Fiscal Year 2022

Ver. 2023-01-23a

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

Advanced Computer Architecture

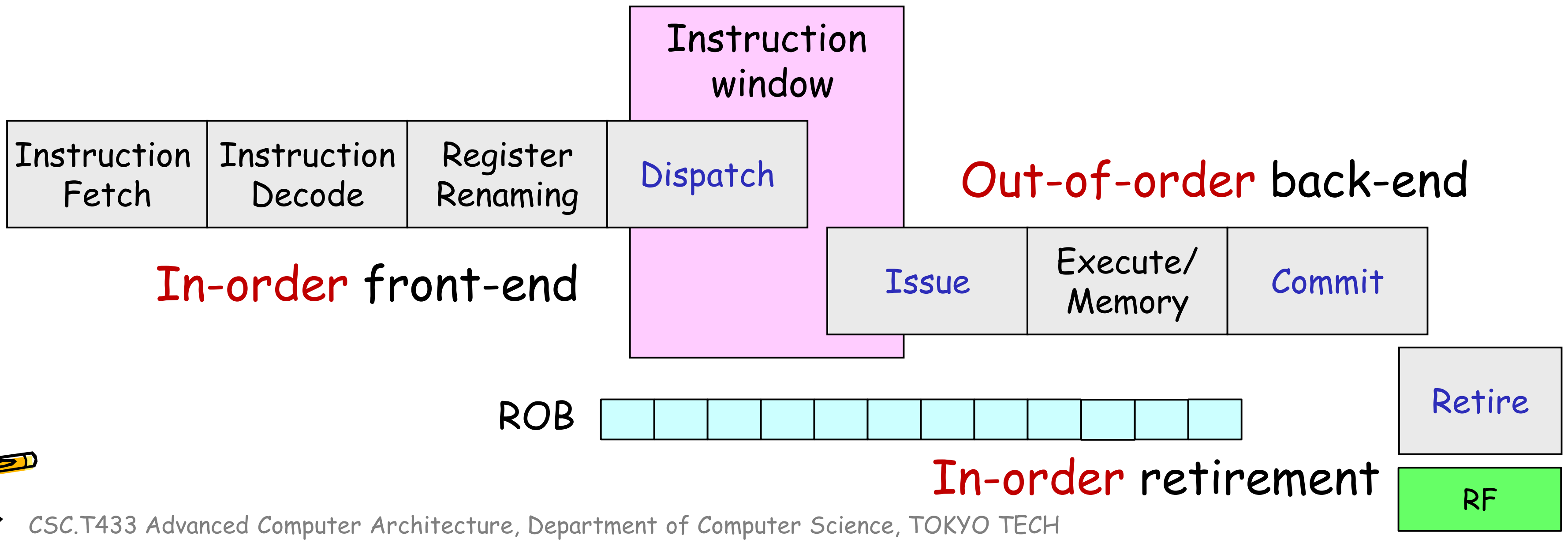
10. Multi-Processor: Distributed Memory and Shared Memory Architecture

www.arch.cs.titech.ac.jp/lecture/ACA/
Room No.W831, HyFlex
Mon 13:45-15:25, Thr 13:45-15:25

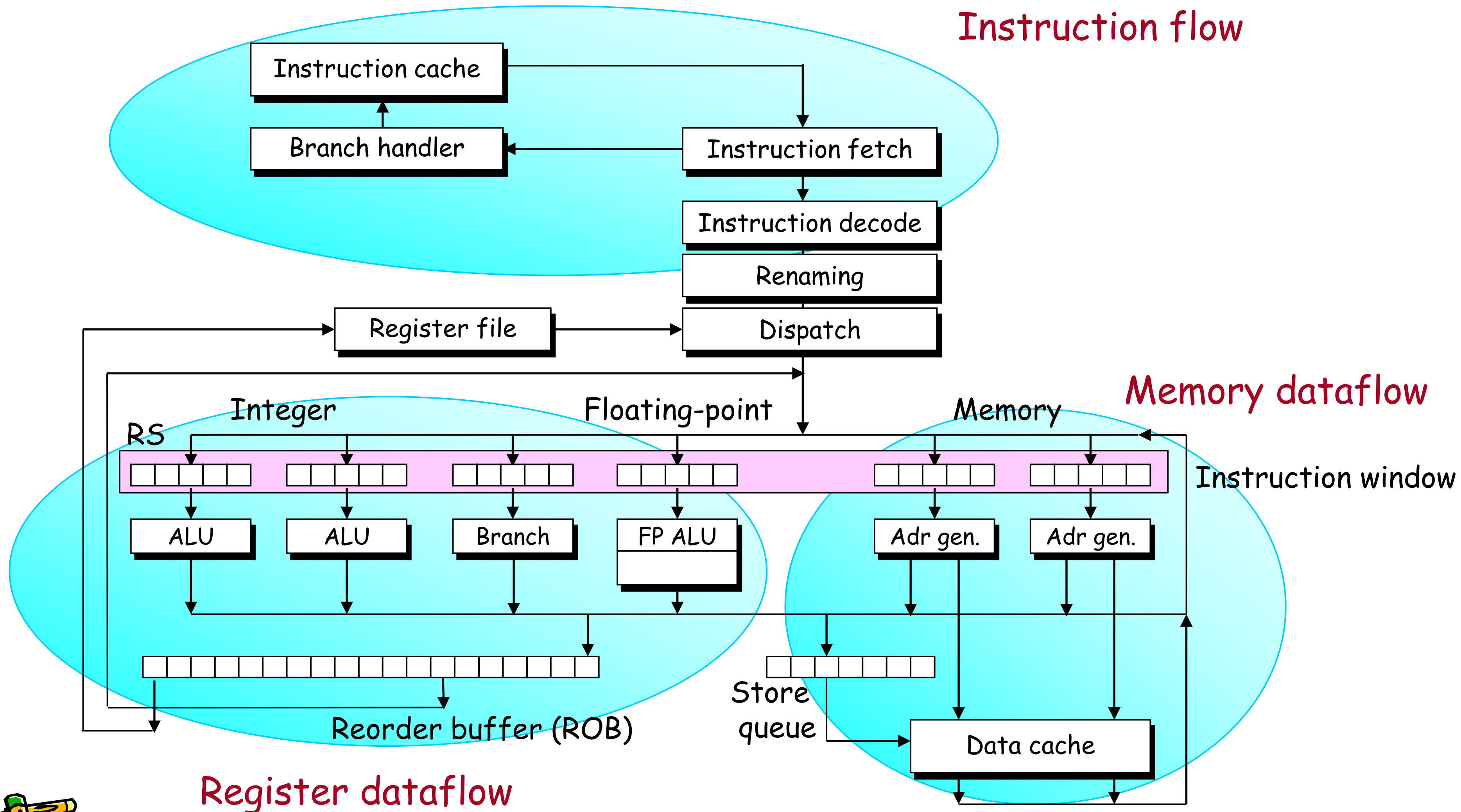
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp

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 $0 to $31.



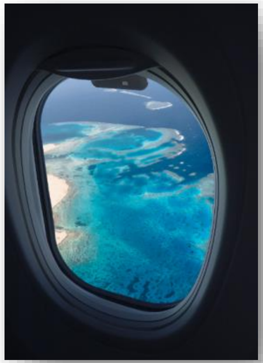
Datapath of OoO execution processor

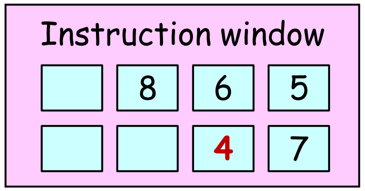


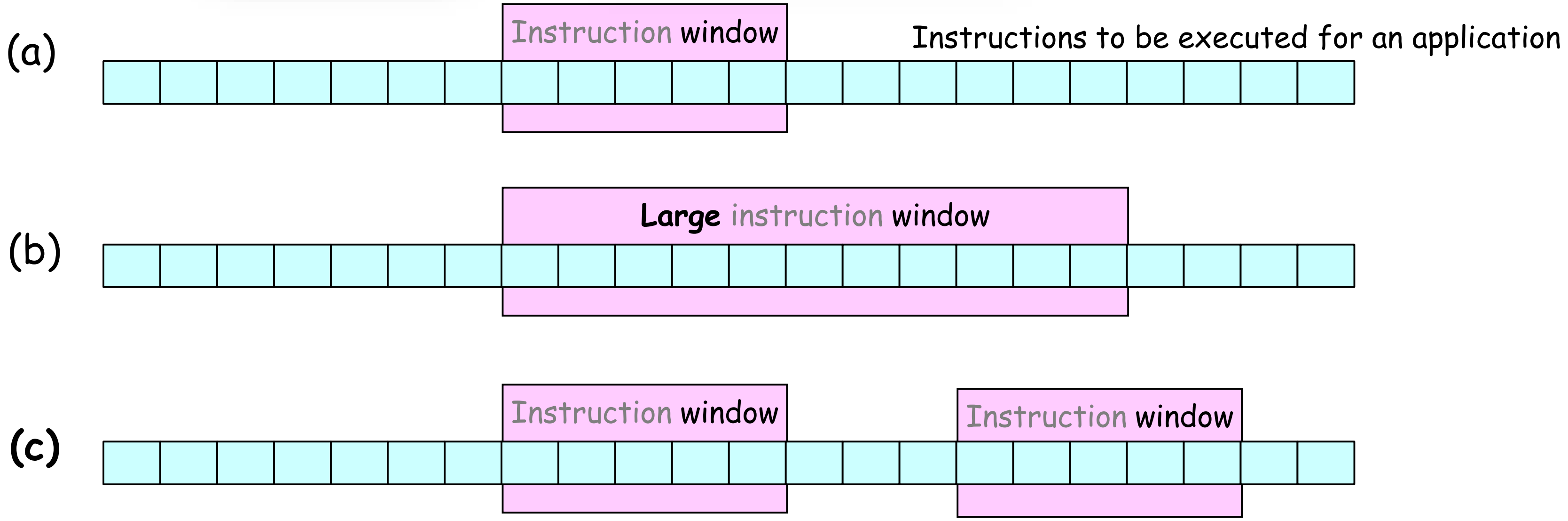
Reservation station (RS)

Aside: What is a window?

- A window is a space in the wall of a building or in the side of a vehicle, which has glass in it so that light can come in and you can see out. (Collins)

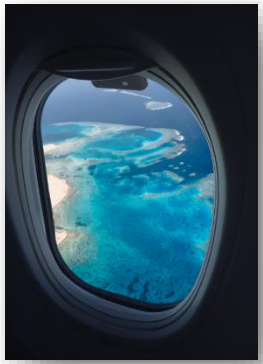


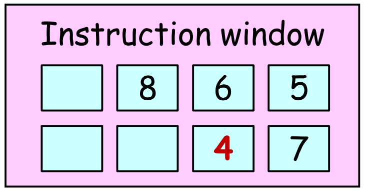


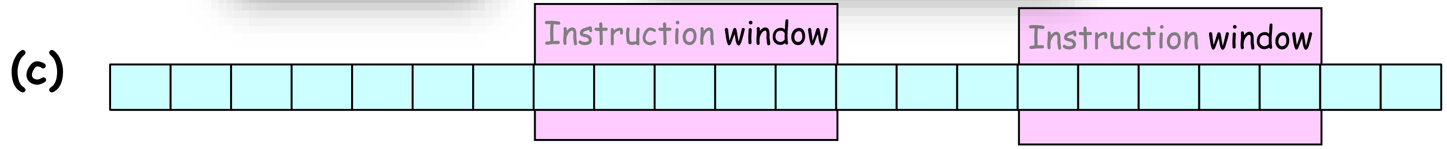


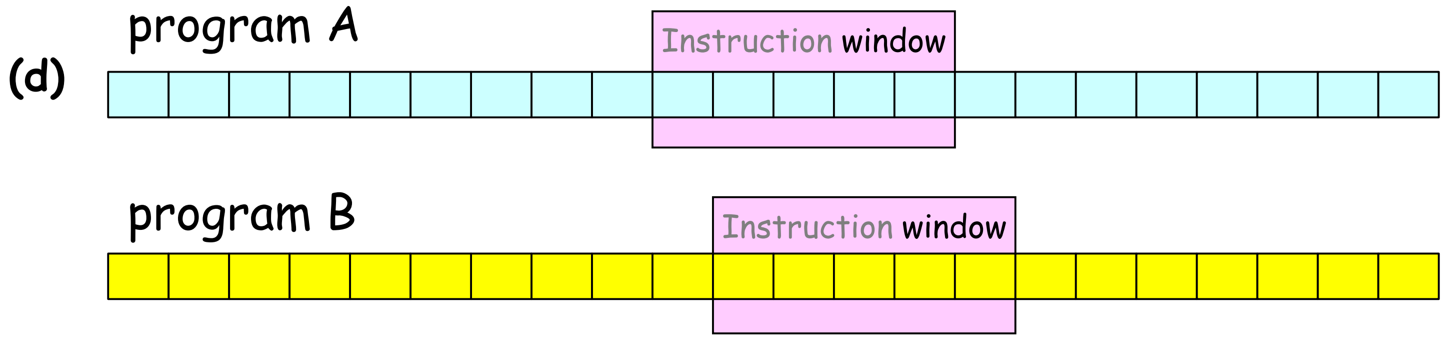
Multiprogramming

- Several independent programs run at the same time.



