilize
- If the UDG has no cycle
  - Topological sort that statically schedules all processes in the DAG

Scheduling the Unified Directed Graph

- Some properties of the UDG:
  - CL processes execute exactly once per cycle
  - RTL processes need to execute until value stabilize
- If the UDG has no cycle
  - Topological sort that statically schedules all processes in the DAG
- If the UDG has cycle
  - Strongly connected components (SCC) algorithm
    - » Shrink cycles into a single node
  - Execute the DAG of SCCs
    - » Topological sort of the DAG
    - » Iteratively Execute the SCC



UMOC Implemented in PyMTL3

- PyMTL3 is a state-of-the-art Python-based hardware generation and simulation framework
- PyMTL3 is very extensible thanks to modular framework architecture
  - Frontend: Embedded domain specific language (EDSL) modeling primitives
  - IR: Native in-memory intermediate representation (NIMIR)
  - Backend: Passes that systematically manipulate NIMIR
- UMOC implemented in PyMTL3:
  - EDSL modeling primitives
  - NIMIR data structures
  - Graph generation and scheduling passes

UMOC Implementation of PyMTL3 EDSL Primitives

- UMOC PyMTL3 EDSL primitives:
  - Inherit from Component
  - InPort, OutPort, Wire, CalleePort, CallerPort
  - @update_ff, @update, @update_once, @method_port
  - add_constraints

```
class RegIncrRTL( Component ):
  def construct( s ):
    s.in_ = InPort (32)
    s.out = OutPort(32)

    s.reg = Wire(32)

    @update_ff
    def seq_reg():
      s.reg <<= s.in_

    @update
    def comb_out():
      s.out @= s.reg + 1
```

```
class RegIncrCL( Component ):
  def construct( s ):
    # Model sequential behavior!
    s.add_constraints(
      M(s.read) < M(s.write),
    )

  @method_port
  def read( s ):
    return s.v + 1

  @method_port
  def write( s, v ):
    s.v = v
```

```
class RegIncrCLRTL( Component ):
  def construct( s ):
    s.write = CalleePort()
    s.out   = OutPort(32)

    s.r1 = RegIncrCL()
    s.r2 = RegIncrRTL()

    connect( s.write, s.r1.write )
    connect( s.out,   s.r2.out   )

    @update_once
    def send_to_r2():
      s.r2.in_ @= s.r1.read()
```

UMOC Implementation of PyMTL3 NIMIR & Passes

UMOC Implementation of PyMTL3 NIMIR & Passes

- Supporting UMOC in PyMTL3 NIMIR elaboration

UMOC Implementation of PyMTL3 NIMIR & Passes

- Supporting UMOC in PyMTL3 NIMIR elaboration
  - Collecting all the update blocks and ordering constraints

UMOC Implementation of PyMTL3 NIMIR & Passes

- Supporting UMOC in PyMTL3 NIMIR elaboration
  - Collecting all the update blocks and ordering constraints
  - Exposing all metadatas with APIs

UMOC Implementation of PyMTL3 NIMIR & Passes

- Supporting UMOC in PyMTL3 NIMIR elaboration
  - Collecting all the update blocks and ordering constraints
  - Exposing all metadatas with APIs
- UMOC passes

UMOC Implementation of PyMTL3 NIMIR & Passes

- Supporting UMOC in PyMTL3 NIMIR elaboration
  - Collecting all the update blocks and ordering constraints
  - Exposing all metadatas with APIs
- UMOC passes
  - GenUDGPass to generate the unified directed graph
    - Update blocks as vertices, explicit/implicit constraints as edges
  - UMOCSchedulingPass to schedule the UDG

UMOC Implementation of PyMTL3 NIMIR & Passes

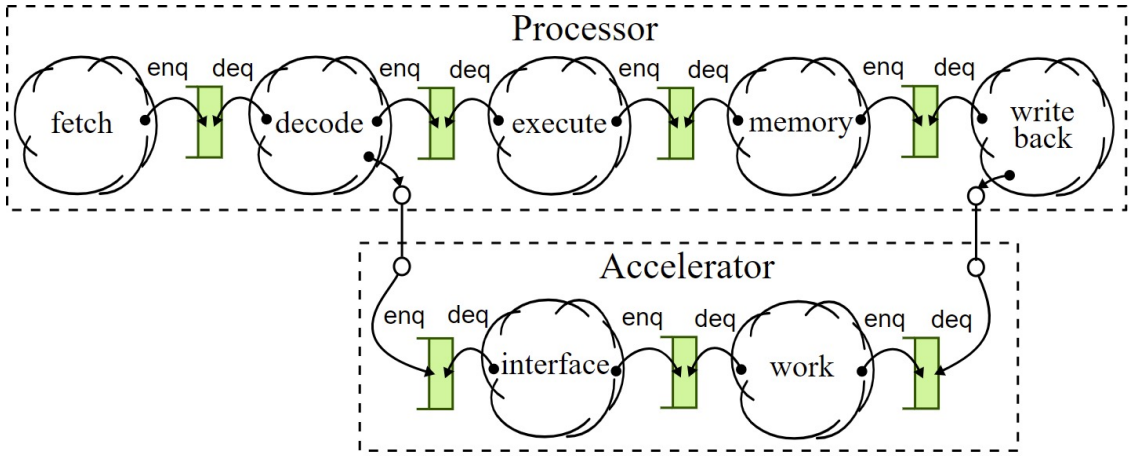
- Supporting UMOC in PyMTL3 NIMIR elaboration
  - Collecting all the update blocks and ordering constraints
  - Exposing all metadatas with APIs
- UMOC passes
  - GenUDGPass to generate the unified directed graph
    - Update blocks as vertices, explicit/implicit constraints as edges
  - UMOCSchedulingPass to schedule the UDG

