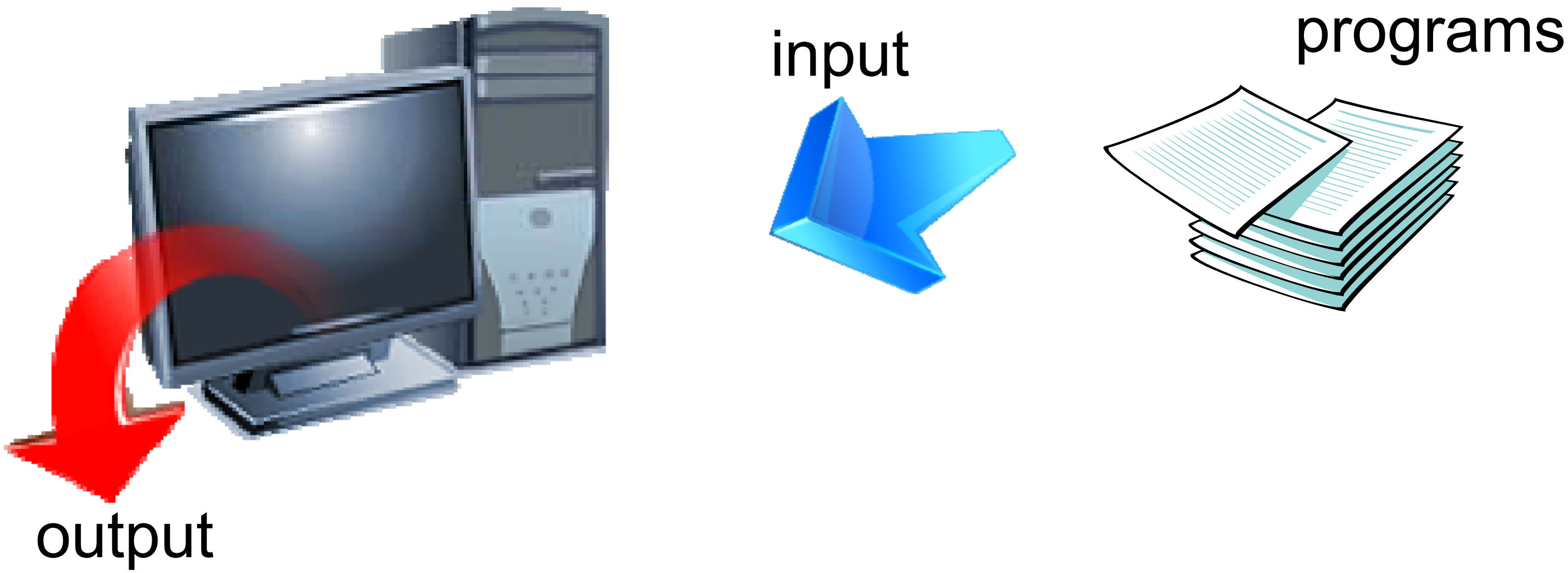
ECE 432/532
Programming for Parallel Processors

Roadmap

- Some background
- Modifications to the von Neumann model
- Parallel hardware
- Parallel software
- Input and output
- Performance
- Parallel program design
- Writing and running parallel programs
- Assumptions

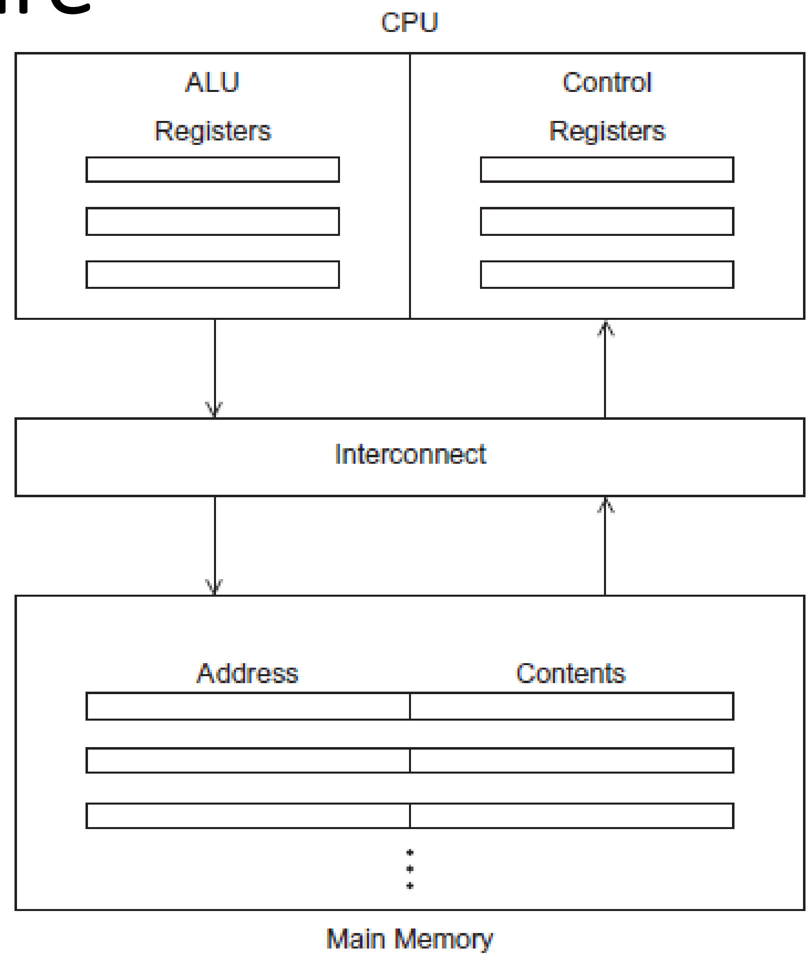
Serial hardware and software

- Parallel hardware and software have grown out of conventional **serial** hardware and software
  - hardware and software that runs (more or less) a single job at a time.



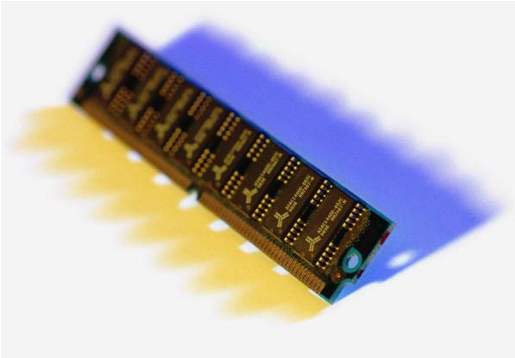
The von Neumann Architecture

- The “classical” **von Neumann architecture** consists of
  - main memory
  - a central processing unit (CPU)
  - an interconnection between the memory and the CPU.



Main memory

- Consists of a collection of locations, each of which is capable of storing both instructions and data.
- Every location consists of an address, which is used to access the location, and the contents of the location.



Central processing unit (CPU)

- Divided into two parts.

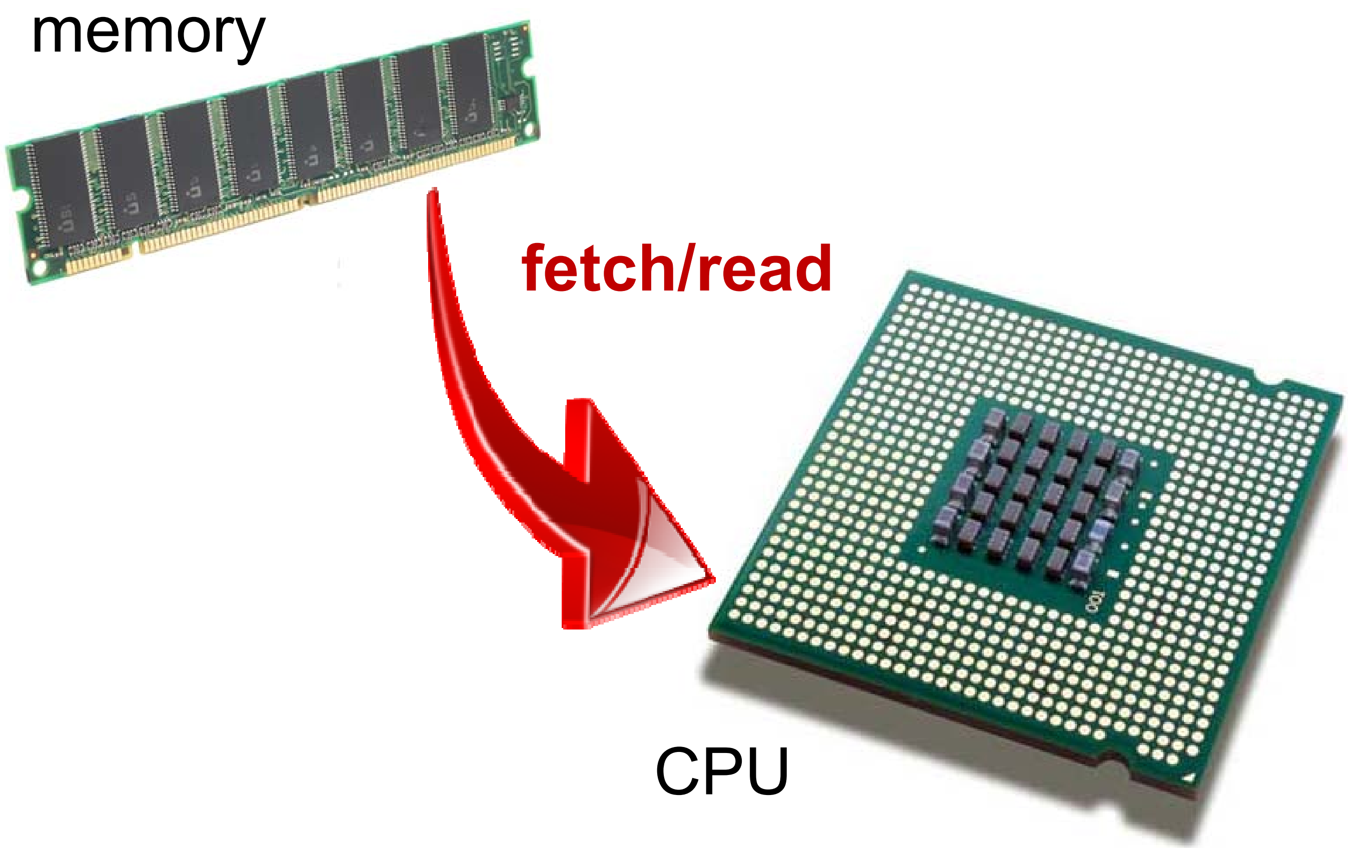
- **Control unit** - responsible for deciding which instruction in a program should be executed. (*the boss*)

