Concurrency

D.N.Rakhmatov

Adopted (with modifications) from:
R. Bryant, CMU
D. Patterson, UC-Berkeley
https://computing.llnl.gov/tutorials/parallel_comp
C. Hamacher et al, *Computer Organization*, 6/E, © 2011 McGraw-Hill
W. Stallings, *Computer Organization and Architecture*, 8/E, © 2010 Pearson
A. Silberschatz et al, *Operating System Concepts Essentials*, 8/E, © 2011 Wiley

Fetch-Decode-Execute Cycle

- Program execution = cycling through 3 basic steps
  - **Fetch:** get the next instruction from memory
  - **Decode:** figure out what the instruction bits mean
  - **Execute:** perform the operation
- Required registers
  - **PC:** Program Counter
    - Contains the memory address of the next instruction (automatically incremented)
    - Changed by branch instructions and pushed onto stack for subroutine calls
  - **IR:** Instruction Register
    - Holds the current instruction (temporary storage during decoding process)

Datapath Stages: MIPS Example

- **Stage 1:** Instruction Fetch
  - No matter what the instruction, an instruction word must first be fetched from memory into **IR**
  - Also, this is where we usually increment **PC** to point to the next instruction
- **Stage 2:** Instruction Decode
  - Upon fetching the instruction word, we next gather and decode information from its fields
    - Read the opcode to determine instruction type and field lengths
    - Read in data from all necessary registers

Datapath Stages: MIPS Example

- **Stage 3:** Execute ALU operation (if needed)
  - The real work of most instructions is done here: arithmetic, shifting, logic, comparisons
  - What about loads/stores? Example: `lw $t0,40($t1)`
    - The address in memory = the value in `$t1` + the value 40
    - We do this addition in this stage
- **Stage 4:** Execute memory access (if needed)
  - Only the load and store instructions do something during this stage; the others remain idle
  - As a result of using the cache memory, this stage is expected to be just as fast (on average) as the others

Datapath Stages: MIPS Example

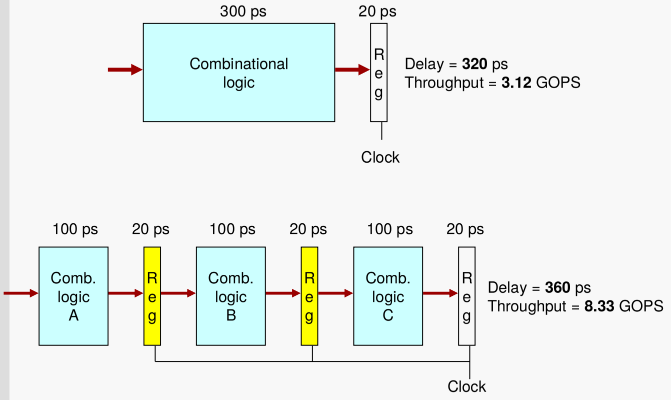
- **Stage 5:** Execute register write (if needed)
  - Most instructions write the result of some computation into a register
    - Examples: arithmetic, logical, shifts, loads
  - What about stores, branches, jumps?
    - Don't write anything into a register at the end
    - These remain idle during this stage
- **Note:**
  - Different machines may have a different number of datapath stages and may do different things per stage
  - **Fetch-Decode-Execute** are the three basic stages found in every computer

Example: STORE instruction

```
sw $r3, 17($r1)
```

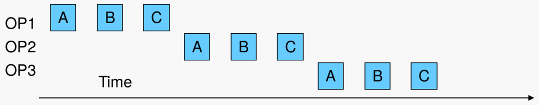
- **Stage 1:** fetch instruction into IR, increment PC
- **Stage 2:** decode to find it's a `sw`, then read registers `$r1` and `$r3`
- **Stage 3:** add 17 to value in register `$r1` (value retrieved in **Stage 2**)
- **Stage 4:** write value in register `$r3` (value retrieved in **Stage 2**) into memory address computed in **Stage 3**
- **Stage 5:** go idle (nothing to write into a register)

