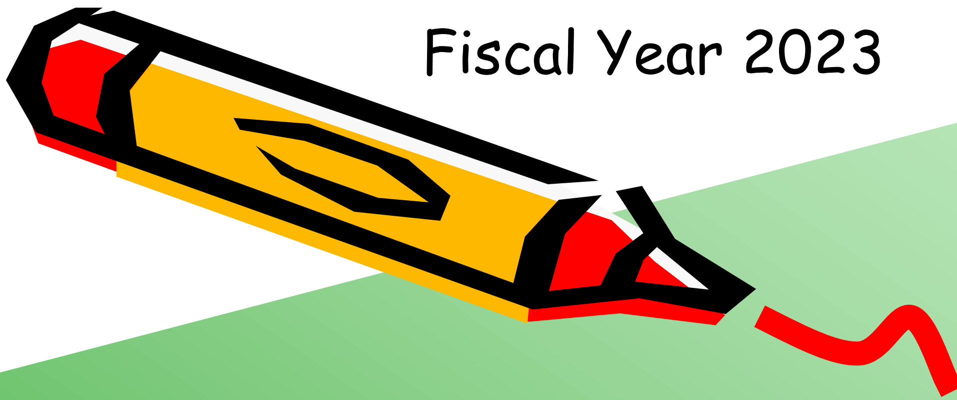


Fiscal Year 2023

Course number: CSC.T433
School of Computing,
Graduate major in Computer Science

Advanced Computer Architecture

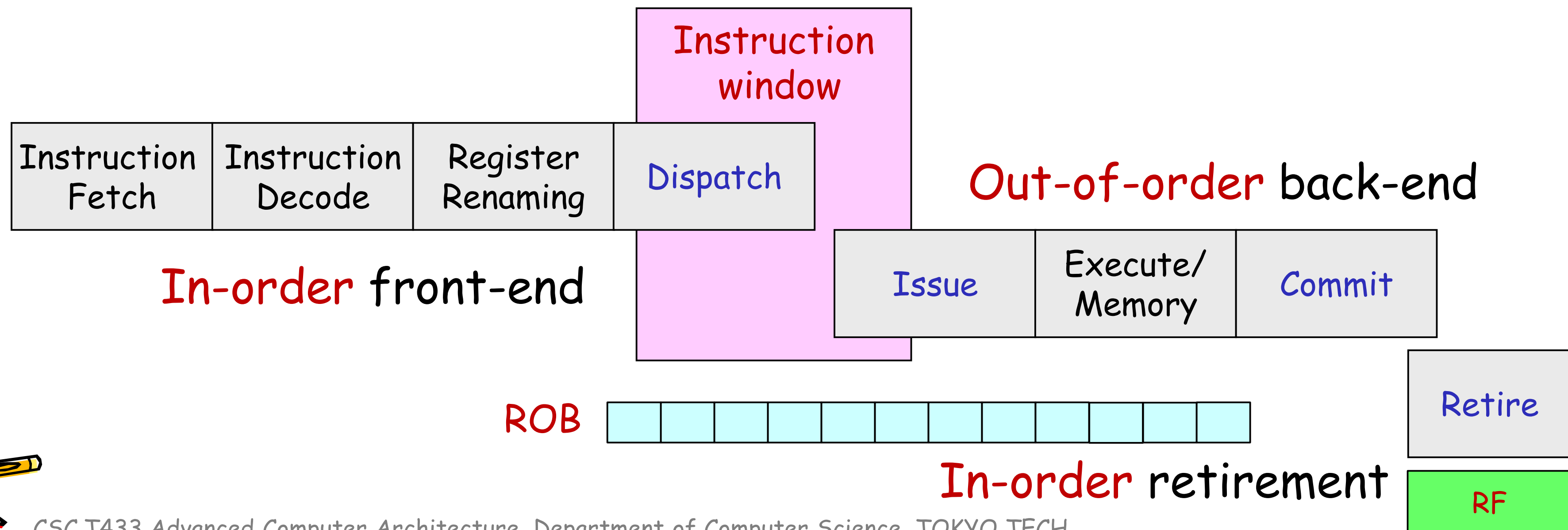
10. Multi-Processor: Distributed Memory and Shared Memory Architecture

www.arch.cs.titech.ac.jp/lecture/ACA/
Room No.W834, Lecture (Face-to-face)
Mon 13:30-15:10, Thr 13:30-15:10

Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp

Instruction pipeline of OoO execution processor

- Allocating instructions to instruction window is called dispatch
- Issue or fire wakes up instructions and their executions begin
- In commit stage, *the computed values are written back to ROB (reorder buffer)*
- The last stage is called retire or graduate.
  The completed consecutive instructions can be retired.
  The result is written back to register file (*architectural register file of 32 registers*) using a logical register number from x0 to x31.



Datapath of OoO execution processor

Instruction flow

Instruction cache

Branch handler

Instruction fetch

Instruction decode

Renaming

Register file

Dispatch

Memory dataflow

Integer

Floating-point

Memory

RS

Instruction window

ALU

ALU

Branch

FP ALU

Adr gen.

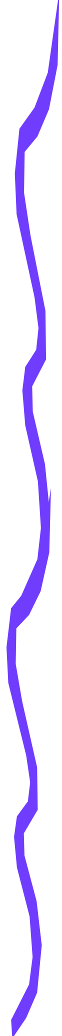
Adr gen.

Store queue

Reorder buffer (ROB)

Data cache

Register dataflow



Reservation station (RS)

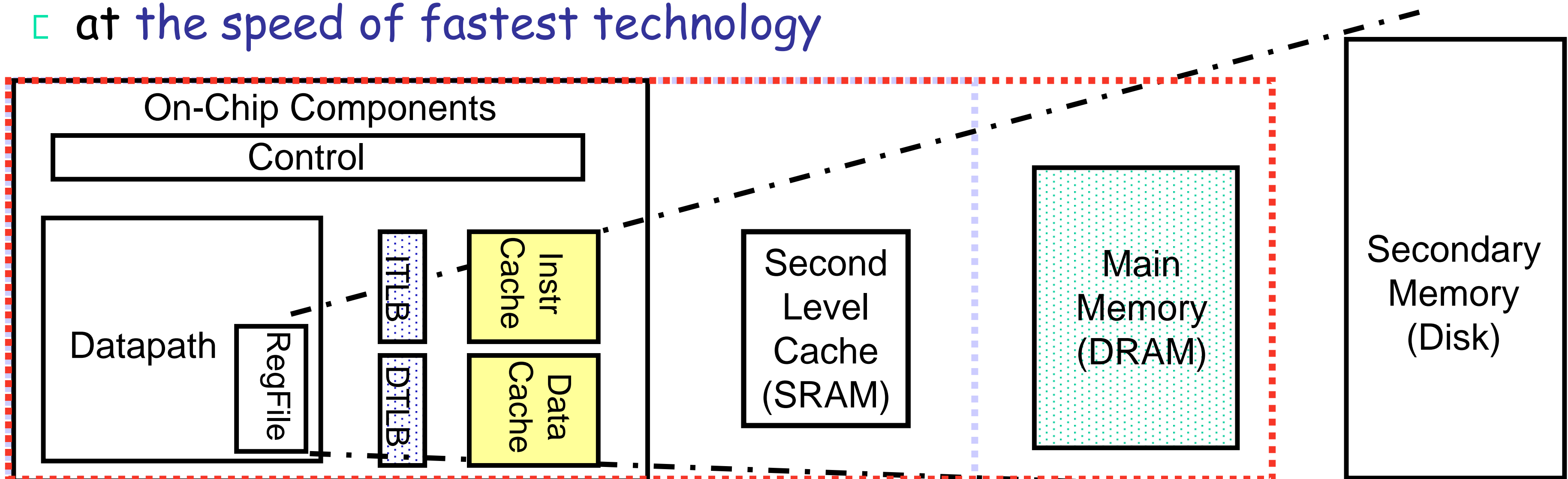
The Memory System's Fact and Goal

- Fact:
  Large memories are slow, and fast memories are small
- How do we create a memory that gives the illusion of being large, fast, and cheap ?

- Temporal Locality (Locality in Time):
  - Keep most recently accessed data items closer to the processor
- Spatial Locality (Locality in Space)
  - Move blocks consisting of contiguous words to the upper levels

A Typical Memory Hierarchy

- By taking advantage of the principle of locality in time and space
  - Present much memory in the cheapest technology
  - at the speed of fastest technology

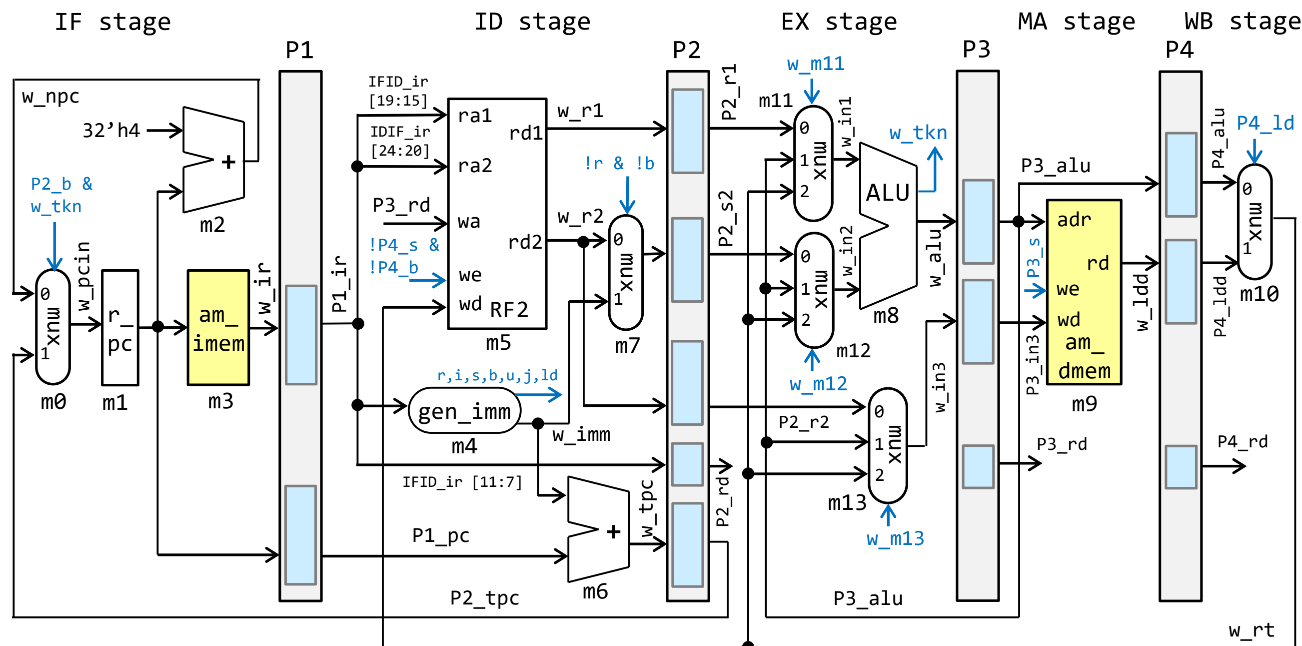


Speed (%cycles):	½'s	1's	10's	100's	1,000's
Size (bytes):	100's	K's	10K's	M's	G's to T's
Cost:	highest				lowest