- **Arithmetic and logic unit** (ALU) - responsible for executing the actual instructions. (*the worker*)

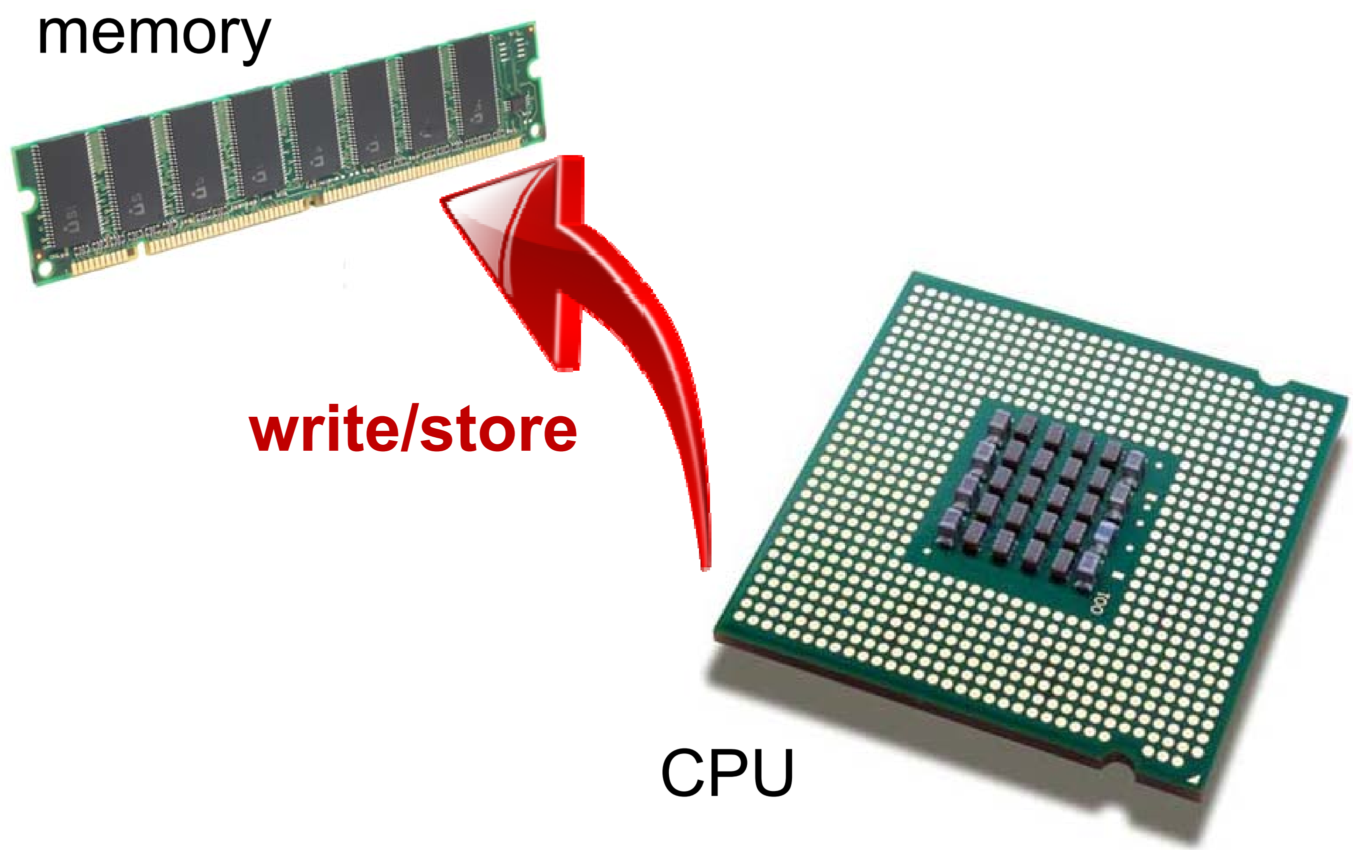
Key terms

- **Register** – very fast storage, part of the CPU.
- **Program counter** – stores address of the next instruction to be executed.
- **Bus** – wires and hardware that connects the CPU and memory.

Transfer data from memory

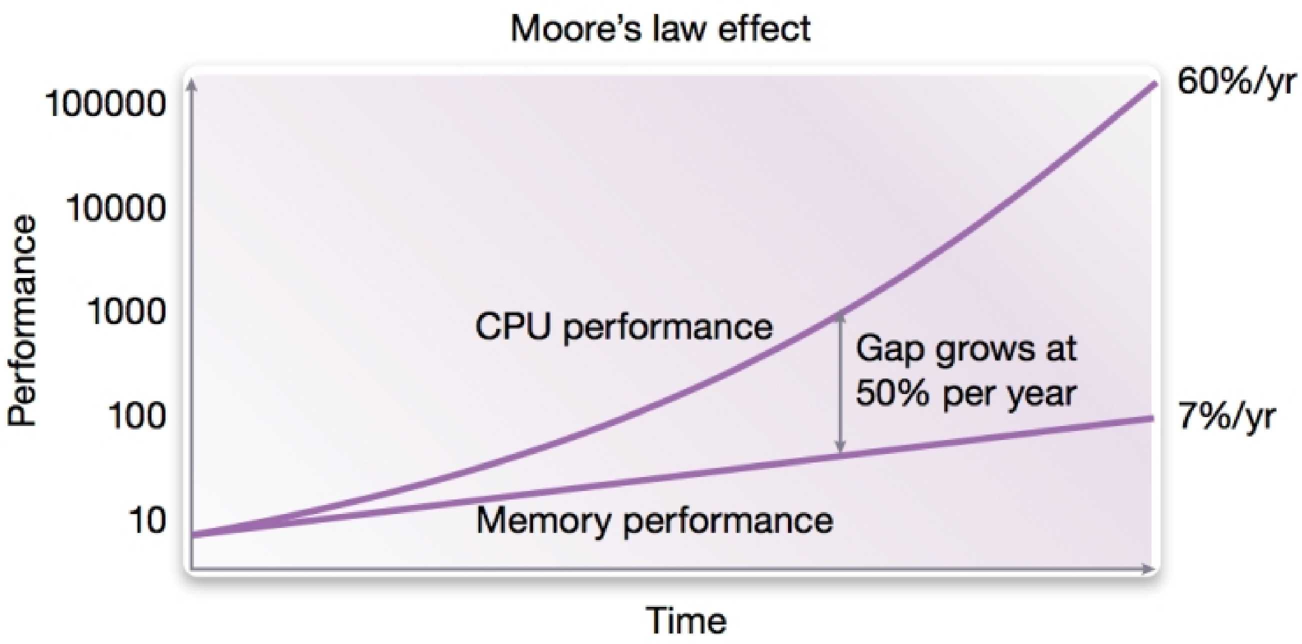


Transfer data to memory



von Neumann bottleneck

- Do you remember the memory gap?



Background information

- Processes
- Multitasking
- Threads

Processes

- An instance of a computer program that is being executed.
- Components of a process:
  - The executable machine language program.
  - A block of memory.
  - Descriptors of resources the OS has allocated to the process.
  - Security information.
  - Information about the state of the process.

Multitasking

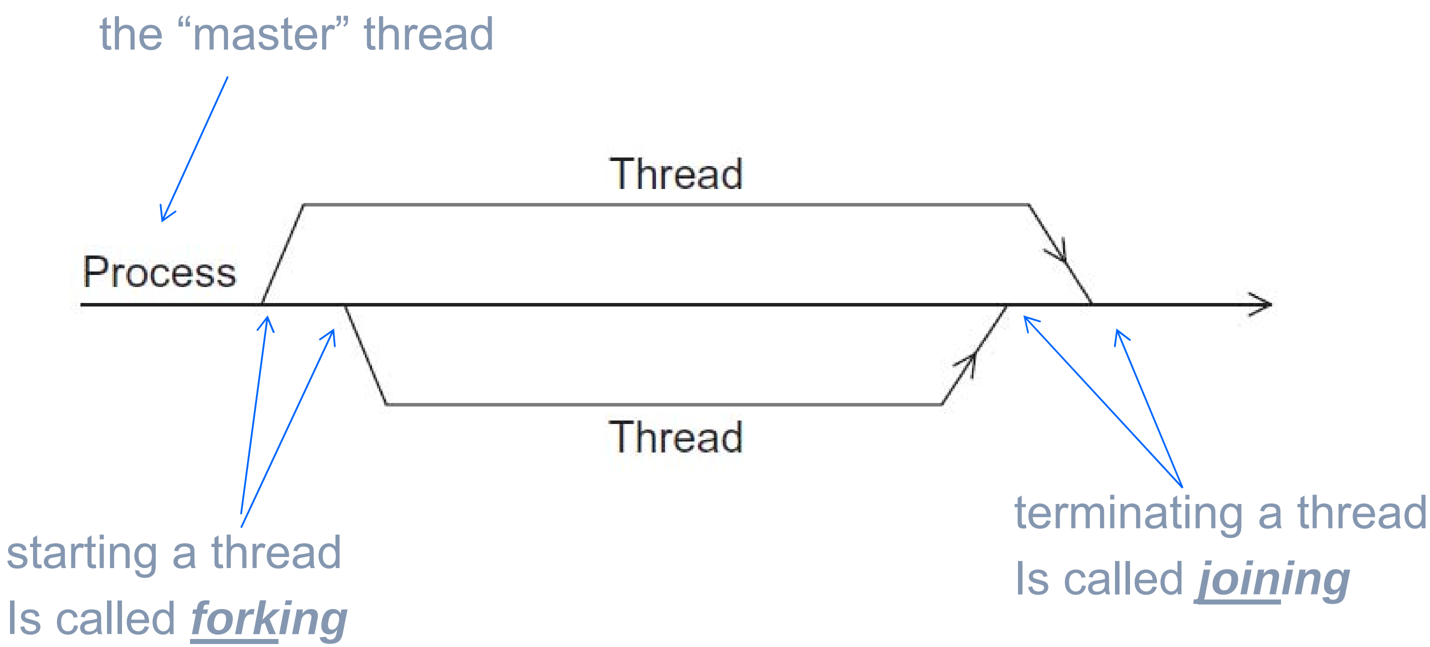
- Gives the illusion that a single processor system is running multiple programs simultaneously.
- Each process takes turns running. (**time slice**)
- After its time is up, it waits until it has a turn again. (**blocks**)

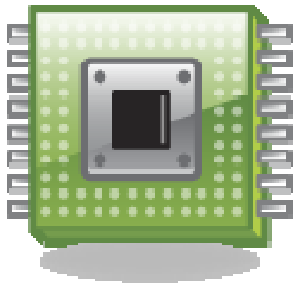
Threading

- Threads are contained within processes.