Pipelining Idea

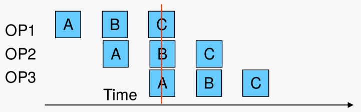


Pipelining Example

- Unpipelined



  - Cannot start new operation until previous one completes
- 3-Way Pipelined



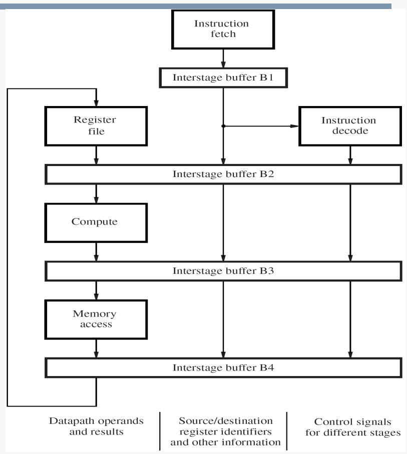
  - Up to 3 operations in process simultaneously

Pipelined Datapath Stages

Clock cycle	1	2	3	4	5	6	7
I_j	Fetch	Decode	Compute	Memory	Write		
I_{j+1}		Fetch	Decode	Compute	Memory	Write	
I_{j+2}			Fetch	Decode	Compute	Memory	Write

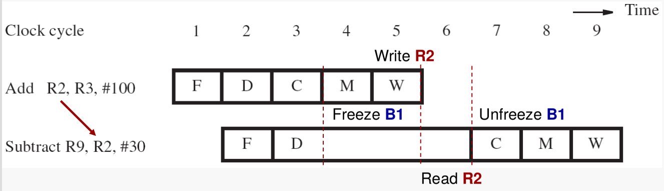
- Need independent stages for pipelining to work
  - All pipeline stages are to take the same amount of time
    - The clock period must accommodate the slowest stage delay
    - Once finished, faster stages sit idle till the end of a clock cycle
  - Challenge: partition system into balanced stages
- Pipelining improves throughput, not latency
  - **Latency:** time to completely execute a certain operation
  - **Throughput:** amount of work done over a period of time

Interstage Buffers

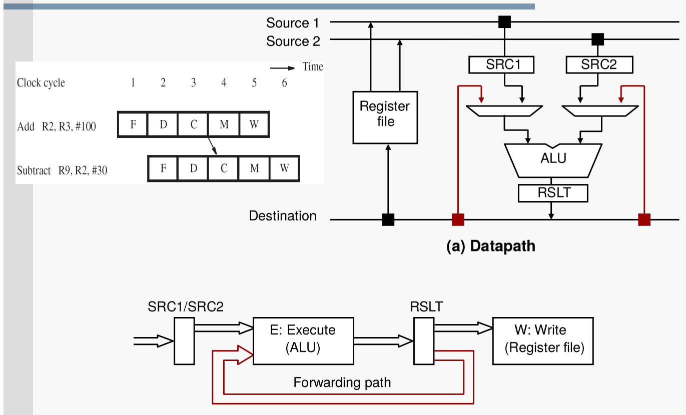


Data Hazards

- Consider two successive instructions I_j and I_{j+1}
  - Assume that the destination register of I_j is the same as the source registers of I_{j+1}
  - Problem: I_j is writing its result to the destination register in cycle 5 (I_j's **Write**), but pipelined I_{j+1} is reading that register in cycle 3 (I_{j+1}'s **Decode**)
  - Solution: stall (delay) I_{j+1} until I_j writes its result



Operand Forwarding



(a) Datapath

(b) Position of source and result registers in pipeline

Software-Based Fix

Clock cycle	1	2	3	4	5	6	7	8	9 → Time
Add R2, R3, #100	F	D	C	M	W				
NOP		F	D	C	M	W			
NOP			F	D	C	M	W		
NOP				F	D	C	M	W	
Subtract R9, R2, #30					F	D	C	M	W

Another Example I

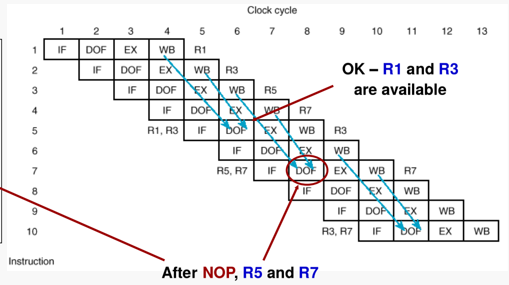
Instruction \ Clock cycle	1	2	3	4	5	6	7	8	9	10
1	IF	DOF	EX	WB						
2		IF	DOF	EX	WB					
3			IF	DOF	EX	WB				
4				IF	DOF	EX	WB			
5					IF	DOF	EX	WB		
6						IF	DOF	EX	WB	
7							IF	DOF	EX	WB

