1 Mem+Net: Banked Memory Systems

2 Proc+Net: Message-Passing Systems

3 Proc+Mem+Net: Shared-Memory Systems

4 Memory Synchronization, Consistency, and Coherence
   4.1. Memory Synchronization
   4.2. Memory Consistency
   4.3. Memory Coherence
1. **Mem+Net: Banked Memory Systems**

- SWMR/MWSR buses used for memory request/response messages
- Addresses usually cache-line interleaved across banks
- Address indicates destination for req message
- Assume 16 × 16B cache lines, address mapping:

Assume all transactions hit in the cache, unpipelined FSM cache with TC/RD states on hit path, single-cycle request/response bus.

```
rd 0x1000
rd 0x1010
rd 0x1014
rd 0x1018
rd 0x1020
```
• We can use queues to help decouple the network from the cache banks

• We queue up transactions destined for the same bank (bank conflicts) to enable moving on to other transactions

Assume all transactions hit in the cache, unpipelined FSM cache with TC/RD states on hit path, single-cycle request/response bus.

```
rd 0x1000
rd 0x1010
rd 0x1014
rd 0x1018
rd 0x1020
```
2. **Proc+Net: Message-Passing Systems**

- Use explicit messages to communicate between the processors
- Each processor has its own local memory that is not accessible by other processors

```c
// Assumes four processors and arrays have 64 elements

// Processor 0 executes this function
vvadd_p0( int* dest, int* src0, int* src1 ) {
    send( 1, src0[16], 16 ); send( 1, src1[16], 16 ); // Distribute
    send( 2, src0[32], 16 ); send( 2, src1[32], 16 ); // source
    send( 3, src0[48], 16 ); send( 3, src1[48], 16 ); // data

    vvadd_serial( dest, src0, src1, 16 );
}

recv( 1, dest[16], 16 ); // Collect
recv( 2, dest[32], 16 ); // result
recv( 3, dest[48], 16 ); // data
}

// Processors 1-3 execute this function
vvadd_pN() {
    int local_dest[16]; int local_src0[16]; int local_src1[16];

    recv( 0, local_src0, 16 );
    recv( 0, local_src1, 16 );

    vvadd_serial( local_dest, local_src0, local_src1, 16 );

    send( 0, local_dest, 16 );
}
```
3. Proc+Mem+Net: Shared-Memory Systems

- Processors implicitly communicate through a globally shared memory
- Can map a high-level message passing framework to shared memory

```
1 // Assumes four processors and each array has 64 elements
2 int done[4] = { 0, 0, 0, 0 };
3
4 // Processor 0 executes this function
5 vvadd_p0( int* dest, int* src0, int* src1 ) {
6    vvadd_serial( dest, src0, src1, 16 );
7 }
8
9 // Wait for other processors to finish
10 for ( int i = 1; i < 4; i++ ) {
11    while ( done[i] != 1 ) { }
12 }
13
14 // Processors 1-3 executes this function
15 vvadd_pN() {
16    int idx = 16 * processor_id;
17    vvadd_serial( dest[idx], src0[idx], src1[idx], 16 );
18    done[processor_id] = 1;
19 }
```
3. Proc+Mem+Net: Shared-Memory Systems

**Unified Shared I/D Cache**

- Unified Shared Instruction and Data Cache
- Req/Resp Memory Network

**Shared I/D Caches**

- Shared Instruction Cache
- Req/Resp Cache Network
- Shared Data Cache

**Shared/Banked I/D Caches**

- Req/Resp Mem Network
- I$ Bank
- Req/Resp Cache Network
- P
- Req/Resp Cache Network
- Req/Resp Main Memory Network
- D$ Bank

**Private I/D Caches**

- Req/Resp Mem Network
- I$
- Req/Resp Mem Network
- P
- Req/Resp Mem Network
- D$
4. Memory Synchronization, Consistency, and Coherence

- **Memory Synchronization**: How processors “hand-shake” at certain points to reach an agreement or commit to a certain sequence of actions

- **Memory Consistency**: The order in which a single processor appears to update memory addresses (consistency is usually focused on how the architecture handles memory transactions to different addresses)

- **Memory Coherence**: All processors always have the same view of a given memory address (coherence is usually focused on how the architecture handles memory transactions to the same memory address)
4. Memory Synchronization, Consistency, and Coherence  

4.1. Memory Synchronization

Assume we wish to have processors 0–2 be able to send a single word of data to processor 3.

```c
// Processor sends single word of data to processor 3
void send( int data ) {
    while ( *lock_ptr != 0 ) {} 
    *lock_ptr = 1;
    while ( *flag_ptr != 0 ) {} 
    *flag_ptr = 1;
    *buf_ptr = data;
    *lock_ptr = 0;
}

// Processor 3 receives single word of data
int recv() {
    while ( *flag_ptr != 1 ) {} 
    int data = *buf_ptr;
    *flag_ptr = 0;
    return data;
}
```
Multiple processors can try to send data to processor 3 at the same time.

```
# assembly for beginning of send function
addi x3, x0, 1    # initialize x3 to 1

loop:
  lw  x1, 0(x2)    # x2 is lock_ptr
  bne x1, x0, loop # check if lock is set
  sw  x3, 0(x2)    # set lock
```

Effective \texttt{lw/sw} interleaving:
- P0: \texttt{lw x1, 0(x2)}
- P1: \texttt{lw x1, 0(x2)}
- P0: \texttt{sw x3, 0(x2)}
- P1: \texttt{sw x3, 0(x2)}
Need to provide **mutual exclusion** to ensure only one processor is updating the flag at any given time.

- Carefully crafted software solutions
- Hardware support through special instructions

- Summary : Atomic fetch & or
- Assembly : `amoor.w rd, rs1, rs2`
- Semantics : `atomic {
  temp = M_4B[ R[rs1] ]
  M_4B[ R[rs1] ] = temp | R[rs2]
  R[rd] = temp
}
- Format : R-Type

```
+-----+-----+---------+---------+-----+---------+-------------+
|01000|order| rs2     | rs1     | 010 | rd      | 0101111     |
+-----+-----+---------+---------+-----+---------+-------------+
```

Atomic instructions are a series of operations that all perform atomically with respect to other memory operations. The `amoor.w` instruction will perform a fetch and an OR operation which looks like they both happened at once to other memory operations.
4. Memory Synchronization, Consistency, and Coherence  

4.1. Memory Synchronization

```assembly
1  # assembly for send function
2  addi  x3, x0, 1    # initialize x3 to 1
3  loop:
4  amoor.w x1, x2, x3    # atomic test-and-set
5  bne    x1, x0, loop  # check if got lock
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>x3, x0, 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>amoor.w</td>
<td>x1, x2, x3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td>x1, x0, loop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>x3, x0, 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>amoor.w</td>
<td>x1, x2, x3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td>x1, x0, loop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4. Memory Synchronization, Consistency, and Coherence

4.2. Memory Consistency

A memory consistency model is part of the instruction set, and specifies the valid order in which a microarchitecture can update memory.

Sequential consistency requires that a microarchitecture ensure that all updates to memory appear to happen in program order.
4.3. Memory Coherence

- Cache coherence should ideally not be exposed in the instruction set
- **Cache coherence protocol** ensures that all processors see an updated view of any given memory address