- They allow programmers to divide their programs into (more or less) independent tasks.
- The hope is that when one thread blocks because it is waiting on a resource, another will have work to do and can run.

A process and two threads





Modifications to the von neumann model

Modifications to the von Neumann model

- Do you remember the von Neumann bottleneck problem?

- Caching is one of the most widely used methods of addressing this issue
  - A collection of memory locations that can be accessed in less time than some other memory locations
  - A CPU cache is typically located on the same chip, or one that can be accessed much faster than ordinary memory.

Memory Hierarchy

Cache	Hit Cost	Size
1st level cache/first level TLB	1 ns	64 KB
2nd level cache/second level TLB	4 ns	256 KB
3rd level cache	12 ns	2 MB
Memory (DRAM)	100 ns	10 GB
Data center memory (DRAM)	100 μs	100 TB
Local non-volatile memory	100 μs	100 GB
Local disk	10 ms	1 TB
Data center disk	10 ms	100 PB
Remote data center disk	200 ms	1 XB

i7 has 8MB as shared 3rd level cache; 2nd level cache is per-core

Caching

- Once we have a cache, an obvious problem is deciding which data and instructions should be stored in the cache!!

Caching

- Once we have a cache, an obvious problem is deciding which data and instructions should be stored in the cache!!

- Observation: Programs tend to use data and instructions that are physically close to recently used data and instructions
  - After executing an instruction, programs typically execute the next instruction

Caching example

- Consider the loop:

```
float z[1000];
. . .
sum = 0.0;
for (i = 0; i < 1000; i++)
   sum += z[i];
```

- The location storing z[1] immediately follows the location z[0]
- Thus as long as i < 999, the read of z[i] is immediately followed by a read of z[i+1].

Cache locality

- The principle that an access of one location is followed by an access of a nearby location is often called **locality**

- **Spatial locality** – accessing a nearby location.

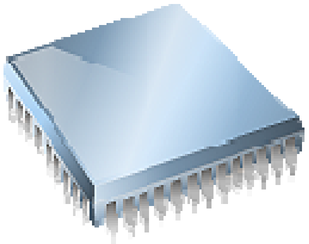
- **Temporal locality** – accessing in the near future.

Cache locality

- In order to exploit the principle of locality, cache operates on blocks of data and instructions instead of individual instructions and individual data items
  - Cache blocks
  - Cache lines

- In our example, if a cache line stores 16 floats, then when we first go to add sum += z[0]
  - The system reads the first 16 elements of z, z[0], z[1], . . . ,z[15] into cache
  - So the next 15 additions will use elements of z that are already in the cache

Cache hit



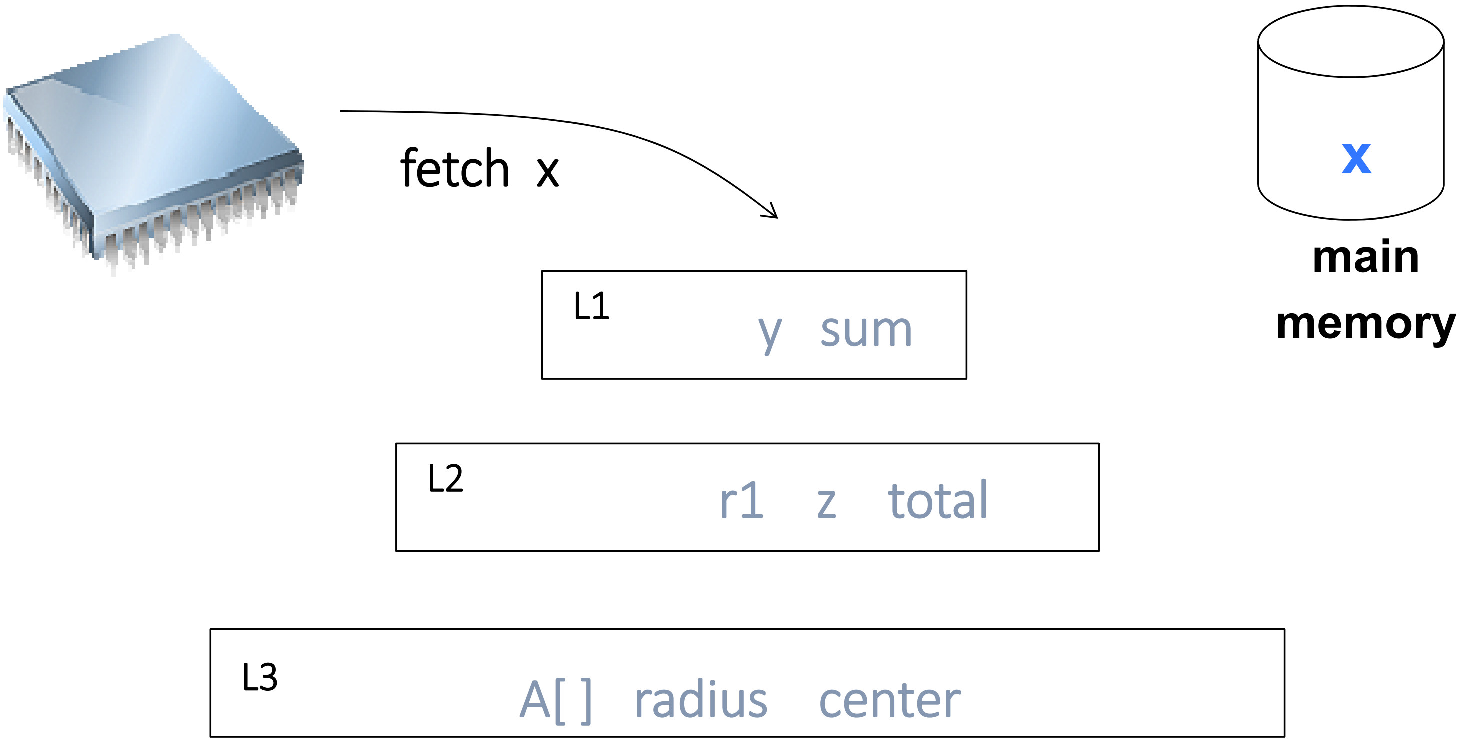
fetch x

L1 x sum

L2 y z total

L3 A[ ] radius r1 center

Cache miss



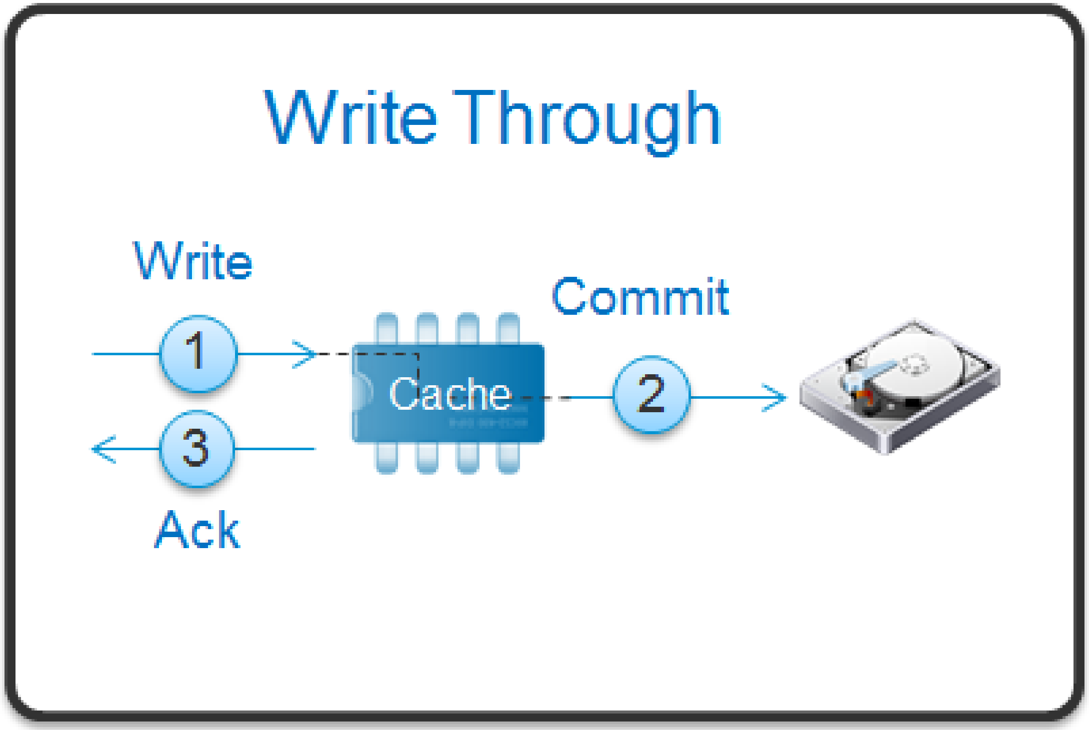
Cache issues

- When a CPU writes data to cache, the value in cache may be inconsistent with the value in main memory.

