Patterns Review
The shift register is an abstraction of any synchronous pipelined block of logic that accepts input data and produces output data, where input and output might not be ready every clock cycle.

How to manage growth in control logic complexity?
Solution: Decouple Units with FIFOs

- Pipeline only cares whether space in FIFO, not about whether consumer can take next value
- Breaks combinational path between pipeline control logic and consumer control logic
- For full throughput with decoupling, need at least two elements in FIFO
  - With only one element, have to ping-pong between pipeline enqueue and consumer dequeue
  - Allowing both enqueue and dequeue in same cycle to single-element FIFO destroys decoupling (back to a synchronous connection)
Many large digital designs are divided into local synchronous pipelines, or units, connected via decoupling FIFOs.

- Approx. 10K-100K gates per unit.

Decoupled units may have different clocks.
- In which case, need asynchronous FIFOs.
Hardware Design Patterns

- Decoupled units are an example of a design pattern
- Pattern: Solution to a commonly recurring design problem
- Idea of patterns and a “pattern language” first proposed for building architecture (Christopher Alexander)
  - “Pattern language” is an interlocking set of design patterns
    - Probably better named a “pattern hierarchy”
  - Alexander proposed single pattern language covering architecture from design of cities to design of roof caps
- Patterns popular in software engineering (“Gang of Four”) and now being used in Par Lab (“Our Pattern Language (OPL)”) to architect parallel software
- This semester continues an experiment to see if we can teach hardware design using patterns
Digital Design Through Patterns

Application(s)

Berkeley Hardware Pattern Language (BHPL)

MP3 bit string

Hardware (RTL)

Audio
BHPL Goals

- BHPL captures problem-solution pairs for creating hardware designs (machines) to execute applications

BHPL Non-Goals

- Doesn’t describe applications themselves, only machines that execute applications and strategies for mapping applications onto machines
BHPL describes Machines not Applications

### Applications (including OPL patterns)

#### Structural Patterns
- Pipelines
- Agent&Repository
- Model-View-Controller
- Event Based
- Process Control
- Iteration
- Map-Reduce
- Layered Systems
- Task Graphs

#### Computational Patterns
- Circuits
- Dense Linear Algebra
- N-Body Methods
- Sparse Linear Algebra
- Spectral Methods
- Unstructured Grids
- Graph Traversal
- Structured Grids
- Graph Algorithms
- FSMs

### Mapping Patterns

#### Machines

- [Diagram showing mapping patterns]

---

BHPL, 2013
BHPL 0.5 Overview

Applications (including OPL patterns)

BHPL

App-to-UTL
Mappings Layer

FFT to SIMD array

Problem: Application Computation
Solution: UTL Machine

UTL-to-UTL
Transformation Layer

Time-Multiplexing
Unrolling

Problem: UTL violates constraint
(too big, too slow)
Solution: Transformed UTL

UTL-to-RTL
Transformation Layer

Microcoded Engine
In-Order Pipeline Engine

Problem: UTL design
Solution: RTL behavior

RTL-to-Technology

Interleaved Memory
FIFO
CAM

Problem: RTL behavior
Solution: Structural design

Lecture 4, Patterns-I

CS294-88, UC Berkeley, Spring 2013
Pattern Template (Name here)

Problem: Describe the particular problem the pattern is meant to solve. Should include some context (small, high throughput), and also the layer of the pattern hierarchy where it fits.

Solution: Describe the solution, which should be some hardware structure with a figure. Solution is usually the pattern name. Should not provide a family of widely varying solutions - these should be separate patterns, possibly grouped under a single more abstract parent pattern.

Applicability: Longer discussion of where this particular solution would normally be used, or where it would not be used.

Consequences: Issues that arise when using this pattern, but only for cases where it is appropriate to use (use Applicability to delineate cases where it is not appropriate to use). These might point at sub-problems for which there are sub-patterns. There might also be limitations on resulting functionality, or implications in design complexity, or CAD tool use etc.
Decoupled Units

**Problem:** Difficult to design a large unit with a single controller, especially when components have variable processing rates. Large controllers have long combinational paths.

