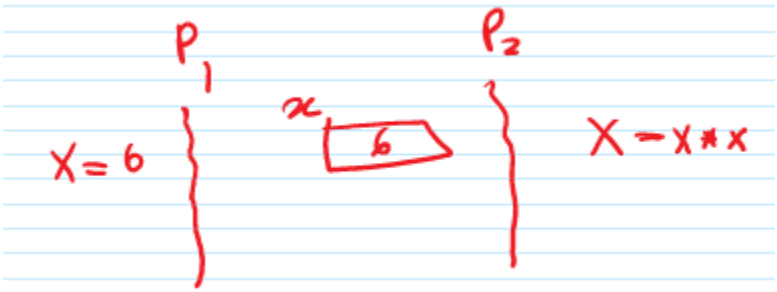
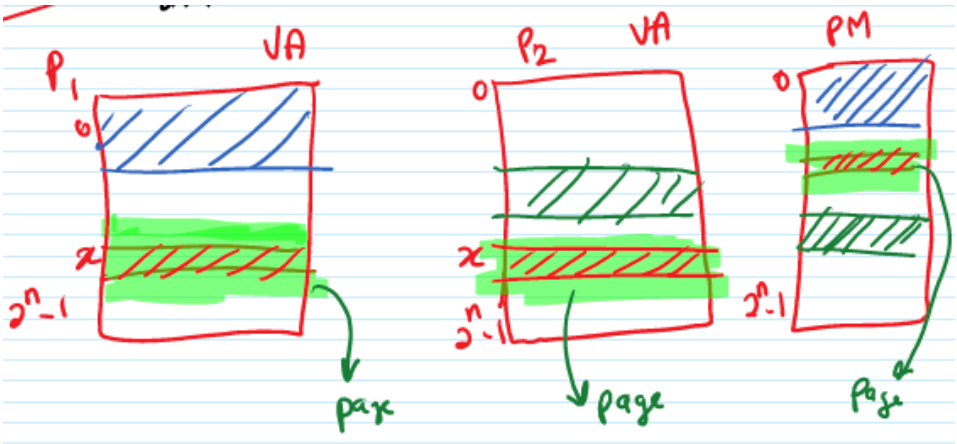
Lecture 7: Concurrent Programming Intro

Lecture 7, Part 1: Inter Process Communication (IPC)

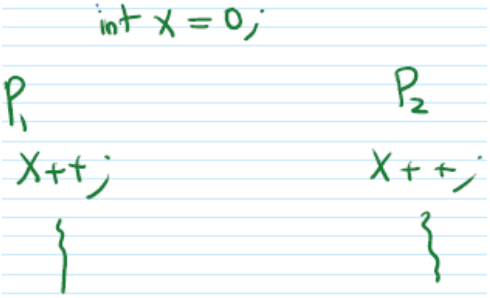
1. Process can communicate using Files.
    - **Example**: communication between a parent and child process.
    - Child inherits file pointer from parent and can use one File for the parent to write into and one for the child to read from.
2. OS supports something called a pipe.
    - Corresponds to 2 file descriptors (int fd [2]).
3. Process could communicate through variables that are shared between them.
    - Shared variable
    - Private variables



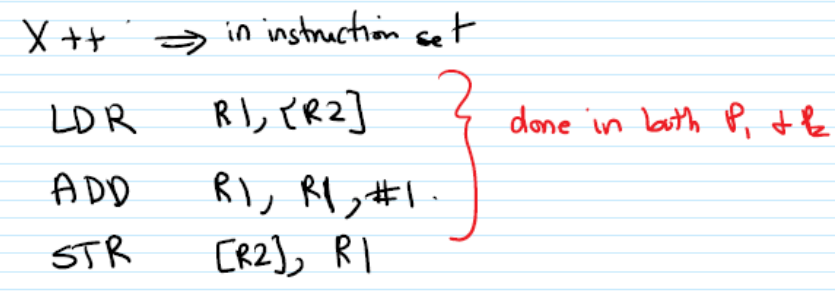
    - Problem→ address translation to protect one process from another.
    - Solution → OS provides the mechanism.