TLB: Translation Lookaside Buffer

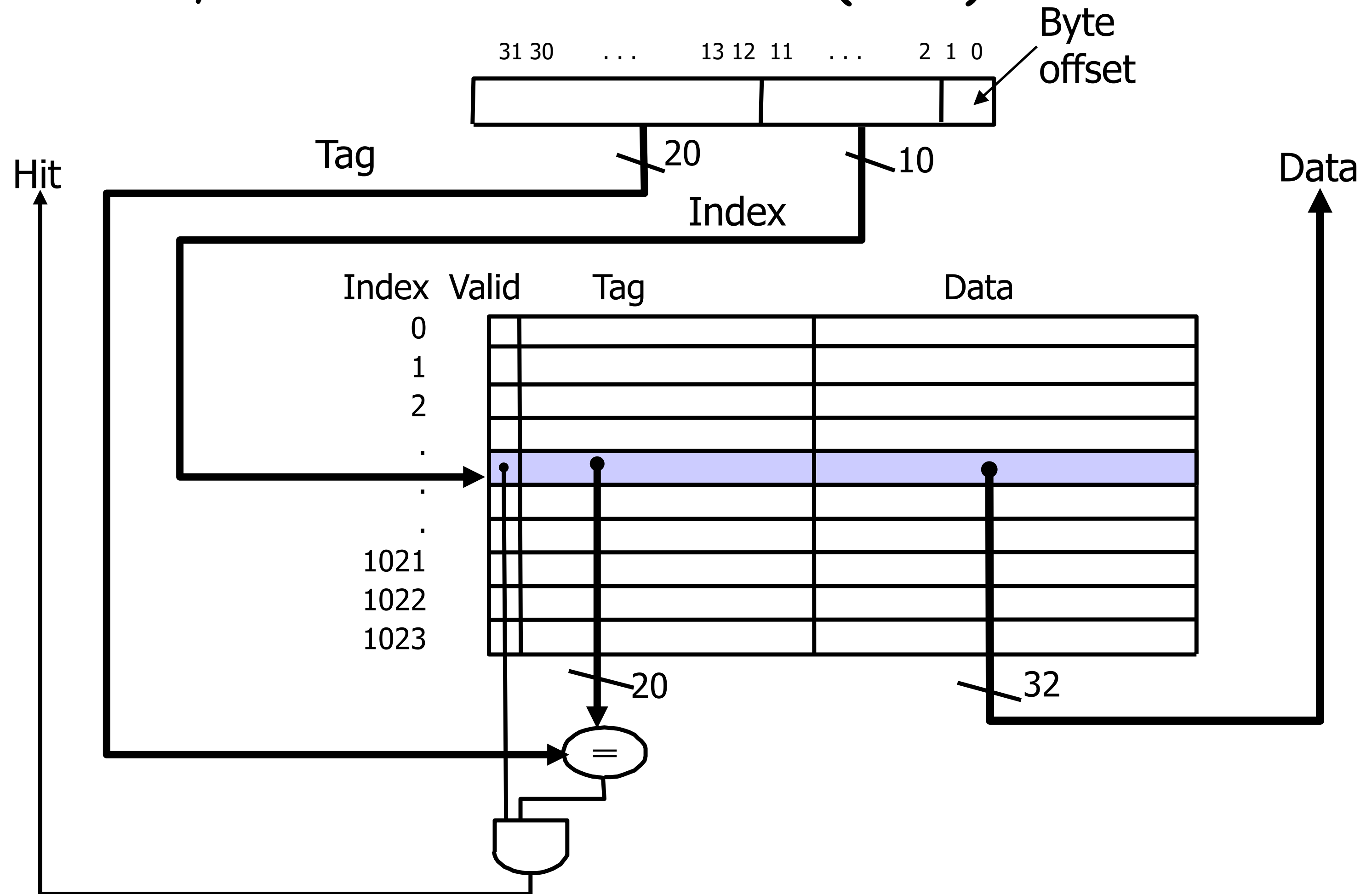
proc9: 5-stage pipelining processor

- The strategy is to separate instruction fetch step (IF), instruction decode step (ID), execution step (EX), memory access step (EX), and write back step (WB).
- Use the pipeline register P3 between EX and MA, and pipeline register P4 between EX and WB.



MIPS Direct Mapped Cache Example

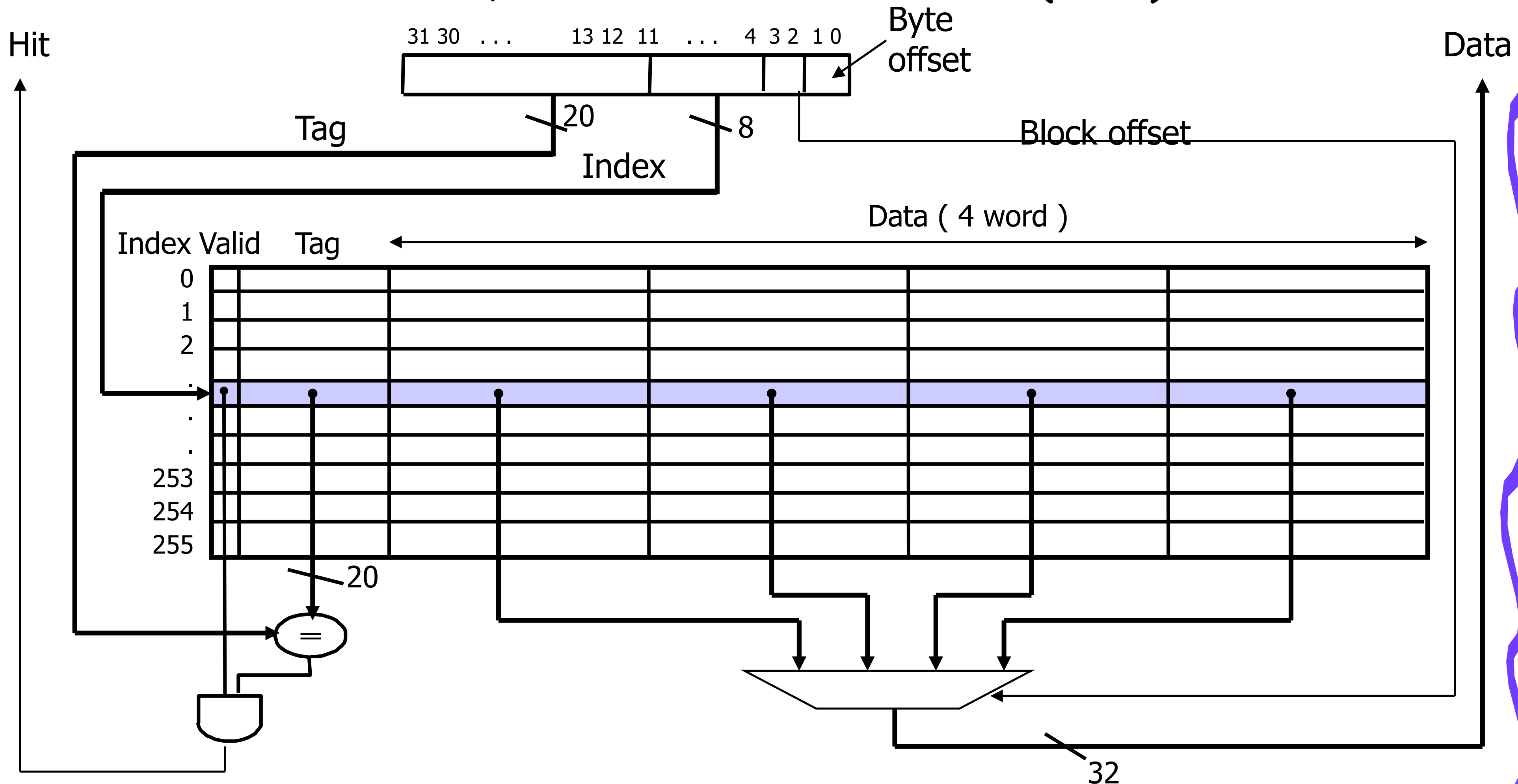
- One word/block, cache size = 1K words (4KB)



What kind of locality are we taking advantage of?

Multiword Block Direct Mapped Cache

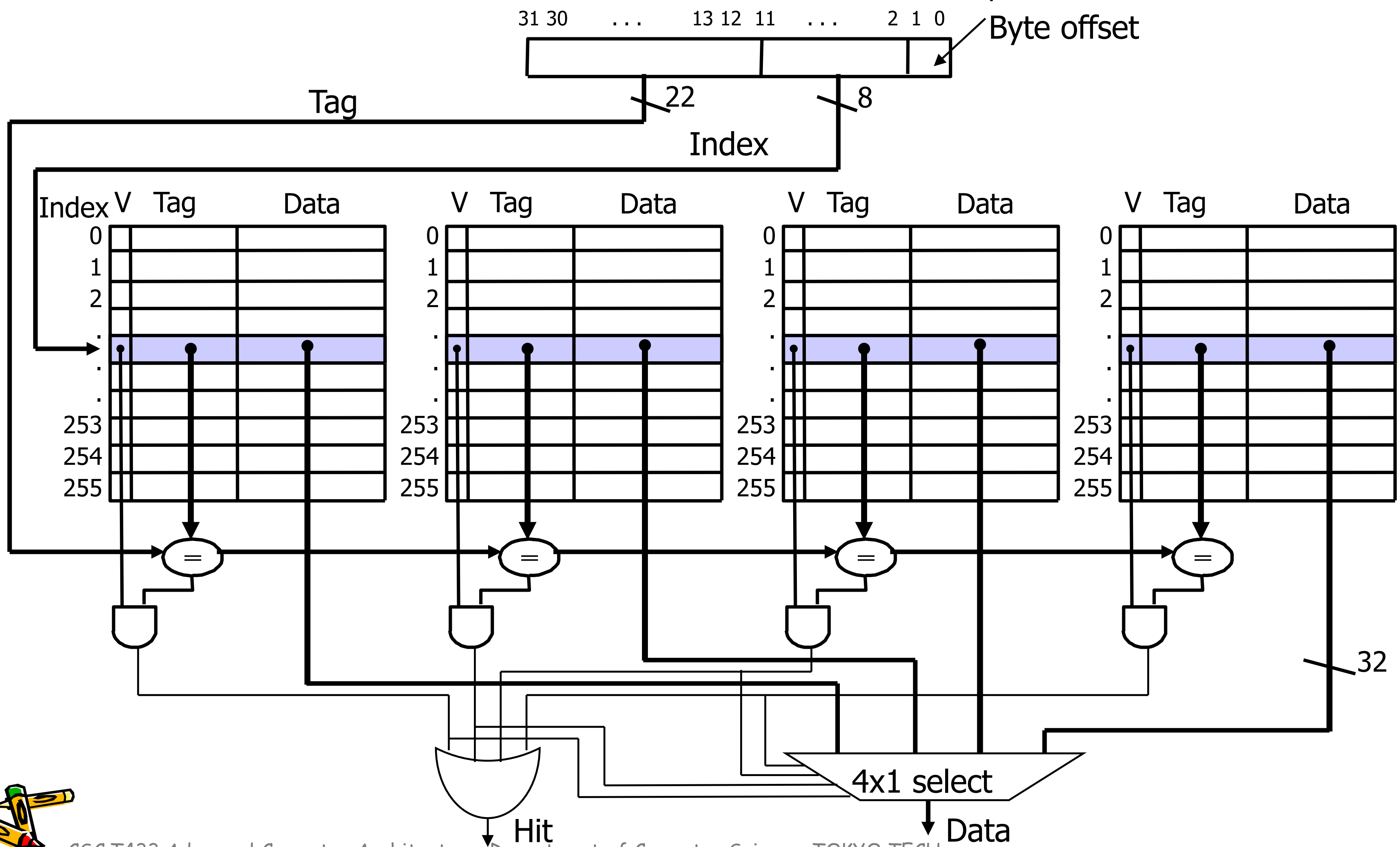
- Four words/block, cache size = 1K words (4KB)



What kind of locality are we taking advantage of?