- Two policies:
  - **Write-through**
  - **Write-back**

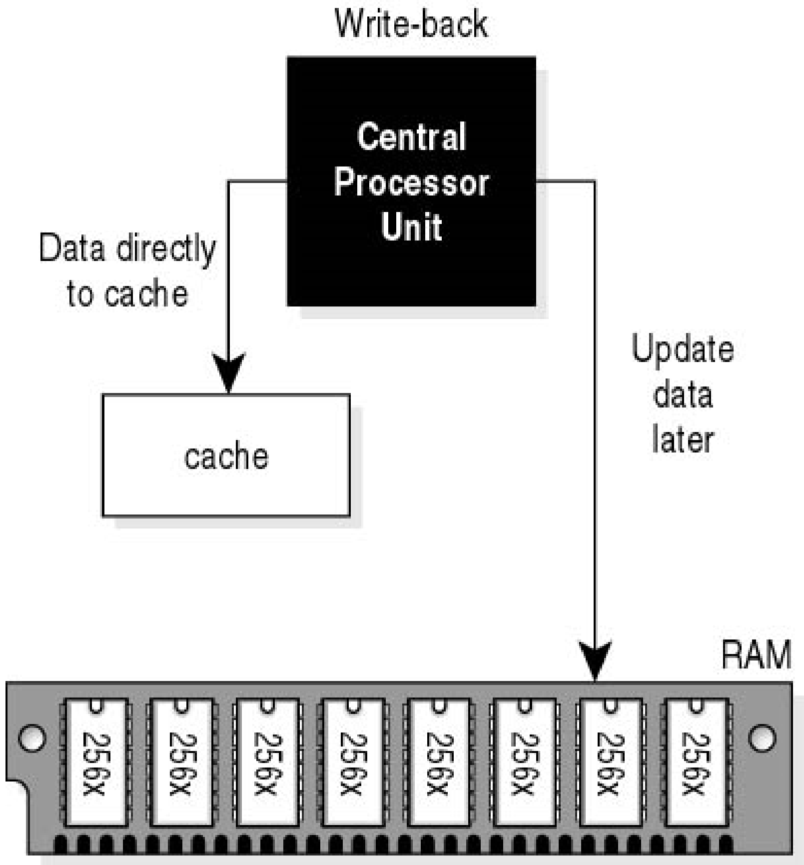
Write-through

- **Write-through** caches handle this by updating the data in main memory at the time it is written to cache.



Write-back

- **Write-back** caches mark data in the cache as dirty. When the cache line is replaced by a new cache line from memory, the dirty line is written to memory.



Cache mappings

- Another issue in cache design is deciding where lines should be stored
- If we fetch a cache line from main memory, where in the cache should it be placed?

Cache mappings

- **Full associative** – a new line can be placed at any location in the cache
- **Direct mapped** – each cache line has a unique location in the cache to which it will be assigned
- ***n*-way set associative** – each cache line can be place in one of *n* different locations in the cache
  - we also need to be able to decide which line should be replaced or evicted

Cache mappings - Example

- MM consists of 16 lines (0–15), and cache consists of 4 lines (0–3):

Memory Index	Cache Location		
	Fully Assoc	Direct Mapped	2-way
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

Cache mappings - Example

- **Fully associative cache**:
  - line 0 can be assigned to cache location 0, 1, 2, or 3

Memory Index	Cache Location		
	Fully Assoc	Direct Mapped	2-way
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

Cache mappings - Example

- **Direct mapped cache**:
  - Might assign lines by looking at their remainder after division by 4

Memory Index	Cache Location		
	Fully Assoc	Direct Mapped	2-way
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

Cache mappings - Example

- **2-way set associative cache:**
  - Group the cache into two sets:
  - indexes 0 and 1 form set 0
  - indexes 2 and 3 form set 1

  - The remainder of the MM index modulo 2, and cache line 0 would be mapped to either cache index 0 or cache index 1

Memory Index	Cache Location		
	Fully Assoc	Direct Mapped	2-way
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

Cache mappings - Example

- **2-way set associative cache:**
  - Group the cache into two sets:
  - indexes 0 and 1 form set 0
  - indexes 2 and 3 form set 1

- When more than one line in memory can be mapped to several different locations in a cache, we need to be able to decide which line should be replaced or **evicted**
  - Least recently used (LRU) policy

Memory Index	Cache Location		
	Fully Assoc	Direct Mapped	2-way
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

Caches and programs

- CPU cache is controlled by hardware → programmers don't directly determine which data and which instructions are in the cache
  - "Noisy neighbor" problem in many-core systems
  - Intel introduced Cache Allocation Technique in new Xeon servers

- However, knowing the principle of spatial and temporal locality allows us to have some indirect control over caching

Caches and programs - Example

```
double A[MAX][MAX], x[MAX], y[MAX];
. . .
/* Initialize A and x, assign y = 0 */
. . .
/* First pair of loops */
for (i = 0; i < MAX; i++)
   for (j = 0; j < MAX; j++)
      y[i] += A[i][j]*x[j];
. . .
/* Assign y = 0 */
. . .
/* Second pair of loops */
for (j = 0; j < MAX; j++)
   for (i = 0; i < MAX; i++)
      y[i] += A[i][j]*x[j];
```

- Which loop has better performance?

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

Caches and programs - Example

```
double A[MAX][MAX], x[MAX], y[MAX];
. . .
/* Initialize A and x, assign y = 0 */
. . .
/* First pair of loops */
for (i = 0; i < MAX; i++)
   for (j = 0; j < MAX; j++)
      y[i] += A[i][j]*x[j];
. . .
/* Assign y = 0 */
. . .
/* Second pair of loops */
for (j = 0; j < MAX; j++)
   for (i = 0; i < MAX; i++)
      y[i] += A[i][j]*x[j];
```

- Which loop has better performance?

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

- Suppose MAX=4
- Direct mapped cache (omit X and Y)
  - 8 elements or 2 cache lines

Virtual memory (1)

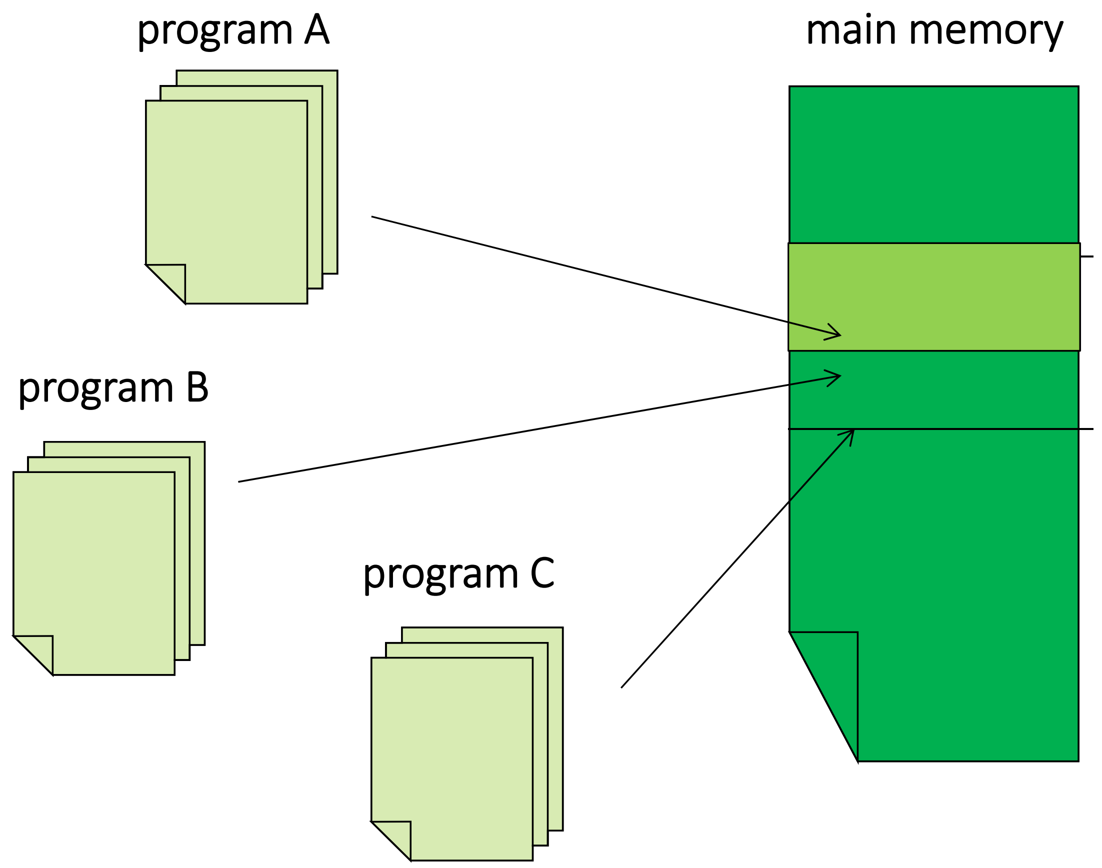
- If we run a very large program or a program that accesses very large data sets, all of the instructions and data may not fit into main memory
  - E.g. multitasking operating systems
- In a multitasking system many running programs must share the available main memory
- This sharing must be done in such a way that each program's data and instructions are protected from corruption by other programs
- Virtual memory functions as a cache for secondary storage

Virtual memory (2)

- It exploits the principle of spatial and temporal locality.
- It only keeps the active parts of running programs in main memory.
- The parts that are idle are kept in a block of secondary storage called **swap space**
- Virtual memory operates on blocks of data and instructions. These blocks are commonly called **pages**
  - fixed page size that currently ranges from 4 to 16 kilobytes.