**Solution:** Break large unit into smaller sub-units where each sub-unit has a separate controller and all channels between sub-units have some form of decoupling (i.e., no assumption about timing of sub-units).

**Applicability:** Large unit where area and performance overhead of decoupling is small compared to benefits of simpler design and shorter controller critical paths.

**Consequences:** Decoupled channels generally have greater communication latency and area/power cost. Sub-unit controllers must cope with unknown arrival time of inputs and unknown time of availability of space on outputs. Sub-units must be synchronized explicitly.
Decoupled Units

Unless shared memory is truly multiported, channels to memory must be decoupled.

Channels to network are usually decoupled in any case.
Related Patterns

- Often multiple solutions to same problem, but which to pick depends on situation

- **Problem** statement and **Applicability** text should help select the correct pattern to use

- Example: Pipelined versus multi-cycle operators when delay through operator exceeds desired cycle time
Pipelined Operator

**Problem:** Combinational function of operator has long critical path that would reduce system clock frequency. High throughput of this function is required.

**Solution:** Divide combinational function using pipeline registers such that logic in each stage has critical path below desired cycle time. Improve throughput by initiating new operation every clock cycle overlapped with propagation of earlier operations down pipeline.

**Applicability:** Operators that require high throughput but where latency is not critical.

**Consequences:** Latency of function increases due to propagation through pipeline registers, adds energy/op. Any associated controller might have to track execution of operation across multiple cycles.
Pipelined Operator

\[ f(g(in)) \]

Clock → \[ f(g(in)) \] → Clock

\[ g(in) \rightarrow f(in) \rightarrow \]

Clock → \[ g(in) \] → Clock → Clock → Clock
Multicycle Operator

**Problem:** Combinational function of operator has long critical path that would reduce system clock frequency. High throughput of this function is not required.

**Solution:** Hold input registers stable for multiple clock cycles of main system, and capture output after combinational function has settled.

**Applicability:** Operators where high throughput is not required, or if latency is critical (in which case, replicate to increase throughput).

**Consequences:** Associated controller has to track execution of operation across multiple cycles. CAD tools might detect false critical path in block.
Multicycle Operator

\[
f(g(\text{in}))
\]

Clock

Block

Clock

Clock

Clock/2

Block

Clock/2
Some Families of Patterns
Multiport Memory Patterns

Provides multiple access ports to a common memory

- True Multiport Memory
- Banked Multiport Memory
  - Interleave lesser-ported banks to provide higher bandwidth
- Cached Multiport Memory
  - Use large single-port main memory, but add cache to service requests for each access port
Controller Patterns

Synchronous control of local datapath

- **State Machine Controller**
  - control lines generated by state machine

- **Microcoded Controller**
  - single-cycle datapath, control lines in ROM/RAM

- **In-Order Pipeline Controller**
  - pipelined control, dynamic interaction between stages

- **Out-of-Order Pipeline Controller**
  - operations within a control stream might be reordered internally

- **Threaded Pipeline Controller**
  - multiple control streams one execution pipeline
  - can be either in-order (PPU) or out-of-order
Network Patterns

Connects multiple units using shared resources

- **Bus**
  - Low-cost, ordered

- **Crossbar**
  - High-performance

- **Multi-stage network**
  - Trade cost/performance
Chiseling Patterns

- How will patterns be implemented in Chisel libraries?
- How will different levels of the pattern language compose
BHPL 0.5 Overview

Applications (including OPL patterns)

BHPL

App-to-UTL Mappings Layer

FFT to SIMD array

Problem: Application Computation
Solution: UTL Machine

UTL-to-UTL Transformation Layer

Time-Multiplexing
Unrolling

Problem: UTL violates constraint (too big, too slow)
Solution: Transformed UTL

UTL-to-RTL Transformation Layer

Microcoded Engine

Problem: UTL design
Solution: RTL behavior

In-Order Pipeline Engine

RTL-to-Technology

Interleaved Memory

Problem: RTL behavior
Solution: Structural design

FIFO 74 CAM

Lecture 4, Patterns-I

CS294-88, UC Berkeley, Spring 2013
App -> UTL in Chisel

- Translating an application computational pattern (e.g., FFT) into a UTL machine
  - For an individual pattern, we can provide a library or framework to convert instance of pattern into a UTL machine description
  - Some patterns require function as a parameter (e.g., structured grid)