Four-Way Set Associative Cache

- One word/block, 2^8 = **256 sets** where each with four ways (each with one block)

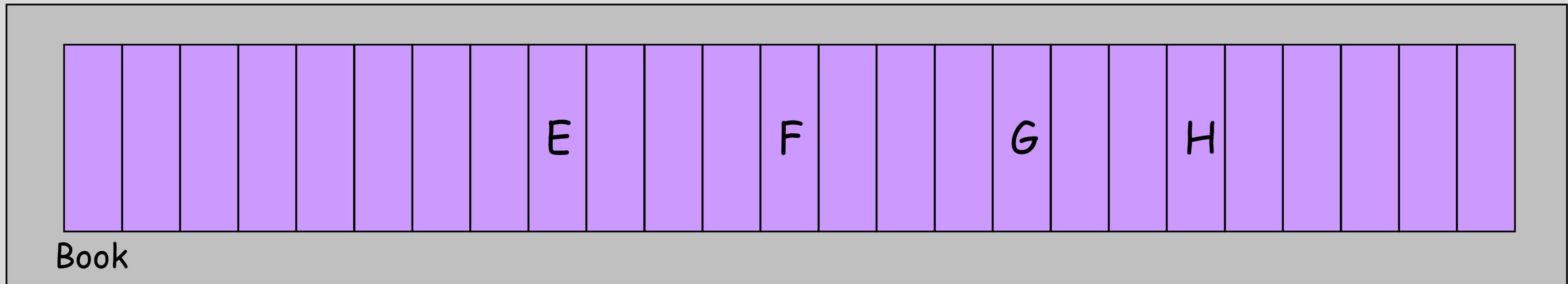


Costs of Set Associative Caches

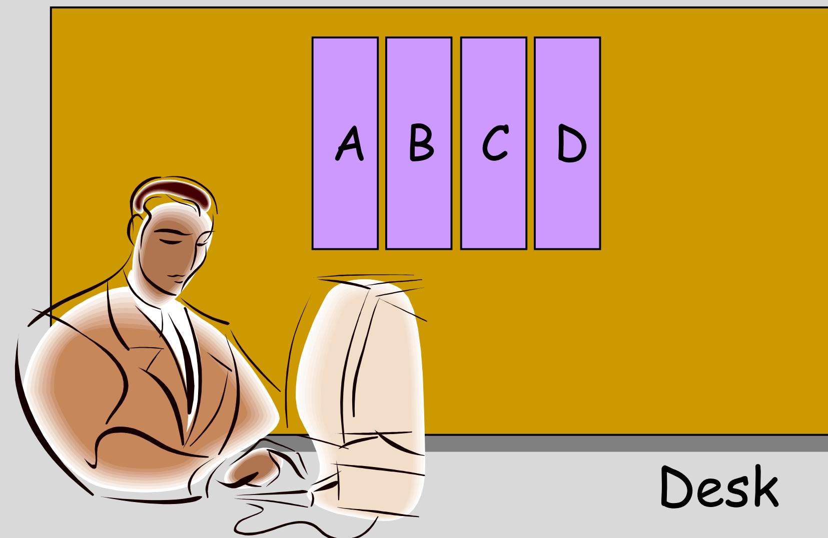
- N-way set associative cache costs
  - N comparators (delay and area)
  - MUX delay (set selection) before data is available
  - Data available after set selection and Hit/Miss decision.
- When a miss occurs,
  which way's block do we pick for replacement ?
  - Least Recently Used (LRU):
    the block replaced is the one that has been unused for the longest time
    - Must have hardware to keep track of when each way's block was used
    - For 2-way set associative, takes one bit per set →
      set the bit when a block is referenced
      (and reset the other way's bit)
  - Random

Exercise 1

Cache Associativity & Replacement Policy



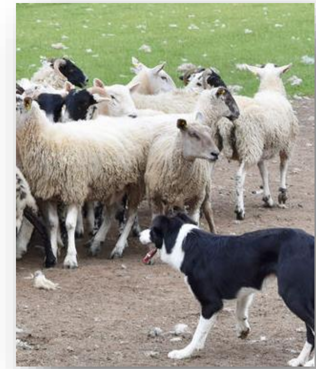
Bookshelf



Desk

Recommended Reading

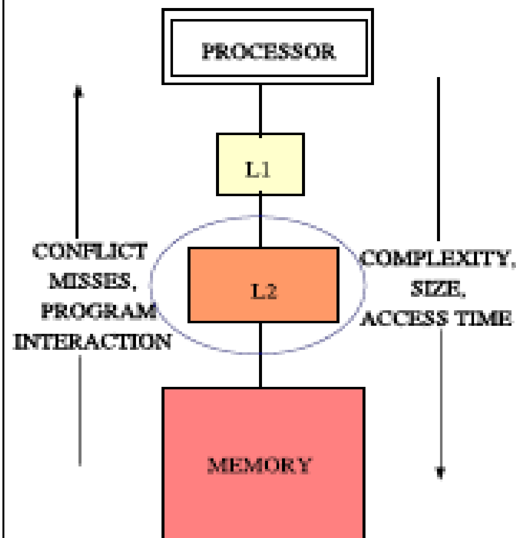
- Emulating Optimal Replacement with a Shepherd Cache
  - Kaushik Rajan, Govindarajan Ramaswamy, Indian Institute of Science
  - MICRO-40, pp. 445-454, 2007
  - Session 8: Cache Replacement Policies



- A quote:
  "The inherent temporal locality in memory accesses is filtered out by the L1 cache. As a consequence, an L2 cache with LRU replacement incurs significantly higher misses than the optimal replacement policy (OPT). We propose to narrow this gap through a novel replacement strategy that mimics the replacement decisions of OPT."

Memory Hierarchy Design

Memory Hierarchy



L2 and lower caches

- Objective : Need to reduce expensive memory accesses
- Design : Large size, Higher associativity, Complex design
- Problem : Do not interact with program directly and observe filtered temporal locality

- High Associativity ⟹ replacement policy crucial to performance
- L1 cache services temporal accesses ⟹ Lack of temporal accesses at L2 ⟹ LRU replacement inefficient
- Replacement decisions are taken off the processor critical path

Emulating Optimal Replacement with a Shepherd Cache, MICRO-2007

LRU has room for improvement

LRU vs OPT

Legend: 512KB-lru16, 512KB-lruFA, 256KB-opt8, 512KB-opt16

MPKI

Off-scale values: 87.5, 87.5, 73.9, 58.4, 80, 79.7, 72.6, 65.9

13%

Benchmarks: art, mcf, gcc, luca, swim, appl, ammp, twolf, vpr, facer, mgrid, apsi, avg26

MPKI for SPEC2000 suite, Benchmarks with MPKI < 5 not plotted but count towards average

Huge performance gap between LRU and OPT
OPT at half the size preferable to LRU at double the size

MPKI: Miss Per Kilo Instructions

Emulating Optimal Replacement with a Shepherd Cache, MICRO-2007

OPT: Optimal Replacement Policy

The Optimal Replacement Policy

1. Replacement Candidates : On a miss any replacement policy could either choose to replace any of the lines in the cache or choose not to place the miss causing line in the cache at all.
2. Self Replacement : The latter choice is referred to as a self-replacement or a cache bypass

Optimal Replacement Policy

On a miss replace the candidate to which an access is least imminent [Belady1966,Mattson1970,McFarling-thesis]

3. Lookahead Window : Window of accesses between miss causing access and the access to the least imminent replacement candidate. Single pass simulation of OPT make use of lookahead windows to identify replacement candidates and modify current cache state [Sugumar-SIGMETRICS1993]

Emulating Optimal Replacement with a Shepherd Cache, MICRO-2007

Example of Optimal Replacement Policy

Understanding OPT

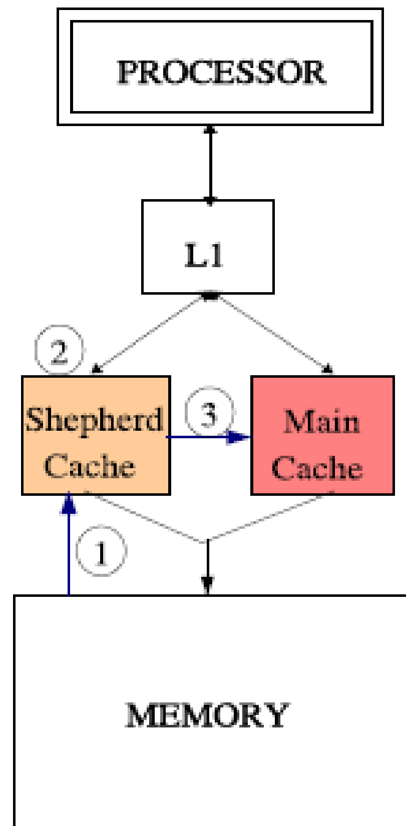
Access Sequence	A_5	A_1	A_6	A_3	A_1	A_4	A_5	A_2	A_5	A_7	A_6	A_8
OPT order for A_5		0		1		2	3	4				
OPT order for A_6				0	1	2	3				4	

- Consider 4 way associative cache with one set initially containing lines (A_1, A_2, A_3, A_4), consider the access stream shown in table
- Access A_5 misses, replacement decision proceeds as follows
    1. Identify replacement candidates : $(A_1, A_2, A_3, A_4, A_5)$
    2. Lookahead and gather imminence order : shown in table, lookahead window circled
    3. Make replacement decision : A_5 replaces A_2
- A_6 self-replaces, lookahead window and imminence order in table

Emulating Optimal Replacement with a Shepherd Cache, MICRO-2007

Shepherd Cache emulation OPT

Emulating OPT with a Shepherd Cache

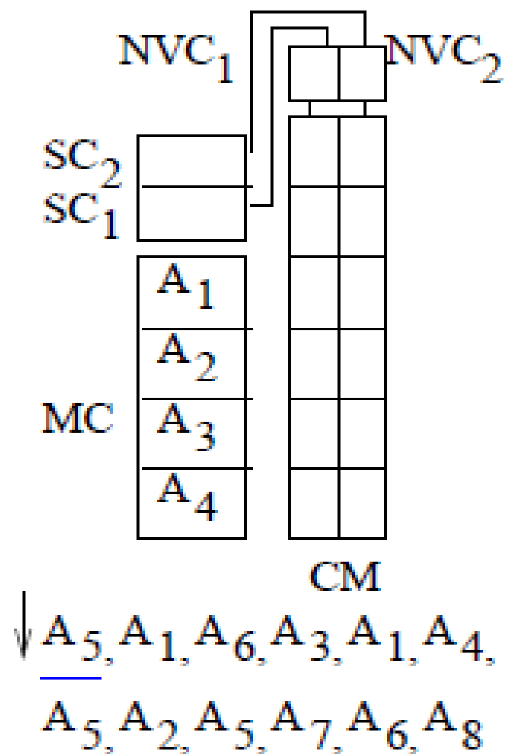


- Split the cache into two logical parts
  - Main Cache (MC) for which optimal replacement is emulated
  - Shepherd Cache (SC) used to provide a lookahead and guide replacements from MC towards OPT
- Operation
  1. Buffer lines temporarily in SC before moving them to MC, SC acts as a FIFO buffer
  2. While in SC, gather imminence information and emulate lookahead
  3. When forced out of SC, make an MC replacement based on the gathered imminence order

Emulating Optimal Replacement with a Shepherd Cache, MICRO-2007

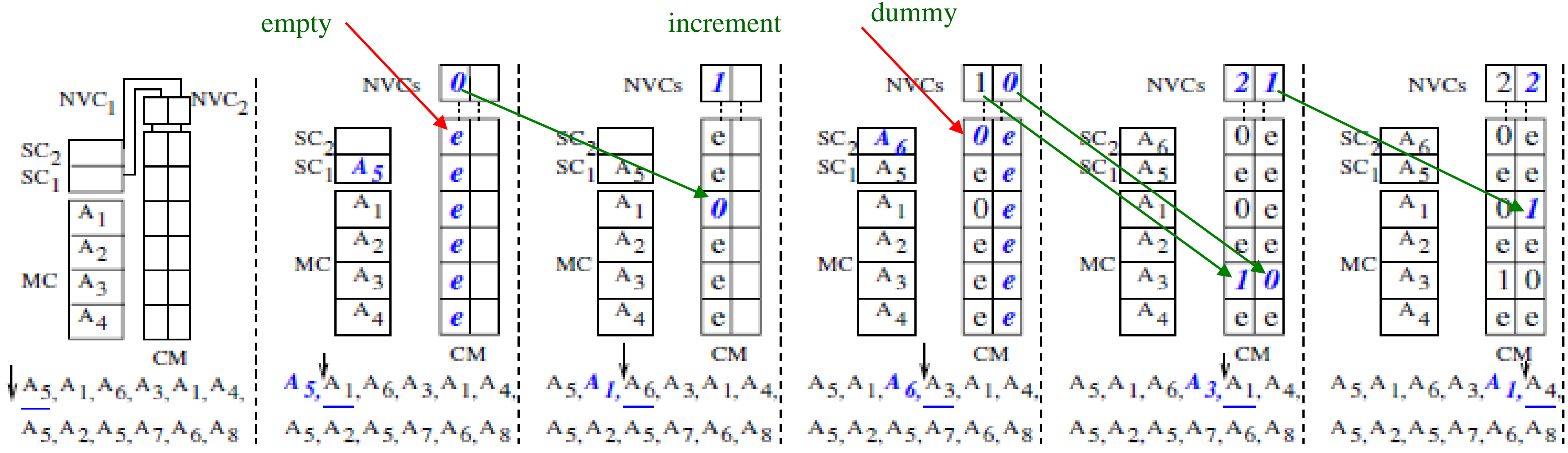
Shepherd Cache Overview

Overview of Shepherd Caching



- To emulate MC with 4 ways per set and 2 SC ways per set
- To gather imminence order add a counter matrix (CM)
- CM has one column per SC way to track imminence order w.r.t to it
- CM has one row per SC and MC line as any of them can be a replacement candidate
- Each column has one Next Value Counter (NVC) to track the next value to assign along column