IF	Instruction fetch
DOF	Decoding and operand fetch
EX	Execution (arithmetic/memory)
WB	Write-back to register file

Note: 7 instructions may take only 10 cycles!
(as opposed to 28 cycles without pipelining)

Another Example II

1 ADD R1, R0, R1
2 ADD R3, R2, R3
3 ADD R5, R4, R5
4 ADD R7, R6, R7
5 ADD R3, R1, R3
6 NOP
7 ADD R7, R5, R7
8 NOP
9 NOP
10 ADD R7, R3, R7



Instruction \ Clock cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
1	IF	DOF	EX	WB	R1								
2		IF	DOF	EX	WB	R3							
3			IF	DOF	EX	WB	R5						
4				IF	DOF	EX	WB	R7					
5				R1, R3	IF	DOF	EX	WB	R3				
6						IF	DOF	EX	WB				
7						R5, R7	IF	DOF	EX	WB	R7		
8								IF	DOF	EX	WB		
9									IF	DOF	EX	WB	
10									R3, R7	IF	DOF	EX	WB

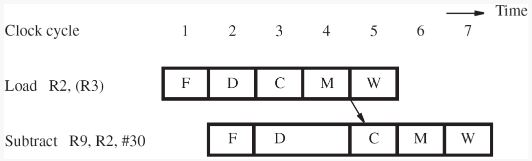
OK – R1 and R3 are available

After NOP, R5 and R7 are available

Note: dependencies and resource conflicts may result in pipeline stalls, which delays instruction completion

(Data) Cache-Related Stalls

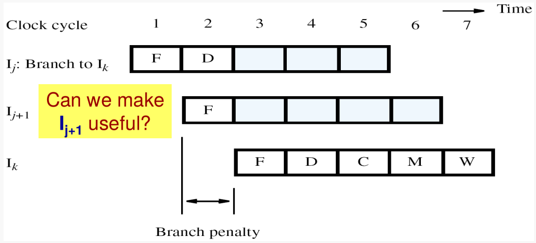
Clock cycle	1	2	3	4	5	6	7	8	9 → Time
I_j: Load R2, (R3)	F	D	C	M			W		
I_{j+1}		F	D	C			M	W	
I_{j+2}			F	D			C	M	W



Clock cycle	1	2	3	4	5	6	7 → Time
Load R2, (R3)	F	D	C	M	W		
Subtract R9, R2, #30		F	D		C	M	W

Control Hazards

- Consider two successive instructions $\mathbf{I_j}$ and $\mathbf{I_{j+1}}$
  - Assume that $\mathbf{I_j}$ is a branch instruction with target $\mathbf{I_k}$
  - Problem: need to resolve the outcome of $\mathbf{I_j}$ as soon as possible and discard $\mathbf{I_{j+1}}$ if the branch to $\mathbf{I_k}$ is taken
  - Solution: modify **Decode** stage to perform branch address calculation and condition checking (if needed)



Clock cycle	1	2	3	4	5	6	7 → Time
I_j: Branch to I_k	F	D					
I_{j+1}		F					
I_k			F	D	C	M	W

Can we make $\mathbf{I_{j+1}}$ useful?

Branch penalty

Filling Delay Slot

```
         Add                  R7, R8, R9
         Branch_if_[R3]=0     TARGET
         I j+1
         :
TARGET:  I k
```

(a) Original sequence of instructions containing a conditional branch instruction

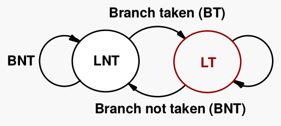
```
         Branch_if_[R3]=0     TARGET
         Add                  R7, R8, R9
         I j+1
         :
TARGET:  I k
```

(b) Placing the Add instruction in the branch delay slot where it is always executed

