ECE 4750 Computer Architecture, Fall 2015
T08 Integrating Processors, Memories, and Networks

School of Electrical and Computer Engineering
Cornell University

revision: 2015-10-28-14-05

1 Mem+Net: Banked Memory Systems **2**

2 Proc+Net: Message-Passing Systems **4**

3 Proc+Mem+Net: Shared-Memory Systems **5**

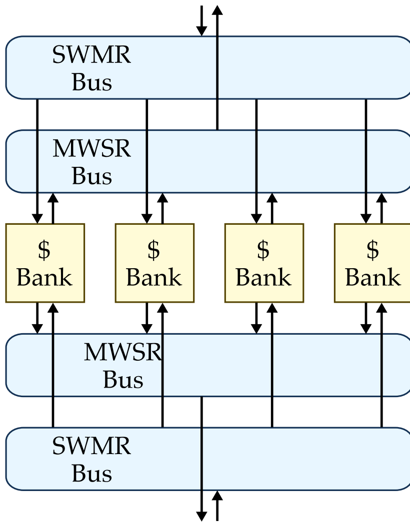
4 Memory Synchronization, Consistency, and Coherence **7**

4.1. Memory Synchronization 8

4.2. Memory Consistency 11

4.3. Memory Coherence 12

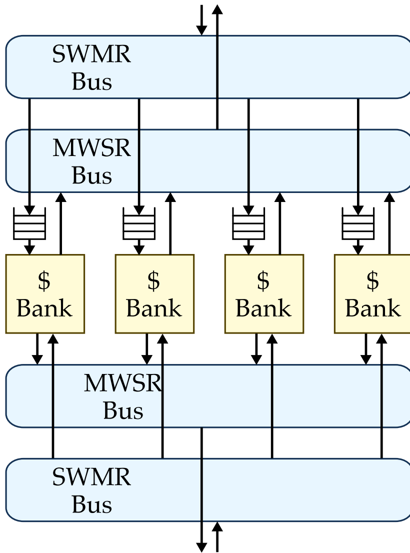
1. Mem+Net: Banked Memory Systems



- 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														

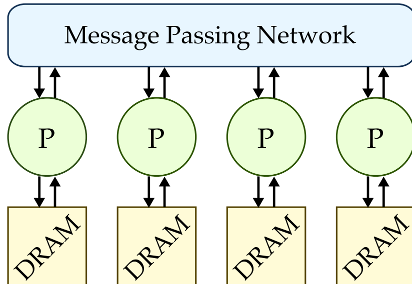


- We can use queues to help decouple the network from the cache banks
- We queue up transactions destinated for the same bank (bank conflicts) to enable moving on to other transactions

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`														

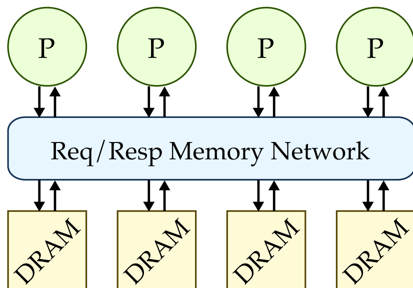
2. Proc+Net: Message-Passing Systems



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

```
1 // Assumes four processors and arrays have 64 elements
2
3 // Processor 0 executes this function
4 vvadd_p0( int* dest, int* src0, int* src1 ) {
5   send( 1, src0[16], 16 ); send( 1, src1[16], 16 );  // Distribute
6   send( 2, src0[32], 16 ); send( 2, src1[32], 16 );  // source
7   send( 3, src0[48], 16 ); send( 3, src1[48], 16 );  // data
8
9   vvadd_serial( dest, src0, src1, 16 );
10
11  recv( 1, dest[16], 16 );                             // Collect
12  recv( 2, dest[32], 16 );                             // result
13  recv( 3, dest[48], 16 );                             // data
14 }
15
16 // Processors 1-3 execute this function
17 vvadd_pN() {
18   int local_dest[16]; int local_src0[16]; int local_src1[16];
19
20   recv( 0, local_src0, 16 );
21   recv( 0, local_src1, 16 );
22
23   vvadd_serial( local_dest, local_src0, local_src1, 16 );
24
25   send( 0, local_dest, 16 );
26 }
```

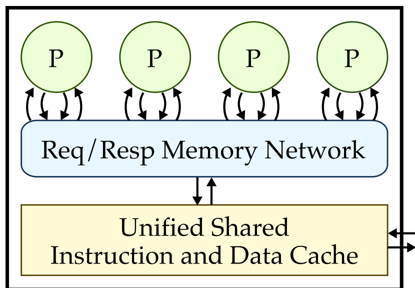
3. Proc+Mem+Net: Shared-Memory Systems



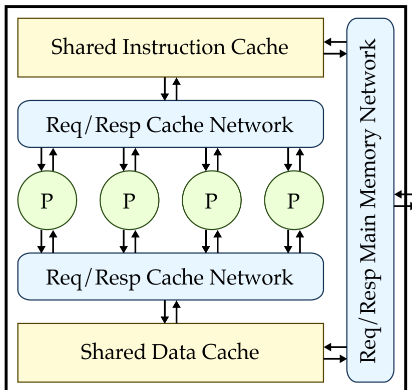
- Processors implicitly communicate through a globally shared memory
- Can map a high-level message passing framework to shared memory

```
1  // Assumes four processors and arrays have 64 elements
2  int done[4] = { 0, 0, 0, 0 };
3
4  // Processor 0 executes this functino
5  vvadd_p0( int* dest, int* src0, int* src1 ) {
6    vvadd_serial( dest, src0, src1, 16 );
7
8    // Wait for other processors to finish
9    for ( int i = 1; i < 4; i++ ) {
10     while ( done[i] != 1 ) { }
11   }
12 }
13
14 // Processors 1-3 executes this functino
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 }
```