Emulating Optimal Replacement with a Shepherd Cache, MICRO-2007



(a) Initial State

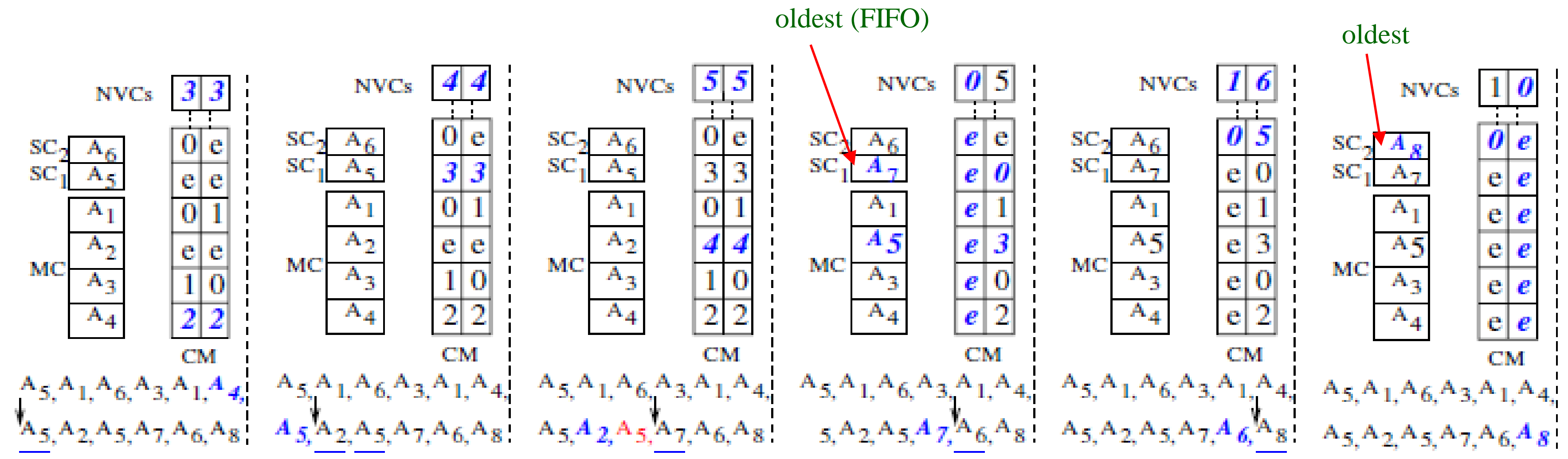
(b) A_5 inserted at SC_1

(c) A_1 added to the optimal order of SC_1

(d) A_6 inserted at SC_2

(e) A_3 added to the optimal order of SC_1,SC_2

(f) A_1 added to optimal order of SC_2



(g) A_4 added to optimal order of SC_1,SC_2

(h) A_5 added to optimal order of SC_1,SC_2

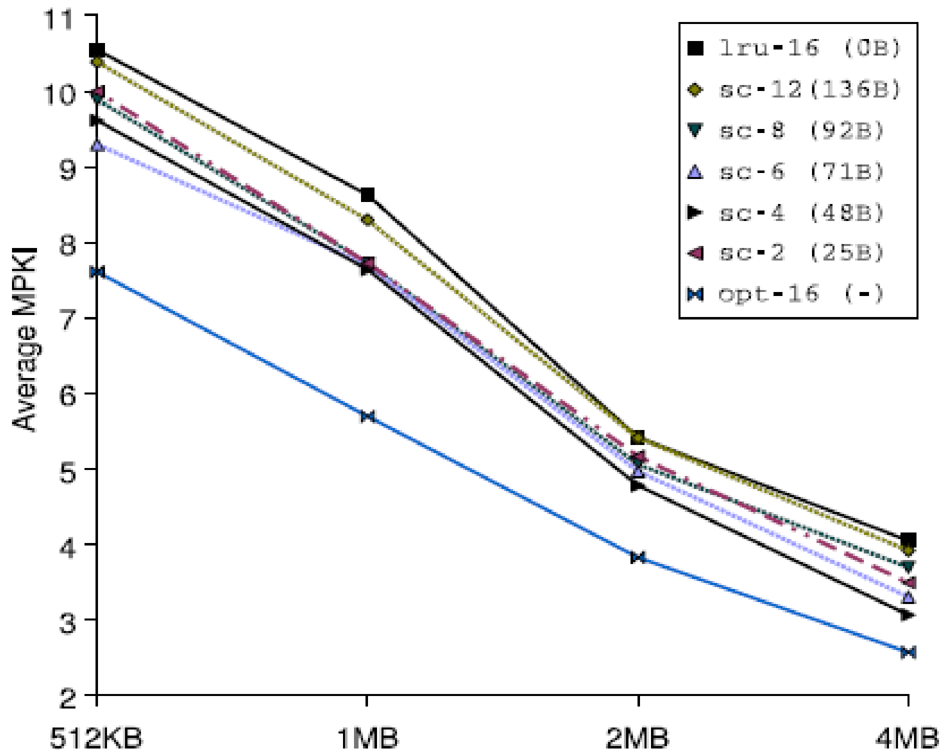
(i) A_2 added to optimal order of SC_1,SC_2

(j) from repla

Access Sequence	A_5	A_1	A_6	A_3	A_1	A_4	A_5	A_2	A_5	A_7	A_6	A_8
OPT order for A_5		0		1		2	3	4				
OPT order for A_6				0	1	2	3				4	

Shepherd cache bridges 32 – 52% of the gap

Bridging the performance gap



Avg MPKI over SPEC2000 suite

Bridging the LRU-OPT gap

- *SC-4 bridges 32-52% of gap*
- SC moves closer to OPT as cache size increases

MPKI: Miss Per Kilo Instructions

Emulating Optimal Replacement with a Shepherd Cache, MICRO-2007

Datapath of OoO execution processor

Instruction flow

Instruction cache

Branch handler

Instruction fetch

Instruction decode

Renaming

Register file

Dispatch

Memory dataflow

Integer

Floating-point

Memory

RS

Instruction window

ALU

ALU

Branch

FP ALU

Adr gen.

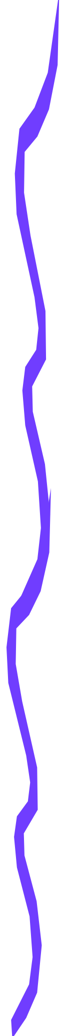
Adr gen.

Store queue

Reorder buffer (ROB)

Data cache

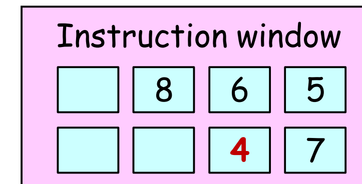
Register dataflow

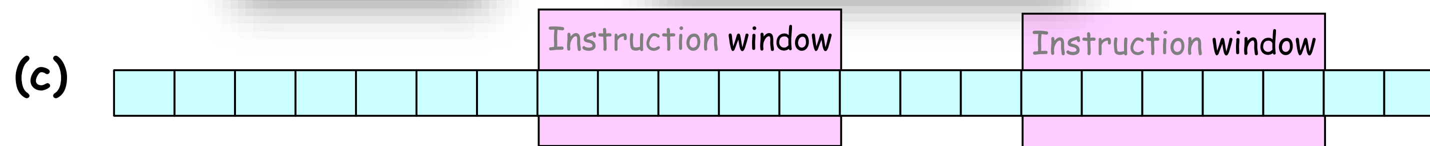


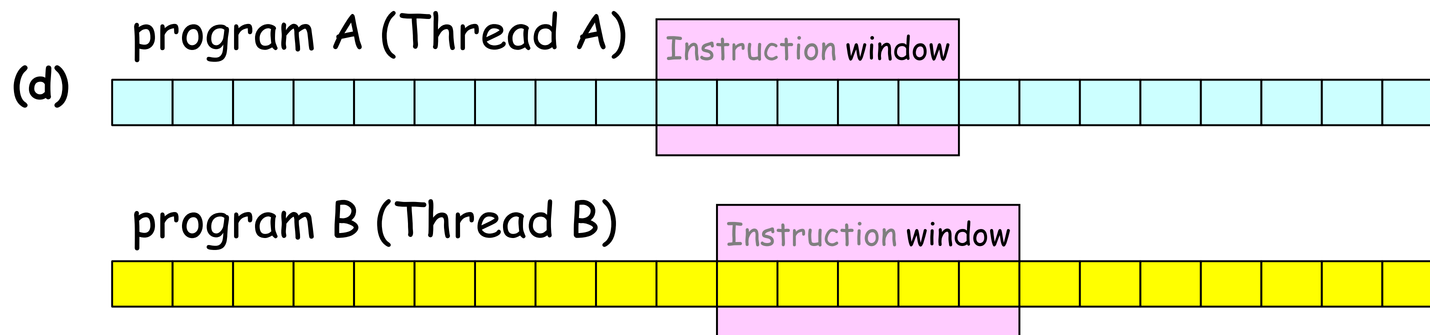
Reservation station (RS)

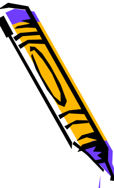
Multiprogramming

- Several independent programs run at the same time.



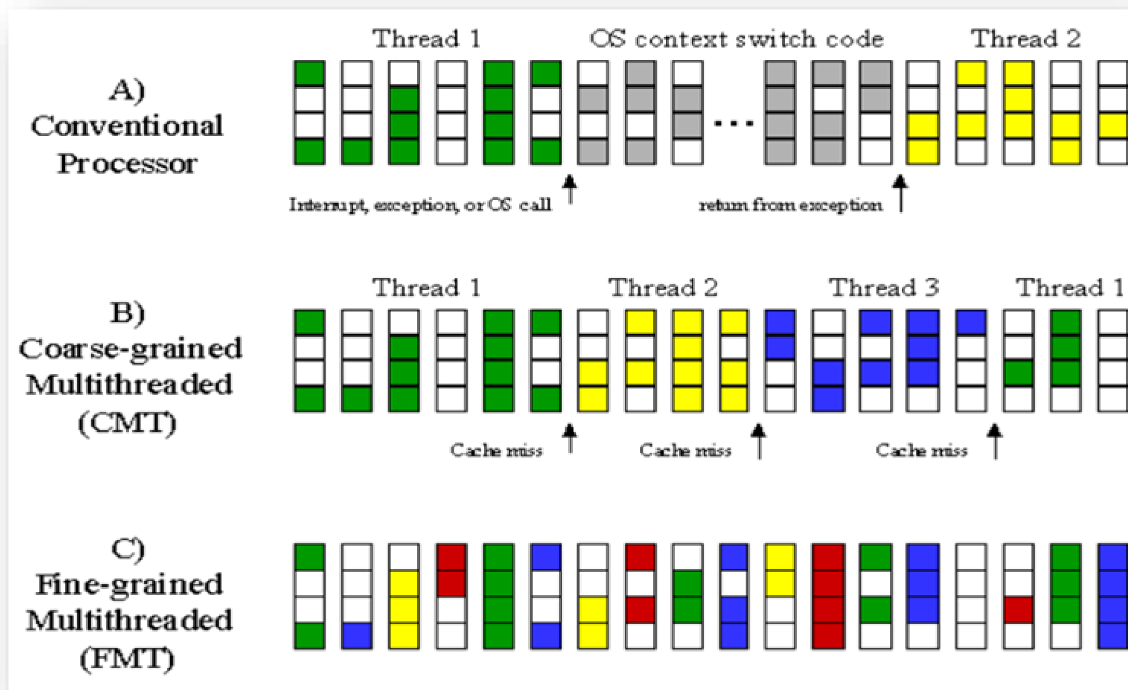






Multithreading (1/2)

- During a branch miss recovery and access to the main memory by a cache miss, ALUs have no jobs to do and have to be idle.
  - interrupt, exception, or OS call
- Executing multiple independent threads (programs) will mitigate the overhead.
- They are called coarse-grained and fine-grained multithreaded processors having multiple architecture states.



Execution Units
Time

http://www.realworldtech.com/alpha-ev8-smt/

Multithreading (2/2)

- Simultaneous Multithreading (SMT) can improve hardware resource usage.