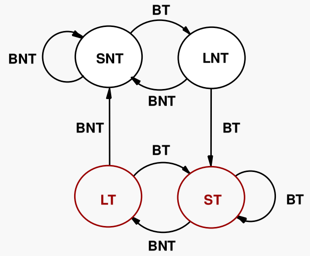
Conditional Branch Prediction I

- Branch outcome is decided in **Decode** stage, while the following instruction is always fetched
  - Problem: the following instruction may have to be discarded, or it may be a NOP (delayed branching)
  - Solution: instead of discarding the following instruction, we predict branch decision in **Fetch** stage, anticipating the next actual instruction
- Dynamic branch prediction
  - Simplest approach: use most recent outcome for likely taken (LT) or likely not-taken (LNT)
    - For branch at end of loop, we mispredict in the last pass, and in the first pass if loop is re-entered
  - Better approach: use strongly/likely taken/not-taken states (ST, LT, SNT, LNT) to avoid misprediction

Conditional Branch Prediction II

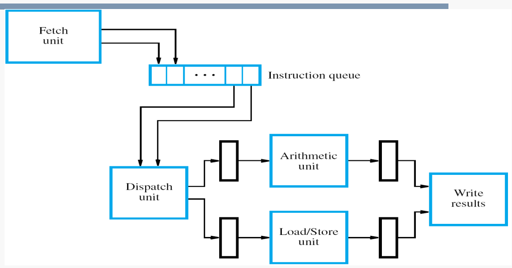


(a) 2-state algorithm



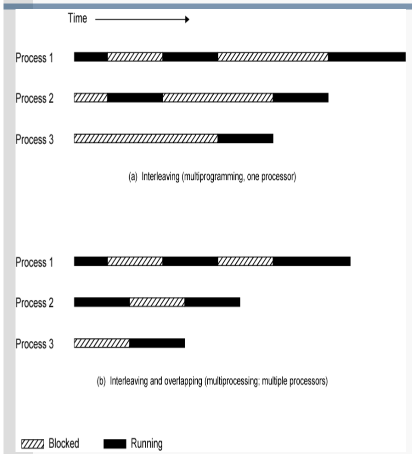
(b) 4-state algorithm

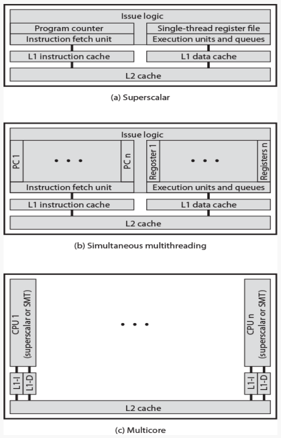
Superscalar Processing Example



Clock cycle	1	2	3	4	5	6 → Time
Add R2, R3, #100	F	D	C	W		
Load R5, 16(R6)	F	D	C	M	W	
Subtract R7, R8, R9		F	D	C	W	
Store R10, 24(R11)		F	D	C	M	W

Beyond Superscalar Processing





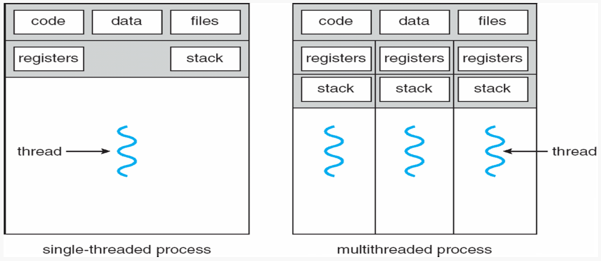
Parallel Programming

- The **programmer** has the responsibility to partition overall computation into tasks and to specify how they are executed in parallel
- Two issues for **parallel programming**:
  - Enabling execution on multiple processors
  - Determining task completion (synchronization)
- Recall: **process** = program and its current state
  - Each process has a corresponding thread, which is an independent path of execution
  - For efficient handling of multiple threads, processors can use hardware multithreading
    - Context switching between threads becomes less punitive

Multithreading I

- Thread creation = issue a system call to OS (e.g., `create_thread`) with a subroutine pointer as a parameter
  - New thread executes that subroutine, calls others (if necessary) using its own stack for local variables
  - Global variables shared with other threads
- Hardware multithreading:
  - Hardware includes multiple sets of registers, including multiple program counters, stack pointers, etc
  - Each set of registers is dedicated to a different thread
    - No time is wasted saving/restoring registers
    - Context switch involves simply changing a hardware pointer to the active set of registers

