Memory Design Patterns
Multiport Memory Design Patterns

Often we require multiple access ports to a common memory

- True Multiport Memory
  - Completely independent read and write port circuitry

- Banked Multiport Memory
  - Interleave lesser-ported banks to provide higher bandwidth

- Stream-Buffered Multiport Memory
  - Use single wider access port to provide multiple narrower streaming ports

- Cached Multiport Memory
  - Use large single-port main memory, but add cache to service requests for each access port
True Multiport Memory

**Problem:** Require simultaneous read and write access by multiple independent agents to a shared common memory.

**Solution:** Provide separate read and write ports to each bit cell for each requester

**Applicability:** Where unpredictable access latency to the shared memory cannot be tolerated.

**Consequences:** High area, energy, and delay cost for large number of ports. Must define behavior when multiple writes on same cycle to same word (e.g., prohibit, provide priority, or combine writes).
True Multiport Example: Itanium-2 Regfile

- Intel Itanium-2 [Fetzer et al, IEEE JSSCC 2002]

**B. Operand Bypass Datapath**

The integer datapath bypassing is divided into four stages, to afford more timing critical inputs the least possible logic delay to the consuming ALUs. Critical L1 cache return data must flow through only one level of muxing before arriving at the ALU inputs, while DET and WRB data, available from staging latches, have the longest logic path to the ALUs. This allows the bypassing of operands from 34 possible results to occur in a half clock cycle, enabling a single-cycle cache access and instruction execution.
**Banked Multiport Memory**

**Problem:** Require simultaneous read and write access by multiple independent agents to a large shared common memory.

**Solution:** Divide memory capacity into smaller banks, each of which has fewer ports. Requests are distributed across banks using a fixed hashing scheme. Multiple requesters arbitrate for access to same bank/port.

**Applicability:** Requesters can tolerate variable latency for accesses. Accesses are distributed across address space so as to avoid “hotspots”.

**Consequences:** Requesters must wait arbitration delay to determine if request will complete. Have to provide interconnect between each requester and each bank/port. Can have greater, equal, or lesser number of banks*ports/bank compared to total number of external access ports.
Banked Multiport Memory

Port A

Bank 0

Bank 1

Bank 2

Bank 3

Arbitration and Crossbar

Port B
Banked Multiport Memory Example

Pentium (P5) 8-way interleaved data cache, with two ports

Figure 7. Dual-access data cache.

[Alpert et al, IEEE Micro, May 1993]
Stream-Buffered Multiport Memory

**Problem:** Require simultaneous read and write access by multiple independent agents to a large shared common memory, where each requester usually makes multiple sequential accesses.

**Solution:** Organize memory to have a single wide port. Provide each requester with an internal stream buffer that holds width of data returned/consumed by each memory access. Each requester can access own stream buffer without contention, but arbitrates with others to read/write stream buffer from memory.

**Applicability:** Requesters make mostly sequential requests and can tolerate variable latency for accesses.

**Consequences:** Requesters must wait arbitration delay to determine if request will complete. Have to provide stream buffers for each requester. Need sufficient access width to serve aggregate bandwidth demands of all requesters, but wide data access can be wasted if not all used by requester. Have to specify memory consistency model between ports (e.g., provide stream flush operations).
Stream-Buffered Multiport Memory

- Port A
- Port B
- Stream Buffer A
- Stream Buffer B
- Arbitration
- Wide Memory
Stream-Buffered Multiport Examples

- IBM Cell microprocessor local store

[Chen et al., IBM, 2005]
Cached Multiport Memory

**Problem:** Require simultaneous read and write access by multiple independent agents to a large shared common memory.

**Solution:** Provide each access port with a local cache of recently touched addresses from common memory, and use a cache coherence protocol to keep the cache contents in sync.

**Applicability:** Request streams have significant temporal locality, and limited communication between different ports.

**Consequences:** Requesters will experience variable delay depending on access pattern and operation of cache coherence protocol. Tag overhead in both area, delay, and energy/access. Complexity of cache coherence protocol.
Cached Multiport Memory

Port A

Cache A

↓

Arbitration and Interconnect

↓

Common Memory

↑

↓

Arbitration and Interconnect

↑

↓

Cache B

Port B
Replicated-State Multiport Memory

**Problem:** Require simultaneous read and write access by multiple independent agents to a small shared common memory. Cannot tolerate variable latency of access.