4. Processes could possibly communicate by sending and receiving messages to each other
    - OS provides the mechanism.
    - Program with shared variables:
        - Consider a 2-process program in which both processes increment a shared variable.



- Question → what is the value of x? → 2
- Problem → x = 1 or 2?



- Why can't X be 1?
  - P1 loads X into R1, increments R1.
  - P2 loads X into register before P1 stores a new value into X.
- Need to synchronize processes that are interacting using shared variables.
- Critical section→ part of a program where a shared variable is accessed.

5. Sometimes process don't explicitly share values to cooperate.

   Example. Synchronization. P1 reads (2 matrices), P2 multiples (2 matrices), P3 writes (a matrix). Process 2 should not start work until Process 1 finishes reading. This is called process synchronization. Synchronization primitives: mutex lock, semaphore, barrier.

Critical Sections:

- Must synchronize process so that they access shared variables one at a time one a critical section.
  - Mutual Exclusion
- Mutex Lock→ synchronization primitives
  - Acquire Lock (L)→ 1. Done before a critical section code; 2. Returns a value when safe for processes to enter critical section.
  - Release Lock→ 1. Done after the critical section; 2. Allows another process to acquire lock.

Implementing Lock:

```
int L = 0;          // 0 => lock is available
                    // 1 => lock is in use.
AcquireLock(L):
    while (L==1);
    L = 1;
ReleaseLock(L)
    L = 0;
```

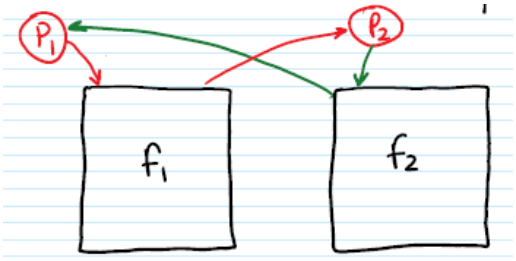
Lecture 8: Concurrent Program Continued

Lecture 8, Part 1: Inter Process Communication (IPC)

- Concurrent programming is about programs with multiple flows of control.
- One way to setup a concurrent program might be to set it up as a program that runs as multiple processes to achieve a common goal.
- To cooperate, process must somehow communicate.

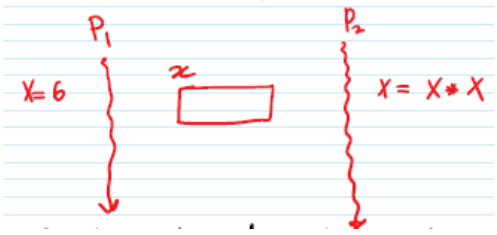
Process Communication:

1. Process can communicate using files.
    a. Example: communication between a parent process and child process.
        i. Parent process creates 2 files before forking child process.
        Why? It's conceivable the parent process may want to be able to send data to the child and the child may want to send the data to the parent.
        ii. Child inherits file descriptors from parent, and they share the file pointers.
        iii. Can use one file for parent to write to and child to read from and the other file for the child to write to and parent to read from.