Multithreading II



- One method to determine when threads complete their tasks is a **barrier** routine
  - The barrier forces threads to wait until all of them have reached the point where this routine is called
  - Threads at the barrier enter a busy-wait loop until the last one arrives to signal all to continue

Programming Example

```
#include     < stdio.h>      /* Routines for input/output. */

#define      N    100        /* Number of elements in each vector. */

double       a[N], b[N];     /* Vectors for computing the dot product. */

void         main (void)
{
             int i;
             double dot_product;

             < Initialize vectors a[], b[] – details omitted.>
             dot_product = 0.0;
             for (i = 0; i <  N; i++)
                dot_product = dot_product + a[i] * b[i];
             printf ("The dot product is %g\n", dot_product);
}
```

4 Processors (Shared Memory) I

```
#include   < stdio.h>          /* Routines for input/output. */
#include   "threads.h"         /* Routines for thread creation/synchronization. */

#define    N    100            /* Number of elements in each vector. */
#define    P    4              /* Number of processors for parallel execution. */

double     a[N], b[N];         /* Vectors for computing the dot product. */
double     partial_sums[P];    /* Array of results computed by threads. */
➡Barrier   bar;                /* Shared variable to support barrier synchronization. */

void       ParallelFunction (void)
{
           int my_id, i, start, end;
           double s;

           my_id = get_my_thread_id ();       /* Get unique identifier for this thread. */
           start = (N/P) * my_id;             /* Determine start/end using thread identifier. */
           end = (N/P) * (my_id + 1) – 1;     /* N is assumed to be evenly divisible by P .* /
           s = 0.0;
           for (i = start; i <= end; i++)
              s = s + a[i] * b[i];
           partial_sums[my_id] = s;           /* Save result in array. */
        ➡  barrier (&bar, P);                 /* Synchronize with other threads. */
}
```

4 Processors (Shared Memory) II

```
void       main (void)
{
           int i;
           double dot_product;

           < Initialize vectors a[], b[] – details omitted.>
        ➡  init_barrier (&bar);
           for (i = 1; i < P; i++)          /* Create P – 1 additional threads. */
            ➡  create_thread (ParallelFunction);
           ParallelFunction();              /* Main thread also joins parallel execution. */
           dot_product = 0.0;               /* After barrier synchronization, compute final result. */
           for (i = 0; i <  P; i++)
              dot_product = dot_product + partial_sums[i];
           printf ("The dot product is %g\n", dot_product);
}
```

Mutex: Mutual Exclusion

- Critical sections
  - Critical sections are code segments of a given process to be executed atomically, i.e., without interference from other concurrent processes that potentially could access the same (shared) resources
- Mutex
  - Mutex is a global (shared) variable indicating whether it is “locked” or “unlocked”
- A process must “lock” an appropriate mutex before entering a critical section, and then “unlock” it when exiting that critical section
  - If a mutex is already “locked” by another process, the requesting process waits until that mutex is “unlocked”

POSIX **Pthreads** Example I

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

/*
The following structure contains the necessary information
to allow the function "dotprod" to access its input data and
place its output into the structure.
*/

typedef struct
 {
   double      *a;
   double      *b;
   double     sum;
   int     veclen;
 } DOTDATA;

/* Define globally accessible variables and a mutex */

#define NUMTHRDS 4
#define VECLEN 100
   DOTDATA dotstr;
   pthread_t callThd[NUMTHRDS];
   pthread_mutex_t mutexsum;
```

POSIX **Pthreads** Example II

```
/*
The function dotprod is activated when the thread is created.
All input to this routine is obtained from a structure
of type DOTDATA and all output from this function is written into
this structure. The benefit of this approach is apparent for the
multi-threaded program: when a thread is created we pass a single
argument to the activated function - typically this argument
is a thread number. All  the other information required by the
function is accessed from the globally accessible structure.
*/
void *dotprod(void *arg)
{

   /* Define and use local variables for convenience */

   int i, start, end, len ;
   long offset;
   double mysum, *x, *y;
   offset = (long)arg;

   len = dotstr.veclen;
   start = offset*len;
   end   = start + len;
   x = dotstr.a;
   y = dotstr.b;
```

POSIX **Pthreads** Example III