**Solution:** Replicate storage and divide read ports among replicas. Each replica has enough write ports to keep all replicas in sync.

**Applicability:** Many read ports required, and variable latency cannot be tolerated.

**Consequences:** Potential increase in latency between some writers and some readers.
Replicated-State Multiport Memory

Write Port 0  Write Port 1

Copy 0

Copy 1

Read Ports

Example: Alpha 21264 Regfile clusters
Memory Hierarchy Design Patterns

Use small fast memory together large slow memory to provide illusion of large fast memory.

- Explicitly managed local stores
- Automatically managed cache hierarchies
Interconnect Design Patterns
Implementing Communication Queues

- Queue can be implemented as centralized FIFO with single control FSM if both ends are close to each other and directly connected:

- In large designs, there may be several cycles of communication latency from one end to other. This introduces delay both in forward data propagation and in reverse flow control.

- Control split into send and receive portions. A credit-based flow control scheme is often used to tell sender how many units of data it can send before overflowing receiver’s buffer.
For one-way latency of \( N \) cycles, need \( 2^*N \) buffers at receiver to ensure full bandwidth
- Will take at least \( 2N \) cycles before sender can be informed that first unit sent was consumed (or not) by receiver

If receive buffer fills up and stalls communication, will take \( N \) cycles before first credit flows back to sender to restart flow, then \( N \) cycles for value to arrive from sender
- meanwhile, receiver can work from \( 2^*N \) buffered values
An alternative to end-end control is distributed flow control (chain of FIFOs)

Requires less storage, as communication flops reused as buffers, but needs more distributed control circuitry

– Lots of small buffers also less efficient than single larger buffer

Sometimes not possible to insert logic into communication path

– e.g., wave-pipelined multi-cycle wiring path, or photonic link
Network Patterns

Connects multiple units using shared resources

- Bus
  - Low-cost, ordered

- Crossbar
  - High-performance

- Multi-stage network
  - Trade cost/performance
- Buses were popular board-level option for implementing communication as they saved pins and wires
- Less attractive on-chip as wires are plentiful and buses are slow and cumbersome with central control
- Often used on-chip when shrinking existing legacy system design onto single chip
- Newer designs moving to either dedicated point-point unit communications or an on-chip network
On-Chip Network

- On-chip network multiplexes long range wires to reduce cost
- Routers use distributed flow control to transmit packets
- Units usually need end-end credit flow control in addition because intermediate buffering in network is shared by all units
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 5, Patterns-II

CS294-88, UC Berkeley, Spring 2013
Unit-Transaction Level (UTL)

- A UTL design’s functionality is specified as sequences of atomic transactions performed at each unit, affecting only local state and I/O of unit
  - i.e., *serializable*: can reach any legal state by single-stepping entire system, one transaction at a time
  - High-level UTL spec admits various mappings into RTL with various cycle timings and overlap of transactions’ executions
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
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)
From BHPL to Chisel Libraries

- Build an interlocking Chisel library stack to mirror BHPL pattern hierarchy
Chisel Pattern Stack

Applications (including OPL patterns)

UTL-to-UTL Transformations

UTL-to-RTL Transformations

RTL-to-Technology

parameters

Detailed Design

Latencies, Area, Energy

FFT to SIMD array

Time-Multiplexing

Unrolling

Microcoded Engine

In-Order Pipeline Engine

Interleaved Memory

FIFO

CAM
First Pass: Architectural Parameters Down

- From high-level UTL design, propagate sizes of *architectural* memories, functional units, communication links down to block generators.
- These sizes independent of microarchitectural implementation.
- Perhaps passed in skeletal units containing only architectural information.
  - E.g., state, links, transaction code, bit widths.
- Pass skeletal units to RTL generators to estimate physical parameters.
Second Pass: Leaf Parameters Up

- Leaf block generators determine latencies, throughputs, area, energy/op
  - E.g. for a memory of width*depth, or functional unit of certain complexity, or link spanning certain distance
- Leaf blocks may do local design-space exploration, or use precomputed tables of known good points
- Can use crude area and place+route information to estimate communication link latencies and energy
- Feed leaf block numbers back up to top-level synthesis
- May repeat first/second pass for coarse-grain design-space exploration
Third Pass: Elaborate Detailed Unit Design

- Flesh out unit designs
  - E.g., add instruction memories to hold transaction code, pipeline registers, buffers for ongoing transactional state
- Pass down to detailed unit generators to expand into functional RTL