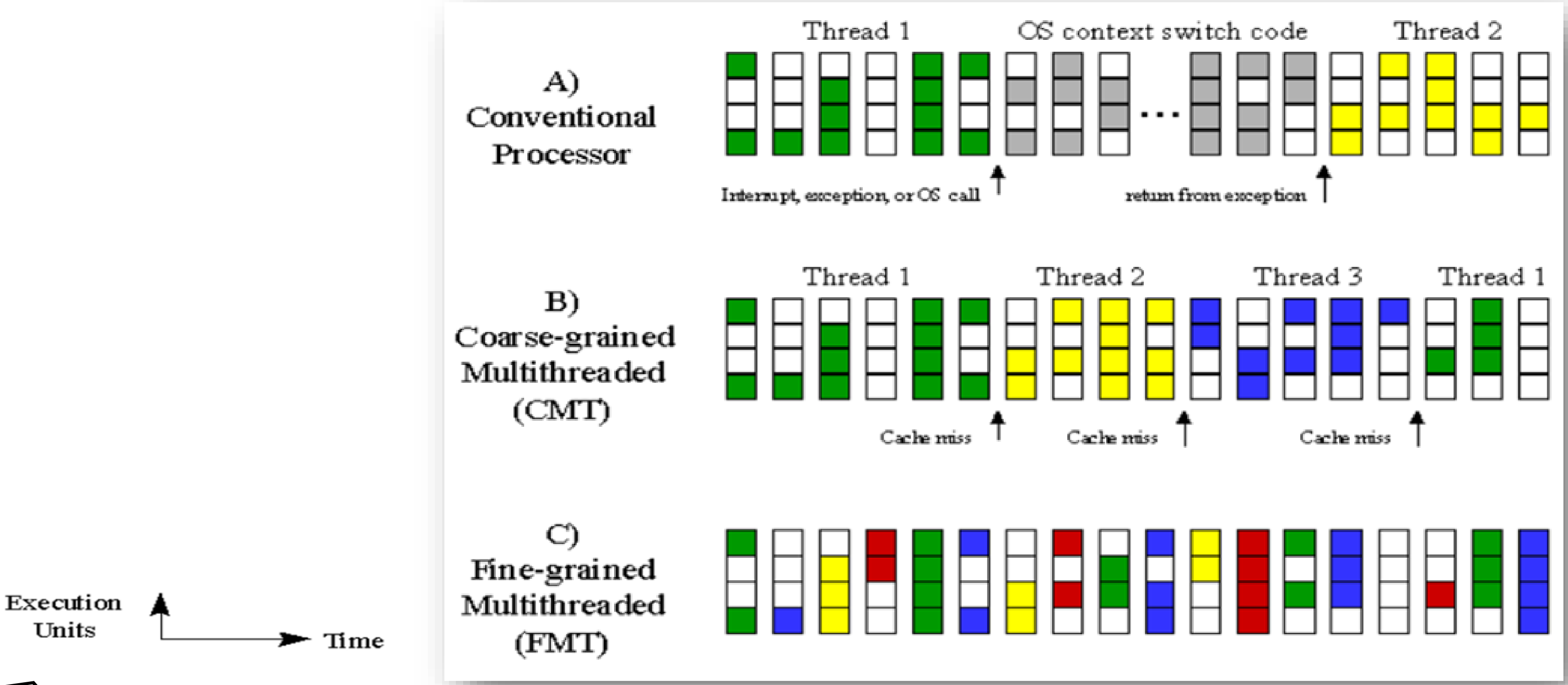






Multithreading (1/2)

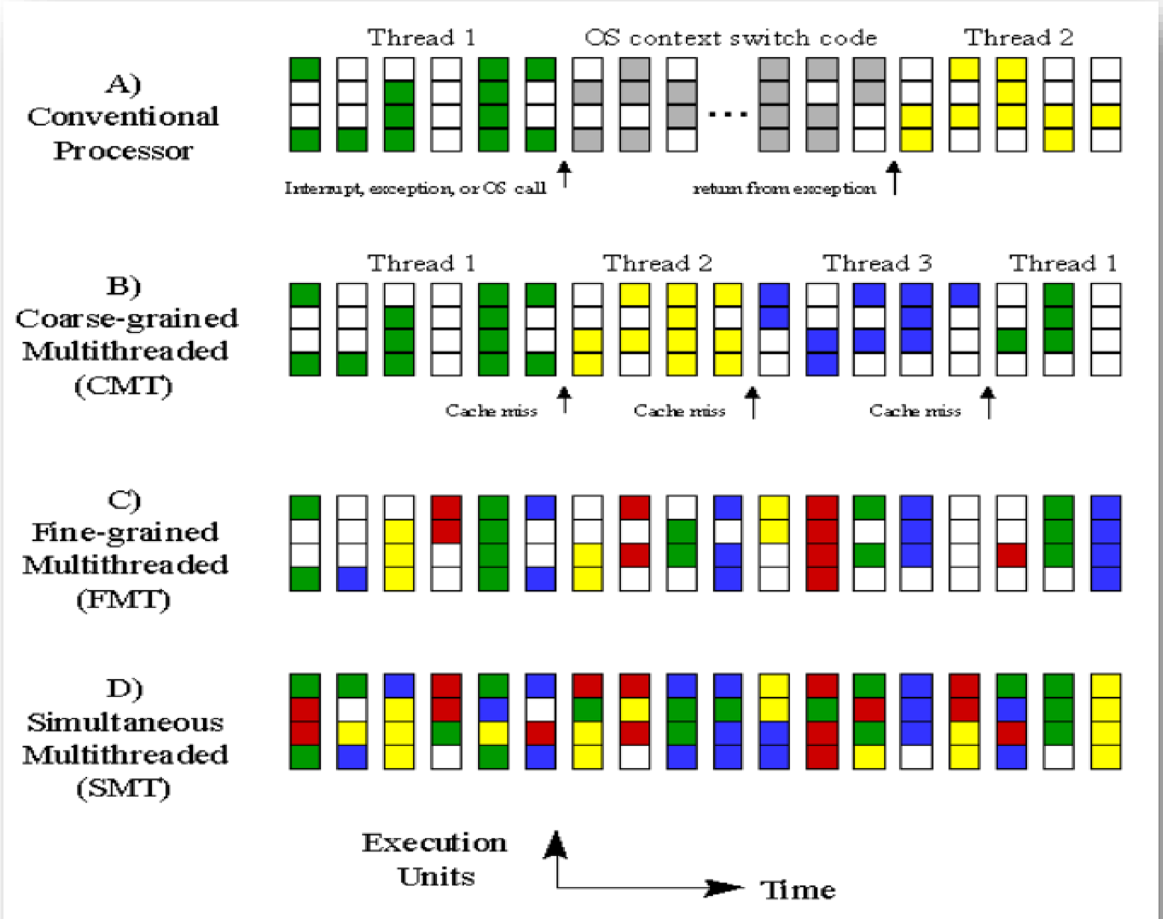
- During a branch miss recovery and access to the main memory by a cache miss, ALUs have no jobs to do and have to be idle.
- Executing multiple independent threads (programs) will mitigate the overhead.
- They are called coarse-grained and fine-grained multithreaded processors having multiple architecture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



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

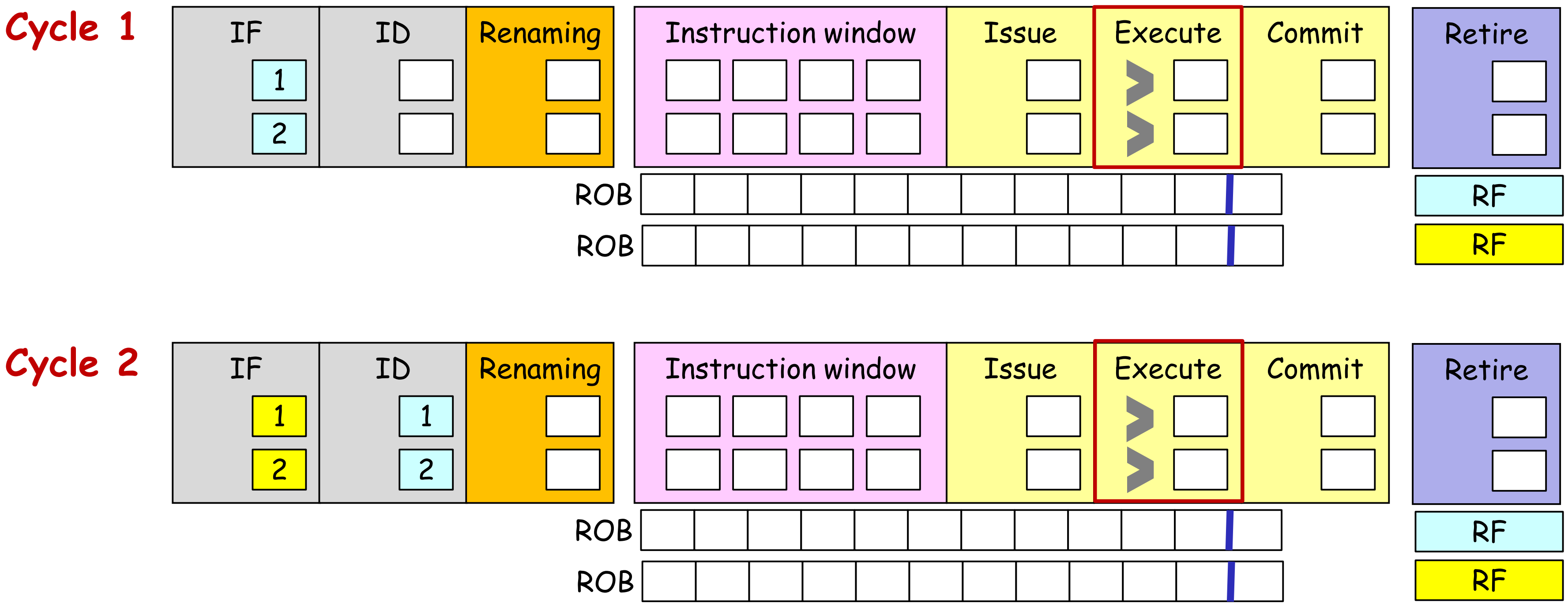
Multithreading (2/2)

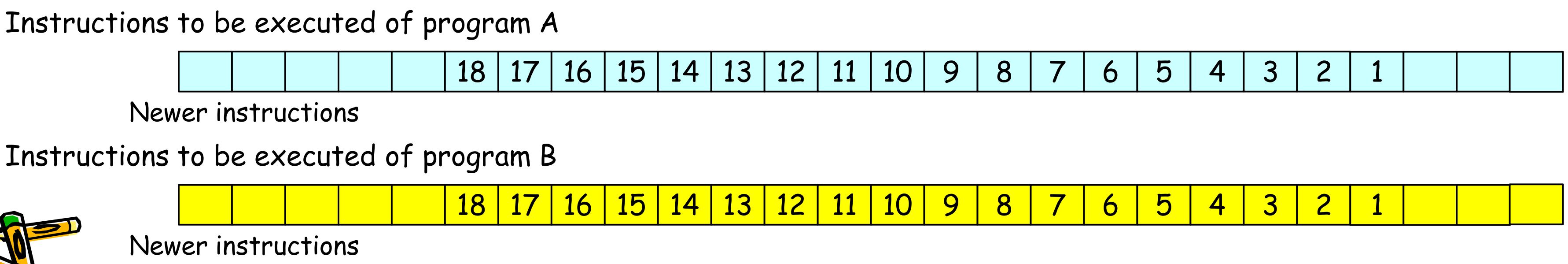
- Simultaneous Multithreading (SMT) can improve hardware resource usage.