Virtual memory (3)

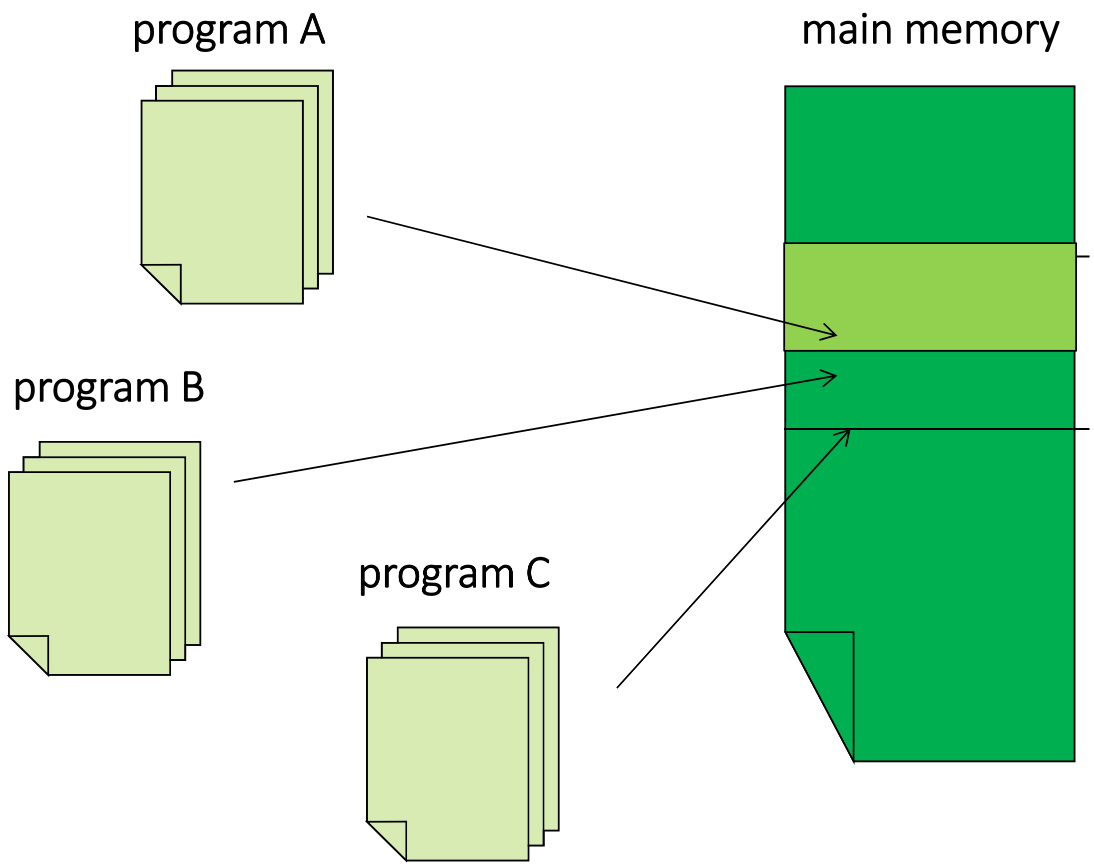
How to assign physical memory addresses to pages?



Virtual memory (3)

How to assign physical memory addresses to pages?

- When a program is compiled its pages are assigned *virtual* page numbers.
- When the program is run, a table is created that maps the virtual page numbers to physical addresses.
- A **page table** is used to translate the virtual address into a physical address.



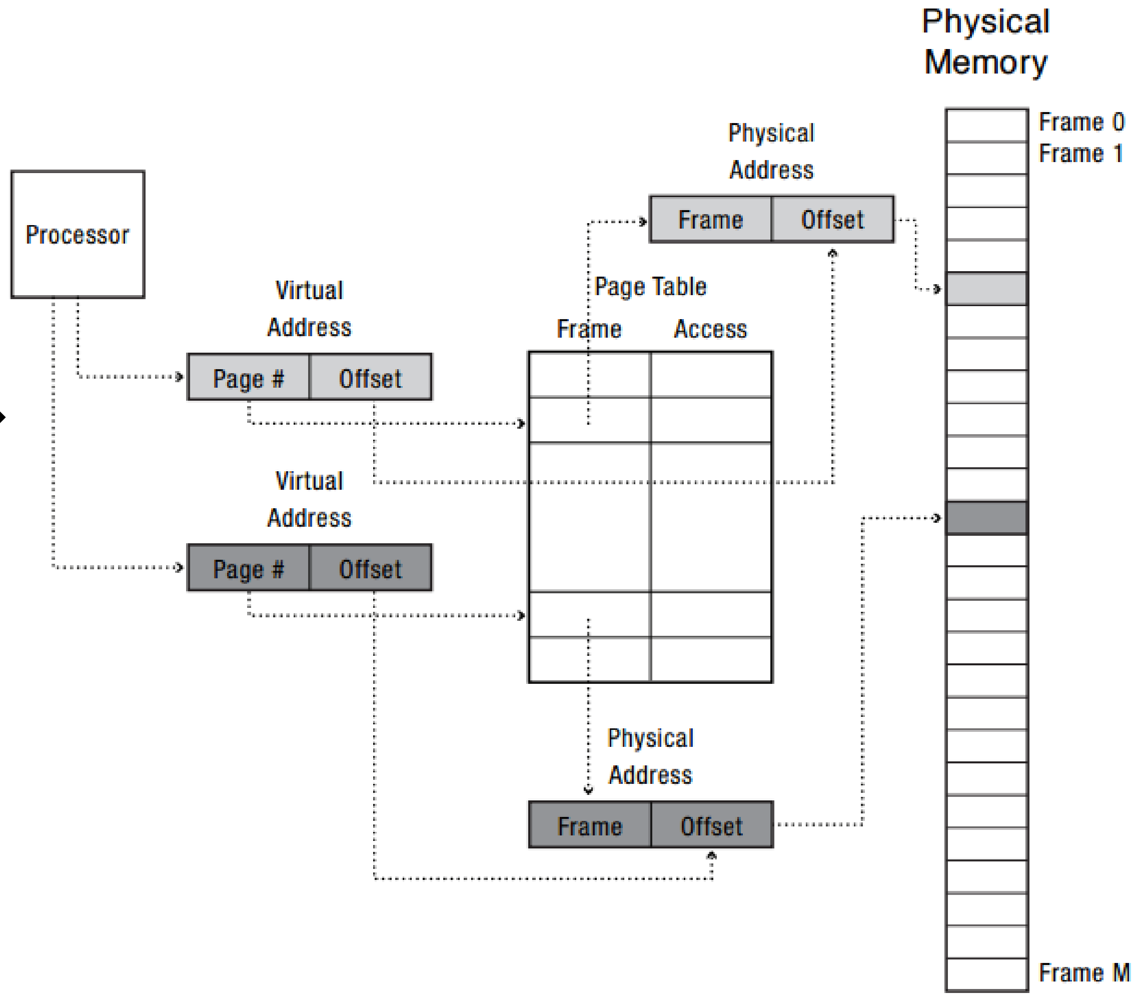
Virtual memory – Page table

- Suppose our addresses are 32 bits and our pages are 4 kilobytes (4096 bytes)
- Then each byte in the page can be identified with 12 bits (2^{12}= 4096)
- We can use the low-order 12 bits of the virtual address to locate a byte within a page
- The remaining bits of the virtual address can be used to locate an individual page

Virtual Address									
Virtual Page Number					Byte Offset				
31	30	$\cdots$	13	12	11	10	$\cdots$	1	0
1	0	$\cdots$	1	1	0	0	$\cdots$	1	1

Virtual memory – Page table

Virtual Address								
Virtual Page Number				Byte Offset				
31	30	⋯	13	12	11	10	⋯	1	0
1	0	⋯	1	1	0	0	⋯	1	1



Translation-lookaside buffer (TLB)

- Using a page table has the potential to significantly increase each program's overall run-time.

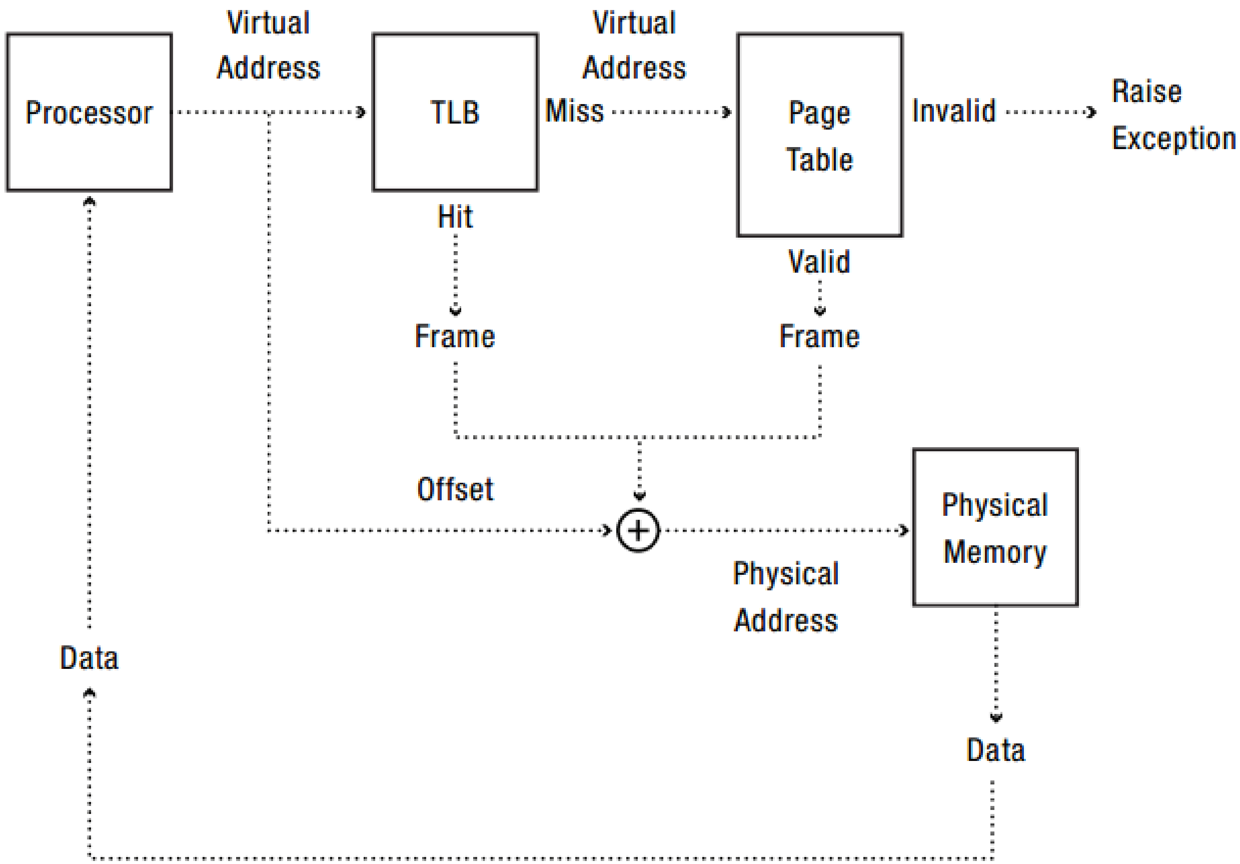
- A special address translation cache in the processor → Translation-lookaside buffer (TLB)

- It caches a small number of entries (typically 16–512) from the page table in very fast memory.