- The user only need to set local explicit ordering constraints. No global scheduling required.

UMOC Case Study in PyMTL3

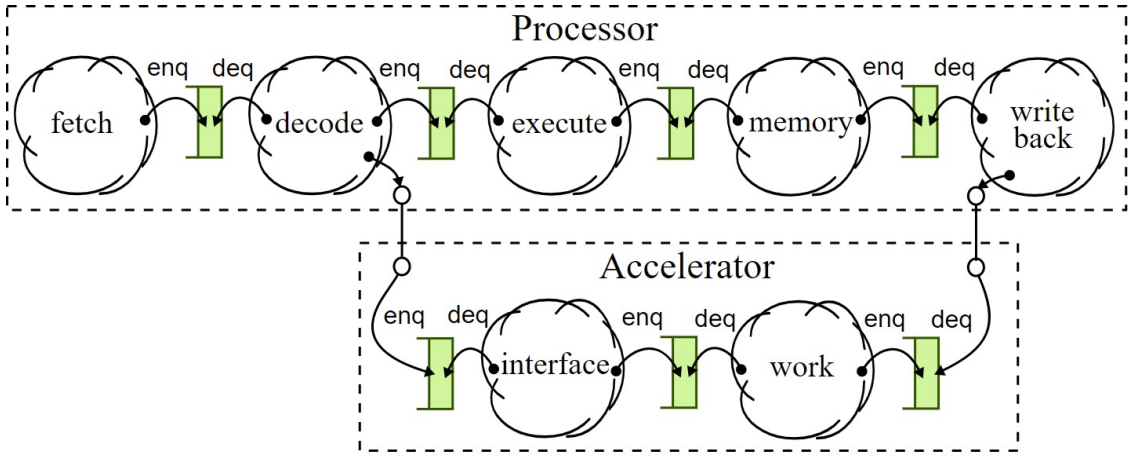
UMOC Case Study in PyMTL3

- 5-stage RTL Processor, 3-stage CL processor



UMOC Case Study in PyMTL3

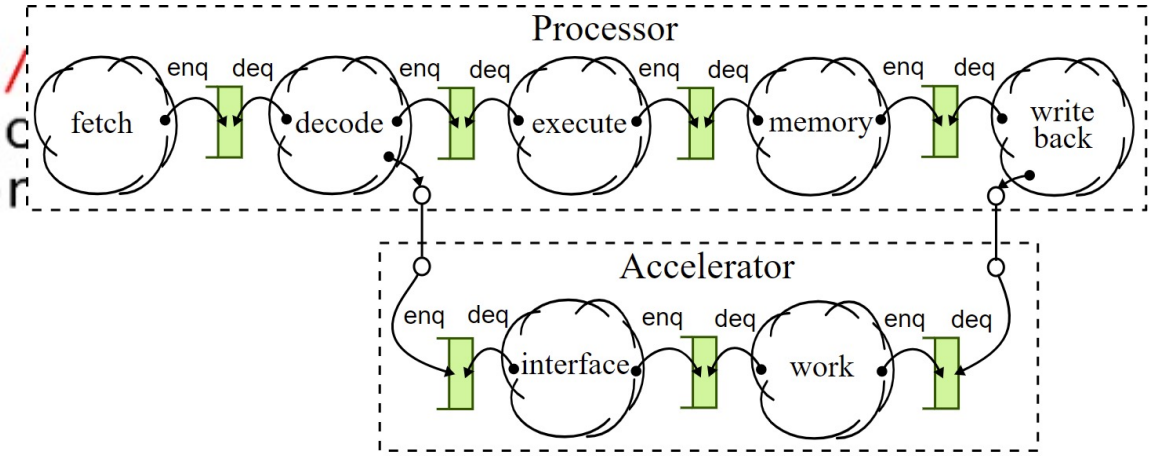
- 5-stage RTL Processor, 3-stage CL processor
- RTL/CL checksum accelerators