A) Conventional Processor
Thread 1
OS context switch code
Thread 2
Interrupt, exception, or OS call
return from exception
B) Coarse-grained Multithreaded (CMT)
Thread 1
Thread 2
Thread 3
Thread 1
Cache miss
Cache miss
Cache miss
C) Fine-grained Multithreaded (FMT)
D) Simultaneous Multithreaded (SMT)
Execution Units
Time
Figure 1. Multithreaded Execution with Increasing Levels of TLP Hardware Support

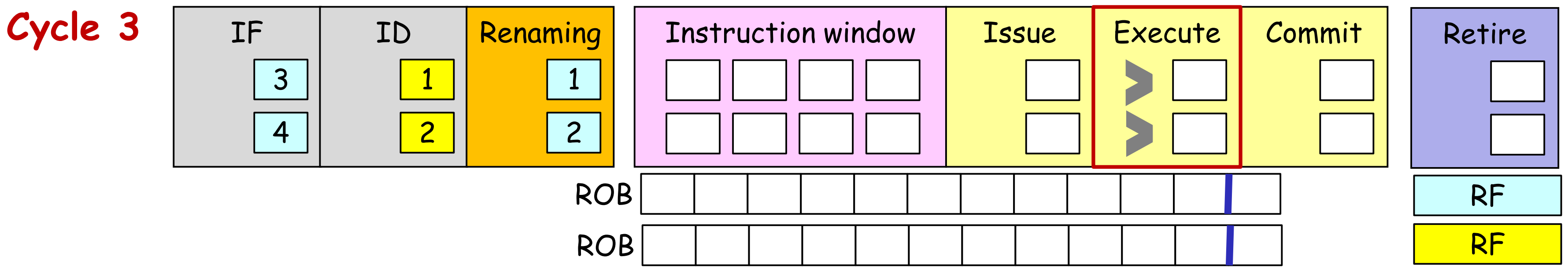
http://www.realworldtech.com/alpha-ev8-smt/

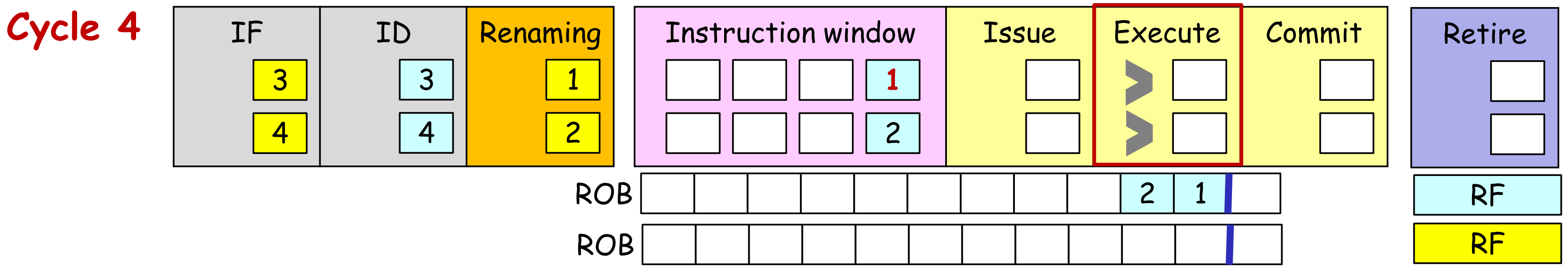
Simultaneous multithreading (SMT)





Simultaneous multithreading (SMT)





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			

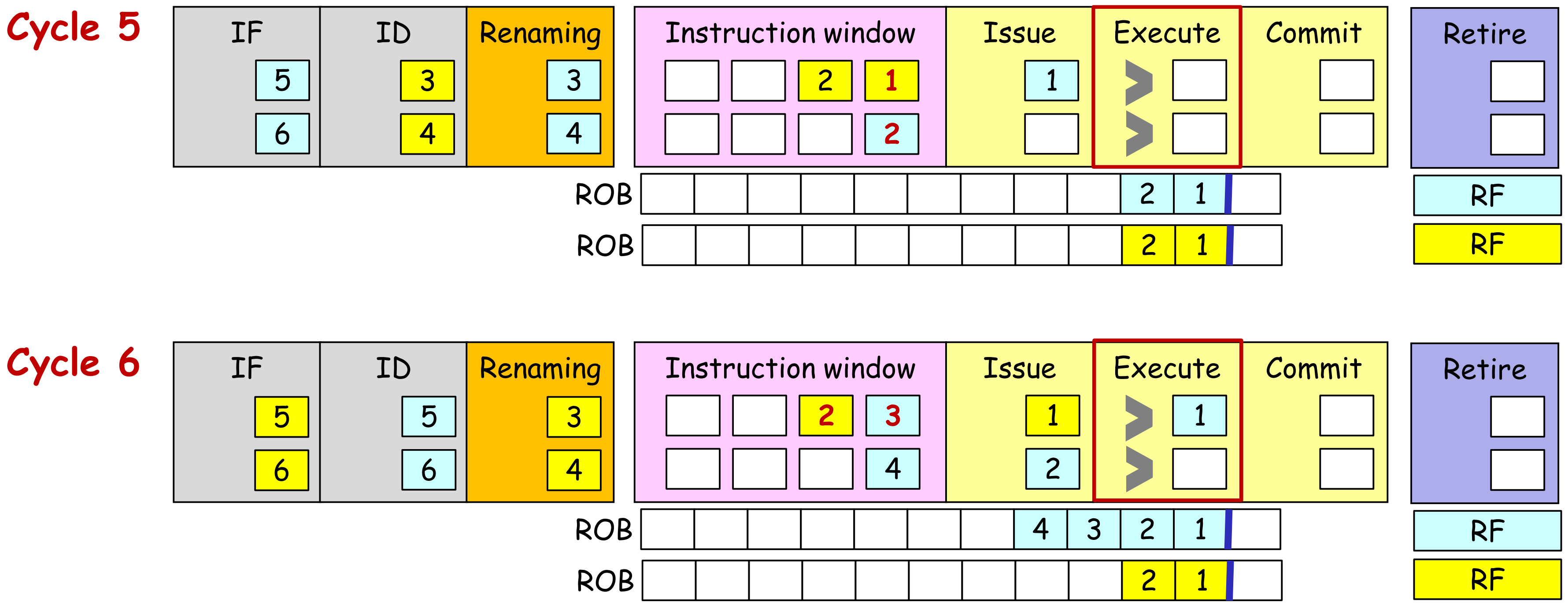
Newer instructions

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)



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			

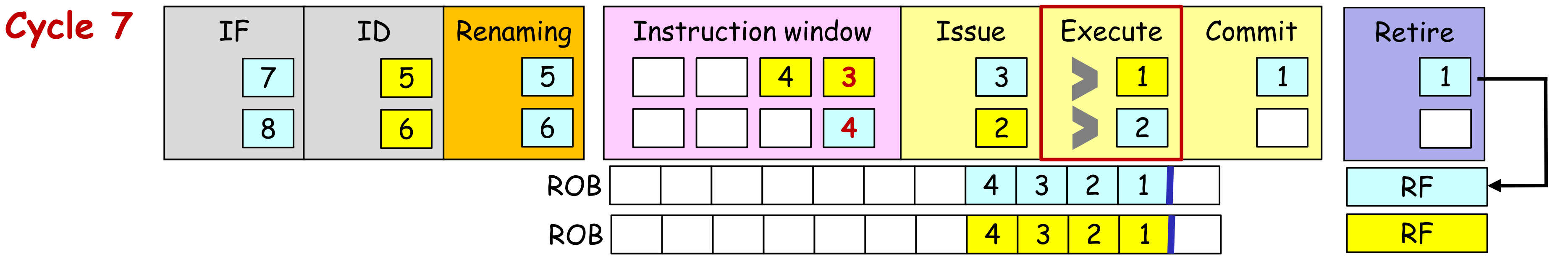
Newer instructions

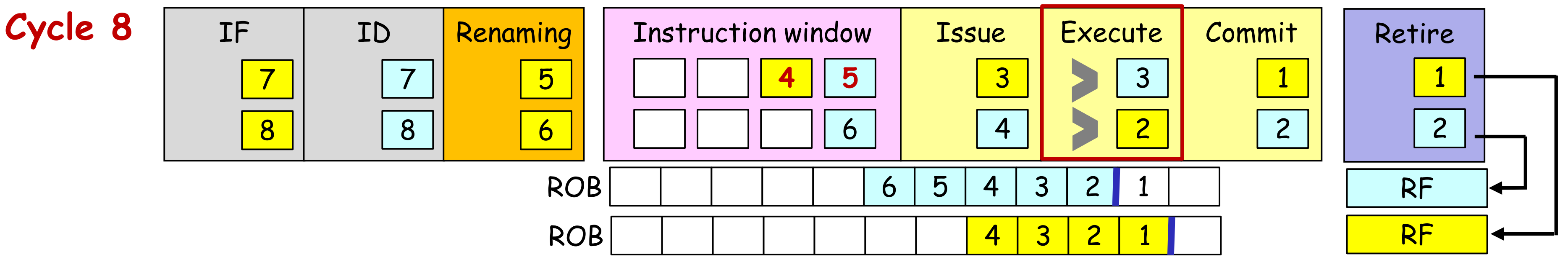
Instructions to be executed of program B

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

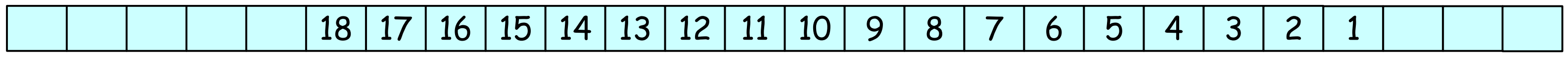
Newer instructions

Simultaneous multithreading (SMT)