- Cost of translation =

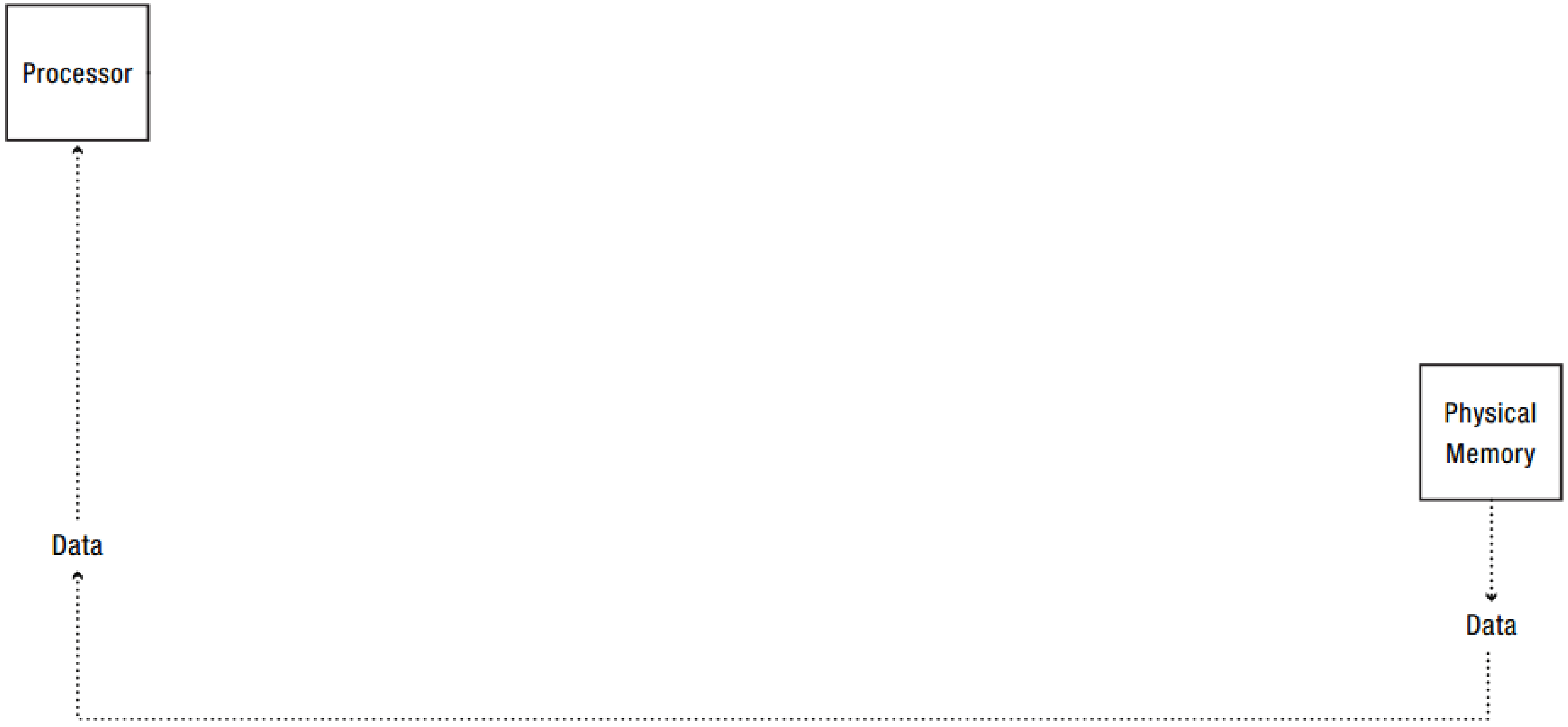
  Cost of TLB + Pr(TLB miss) * cost of page table lookup

Translation-lookaside buffer (TLB)

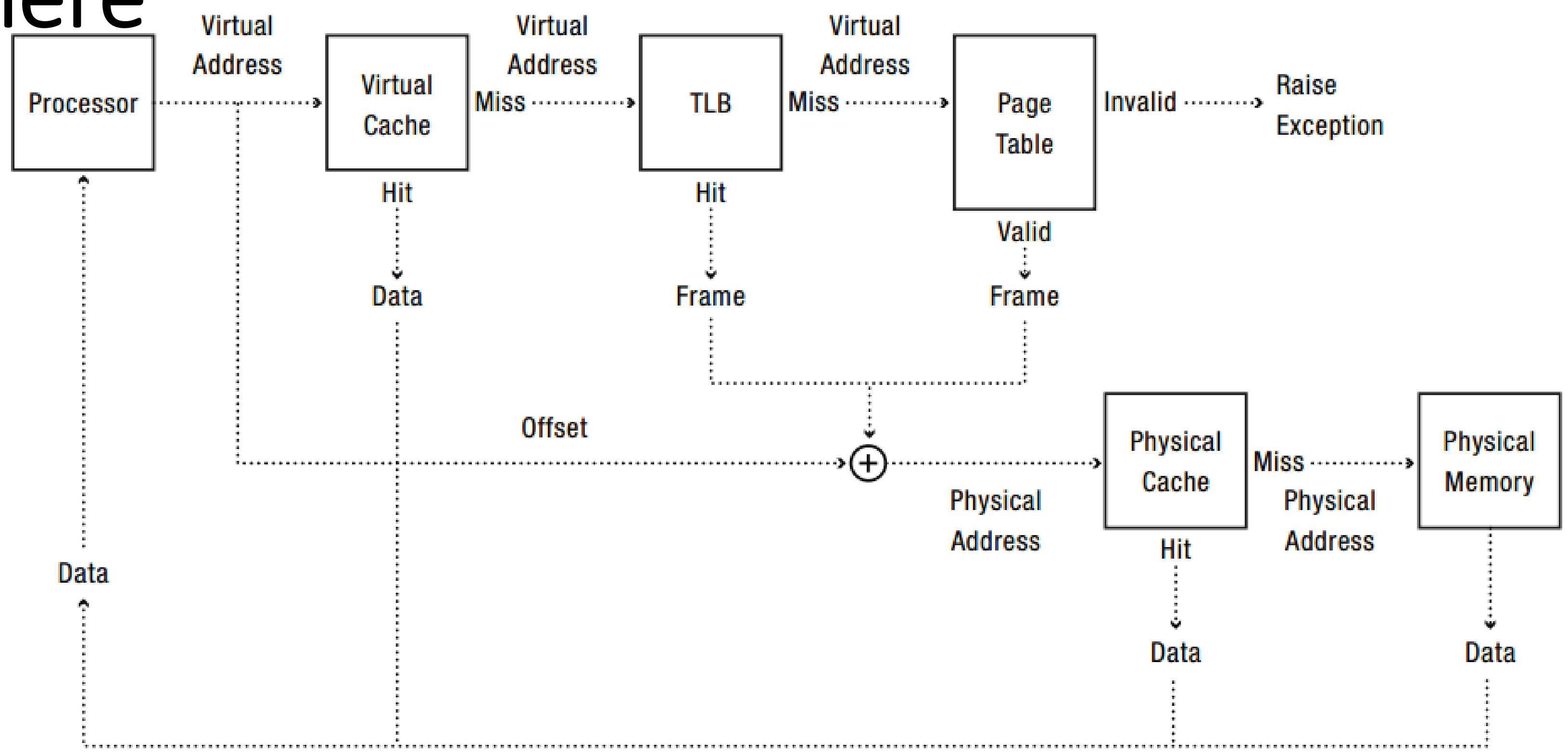


Where do we stand?

From here



To here


Processor
Virtual Address
Virtual Cache
Virtual Address
Miss
TLB
Virtual Address
Miss
Page Table
Invalid
Raise Exception
Hit
Data
Hit
Frame
Valid
Frame
Offset
Physical Address
Physical Cache
Miss
Physical Address
Physical Memory
Hit
Data
Data
Data

Instruction Level Parallelism (ILP)

- Attempts to improve processor performance by having multiple processor components or **functional units** simultaneously executing instructions.

- **Pipelining** - functional units are arranged in stages.

- **Multiple issue** - multiple instructions can be simultaneously initiated

- Both approaches are used in virtually all modern CPUs

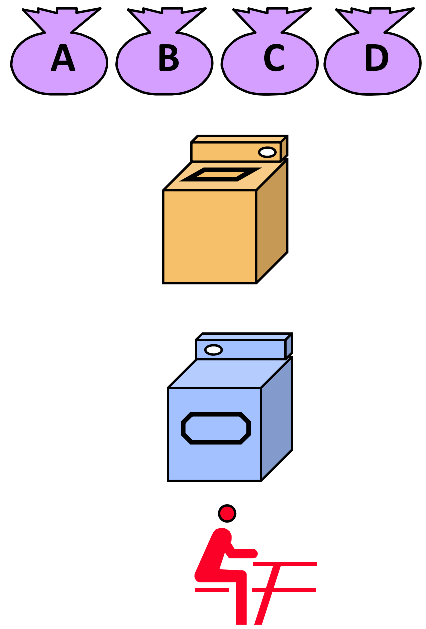
Instruction Level Parallelism (2)