UMOC Case Study in PyMTL3

- 5-stage RTL Processor, 3-stage CL processor
- RTL/CL checksum accelerators
- Manual == Execute everything in accelerator before processor (or vice versa)

```
void Tile::tick()
{
  //
  ac
  pr
}
```



UMOC Case Study in PyMTL3

- 5-stage RTL Processor, 3-stage CL processor
- RTL/CL checksum accelerators
- Manual == Execute everything in accelerator before processor (or vice versa)

```
void Tile::tick()
{
  // modular
  accel.tick();
  proc.tick();
}
```

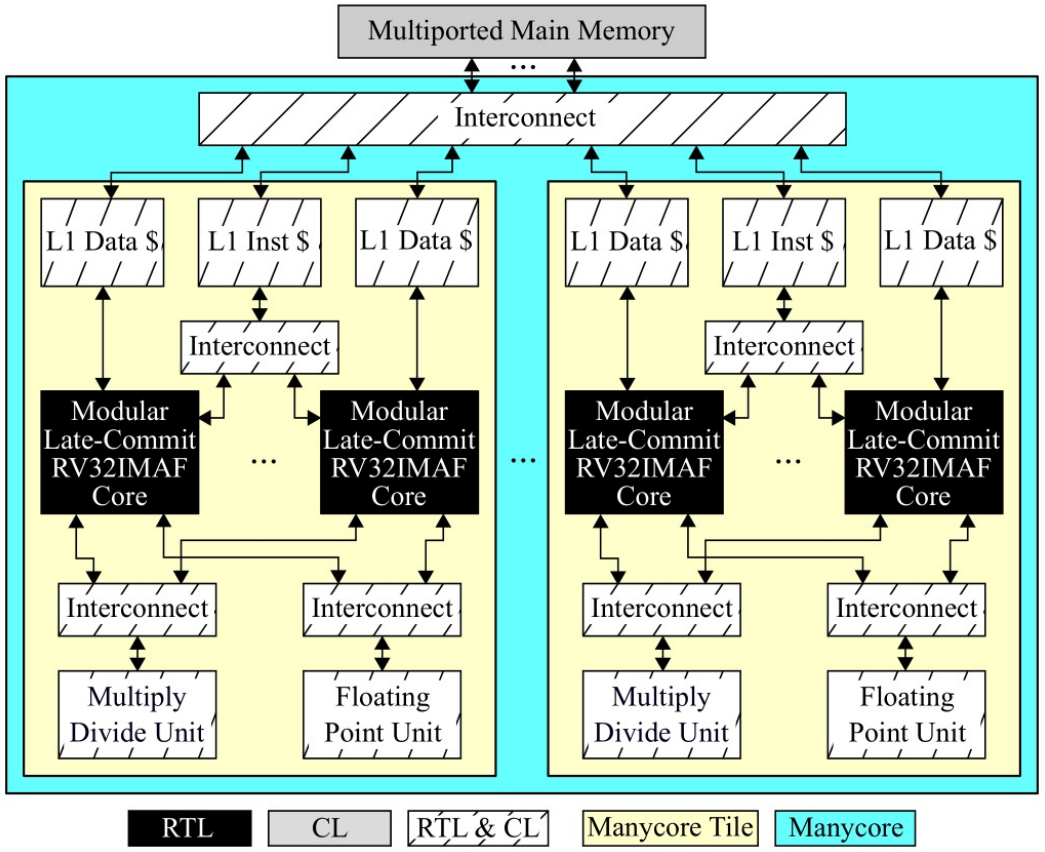
UMOC Case Study in PyMTL3

- 5-stage RTL Processor, 3-stage CL processor
- RTL/CL checksum accelerators
- Manual == Execute everything in accelerator before processor (or vice versa)

Mechanism	Composition	#Cycles	Deviation	Remarks
Event-driven	RTL Proc + RTL Accel	565	-	baseline
UMOC	RTL Proc + RTL Accel	565	0%	same as baseline
UMOC	CL Proc + CL Accel	541	4%	due to 3-stage
Manual **Proc<Accel**	CL Proc + CL Accel	416	26%	modular sub-tick
Manual **Accel<Proc**	CL Proc + CL Accel	416	26%	modular sub-tick
UMOC	CL Proc + RTL Accel	541	4%	same as CL+CL
UMOC	RTL Proc + CL Accel	565	0%	same as RTL+RTL

CL/RTL Compositions Helps Chip Tape-outs

- Main-memory only needs CL
- CL shared MDU/FPU for DSE
- CL cache for DSE
- CL on-chip networks for DSE
- Processor IP already developed



Takeaways & Conclusion

- UMOC's explicit ordering constraints achieves model fidelity and scheduling modularity at once.
- UMOC's implicit & explicit constraints achieves seamless CL/RTL composition.
- UMOC has been implemented in PyMTL3. Many IPs have been built using UMOC scheme.