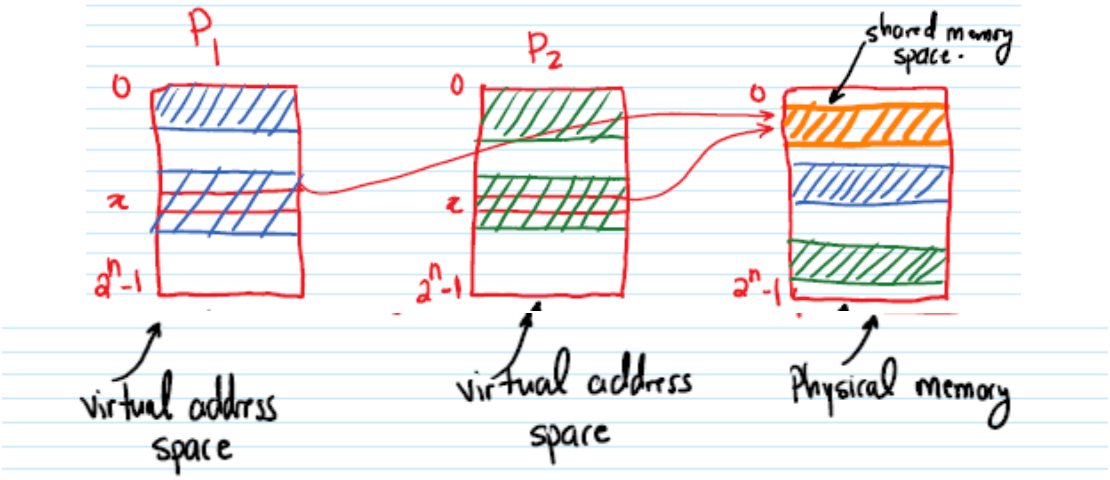


    b. Problem→ Files ~ Hard Disk (both very high overhead)
2. Pipe → technical term
    a. Something supported by all operating systems.
    b. Looks superficially like process can communicate using files.
    c. A pipe provides two file descriptors but not using hard disk.
        int fd [2];
    c. Read from fd[0] accesses data written to fd[1] in FIFO (First in First Out) order and vice versa.

3. Processes could communicate through variables that are shared between them
    a. There will be some variables that can be used between process P1 and P2 for the purposes of communication and we will refer to them as shared variables and the variables of the process that are not shared, we will refer to them as private variables.



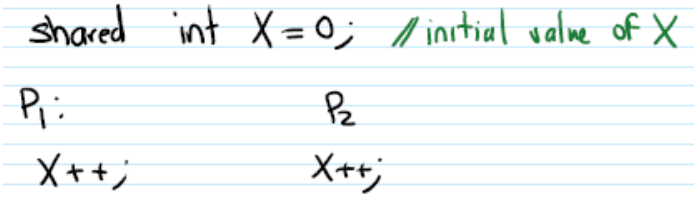
b. Problem: Recall that address translation is used to protect one process from another.



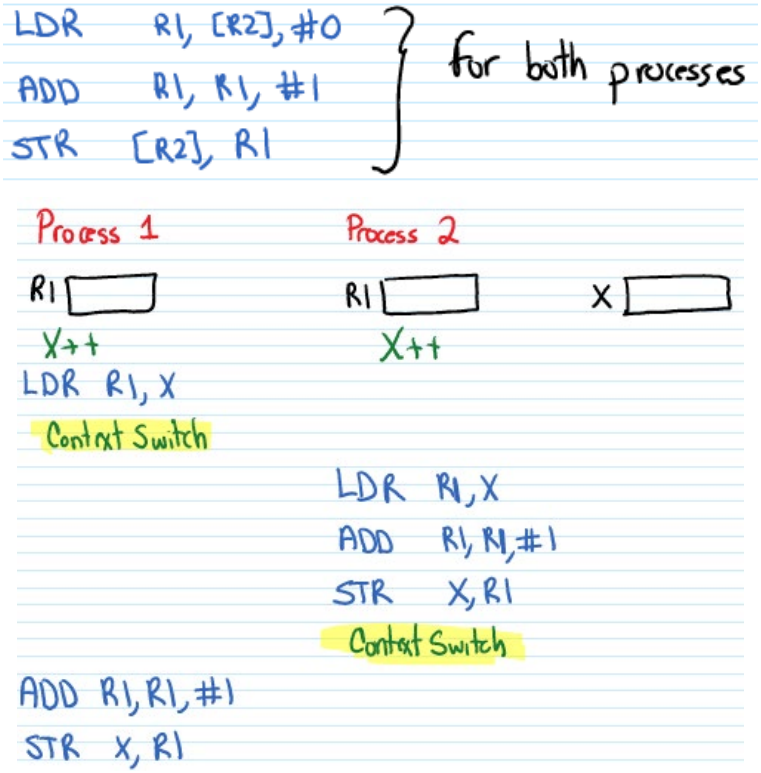
4. Processes could communicate by sending and receiving messages to each other.
    a. Special support for these messages.
5. Sometimes processes may need to communicate but not explicitly communicate values to cooperate.
    a. Processes may just have to synchronize their activities.
    b. Example: Process 1 reads 2 matrices, Process 2 multiplies them, Process 3 write the result to a matrix.
    c. Process 2 should not start work until Process 1 finishes reading.
    d. This is called process synchronization.
    e. To do synchronization, some kind of mechanism has to be provided.
        i. Synchronization primitives:
        Example: mutex lock, semaphores, barrier

Programming with Shared Variables:

- Consider a 2-process program in which both processes increment a shared variable.