- Composition of patterns into an application
  1. Provide an application framework, e.g., a speech recognizer framework, into which we plug constrained computations
  2. Provide a Chisel EDSL
Each transaction has a combinational guard function defined over local state and state of I/O indicating when it can fire
  - e.g., only fire when head of input queue present and of certain type
Transaction mutates local state and performs I/O when it fires
Scheduler is combinational function that picks next ready transaction to fire
Transactions in decreasing scheduler priority

- **Table_Write** (request on table access queue)
  - Writes a given 12-bit value to a given 12-bit address

- **Table_Read** (request on table access queue)
  - Reads a 12-bit value given a 12-bit address, puts response on table reply queue

- **Route** (request on packet input queue)
  - Looks up header in table and places routed packet on correct output queue

*This level of detail is all the information we really need to understand what the unit is supposed to do! Everything else is implementation.*
The reorder buffer, the trie lookup pipeline’s registers, and any control state are microarchitectural state that should not affect function as viewed from outside.

Implementation must ensure atomicity of UTL transactions:
- Reorder buffer ensures packets flow through unit in order
- Must also ensure table write doesn’t appear to happen in middle of packet lookup, e.g., wait for pipeline to drain before performing write
Architectural State

- The architectural state of a unit is that which is visible from outside the unit through I/O operations.
  - i.e., architectural state is part of the spec.
  - (this is the target for “black-box” testing)

- When a unit is refined into RTL, there will usually be additional *microarchitectural* state that is not visible from outside.
  - Intra-transaction sequencing logic
  - Pipeline registers
  - Internal caches and/or buffers
  - (this is the target for “white-box” testing)
UTL->UTL

- Transactional actor as common representation of a unit?
  - e.g., Actor hierarchy (SDF, Kahn, Transactional Actor)

- These are transformations within a single level of representation, analogous to compiler transformations
  - e.g., loop unrolling, time-multiplexing

- Authors provide transformation passes, constrained by types of units they can transform?
  - e.g., loop unrolling for stateless units only

- Design-space exploration tools manipulate the parameters exposed by these transformation passes?
UTL->RTL

- Analogous to back-end of software compiler
- Convert certain types of unit into pipelined RTL engines
Processing Unit Design Patterns
Control+Datapath

**Problem:** Arithmetic operations within transaction require large functional units and memories connected by wide buses. Sequencing of operations within transaction is complex.

**Solution:** Split processing unit into 1) *datapath*, which contains functional units, data memories, and their interconnect, and 2) *control*, which contains all sequencing logic.

**Applicability:** Where there is a clear divide between control logic and data processing logic, with relatively few signals crossing this divide, and mostly from control to datapath not vice versa.

**Consequences:** Most design errors are confined to the control portion. Same datapath design can perform many different transactions by changing control sequencing. Paths between control and datapath, particularly from datapath back to control, are often timing critical.
Controller Patterns

For synchronous control of local datapath

- **State Machine Controller**
  - control lines generated by state machine
- **Microcoded Controller**
  - single-cycle datapath, control lines in ROM/RAM
- **In-Order Pipeline Controller**
  - control pipelined datapath, dynamic interaction between stages
- **Out-of-Order Pipeline Controller**
  - operations within a control stream might be reordered internally
- **Threaded Pipeline Controller**
  - multiple control streams, one execution pipeline
Control Decomposition

Can divide control functions into three categories:

Transaction Scheduling

Transaction Sequencing

Pipeline Control

Pick next transaction to be executed

Sequence operations within transaction

Control execution of operations on pipelined datapath

To datapath

From datapath
State Machine Controller

**Problem:** Control for a simple unit that performs a single transaction at a time.

**Solution:** Construct state machine with a common initial state to select next transaction, and a separate path for each transaction to sequence operations for that transaction.