```
   /*
   Perform the dot product and assign result
   to the appropriate variable in the structure.
   */

   mysum = 0;
   for (i=start; i<end ; i++)
    {
      mysum += (x[i] * y[i]);
    }

   /*
   Lock a mutex prior to updating the value in the shared
   structure, and unlock it upon updating.
   */
   pthread_mutex_lock (&mutexsum);
   dotstr.sum += mysum;
   pthread_mutex_unlock (&mutexsum);

   pthread_exit((void*) 0);
}
```

POSIX **Pthreads** Example IV

```
int main (int argc, char *argv[])
{
   long i;
   double *a, *b;
   void *status;
   pthread_attr_t attr;

   /* Assign storage and initialize values */
   a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
   b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));

   for (i=0; i<VECLEN*NUMTHRDS; i++)
    {
     a[i]=1.0;
     b[i]=a[i];
    }

   dotstr.veclen = VECLEN;
   dotstr.a = a;
   dotstr.b = b;
   dotstr.sum=0;

   pthread_mutex_init(&mutexsum, NULL);

   /* Create threads to perform the dotproduct  */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
```

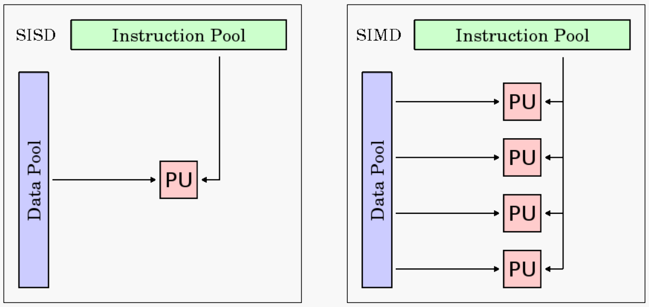
POSIX **Pthreads** Example V

```
        for(i=0; i<NUMTHRDS; i++)
        {
        /*
        Each thread works on a different set of data.
        The offset is specified by 'i'. The size of
        the data for each thread is indicated by VECLEN.
        */
        pthread_create(&callThd[i], &attr, dotprod, (void *)i);
        }

        pthread_attr_destroy(&attr);

        /* Wait on the other threads */
        for(i=0; i<NUMTHRDS; i++)
        {
          pthread_join(callThd[i], &status);
        }