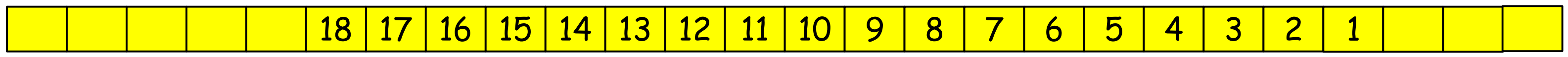


Instructions to be executed of program A



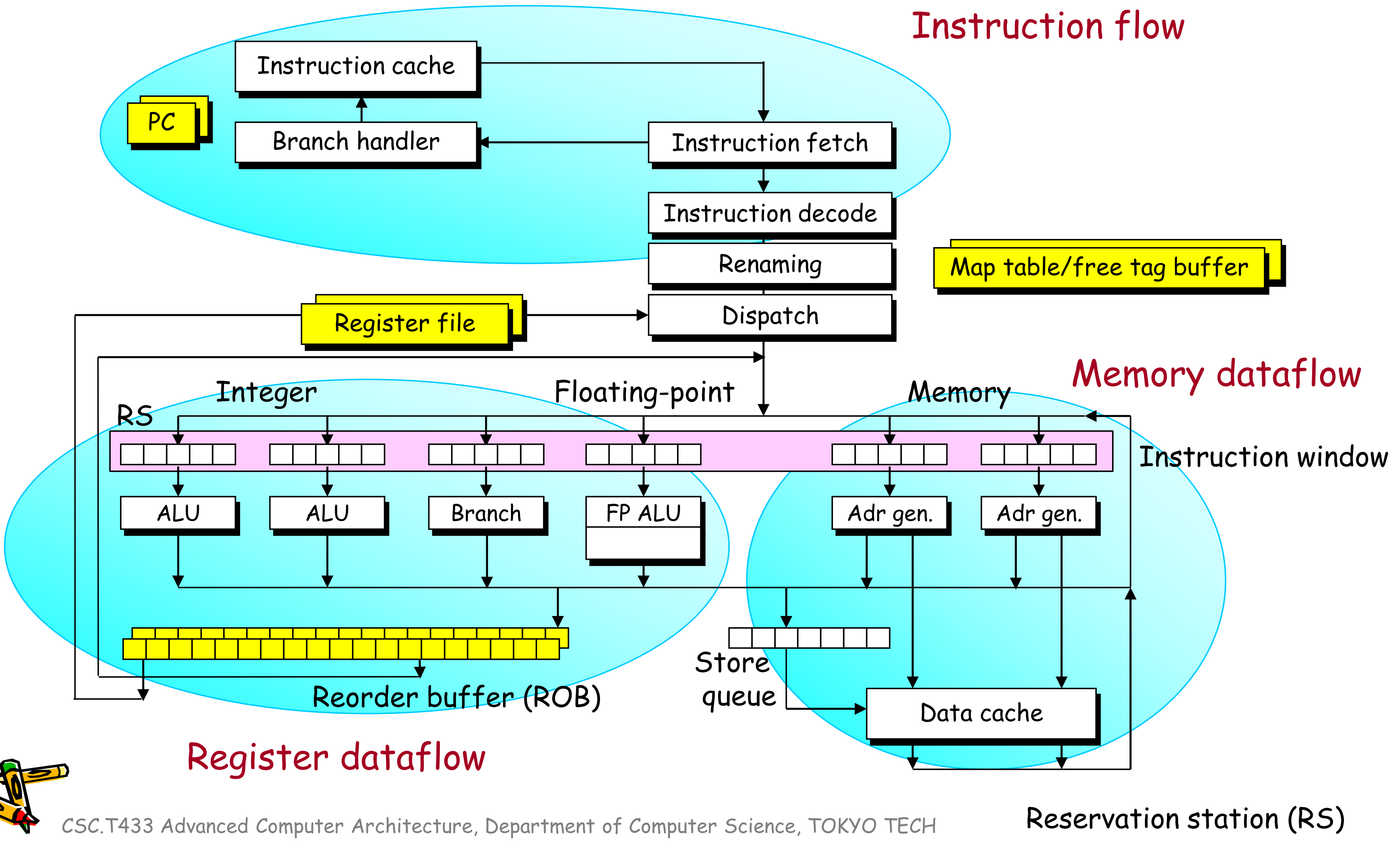
Newer instructions

Instructions to be executed of program B



Newer instructions

Datapath of SMT OoO execution processor



this slide is to be used as a whiteboard

From multi-core era to many-core era

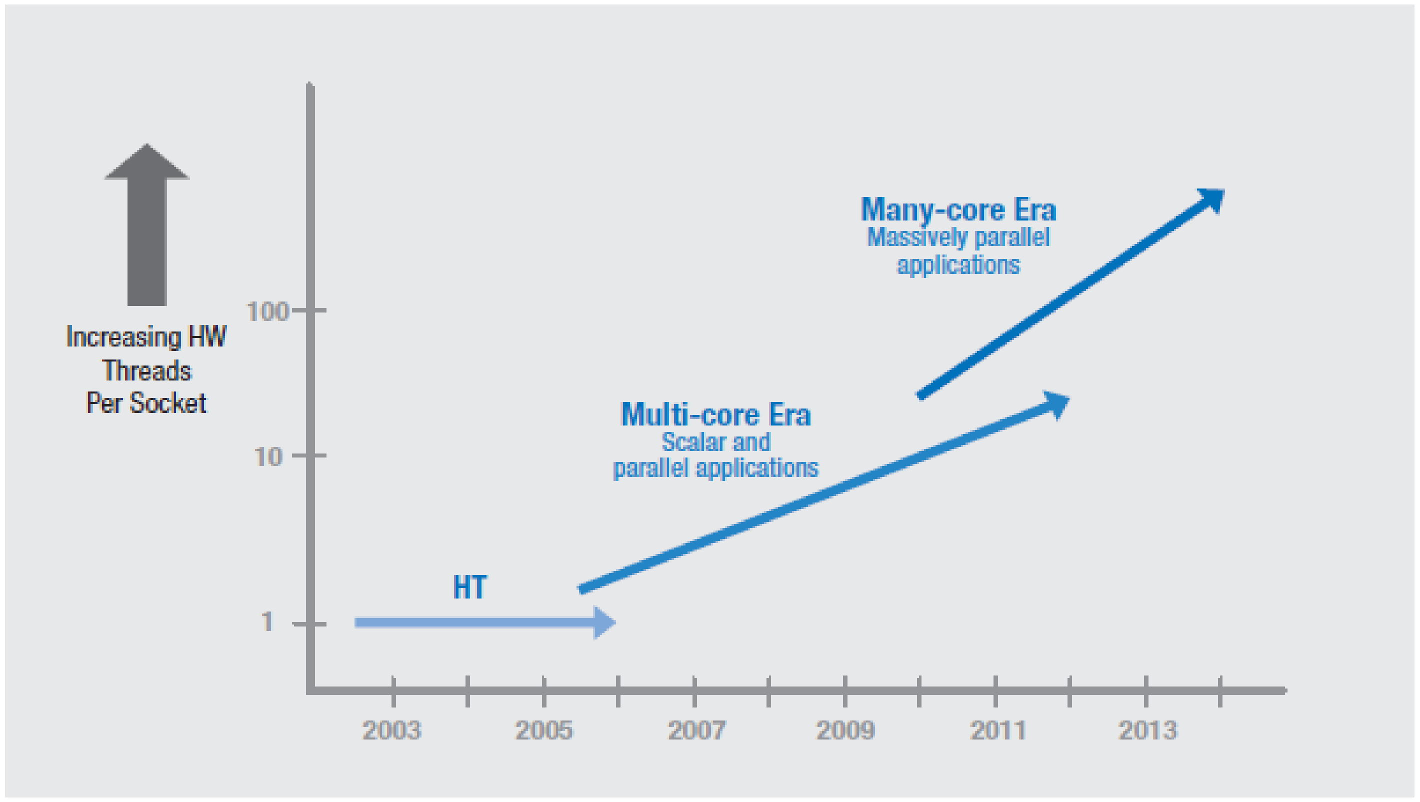
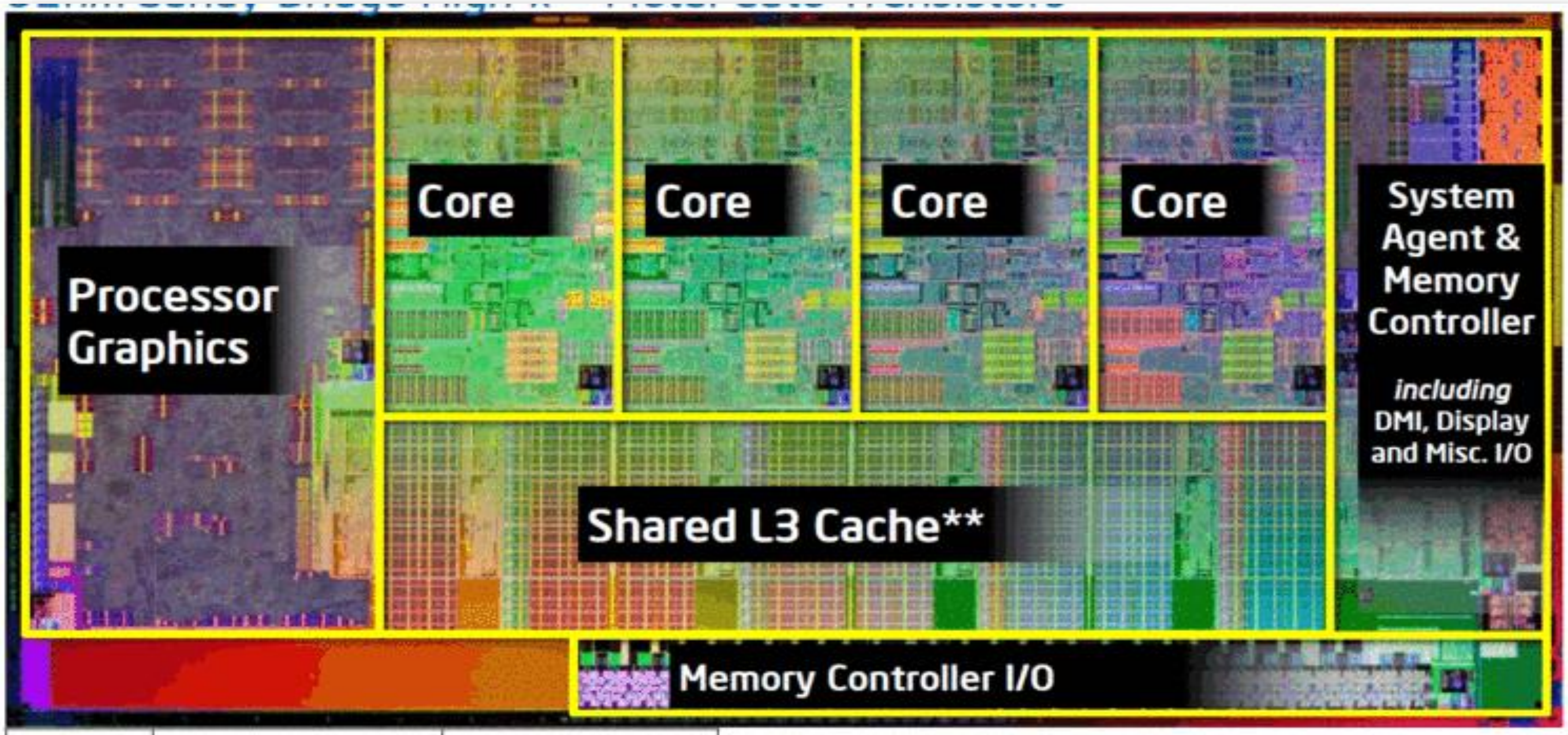