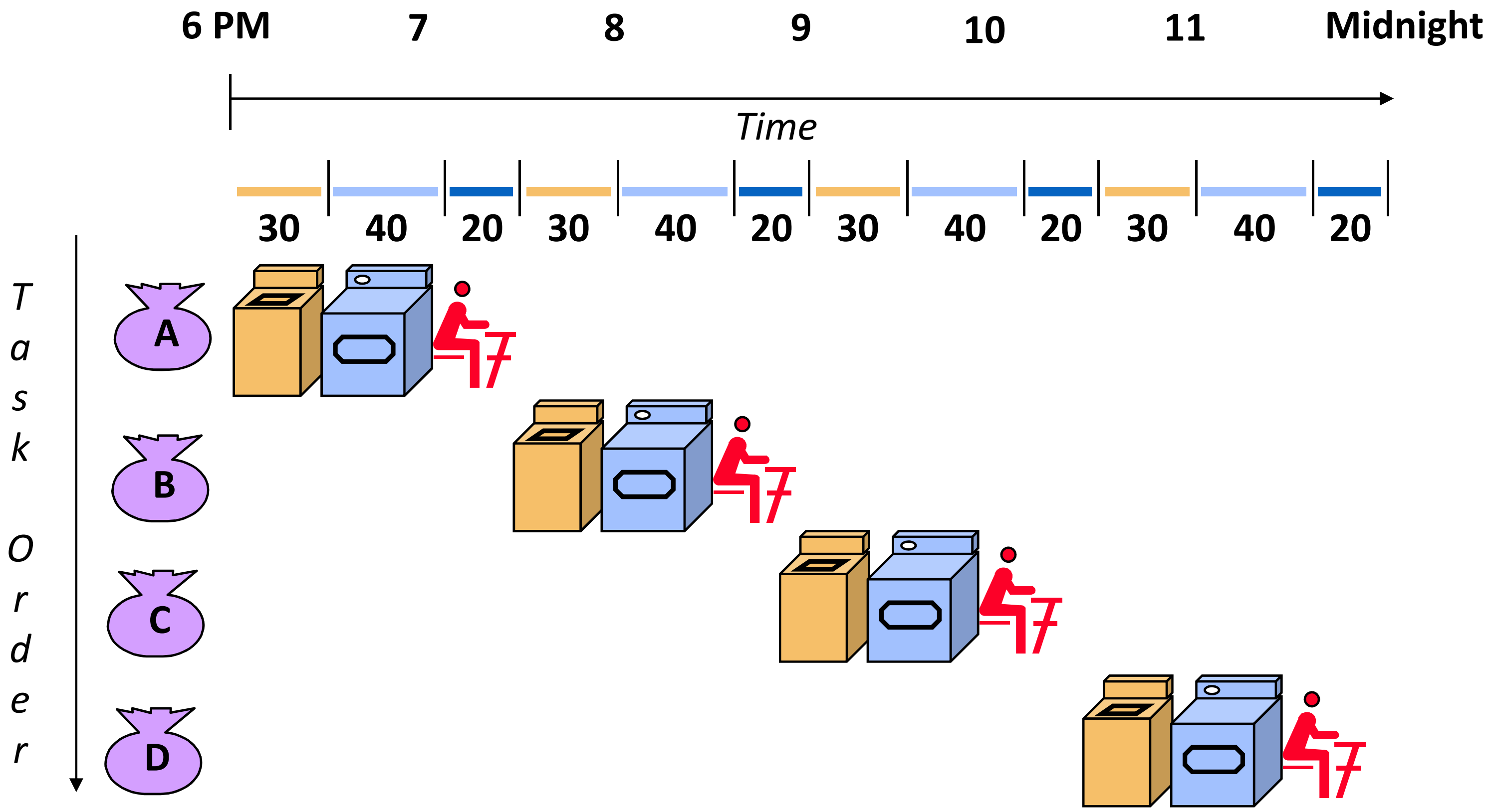
- **Pipelining** - functional units are arranged in stages.

- **Multiple issue** - multiple instructions can be simultaneously initiated.

What Is Pipelining

- Laundry Example
- Ann, Brian, Cathy, Dave
  each have one load of clothes
  to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- “Folder” takes 20 minutes

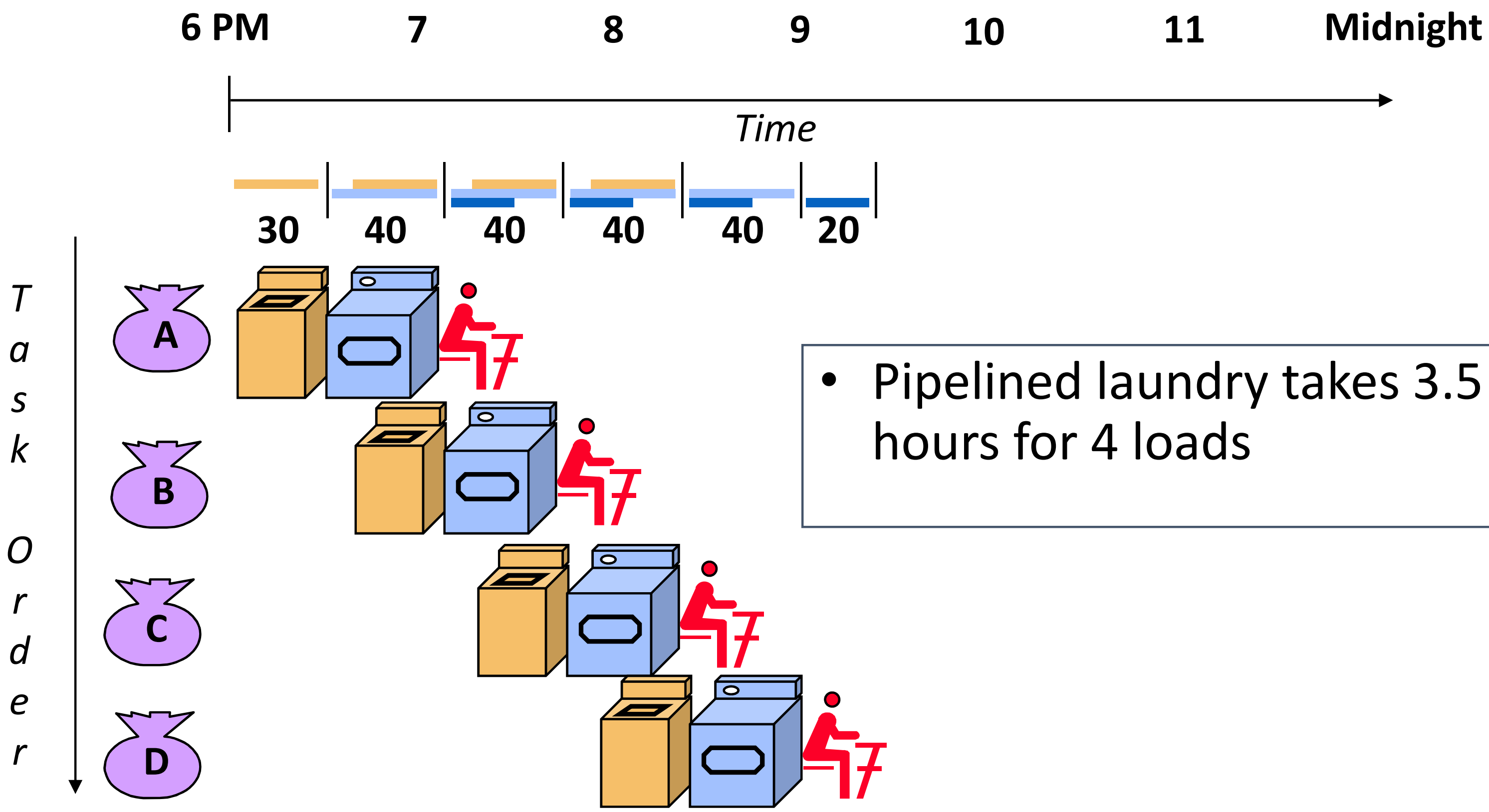




Sequential laundry takes 6 hours for 4 loads

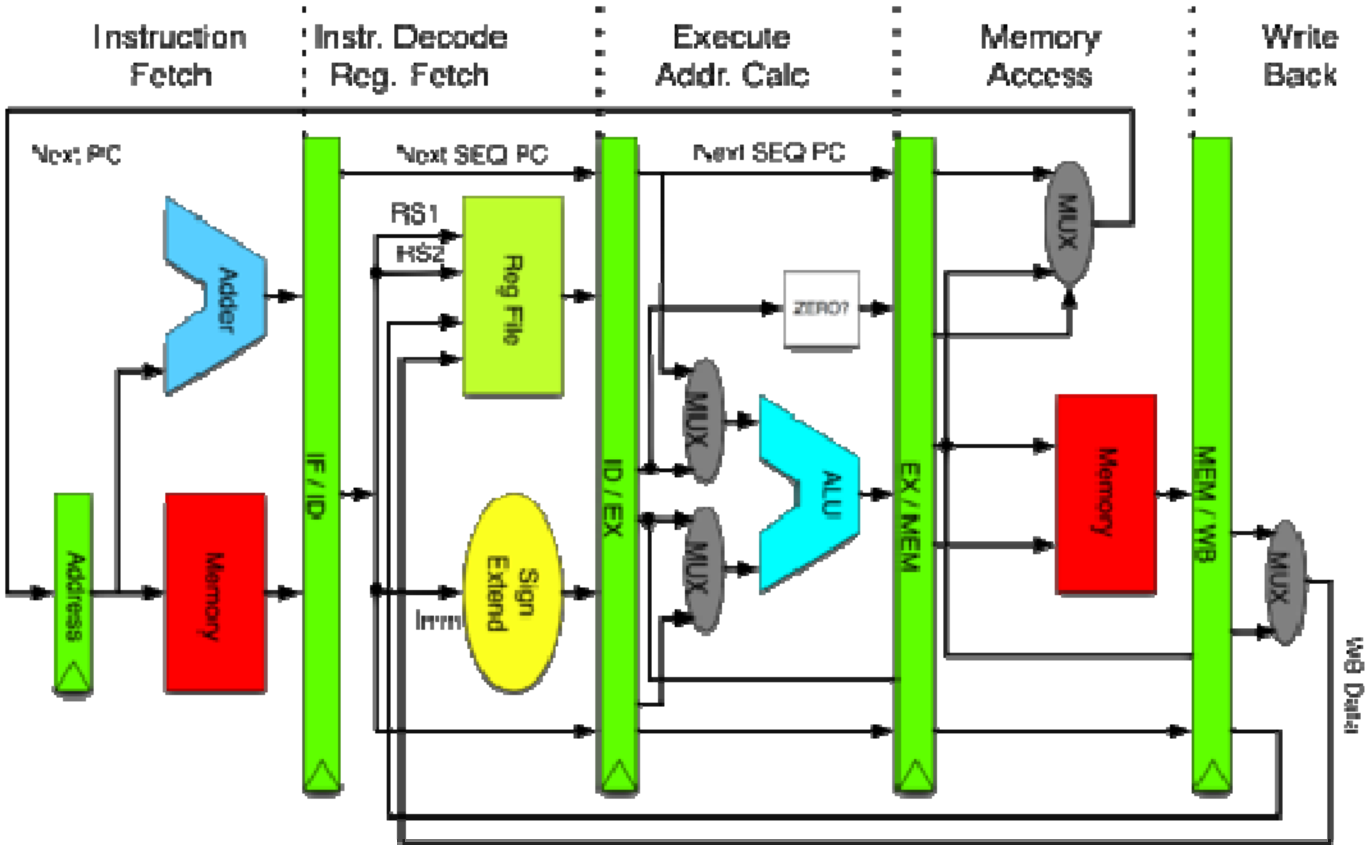
If they learned pipelining, how long would laundry take?

