# T08 Integrating Processors, Memories, and Networks

School of Electrical and Computer Engineering  
Cornell University

revision: 2016-11-13-16-16

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mem+Net: Banked Memory Systems</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Proc+Net: Message-Passing Systems</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Proc+Mem+Net: Shared-Memory Systems</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>Memory Synchronization, Consistency, and Coherence</td>
<td>7</td>
</tr>
<tr>
<td>4.1</td>
<td>Memory Synchronization</td>
<td>8</td>
</tr>
<tr>
<td>4.2</td>
<td>Memory Consistency</td>
<td>12</td>
</tr>
<tr>
<td>4.3</td>
<td>Memory Coherence</td>
<td>13</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>rd 0x1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd 0x1010</td>
</tr>
<tr>
<td>rd 0x1014</td>
</tr>
<tr>
<td>rd 0x1018</td>
</tr>
<tr>
<td>rd 0x1020</td>
</tr>
</tbody>
</table>
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

---

```c
// Assumes four processors and each array has 64 elements
int done[4] = { 0, 0, 0, 0 };

// Processor 0 executes this function
vvadd_p0( int* dest, int* src0, int* src1 ) {
    vvadd_serial( dest, src0, src1, 16 );

    // Wait for other processors to finish
    for ( int i = 1; i < 4; i++ ) {
        while ( done[i] != 1 ) { }
    }
}

// Processors 1-3 executes this function
vvadd_pN() {
    int idx = 16 * processor_id;
    vvadd_serial( dest[idx], src0[idx], src1[idx], 16 );
    done[processor_id] = 1;
}
```
Unified Shared I/D Cache

Shared I/D Caches

Shared/Banked I/D Caches

Private I/D Caches
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)
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
# 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 **lw/sw** interleaving:

- P0: **lw** x1, 0(x2)
- P1: **lw** x1, 0(x2)
- P0: **sw** x3, 0(x2)
- P1: **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**
- \[
\text{atomic} \{
\text{temp} = \text{M}_4B[ \text{R}[rs1] ] \\
\text{M}_4B[ \text{R}[rs1] ] = \text{temp} \mid \text{R}[rs2] \\
\text{R}[rd] = \text{temp}
\}
\]

**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

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

```
addi  x3, x0, 1
amoor.w x1, x2, x3
bne  x1, x0, loop
```
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.

```assembly
# Assembly frag from send
1  sw x1, 0(x2)  # write data_ptr
2  sw x3, 0(x4)  # write flag_ptr

# Assembly frag from recv
1  lw x2, 0(x3)  # read flag_ptr
2  lw x4, 0(x5)  # read buf_ptr
```
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