Figure 1: Current and expected eras of Intel® processor architectures

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

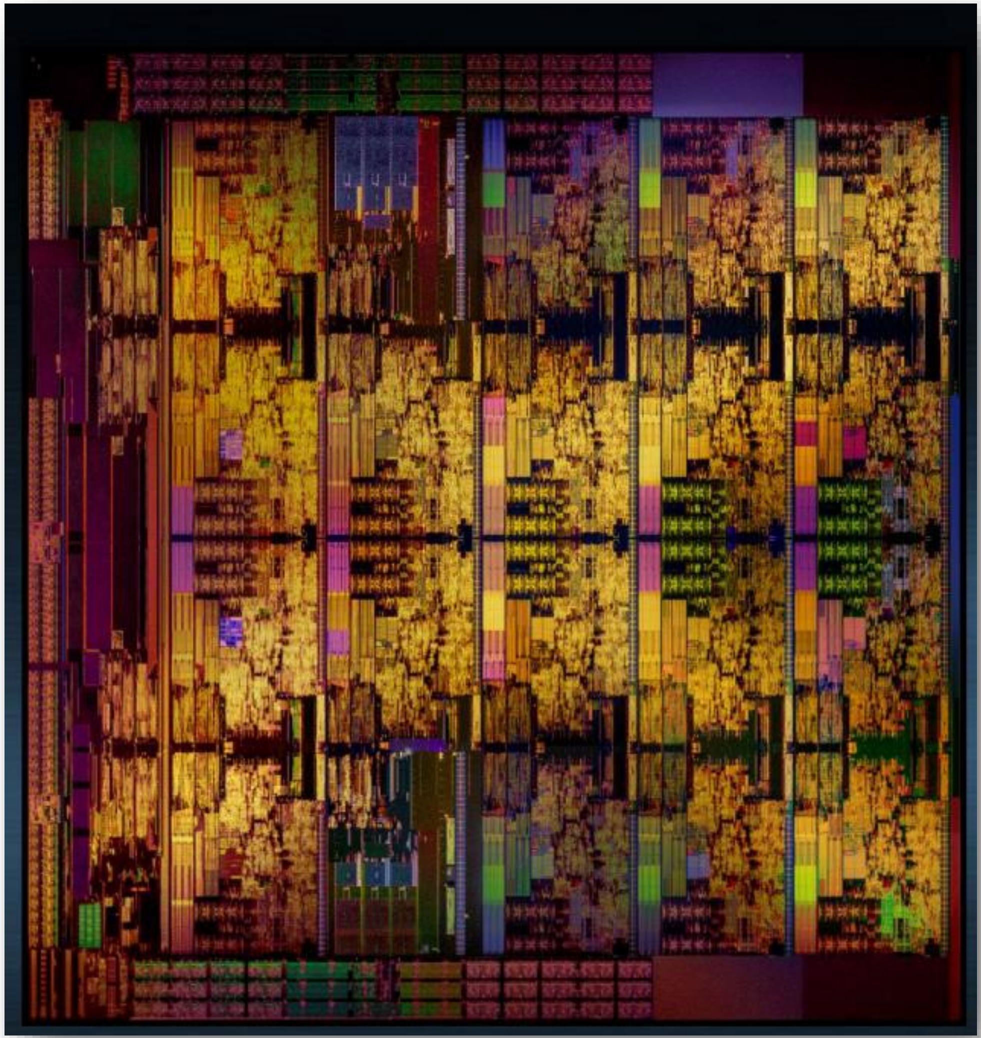
Intel Sandy Bridge, January 2011

- 4 to 8 core

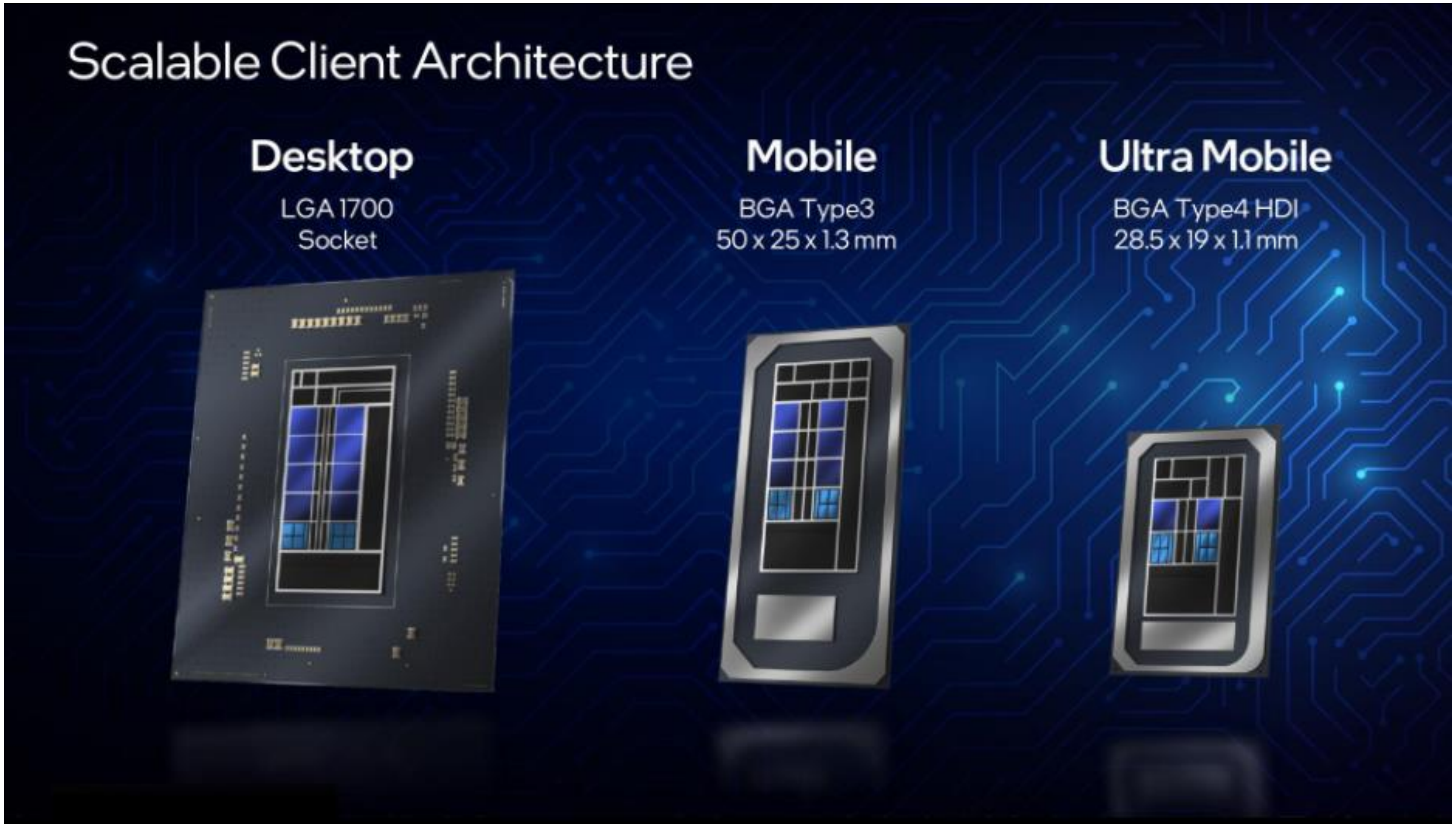


Intel Skylake-X, Core i9-7980XE, 2017

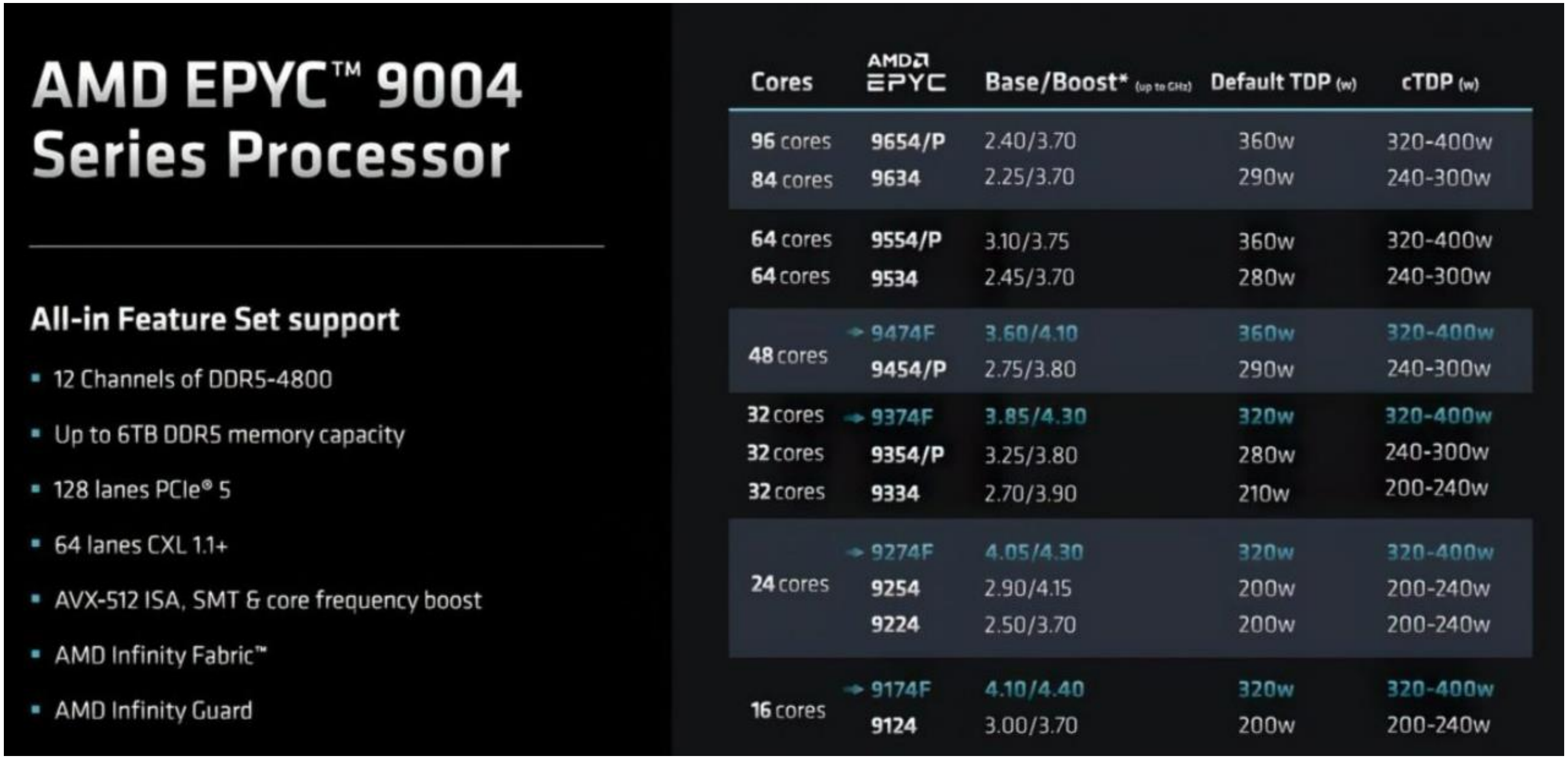
- 18 core



2021.11 Intel Alder Lake processor



2022.11 AMD EPYC 9654 processor with 96 cores



AMD EPYC™ 9004 Series Processor

All-in Feature Set support

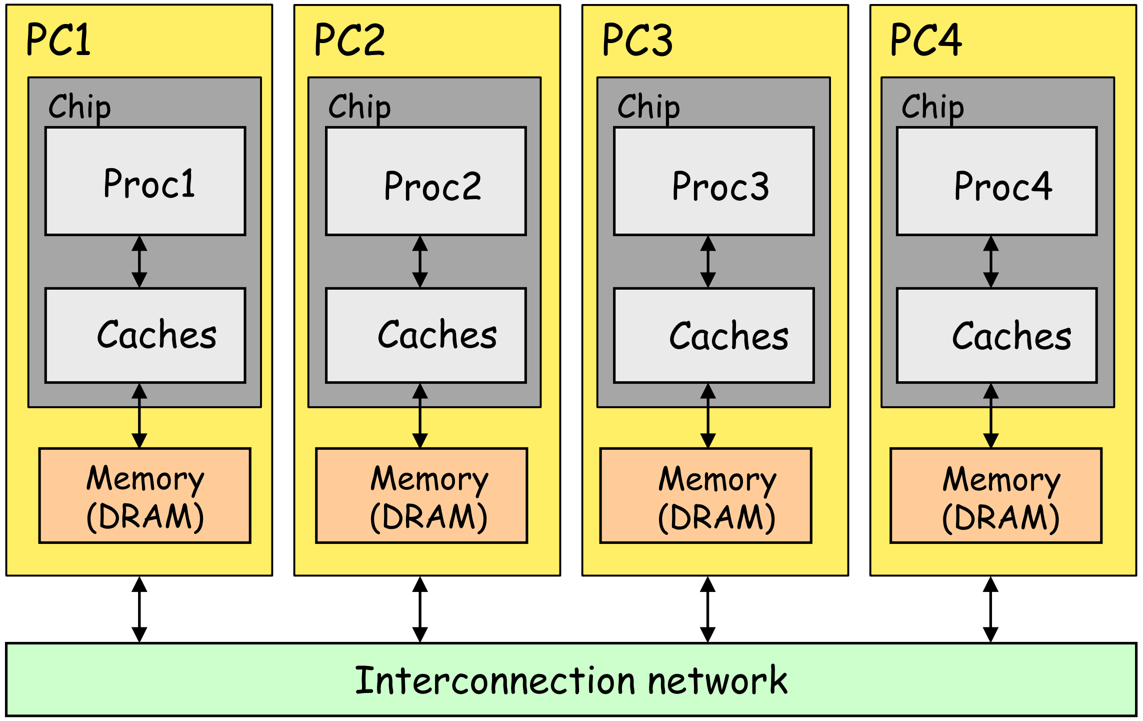
- 12 Channels of DDR5-4800
- Up to 6TB DDR5 memory capacity
- 128 lanes PCIe® 5
- 64 lanes CXL 1.1+
- AVX-512 ISA, SMT & core frequency boost
- AMD Infinity Fabric™
- AMD Infinity Guard

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

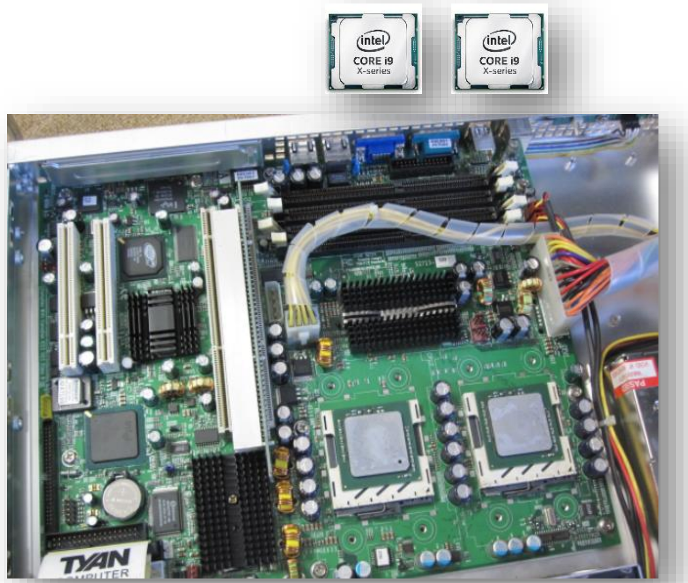
- A PC cluster or parallel computers for higher performance
- Each memory module is associated with a processor
- Using explicit send and receive functions (message passing) to obtain the data required.
  - Who will send and receive data? How?