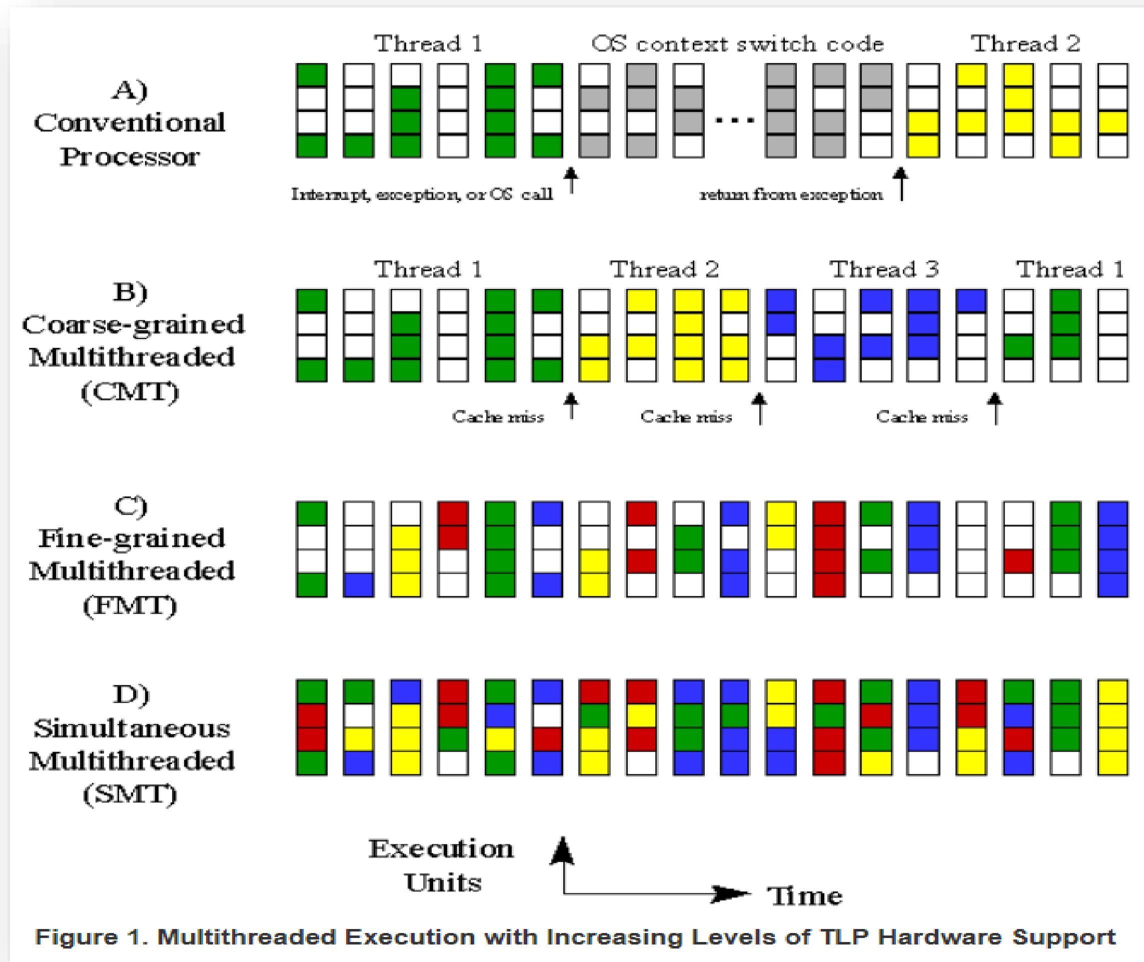
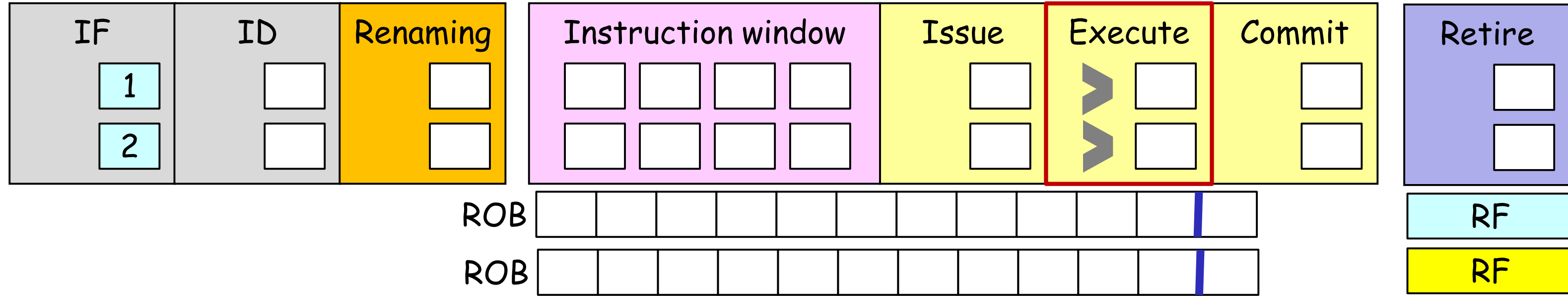


Figure 1. Multithreaded Execution with Increasing Levels of TLP Hardware Support

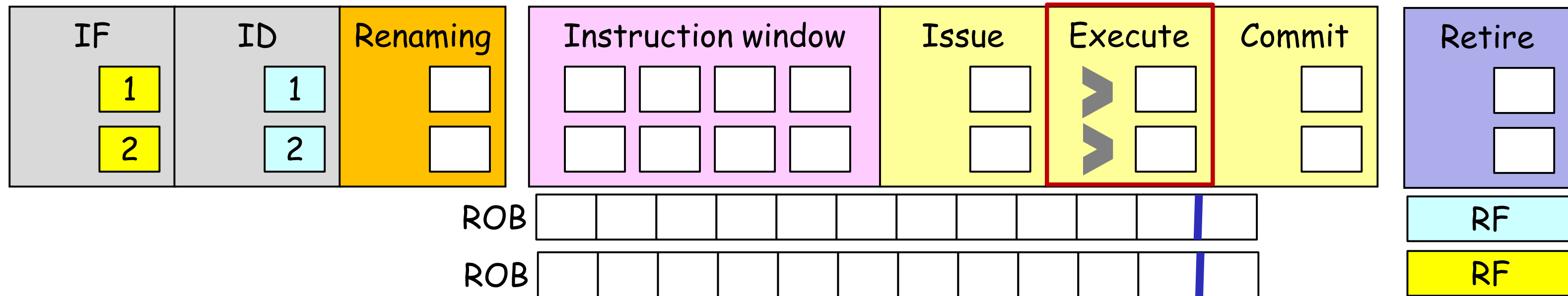
http://www.realworldtech.com/alpha-ev8-smt/

Simultaneous multithreading (SMT)

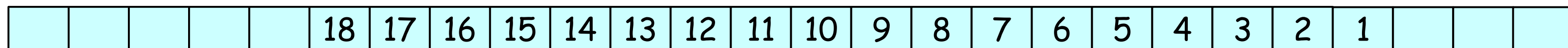
Cycle 1



Cycle 2

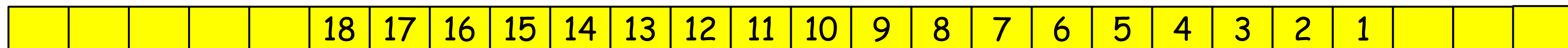


Instructions to be executed of program A



Newer instructions

Instructions to be executed of program B



Newer instructions

Simultaneous multithreading (SMT)

Cycle 3

IF: 3, 4 | ID: 1, 2 | Renaming: 1, 2 | Instruction window | Issue | Execute | Commit | Retire

ROB

ROB

RF

RF

Cycle 4

IF: 3, 4 | ID: 3, 4 | Renaming: 1, 2 | Instruction window: 1, 2 | Issue | Execute | Commit | Retire

ROB: 2, 1

ROB

RF

RF

Instructions to be executed of program A

					18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1			

Newer instructions

Instructions to be executed of program B

					18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1			

Newer instructions

Simultaneous multithreading (SMT)

Cycle 5

IF	ID	Renaming	Instruction window	Issue	Execute	Commit	Retire
5	3	3	2, 1	1			
6	4	4	2				

ROB: 2 1 — RF
ROB: 2 1 — RF

Cycle 6

IF	ID	Renaming	Instruction window	Issue	Execute	Commit	Retire
5	5	3	2, 3	1	1		
6	6	4	4	2			

ROB: 4 3 2 1 — RF
ROB: 2 1 — RF

Instructions to be executed of program A

					18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1			

Newer instructions

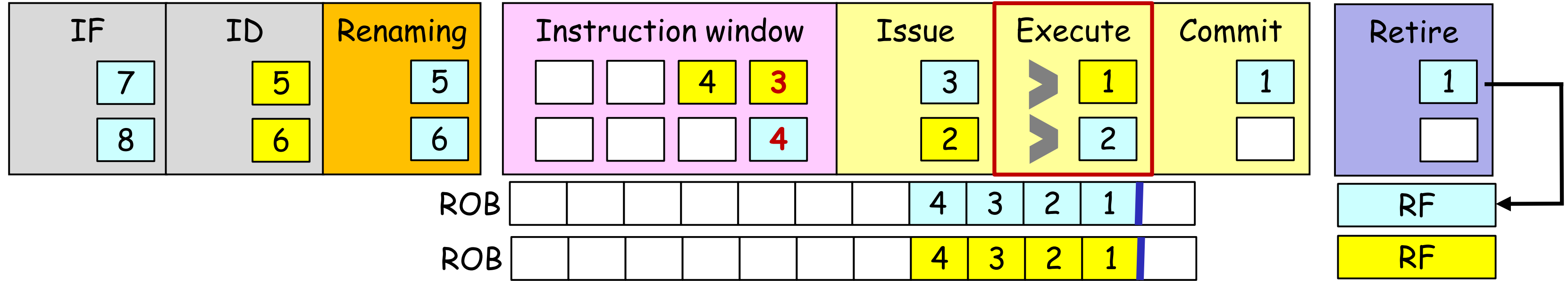
Instructions to be executed of program B

					18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1			

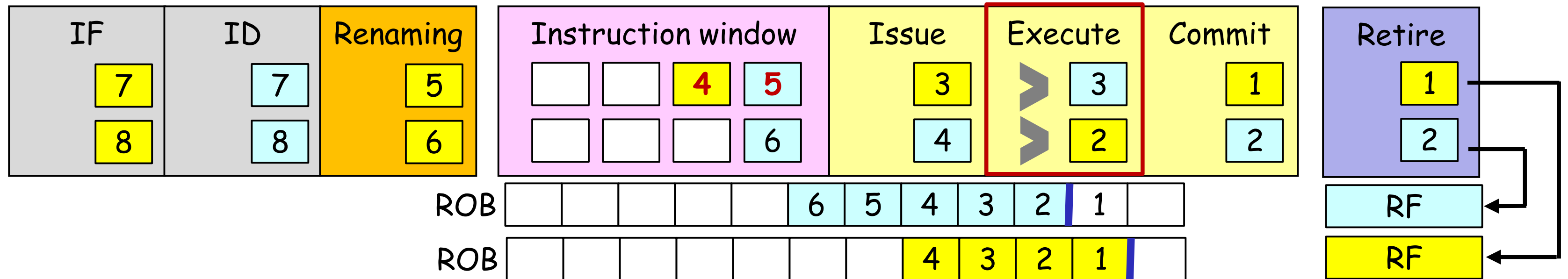
Newer instructions

Simultaneous multithreading (SMT)

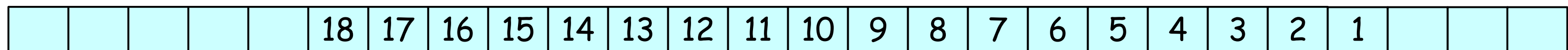
Cycle 7



Cycle 8



Instructions to be executed of program A



Newer instructions

Instructions to be executed of program B

Newer instructions

Datapath of SMT OoO execution processor

Instruction flow

Instruction cache

PC

Branch handler

Instruction fetch

Instruction decode

Renaming

Map table/free tag buffer

Register file

Dispatch

Memory dataflow

Integer

Floating-point

Memory

RS

Instruction window

ALU

ALU

Branch

FP ALU

Adr gen.

Adr gen.