Start work ASAP



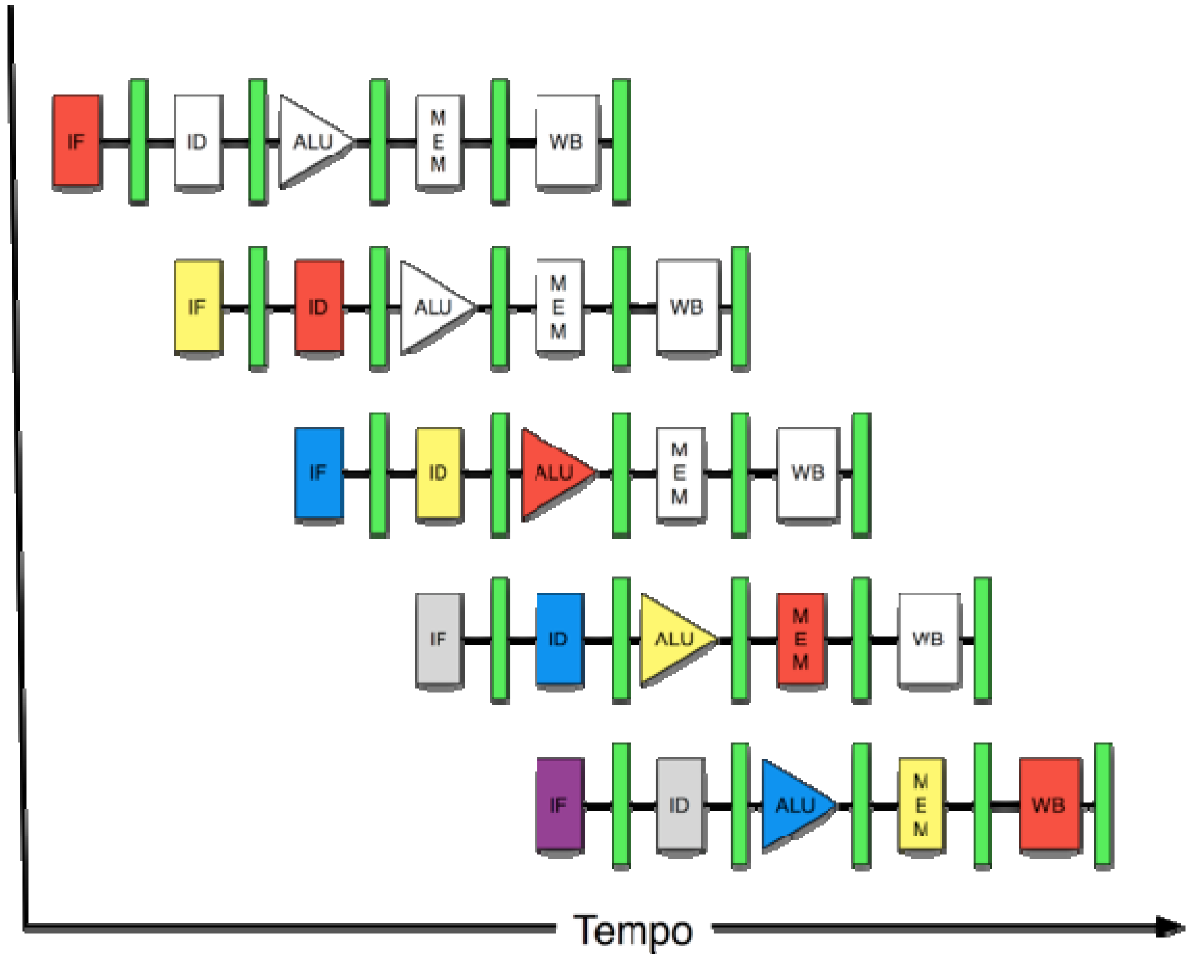
- Pipelined laundry takes 3.5 hours for 4 loads

Pipeline



Instruction Fetch
Instr. Decode Reg. Fetch
Execute Addr. Calc
Memory Access
Write Back
Next PC
Next SEQ PC
Next SEQ PC
RS1
RS2
Adder
Reg File
ZERO?
MUX
MUX
MUX
ALU
MUX
Memory
Memory
Address
IF / ID
ID / EX
EX / MEM
MEM / WB
Sign Extend
Imm
WB Data

Pipeline



Pipelining example

- Add the floating point numbers 9.87×10^4 and 6.54×10^3
- The steps are

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	9.87×10^4	6.54×10^3	
2	Compare exponents	9.87×10^4	6.54×10^3	
3	Shift one operand	9.87×10^4	0.654×10^4	
4	Add	9.87×10^4	0.654×10^4	10.524×10^4
5	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
6	Round result	9.87×10^4	0.654×10^4	1.05×10^5
7	Store result	9.87×10^4	0.654×10^4	1.05×10^5

Pipelining example

- If each of the operation takes 10^{-9} sec, the addition operation will take seven nanoseconds

- The following code takes 7000 nanoseconds

```
float x[1000], y[1000], z[1000];
. . .
for (i = 0; i < 1000; i++)
   z[i] = x[i] + y[i];
```

Pipelining example

- Divide the floating point adder into 7 separate pieces of hardware or functional units.

- First unit fetches two operands, second unit compares exponents, etc.

- Output of one functional unit is input to the next.

Pipelining example

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

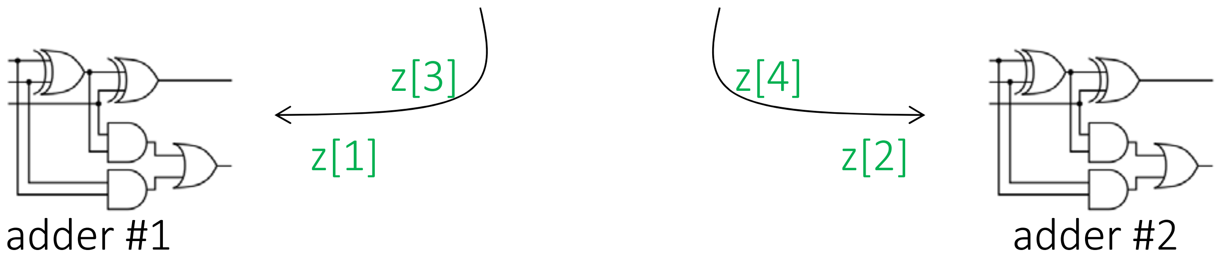
Pipelining example

- One floating point addition still takes 7 nanoseconds.
- But 1000 floating point additions now takes 1006 nanoseconds!
- However things are not that good all times
  - Data dependencies
  - Branch misprediction

Multiple Issue (1)

- Pipelines improve performance by taking individual pieces of hardware or functional units and connecting them in sequence
- Multiple issue processors replicate functional units and try to simultaneously execute different instructions in a program.

```
for (i = 0; i < 1000; i++)
    z[i] = x[i] + y[i];
```



Multiple Issue (2)

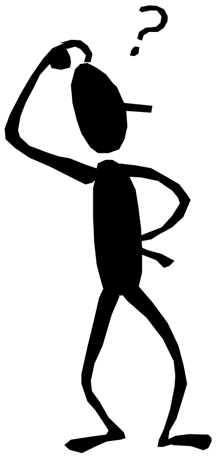
- If the functional units are scheduled at compile time, the multiple issue system is said to use **static** multiple issue.

- If they're scheduled at run-time, the system is said to use **dynamic** multiple issue.

- A processor that supports dynamic multiple issue is sometimes said to be **superscalar**.

Speculation (1)

- In order to make use of multiple issue, the system must find instructions that can be executed simultaneously.



- In speculation, the compiler or the processor makes a guess about an instruction, and then executes the instruction on the basis of the guess.

Speculation (2)

```
z = x + y ;
if ( z > 0)
    w = x ;
else
    w = y ;
```

If the system speculates incorrectly,
it must go back and recalculate w = y.

Speculation (3)

- Speculative execution must allow for the possibility that the predicted behavior is incorrect

- If the compiler does the speculation, it will usually insert code that tests whether the speculation was correct, and, if not, takes corrective action

- If the hardware does the speculation, the processor usually stores the result(s) of the speculative execution in a buffer.

Hardware multithreading (1)

- ILP can be very difficult to exploit
  - A program with a long sequence of dependent statements offers few opportunities

E.g. Fibonacci numbers → no opportunity for simultaneous execution

```
f[0] = f[1] = 1;
for (i = 2; i <= n; i++)
   f[i] = f[i-1] + f[i-2];
```

Hardware multithreading (2)

- There aren’t always good opportunities for simultaneous execution of different threads.

- Hardware multithreading provides a means for systems to continue doing useful work when the task being currently executed has stalled.
  - Ex., the current task has to wait for data to be loaded from memory.

Hardware multithreading (3)

- **Fine-grained** - the processor switches between threads after each instruction, skipping threads that are stalled.
  - <u>Pros</u>: potential to avoid wasted machine time due to stalls.
  - <u>Cons</u>: a thread that’s ready to execute a long sequence of instructions may have to wait to execute every instruction.

Hardware multithreading (4)

- **Coarse-grained** - only switches threads that are stalled waiting for a time-consuming operation to complete.
  - Pros: switching threads doesn't need to be nearly instantaneous.
  - Cons: the processor can be idled on shorter stalls, and thread switching will also cause delays.

Hardware multithreading (5)

- **Simultaneous multithreading** (SMT) - a variation on fine-grained multithreading.

- Allows multiple threads to make use of the multiple functional units.