**Applicability:** Where datapath is not highly pipelined and where unit only executes one non-overlapping transaction at a time. Combinational control logic can expand dramatically as number of states increases, so limited to less pipelined and less concurrent units.

**Consequences:** State machine can be more compact and faster than a microcode controller for small state machines. Changes in unit functionality can cause large changes in size/speed of state machine.
State Machine Controller
Microcoded Controller

**Problem:** Control for a complex unit that performs a single transaction at a time.

**Solution:** Encode control lines in a ROM structure with a small state machine to sequence through locations in ROM. Microcode dispatch function selects next transaction to execute, and each transaction executed by sequence in microcode ROM. Can also use RAM structure to allow post-fabrication modifications to control.

**Applicability:** Where unit only executes one non-overlapping transaction at a time, but where control is complex. Particularly useful in technology where ROM bits are significantly cheaper than logic gates.

**Consequences:** Microcode easily modified to make changes in unit functionality. Unit cycle time can be limited by critical path from ROM readout to ROM address input (can use pipelined microcode engine to speed throughput inbetween control hazards).
Microcoded Controller
In-Order Pipeline Controller

**Problem:** Control for a complex pipelined unit that can overlap execution of multiple transactions, and multiple operations within one transaction.

**Solution:** Generate control signals for each stage of pipeline using control state pipelined along with data state. Use dynamic scoreboard (part of which may be the pipelined control state) to track operations in flight in pipeline. Next operation can only enter pipeline when scoreboard indicates this would not create a pipelining hazard (structural, data, or control).

**Applicability:** Where unit’s datapath is pipelined and either sequence of transactions or sequence of operations within a transaction is dynamically determined by input data. Where in-order processing is required, or sufficient for performance goals.

**Consequences:** The datapath design mandates the hazards generated by an executing operation, and can cause large growth in scoreboard complexity and reduction in performance unless hazards on common sequences are avoided.
In-Order Pipeline Controller

Select next xaction or next op in current xaction
Out-of-Order Processing Unit

- When in-order gives insufficient throughput, buffer operations and issue out-of-order with respect to hazards.
Threaded Processing Unit

- Multiplex multiple transaction streams onto single hardware unit.
- One specific implementation of time-multiplexing.
## Taxonomy of Control Strategies

Increasing levels of control complexity build on each other.

<table>
<thead>
<tr>
<th>SM</th>
<th>(\mu)Code</th>
<th>In-Order</th>
<th>OoO</th>
<th>Threaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next state after idle state</td>
<td>Dispatch on transaction state to (\mu)code address</td>
<td>Initialize current transaction state</td>
<td>Initialize current transaction state</td>
<td>Interleave transactions from multiple units</td>
</tr>
<tr>
<td>State transitions</td>
<td>Step through (\mu)code</td>
<td>Sequence through transaction states (either FSM or (\mu)Code)</td>
<td>same as In-Order</td>
<td>Same as In-Order, except multiple simultaneous xactions expanded</td>
</tr>
<tr>
<td>N/A</td>
<td>N/A</td>
<td>Control state pipelined along with data. Scoreboard controls issue of next in-order operation</td>
<td>Control state pipelined along with data. Issue buffer executes operation out-of-order</td>
<td>Control state pipelined along with data. Issue next operation from ready xaction.</td>
</tr>
</tbody>
</table>
BHPL 0.5 Overview

Applications (including OPL patterns)

BHPL

App-to-UTL Mappings Layer

FFT to SIMD array

Problem: Application Computation
Solution: UTL Machine

UTL-to-UTL Transformation Layer

Time-Multiplexing

Problem: UTL violates constraint (too big, too slow)
Solution: Transformed UTL

UTL-to-RTL Transformation Layer

Microcoded Engine

Problem: UTL design
Solution: RTL behavior

In-Order Pipeline Engine

RTL-to-Technology

Interleaved Memory

Problem: RTL behavior
Solution: Structural design

FIFO

Lecture 4, Patterns-I

CS294-88, UC Berkeley, Spring 2013