Store queue

Reorder buffer (ROB)

Data cache

Register dataflow

Reservation station (RS)

From multi-core era to many-core era

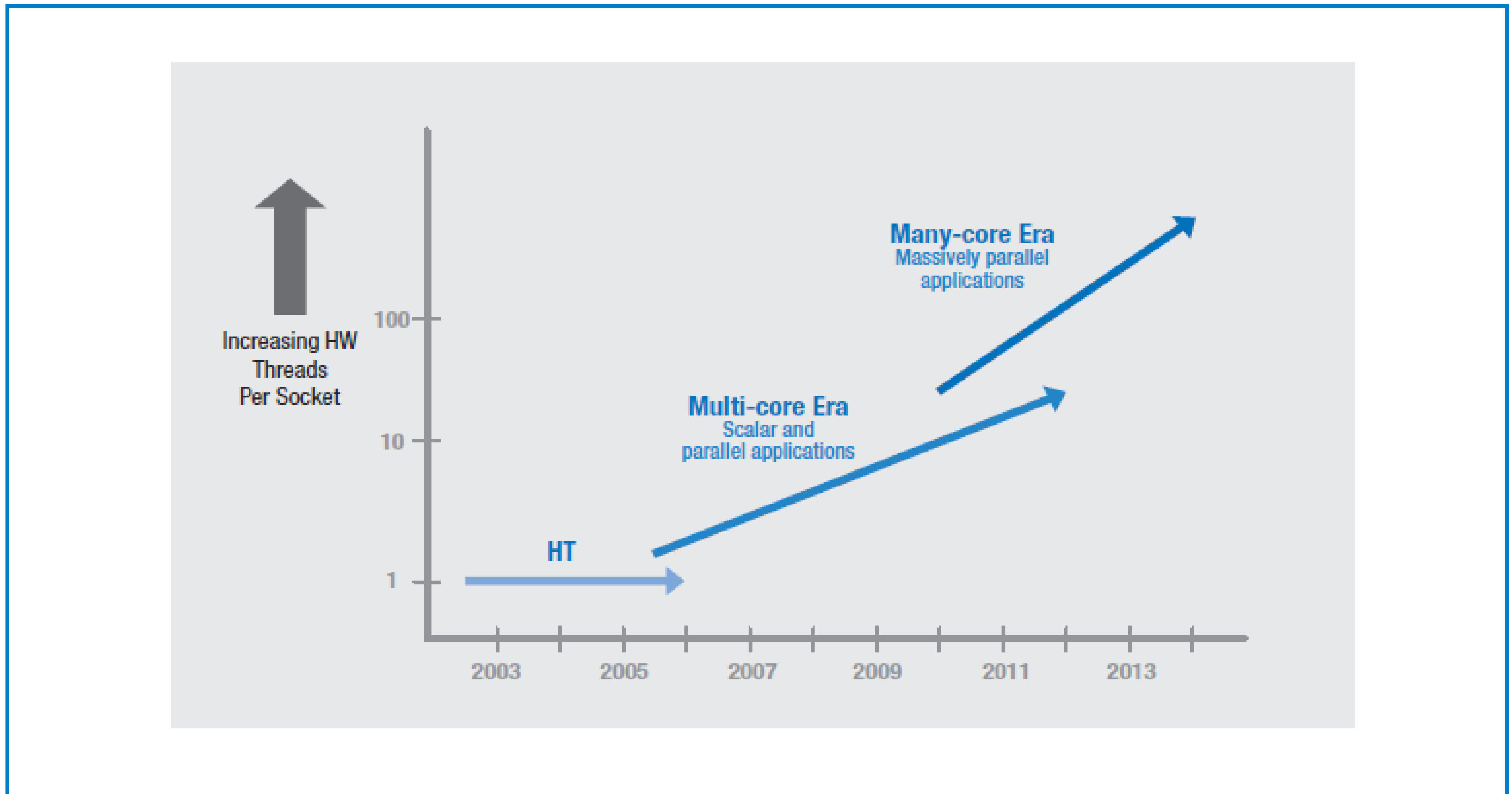
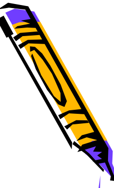


Figure 1: Current and expected eras of Intel® processor architectures

Platform 2015: Intel® Processor and Platform Evolution for the Next Decade, 2005

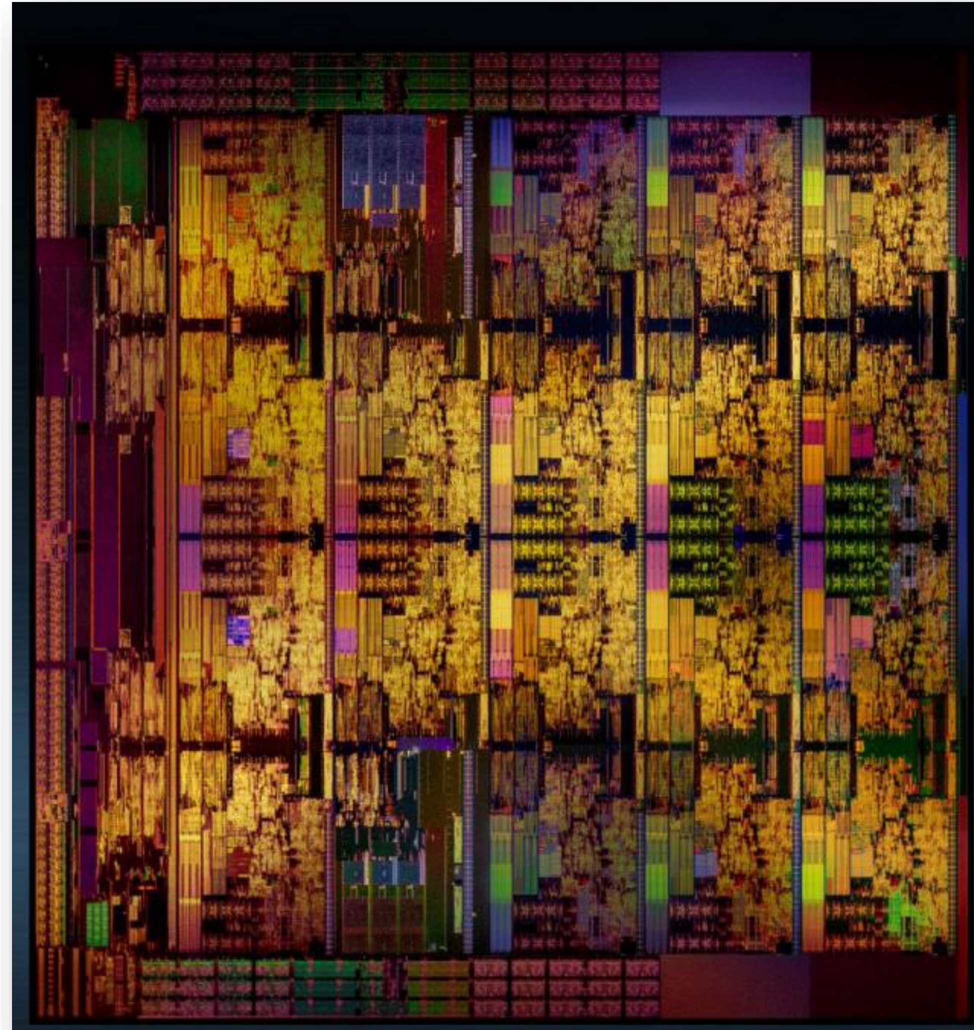
Intel Sandy Bridge, January 2011

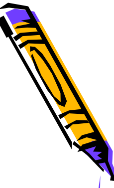


- 4 to 8 core

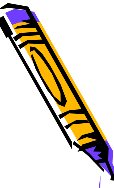
Intel Skylake-X, Core i9-7980XE, 2017

- 18 core





2021.11 Intel Alder Lake processor



Scalable Client Architecture

Desktop	Mobile	Ultra Mobile
LGA 1700 Socket	BGA Type3 50 x 25 x 1.3 mm	BGA Type4 HDI 28.5 x 19 x 1.1 mm

2022.11 AMD EPYC 9654 processor with 96 cores

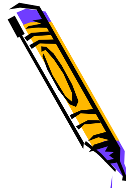
AMD EPYC™ 9004 Series Processor

All-in Feature Set support

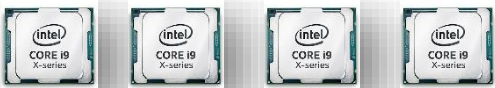
- 12 Channels of DDR5-4800
- Up to 6TB DDR5 memory capacity
- 128 lanes PCIe® 5
- 64 lanes CXL 1.1+
- AVX-512 ISA, SMT & core frequency boost
- AMD Infinity Fabric™
- AMD Infinity Guard

Cores	AMD EPYC	Base/Boost* (up to GHz)	Default TDP (w)	cTDP (w)
96 cores	9654/P	2.40/3.70	360w	320-400w
84 cores	9634	2.25/3.70	290w	240-300w
64 cores	9554/P	3.10/3.75	360w	320-400w
64 cores	9534	2.45/3.70	280w	240-300w
48 cores	9474F	3.60/4.10	360w	320-400w
	9454/P	2.75/3.80	290w	240-300w
32 cores	9374F	3.85/4.30	320w	320-400w
32 cores	9354/P	3.25/3.80	280w	240-300w
32 cores	9334	2.70/3.90	210w	200-240w
24 cores	9274F	4.05/4.30	320w	320-400w
	9254	2.90/4.15	200w	200-240w
	9224	2.50/3.70	200w	200-240w
16 cores	9174F	4.10/4.40	320w	320-400w
	9124	3.00/3.70	200w	200-240w

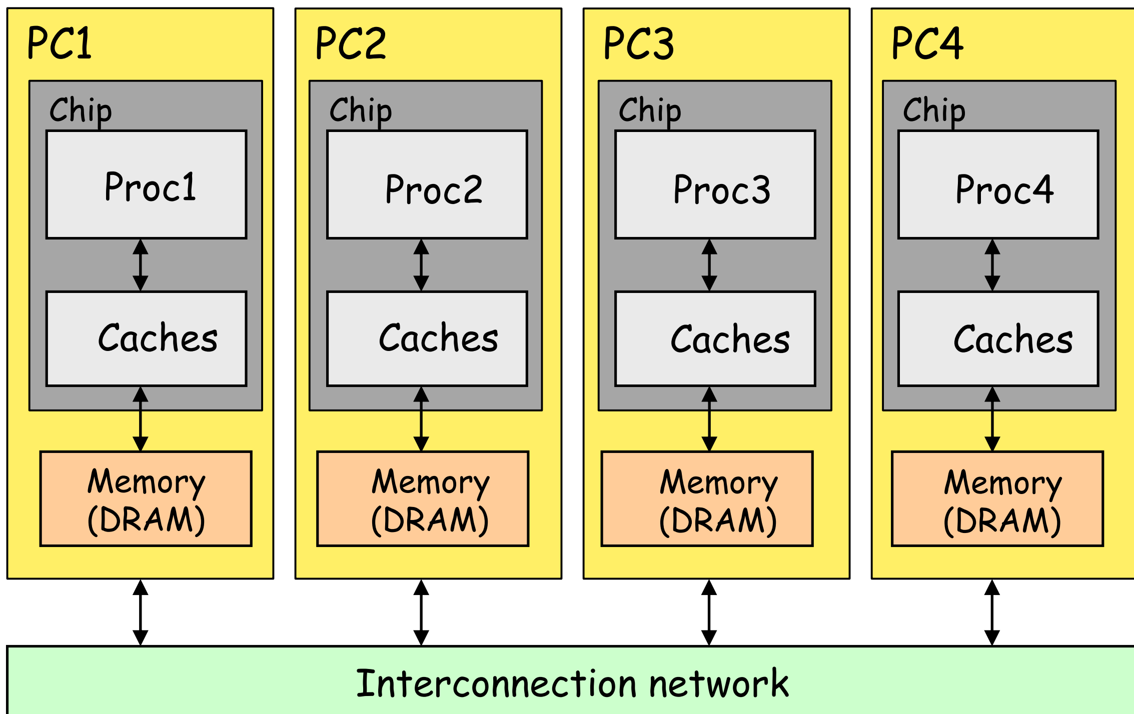
Distributed Memory Multi-Processor Architecture



- A PC cluster or parallel computers for higher performance
- Each memory module is associated with a processor
- Using explicit send and receive functions (message passing) to obtain the data required.
  - Who will send and receive data? How?