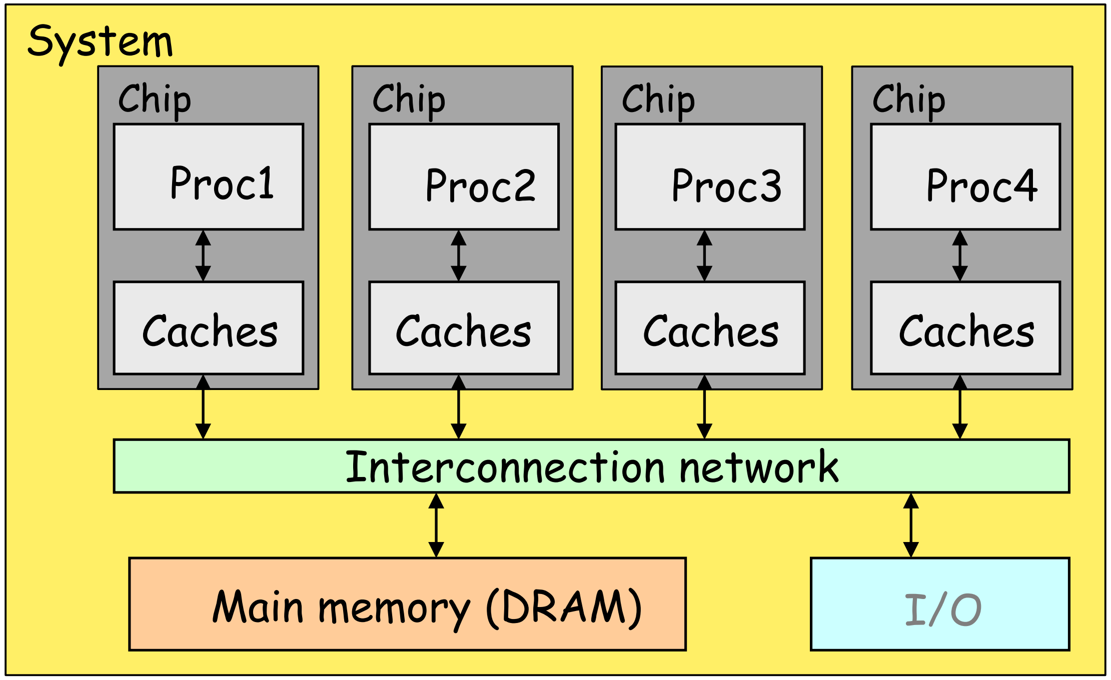
PC cluster



Shared Memory Multi-Processor Architecture

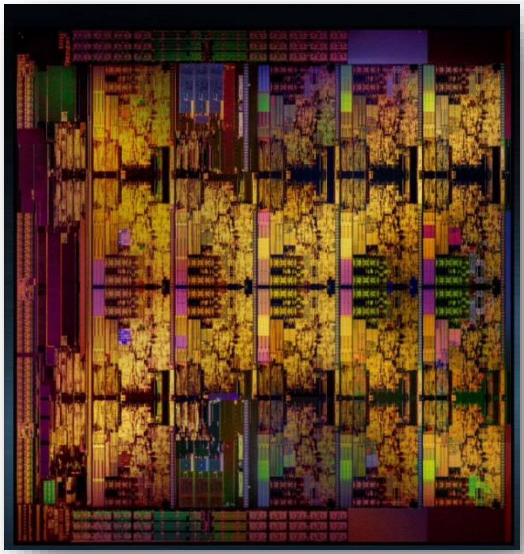
- All the processors can access the same address space of the main memory (shared memory) through an interconnection network.
- The shared memory or shared address space (SAS) is used as a means for communication between the processors.
    - What are the means to obtain the shared data?
    - What are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

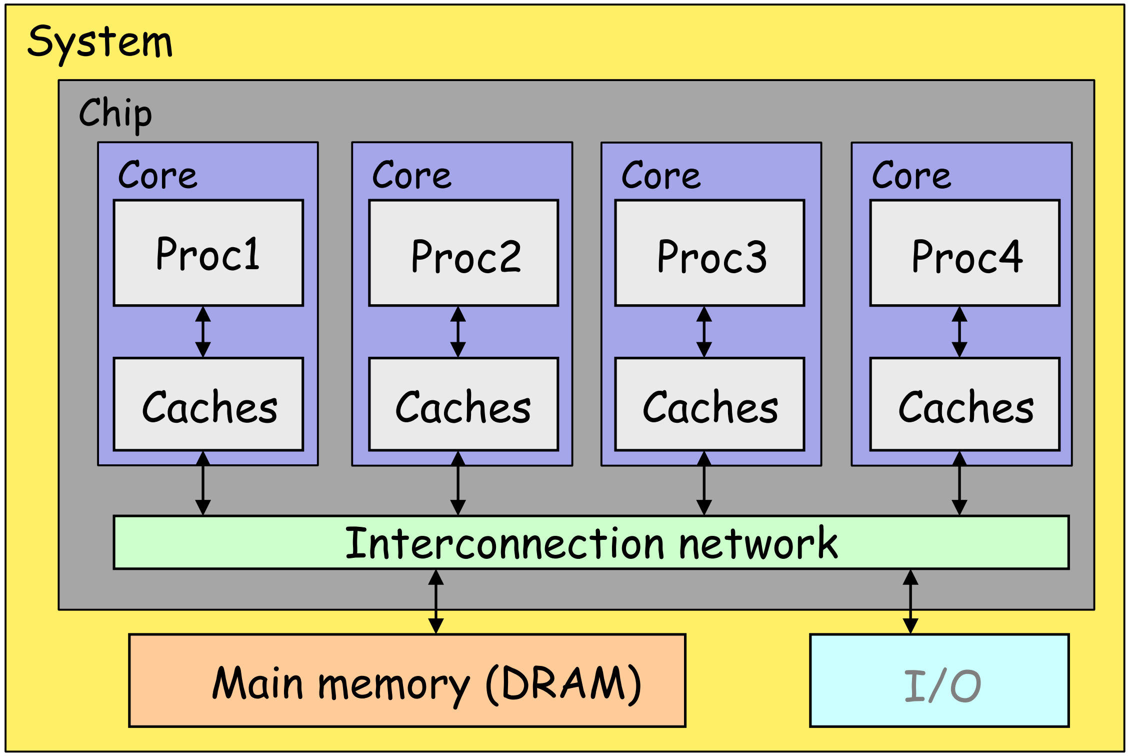




Shared memory many-core architecture

- The single-chip integrates many cores (conventional processors) and an interconnection network.

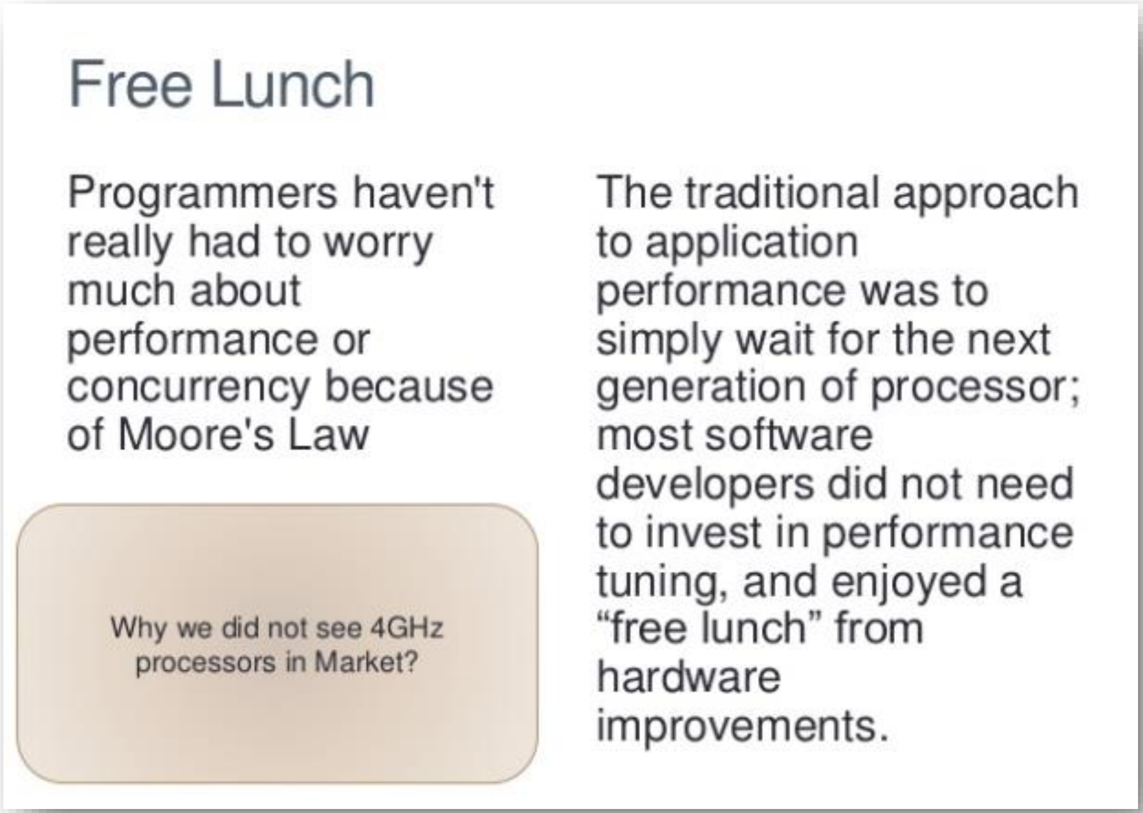

Intel Skylake-X, Core i9-7980XE, 2017



this slide is to be used as a whiteboard

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

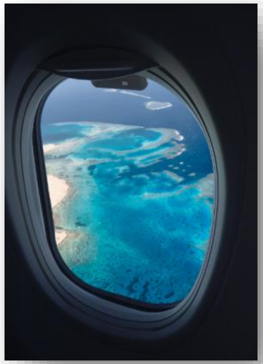
Why we did not see 4GHz processors in Market?

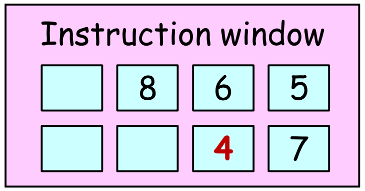
The traditional approach to application performance was to simply wait for the next generation of processor; most software developers did not need to invest in performance tuning, and enjoyed a “free lunch” from hardware improvements.