   /* After joining, print out the results and cleanup */
   printf ("Sum =  %f \n", dotstr.sum);
   free (a);
   free (b);
   pthread_mutex_destroy(&mutexsum);
   pthread_exit(NULL);
}
```

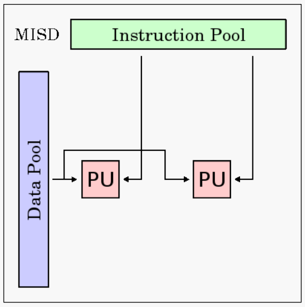
Flynn's Taxonomy I

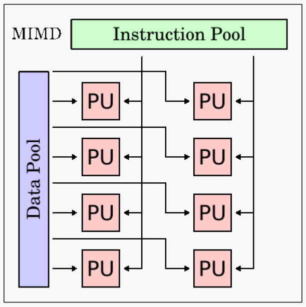
- **SISD:** Single-Instruction Single-Data
  - A single processing unit (PU) executes a single stream of instructions operating on a single data stream
- **SIMD:** Single-Instruction Multiple-Data
  - Multiple PUs execute the same stream of instructions, but each PU works with its own data stream
- **MISD:** Multiple-Instruction Single-Data
  - Multiple PUs work with the same data stream, but each processor executes its own instruction stream
- **MIMD:** Multiple-Instruction Multiple-Data
  - Multiple PUs execute multiple instruction streams operating on multiple data streams

Flynn's Taxonomy II

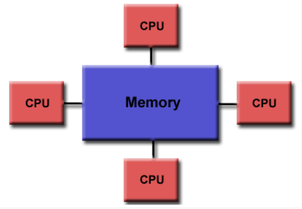


Flynn's Taxonomy III



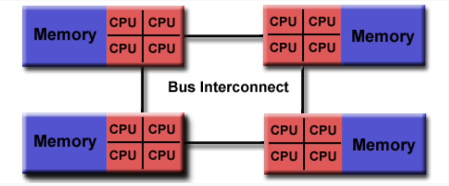


Memory Architecture I

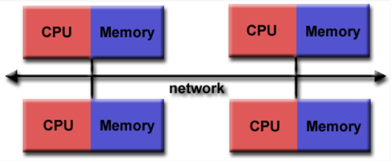


Shared Memory:
Uniform Memory Access (UMA)

Shared Memory:
Nonuniform Memory Access (NUMA)

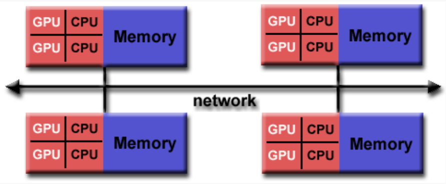


Memory Architecture II



Distributed Memory

Hybrid Memory:
Shared (locally) + Distributed (globally)



Cache Coherence

- Multiprocessors with shared memory allow processors to access the same address locations
- If each processor has a cache, there may be multiple copies of the same shared data
  - When one processor writes to its cache, all other caches will then have incorrect (old) copies that must be either updated or invalidated
- Cache coherence = enforcing consistency of shared data copies residing in multiple caches
  - Example: snoopy caching (bus-based multiprocessors)
  - Example: ownership-based write-back policy
  - Example: directory-based coherence
    - Large-scale distributed-memory systems only

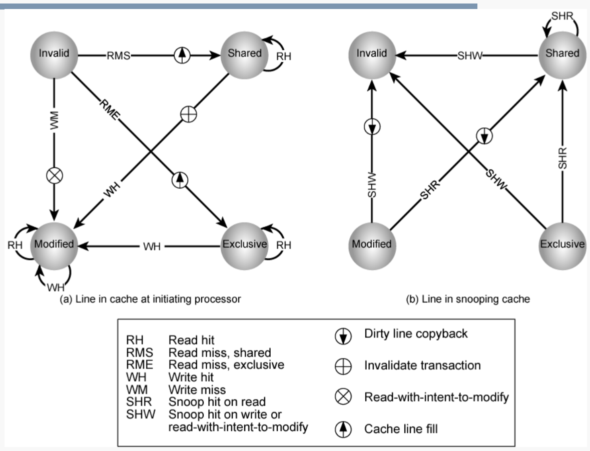
Write-Back Protocol

- Initially, main memory is the owner of all blocks
  - Memory retains ownership of any block cached on read
- Some processor wants to write to its cached block
  - Processor must obtain ownership first, by broadcasting a request to invalidate block's copies cached elsewhere
  - Once the block's owner, processor can modify the block without having to write to main memory or other caches
  - **Read** request from another processor
    - Current owner sends block's copy to requester and memory
      - ✓ Main memory reacquires block's ownership
  - **Write** request from another processor
    - Current owner sends block's copy to requester and invalidates its own copy
      - ✓ Requesting processor acquires block's ownership

Snoopy Caching I

- Broadcasting of requests for cache coherence is most easily accomplished on a single bus
  - Cache controller for each processor snoops (observes) transactions and reacts to enforce coherence
  - Duplicate tags are used to support cache snooping without interfering with processor's own cache accesses
- Each cache block has 4 basic states (**MESI**)
  - Modified: this block is the only copy cached from main memory, and it has been modified
  - Exclusive: this block is the only copy cached from main memory, and it has NOT been modified
  - Shared: this block is one of multiple cached copies, and it has NOT been modified
  - Invalid: this block is an invalid copy

Snoopy Caching II



(a) Line in cache at initiating processor

(b) Line in snooping cache

Amdahl’s Law

- **Speedup = 1/(1 – $f_{enhanced}$ + $f_{enhanced}$/$S_{enhanced}$)**
  - **$S_{enhanced}$** – ideal speedup achievable due to enhancement
  - **$f_{enhanced}$** – fraction of computational time improved due to enhancement
    - Fraction of computational time NOT improved: **1 – $f_{enhanced}$**
- Enhancement: 1 processor → **P** processors
  - Assume: **f** = parallelizable fraction of computation
  - **Speedup = 1/(1 – f + f/P) = P/(P – f(P – 1))**
- Example:
  - **P** = 16, **f** = 0.7, **Speedup** = 2.91
  - **P** = 64, **f** = 0.7, **Speedup** = 3.22
    - Marginal improvement from 2.91 to 3.22 ← **f** is very important!