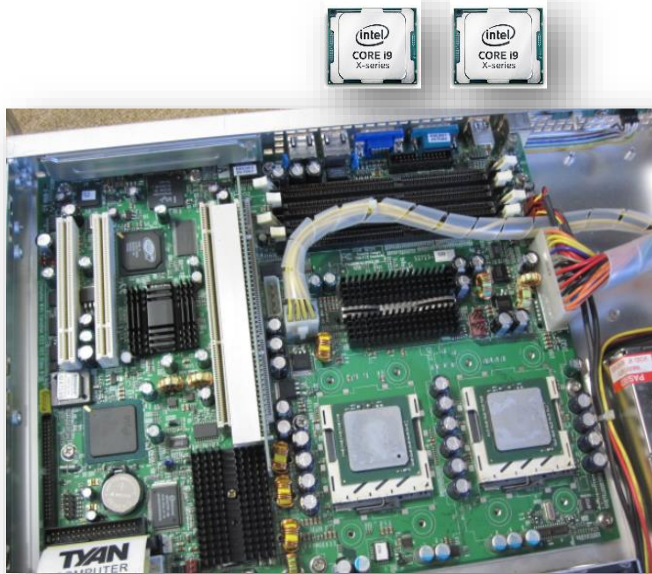


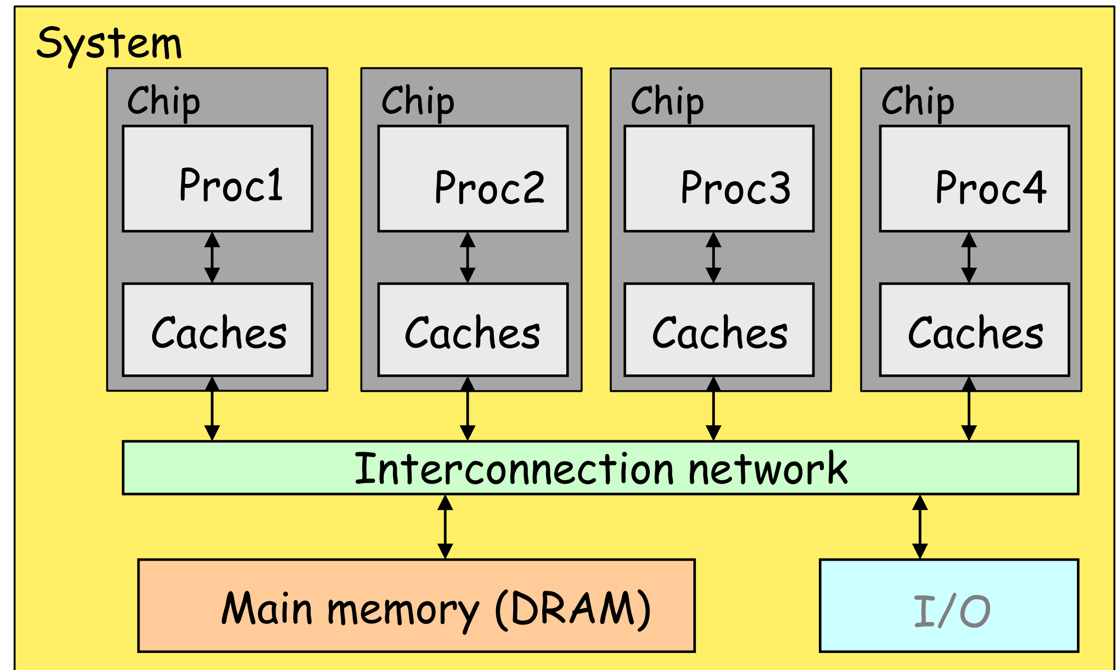
PC cluster



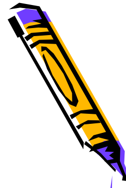
Shared Memory Multi-Processor Architecture

- All the processors can access the same address space of the main memory (shared memory) through an interconnection network.
- The shared memory or shared address space (SAS) is used as a means for communication between the processors.
  - What are the means to obtain the shared data?
  - What are the advantages and disadvantages of shared memory?

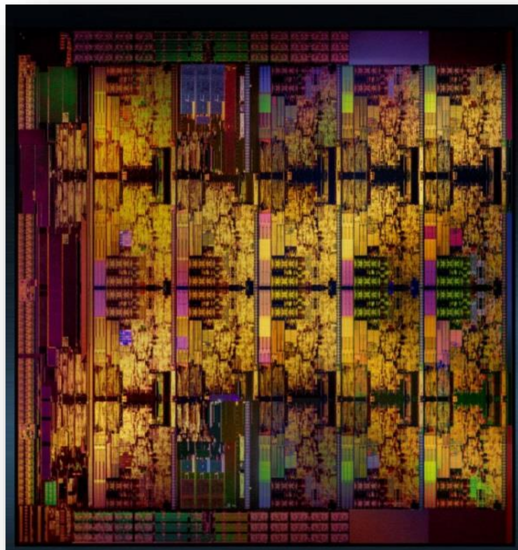




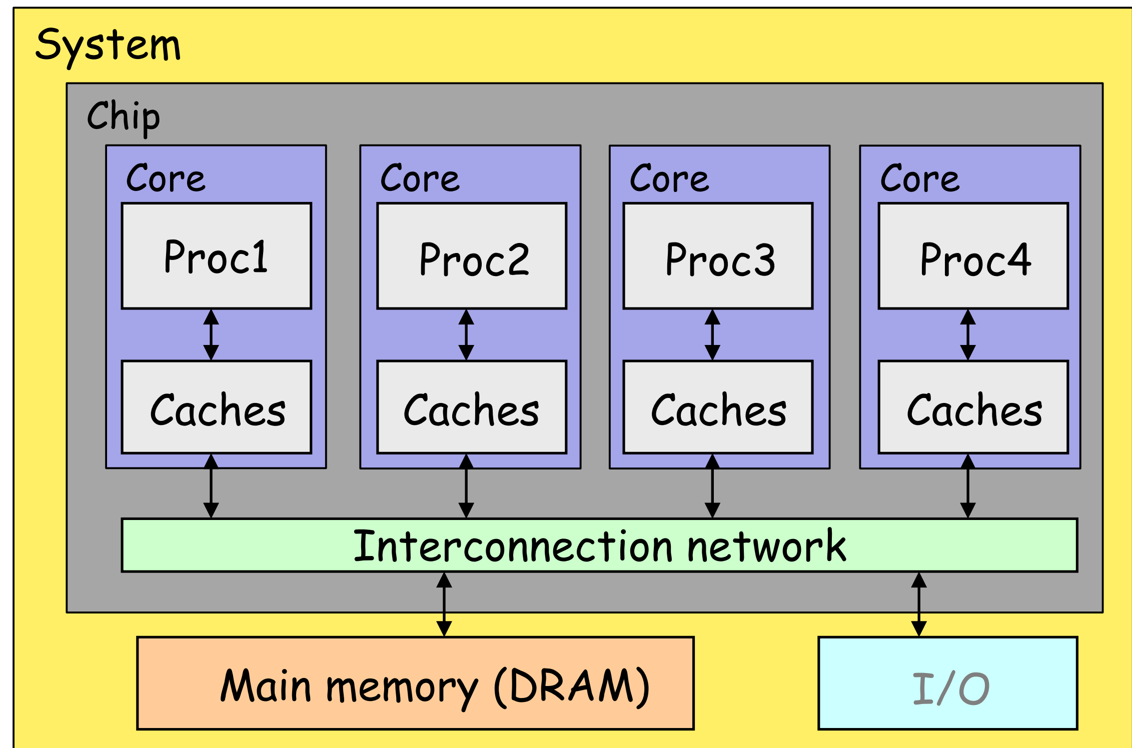
Shared memory many-core architecture

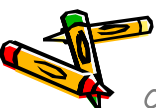


- The single-chip integrates many cores (conventional processors) and an interconnection network.
- The shared memory or shared address space (SAS) is used as a means for communication between the processors.



Intel Skylake-X, Core i9-7980XE, 2017





The free lunch is over

- Programmers have to worry much about performance and concurrency
- Parallel programming & multi-processor (multi-core) architecture

Free Lunch

Programmers haven't really had to worry much about performance or concurrency because of Moore's Law

Why we did not see 4GHz processors in Market?

The traditional approach to application performance was to simply wait for the next generation of processor; most software developers did not need to invest in performance tuning, and enjoyed a “free lunch” from hardware improvements.

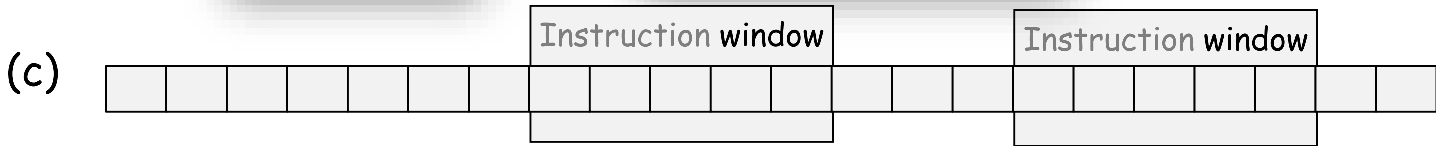
The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software by Herb Sutter, 2005

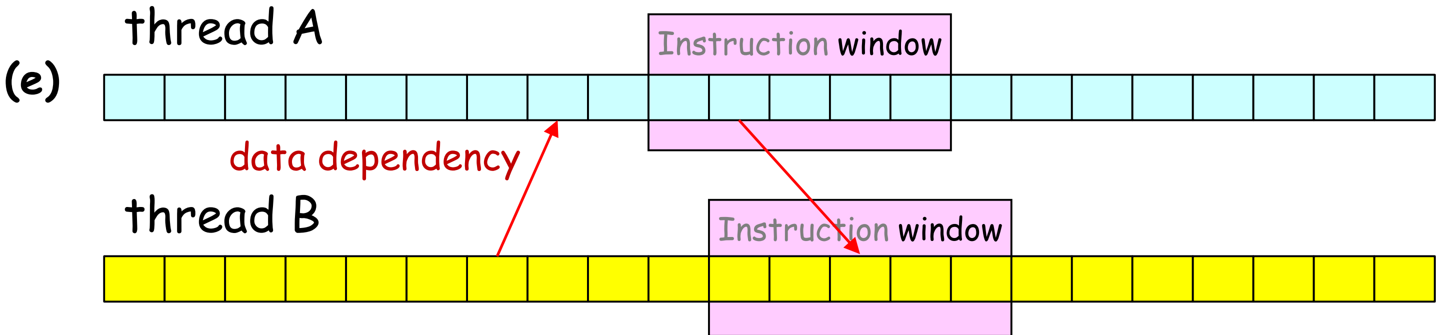
Parallel programming

- Several dependent threads run at the same time on a multi-processor (many-core) system.

Instruction window

	8	6	5
		4	7





Sample of a wrong parallel program using pthread

```
% gcc main1.c -O0 -lpthread -o a.out1
% ./a.out1
main: 20000000
```

Single Program Multiple Data (SPMD)

```c
#include <stdio.h>
#include <pthread.h>
#define N 10000000

int a = 0;

int func1(){
  int i;
  for(i=0; i<N; i++){ a++; }
};

int func2(){
  int i;
  for(i=0; i<N; i++){ a++; }
};

int main(){
  func1();
  func2();

  printf("main: %d¥n", a);
  return 0;
}
```

main1.c
sequential program

```c
#include <stdio.h>
#include <pthread.h>
#define N 10000000    // ten million

int a = 0;

int func1(){
  int i;
  for(i=0; i<N; i++){ a++; }
};

int func2(){
  int i;
  for(i=0; i<N; i++){ a++; }
};

int main(){
  pthread_t t1, t2;
  pthread_create(&t1, NULL, (void *)func1, NULL);
  pthread_create(&t2, NULL, (void *)func2, NULL);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);

  printf("main: %d¥n", a);
  return 0;
}
```