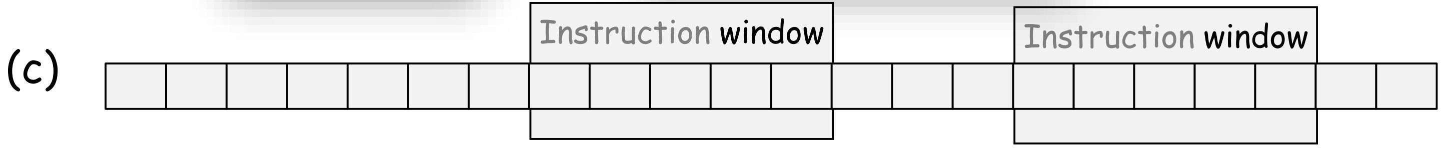
The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software by Herb Sutter, 2005

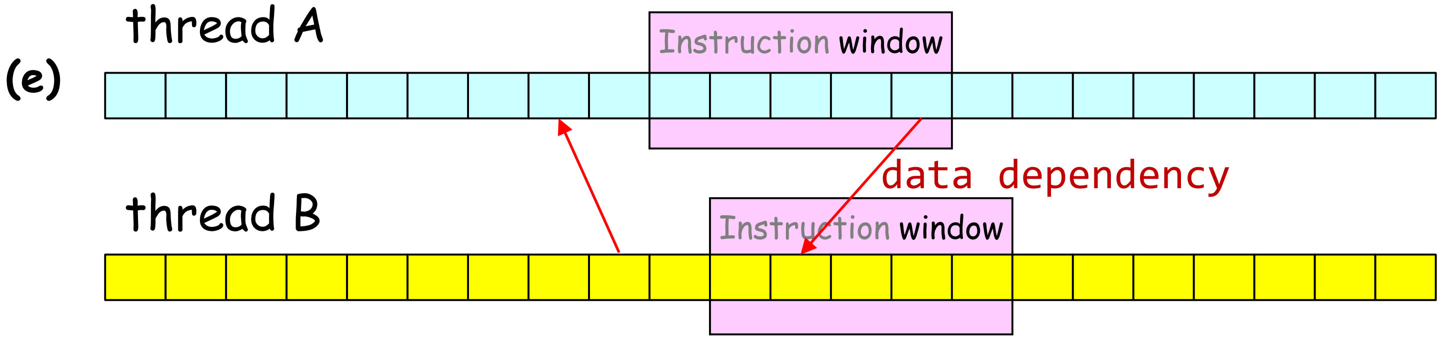
Parallel programming

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

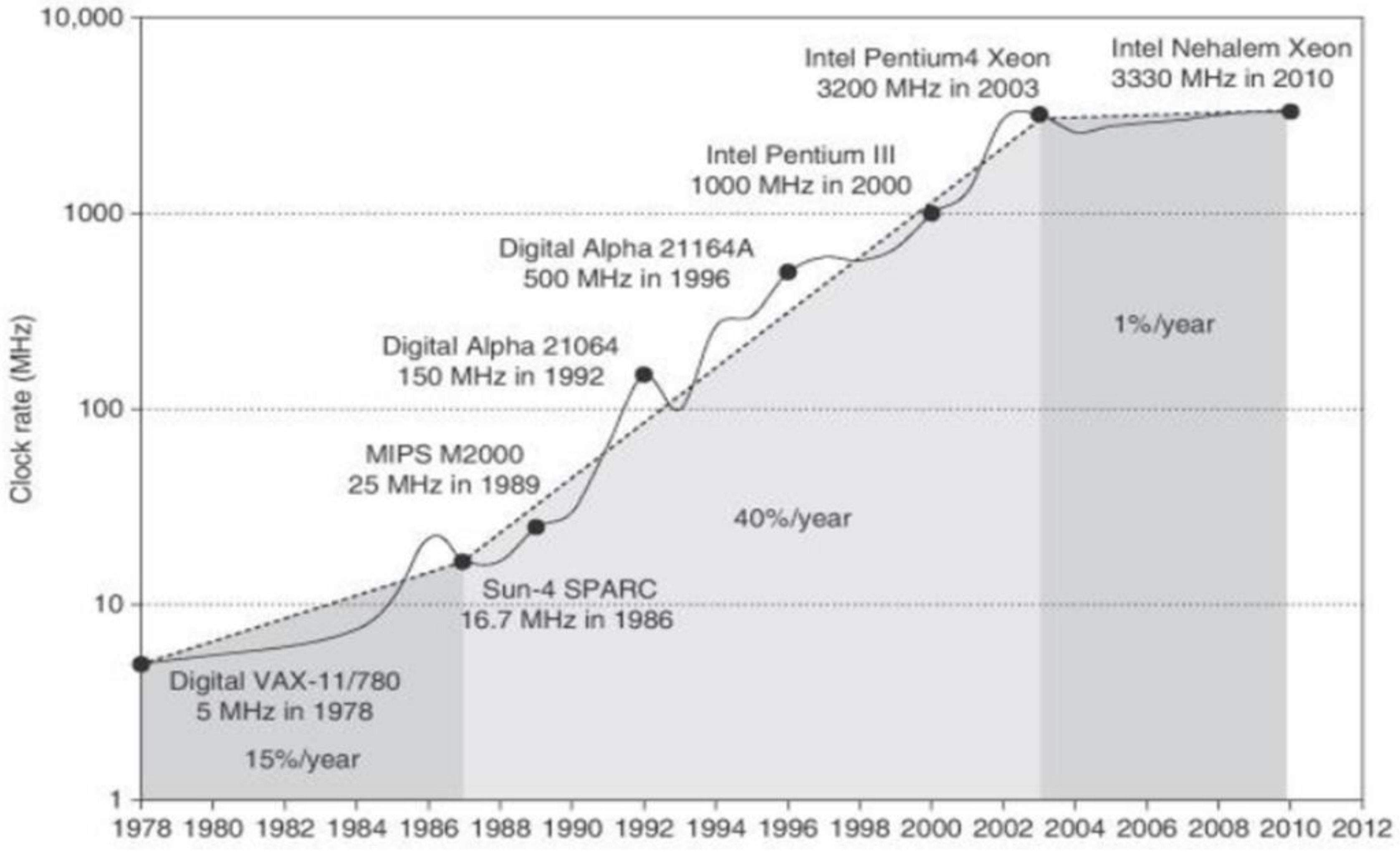








Growth in clock rate of microprocessors



From CAQA 5th edition

Sample of a wrong parallel program using pthread

```
% gcc main.c -lpthread
% ./a.out
main: 20000000
```

```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;
}
```

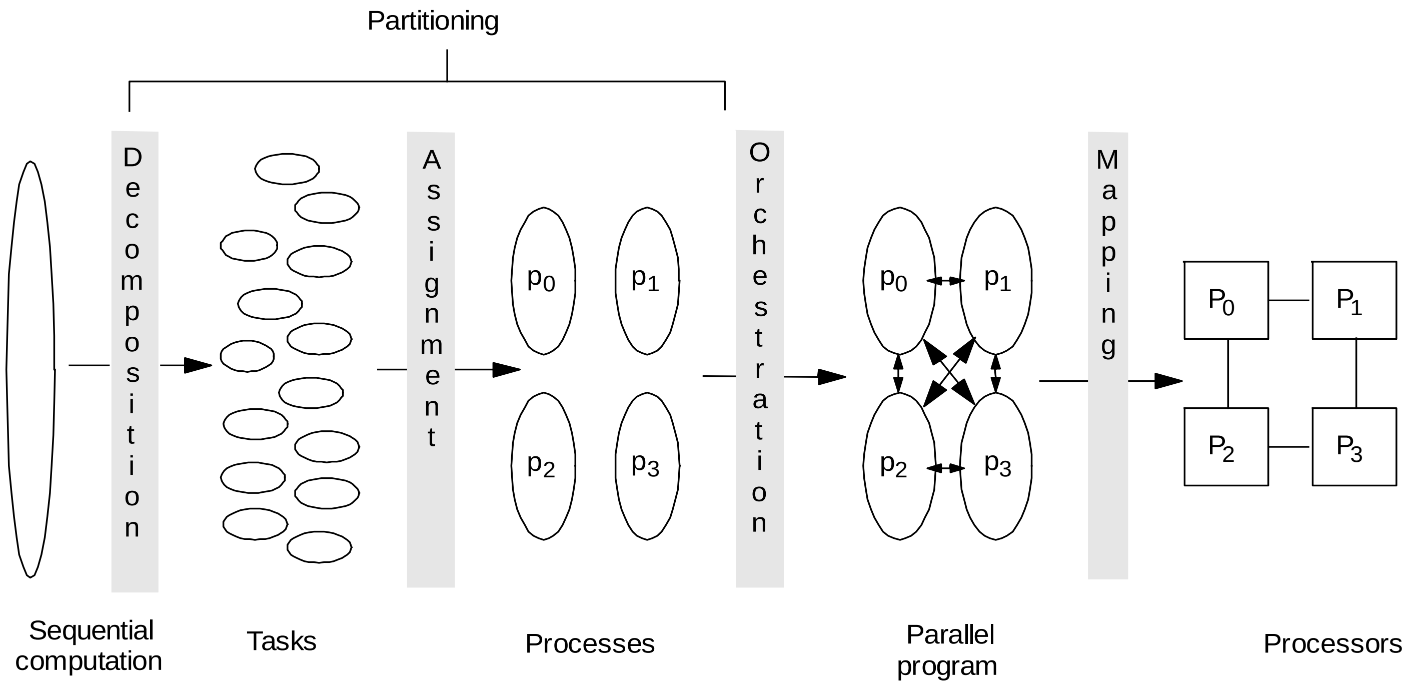
```c
#include <stdio.h>
#include <pthread.h>
#define N 10000000    // ten million

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;
}
```

```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;
}
```

this slide is to be used as a whiteboard

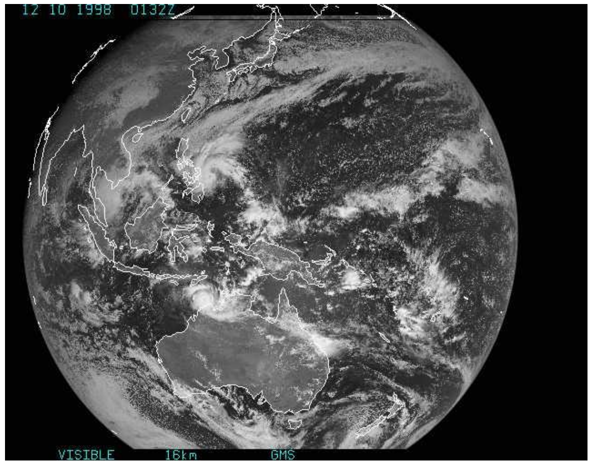
Four steps in creating a parallel program

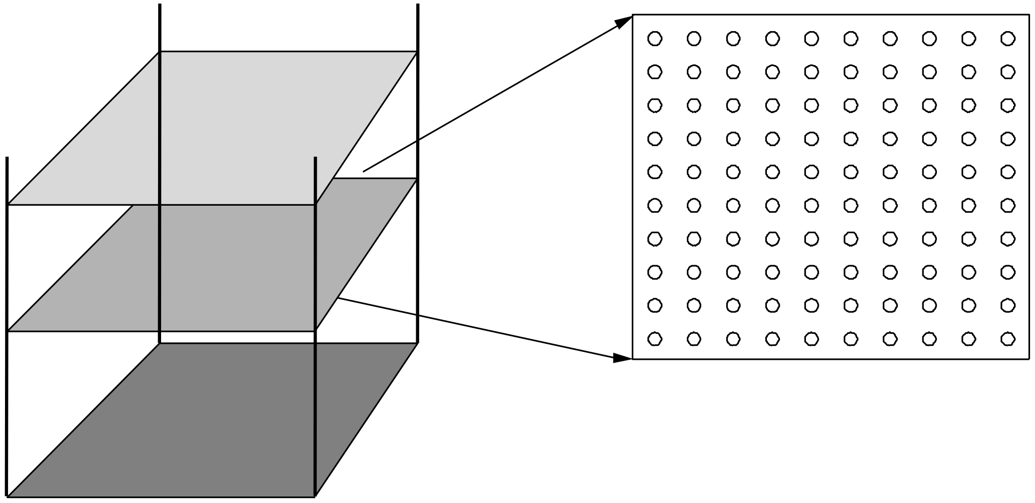
0. Preparing an optimized sequential program (baseline)
1. Decomposition of computation in tasks
2. Assignment of tasks to processes
3. Orchestration of data access, comm, synch.
4. Mapping processes to processors (cores)



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

Simulating ocean currents





(a) Cross sections

(b) Spatial discretization of a cross section

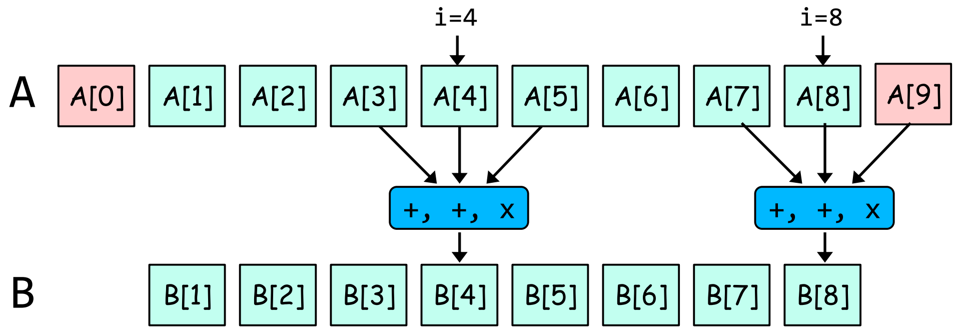
- Model as two-dimensional grids
  - Discretize in space and time
  - finer spatial and temporal resolution enables greater accuracy
- Many different computations per time step
  - Concurrency across and within grid computations
- We use one-dimensional grids for simplicity