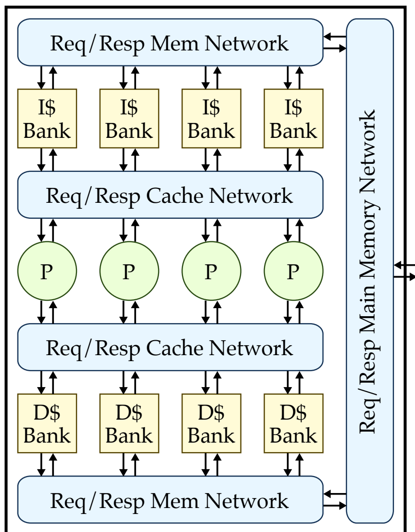
Unified Shared I/D Cache



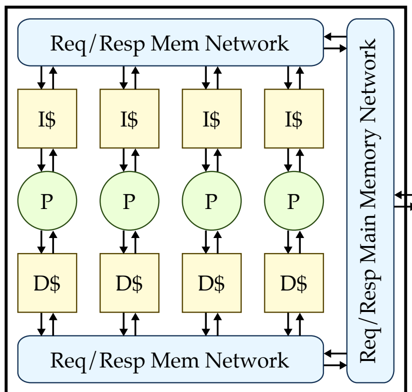
Shared I/D Caches



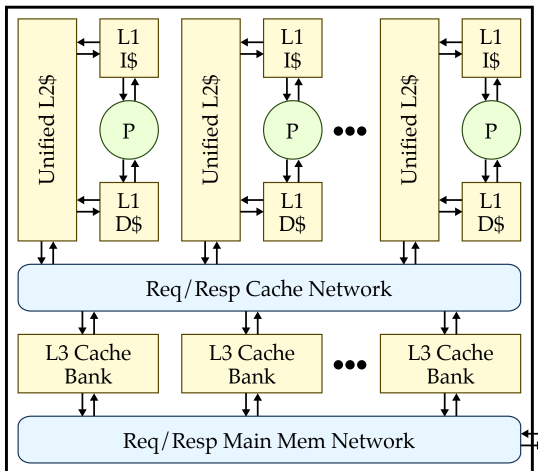
Shared/Banked I/D Caches



Private I/D Caches



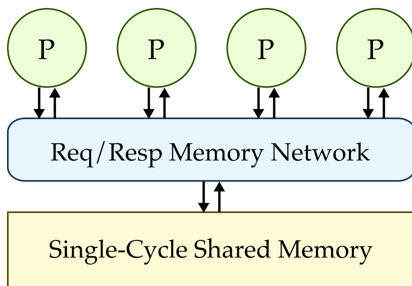
Private L1 I/D, L2 Caches, Shared/Banked L3 Cache



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)

4.1. Memory Synchronization



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

```
1  // Processor sends single word of data to processor 3
2  void send( int data ) {
3    while ( *lock_ptr != 0 ) { }
4    *lock_ptr = 1;
5    while ( *flag_ptr != 0 ) { }
6    *flag_ptr = 1;
7    *buf_ptr  = data;
8    *lock_ptr = 0;
9  }
10
11 // Processor 3 receives single word of data
12 int recv() {
13   while ( *flag_ptr != 1 ) { }
14   int data  = *buf_ptr;
15   *flag_ptr = 0;
16   return data;
17 }
```

Multiple processors can try to send data to processor 3 at the same time.

```
1    # assembly for beginning of send function
2    addiu r3, r0, 1     # initialize r3 to 1
3  loop:
4    lw    r1, 0(r2)     # r2 is lock_ptr
5    bne   r1, r0, loop  # check if lock is set
6    sw    r3, 0(r2)     # set lock
```

addiu																					
lw																					
br																					
opA																					
opB																					
sw																					

addiu																					
lw																					
br																					
opA																					
opB																					
sw																					

Effective `lw`/`sw` interleaving:

- P0: `lw r1` → P1: `lw r1` → P0: `sw r3` →
  P0: `sw r4` → P1: `sw r3` → P1: `sw r4`

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  : amo.or r_dst, r_addr, r_src
- Semantics : atomic {
                temp = M_4B[ R[r_addr] ]
                M_4B[ R[r_addr] ] = temp | R[r_src]
                R[r_dst] = temp
              }
- Format    : R-Type

  31    26 25   21 20   16 15   11 10    6 5       0
 +--------+-------+-------+-------+-------+--------+
 |   op   |  rs   |  rt   |  rd   |  sa   |   cmd  |
 | 100111 | addr  | src   | dst   | 00000 | 000100 |
 +--------+-------+-------+-------+-------+--------+
```

Atomic instructions are a series of operations that all perform atomically with respect to other memory operations. The `amo.and` instruction will perform a fetch and an OR operation which looks like they both happened at once to other memory operations.

```
1   # assembly for send function
2   addiu  r3, r0, 1      # initialize r3 to 1
3 loop:
4   amo.or r1, r2, r3     # atomic test-and-set
5   bne    r1, r0, loop   # check if got lock
```

4.2. Memory Consistency

```
# Assembly frag from send
sw r1, 0(r2)   # write data_ptr
sw r3, 9(r4)   # write flag_ptr
```

```
# Assembly frag from recv
lw r2, 0(r3) # read flag_ptr
lw r4, 0(r5) # read buf_ptr
```

- 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

P P P P

```
  # Assembly frag from send
  sw r1, 0(r2)   # write data_ptr
  sw r3, 9(r4)   # write flag_ptr
```

```
  # Assembly frag from recv
loop:
  lw  r2, 0(r3) # read flag_ptr
  bne r2, r1, loop
  lw  r4, 0(r5) # read buf_ptr
```

- 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