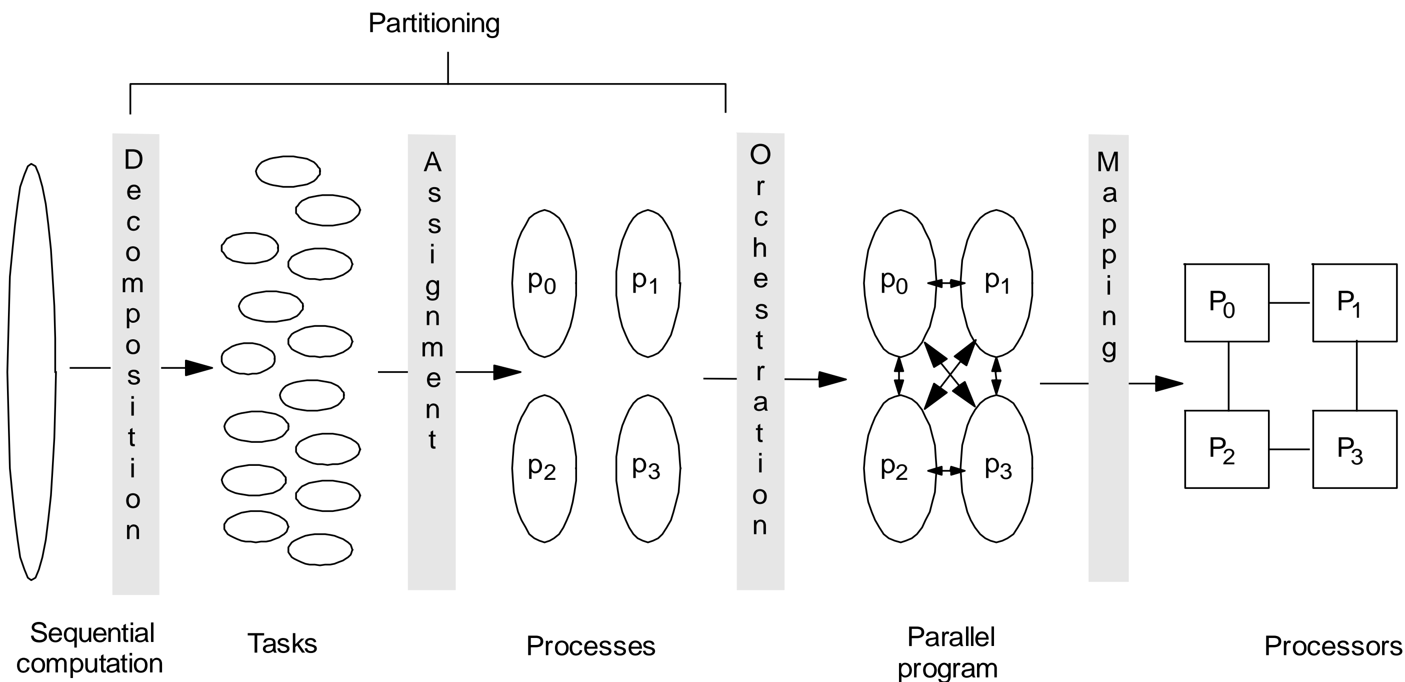
main2.c
parallel program with func1 and func2

```c
#include <stdio.h>
#include <pthread.h>
#define N 10000000    // ten million

int a = 0;

int func1(){
  int i;
  for(i=0; i<N; i++){ a++; }
};

int main(){
  pthread_t t1, t2;
  pthread_create(&t1, NULL, (void *)func1, NULL);
  pthread_create(&t2, NULL, (void *)func1, NULL);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);

  printf("main: %d¥n", a);
  return 0;
}
```

main3.c
parallel program with func1

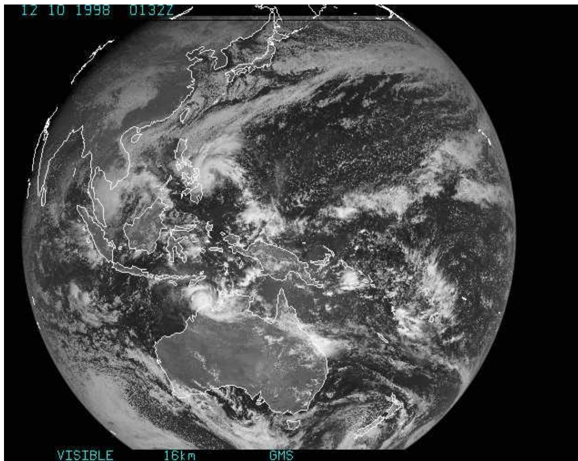
Four steps in creating a parallel program

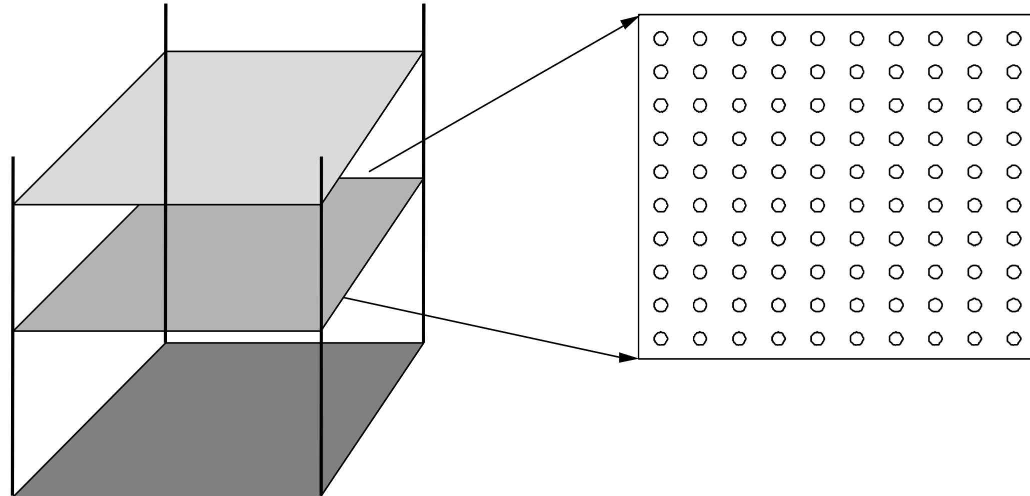
0. Preparing an optimized sequential program (baseline)
1. Decomposition of computation in tasks
2. Assignment of tasks to processes
3. Orchestration of data access, comm, synch.
4. Mapping processes to processors (cores)



Adapted from *Parallel Computer Architecture*, David E. Culler

Simulating ocean currents





(a) Cross sections

(b) Spatial discretization of a cross section

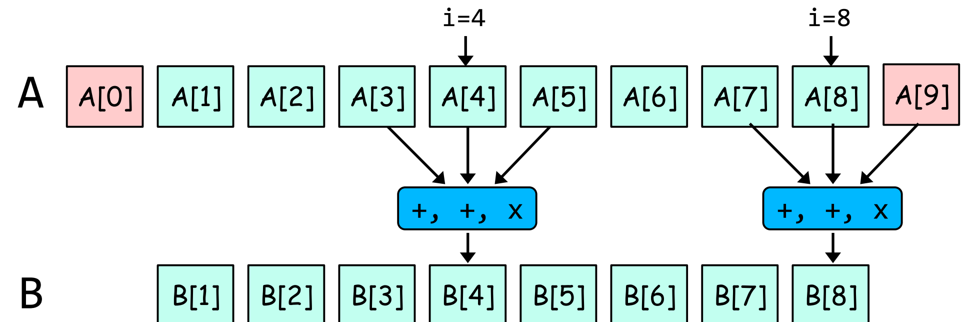
- Model as two-dimensional grids
  - Discretize in space and time
  - finer spatial and temporal resolution enables greater accuracy
- Many different computations per time step
  - Concurrency across and within grid computations
- We use one-dimensional grids for simplicity

Sequential version as the baseline

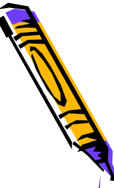
- A sequential program main01.c and the execution result
- Computations in blue color are fully parallel

```c
#define N 8       /* the number of grids */
#define TOL 15.0 /* tolerance parameter */
float A[N+2], B[N+2];

void solve () {
    int i, done = 0;
    while (!done) {
        float diff = 0;
        for (i=1; i<=N; i++) {
            B[i] = 0.333 * (A[i-1] + A[i] + A[i+1]);
            diff = diff + fabsf(B[i] - A[i]);
        }
        if (diff <TOL) done = 1;
        for (i=1; i<=N; i++) A[i] = B[i];

        for (i=0; i<=N+1; i++) printf("%6.2f ", B[i]);
        printf("| diff=%6.2f¥n", diff); /* for debug */
    }
}

int main() {
    int i;
    for (i=1; i<N-1; i++) A[i] = 100+i*i;
    solve();
}
```

```
  0.00  68.26 104.56 109.56 116.55 125.54  86.91  45.29   0.00   0.00 | diff=129.32
  0.00  57.55  94.03 110.11 117.10 109.56  85.83  44.02  15.08   0.00 | diff= 55.76
  0.00  50.48  87.15 106.97 112.14 104.06  79.72  48.26  19.68   0.00 | diff= 42.50
  0.00  45.83  81.45 101.99 107.62  98.54  77.27  49.17  22.63   0.00 | diff= 31.68
  0.00  42.38  76.35  96.92 102.61  94.38  74.92  49.64  23.91   0.00 | diff= 26.88
  0.00  39.54  71.81  91.87  97.87  90.55  72.91  49.44  24.49   0.00 | diff= 23.80
  0.00  37.08  67.67  87.10  93.34  87.02  70.89  48.90  24.62   0.00 | diff= 22.12
  0.00  34.88  63.89  82.62  89.06  83.67  68.87  48.09  24.48   0.00 | diff= 21.06
  0.00  32.89  60.40  78.44  85.03  80.45  66.81  47.10  24.17   0.00 | diff= 20.26
  0.00  31.07  57.19  74.55  81.23  77.35  64.72  45.98  23.73   0.00 | diff= 19.47
  0.00  29.39  54.21  70.92  77.63  74.36  62.62  44.77  23.21   0.00 | diff= 18.70
  0.00  27.84  51.46  67.52  74.23  71.47  60.52  43.49  22.64   0.00 | diff= 17.95
  0.00  26.41  48.89  64.34  71.00  68.67  58.43  42.17  22.02   0.00 | diff= 17.23
  0.00  25.07  46.50  61.35  67.94  65.97  56.37  40.84  21.38   0.00 | diff= 16.53
  0.00  23.83  44.26  58.54  65.02  63.36  54.34  39.49  20.72   0.00 | diff= 15.85
  0.00  22.68  42.17  55.88  62.24  60.85  52.34  38.14  20.05   0.00 | diff= 15.20
  0.00  21.59  40.20  53.38  59.60  58.42  50.39  36.81  19.38   0.00 | diff= 14.58
```



Decomposition and assignment



- Single Program Multiple Data (SPMD)
  - Decomposition: there are eight tasks to compute B[i]
  - Assignment: the first four tasks for core 1, and the last four tasks for core 2

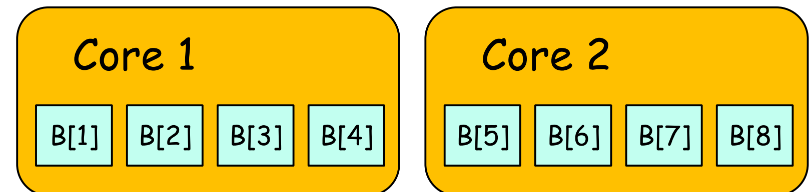
```
float A[N+2], B[N+2]; /* these are in shared memory */
float diff=0;         /* variable  in shared memory */

void solve_pp (int pid, int ncores) {
    int i, done = 0;                    /* private variables */
    int mymin = 1 + (pid * N/ncores);   /* private variable  */
    int mymax = mymin + N/ncores – 1;   /* private variable  */
    while (!done) {
        float mydiff = 0;
        for (i=mymin; i<=mymax; i++) {
            B[i] = 0.333 * (A[i-1] + A[i] + A[i+1]);
            mydiff = mydiff + fabsf(B[i] - A[i]);
        }
        diff = diff + mydiff;

        if (diff <TOL) done = 1;
        if (pid==1) diff = 0;
        for (i=mymin; i<=mymax; i++) A[i] = B[i];
    }
}

int main() { /* solve this using two cores */
    initialize shared data A and B;
    create thread1 and call solve_pp(1, 2);
    create thread2 and call solve_pp(2, 2);
}
```

Computation

Decomposition

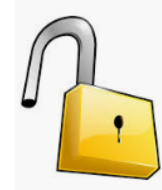
B[1] B[2] B[3] B[4] B[5] B[6] B[7] B[8]

Assignment



Orchestration

- LOCK and UNLOCK around critical section
  - Lock provides exclusive access to the locked data.
  - Set of operations we want to execute atomically
- BARRIER ensures all reach here



```
float A[N+2], B[N+2]; /* these are in shared memory */
float diff=0.0;       /* variable  in shared memory */

void solve_pp (int pid, int ncores) {
    int i, done = 0;                      /* private variables */
    int mymin = 1 + (pid * N/ncores);    /* private variable  */
    int mymax = mymin + N/ncores – 1;    /* private variable  */
    while (!done) {
        float mydiff = 0;
        for (i=mymin; i<=mymax; i++) {
            B[i] = 0.333 * (A[i-1] + A[i] + A[i+1]);
            mydiff = mydiff + fabsf(B[i] - A[i]);
        }
        LOCK();
        diff = diff + mydiff;
        UNLOCK();

        BARRIER();
        if (diff <TOL) done = 1;
        BARRIER();
        if (pid==1) diff = 0;
        for (i=mymin; i<=mymax; i++) A[i] = B[i];
        BARRIER();
    }
}
```

These operations must be executed atomically

```
(1) load diff
(2) add
(3) store diff
```

After all cores update the diff, *if statement* must be executed.

```
if (diff <TOL) done = 1;
```

Key components of many-core processors

- Interconnection network
  - connecting many modules on a chip achieving high throughput and low latency
- Main memory and caches
  - Caches are used to reduce latency and to lower network traffic
  - A parallel program has private data and shared data
  - New issues are cache coherence and memory consistency
- Core
  - High-performance superscalar processor providing a hardware mechanism to support thread synchronization