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
```

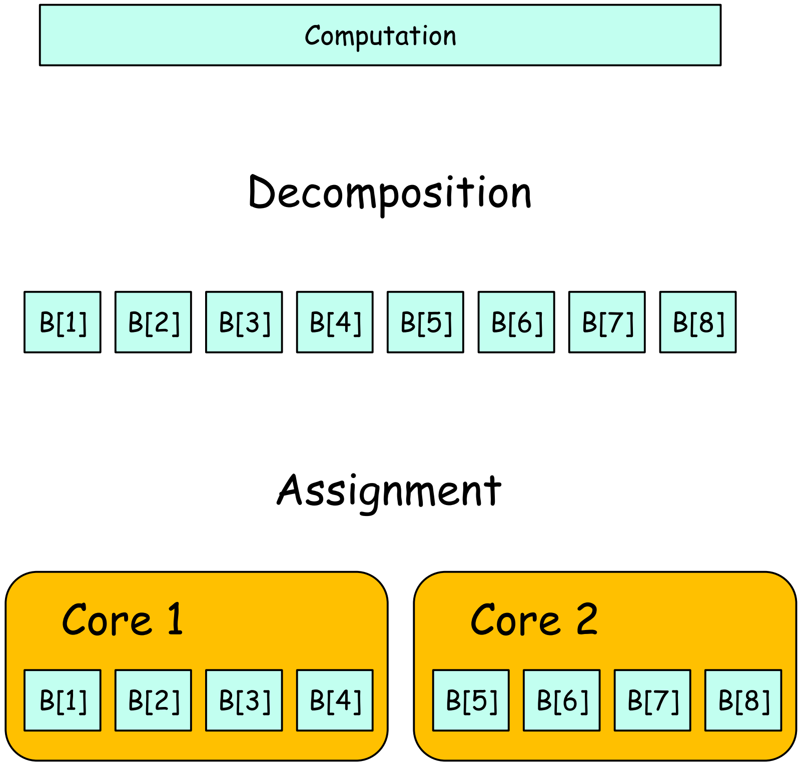


this slide is to be used as a whiteboard

Decomposition and assignment

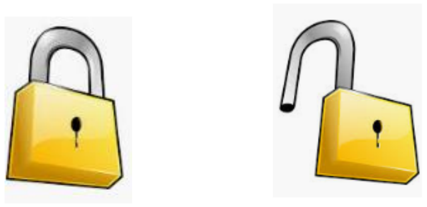
- Single Program Multiple Data (SPMD)
  - Decomposition: there are eight tasks to compute B[i]
  - Assignment: the first four tasks for core 1, and the last four tasks for core 2

```
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);
}
```



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

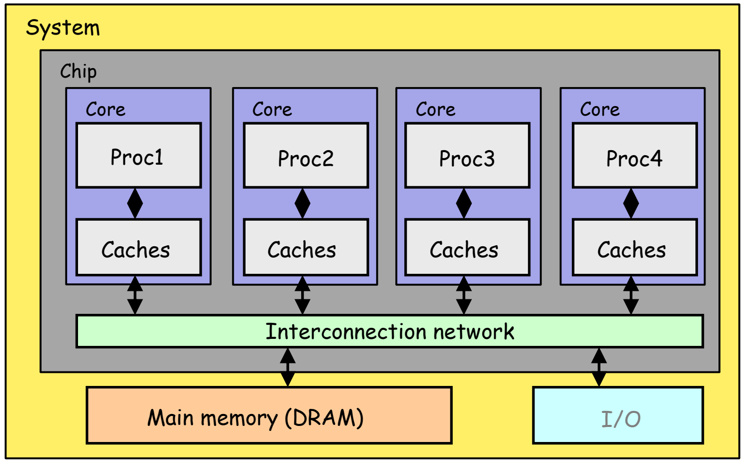
After all cores update the diff, *if statement* must be executed.

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

this slide is to be used as a whiteboard

Key components of many-core processors

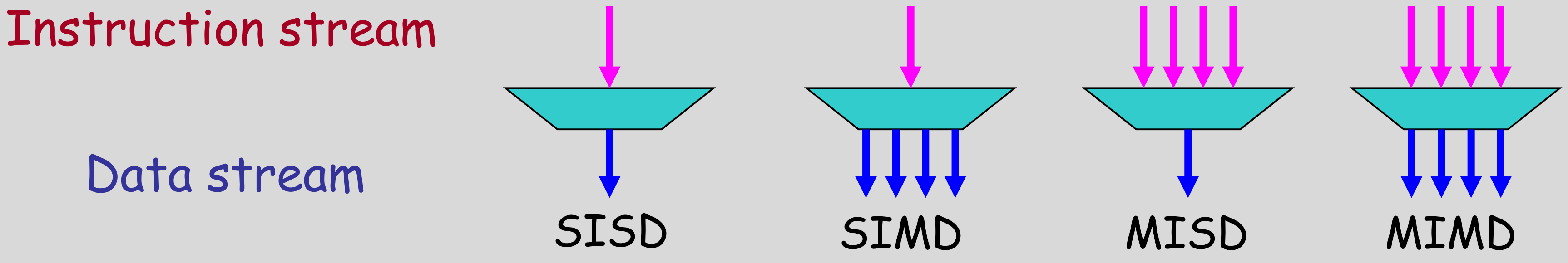
- Interconnection network
  - connecting many modules on a chip achieving high throughput and low latency
- Main memory and caches
  - Caches are used to reduce latency and to lower network traffic
  - A parallel program has private data and shared data
  - New issues are cache coherence and memory consistency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



this slide is to be used as a whiteboard

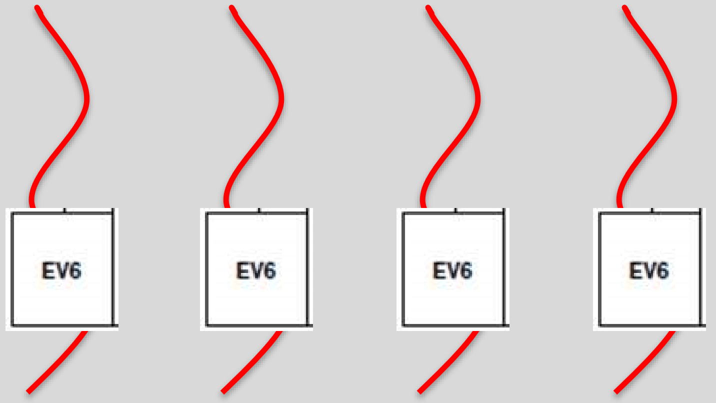
Flynn's taxonomy (1996)

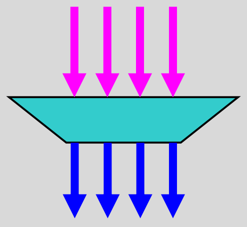
- A classification of computer architectures, proposed by Michael J. Flynn in 1966. The four classifications are based upon the number of concurrent instruction streams and data streams available in the architecture.
  - **SISD (Single Instruction stream, Single Data stream)**
  - **SIMD (Single Instruction stream, Multiple Data stream)**
  - MISD (Multiple Instruction stream, Single Data stream)
  - **MIMD (Multiple Instruction stream, Multiple Data stream)**



Flynn's taxonomy (1996)

- A classification of computer architectures, proposed by Michael J. Flynn in 1966. The four classifications are based upon the number of concurrent instruction streams and data streams available in the architecture.
  - **SISD (Single Instruction stream, Single Data stream)**
  - **SIMD (Single Instruction stream, Multiple Data stream)**
  - MISD (Multiple Instruction stream, Single Data stream)
  - **MIMD (Multiple Instruction stream, Multiple Data stream)**

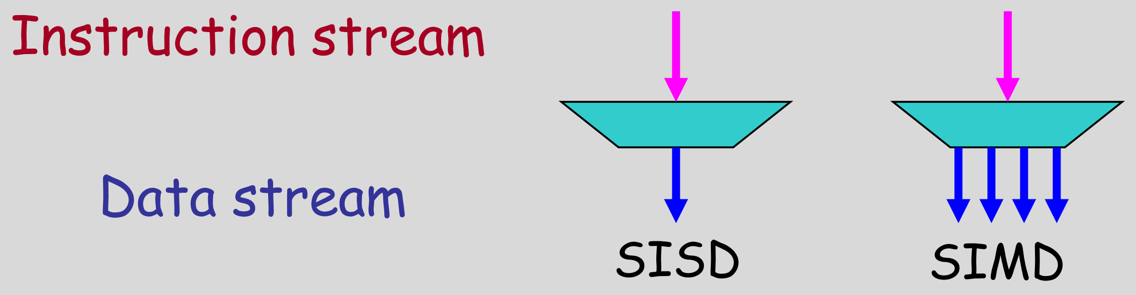




MIMD

Flynn's taxonomy (1996)

- A classification of computer architectures, proposed by Michael J. Flynn in 1966. The four classifications are based upon the number of concurrent instruction streams and data streams available in the architecture.
  - **SISD (Single Instruction stream, Single Data stream)**
  - **SIMD (Single Instruction stream, Multiple Data stream)**
  - MISD (Multiple Instruction stream, Single Data stream)
  - **MIMD (Multiple Instruction stream, Multiple Data stream)**



SIMD Variants

- Vector architectures
- SIMD extensions
- Graphics Processing Units (GPUs)

- SIMD variants exploit data-level parallelism
- Instruction-level parallelism in superscalar processors
- Thread-level parallelism in multicore processors

Vector architecture

- Computers designed by Seymour Cray starting in the 1970s
- Basic idea:
  - Read sets of data elements into "vector registers"
  - Operate on those registers
  - Disperse the results back into memory

Cray Supercomputer

DAXPY in MIPS Instructions

Example: DAXPY (double precision a x X + Y)

```
        L.D       F0,a          ; load scalar a
        DADDIU    R4,Rx,#512    ; upper bound of what to load
Loop:   L.D       F2,0(Rx )     ; load X[i]
        MUL.D     F2,F2,F0      ; a x X[i]
        L.D       F4,0(Ry)      ; load Y[i]
        ADD.D     F4,F2,F2      ; a x X[i] + Y[i]
        S.D       F4,9(Ry)      ; store into Y[i]
        DADDIU    Rx,Rx,#8      ; increment index to X
        DADDIU    Ry,Ry,#8      ; increment index to Y
        SUBBU     R20,R4,Rx     ; compute bound
        BNEZ      R20,Loop      ; check if done
```

- Requires almost 600 MIPS operations

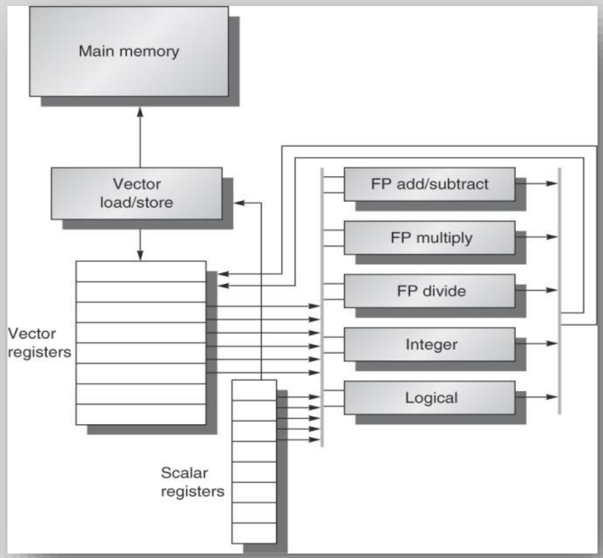
DAXPY in VMIPS (MIPS with Vector) Instructions

- ADDV.D : add two vectors
- ADDVS.D : add vector to a scalar
- LV/SV : vector load and vector store from address

- Example: DAXPY (double precision a*X+Y)

```
L.D       F0,a       ; load scalar a
LV        V1,Rx      ; load vector X
MULVS.D   V2,V1,F0   ; vector-scalar multiply
LV        V3,Ry      ; load vector Y
ADDV.D V4,V2,V3      ; add
SV        Ry,V4      ; store the result
```

- Requires 6 instructions


Main memory
Vector load/store
FP add/subtract
FP multiply
FP divide
Integer
Logical
Vector registers
Scalar registers