- What is the best value of X after this?
- Complications: remember that X++ compiles into something like this:

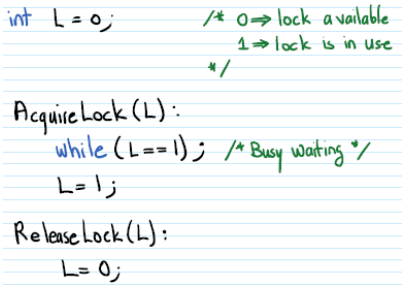


- Final value of X could be 1!!
  - P1 loads X into R1, increments R1.
  - P2 loads X into register before P1, stores new value into X
  - Net result: P1 stores 1, P2 stores 1
- Key takeaway→ it may be necessary to synchronize processes that are interacting using shared variables.
- Problem arises when 2 or more processes try to update a shared variable.
- Critical section→ parts of concurrent programs where shared variables are accessed as described.

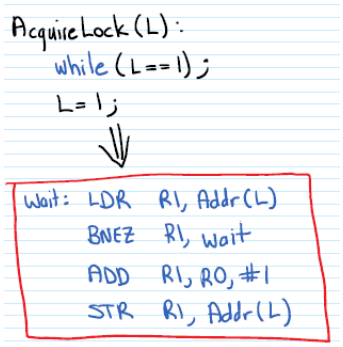
Critical Section Problem: Mutual Exclusion

- Must synchronize processes so that they access a shared variable one at a time in a critical section. This is called mutual exclusion.
- Mutex Lock→ a synchronization primitive.
- Looking at Mutex Lock as a data structure:
    - Acquire Lock (L)
        - Done before critical section of the code
        - Returns when safe for process to enter critical section.
    - Release Lock (L)
        - Done after critical section.
        - Allows another process to acquire lock.

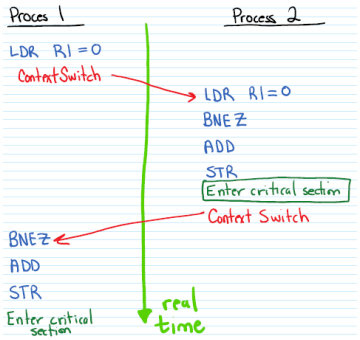
Implementing a Lock:



Why this implementation fails:



- Assume that lock L is currently available (L=0) and that 2 processes, P1 & P2 try to acquire the lock L.



- Therefore, this implementation allows processes P1 & P2 to be in a critical section together.

Lecture 8, Part 2: Concurrent Programming

Busy Wait Lock Implementation:

- Hardware support will be useful to implement a lock.
- Example: Test & Set Instruction
    - A machine instruction with one memory opened.

```
Test&Set  Lock
  tmp = Lock
  Lock = 1
  return tmp
```

    - Where these 3 steps happen atomically or indivisibly.
    - That is, all 3 happen as one operation (with nothing happening in between).
    - This is an example of synchronization primitives.
    - In general instruction which describe this kind of purpose can be described as:
    - Atomic Read-Modify-Write (RMW) instructions

Busy Wait Lock with Test & Set:

- Lock variable declared as int L;
    - L == 0 means that the lock is available
    - L == 1 means that the lock is in use
- Acquire Lock (L)

```
while (Test&Set(L))   /* busy wait */
// busy wait until L has been Test&Set from 0 to 1
// i.e. the return value from Test&Set is 0
```

- Release Lock (L)

```
L=0;   ⇒   STR L, R0;
```

Example

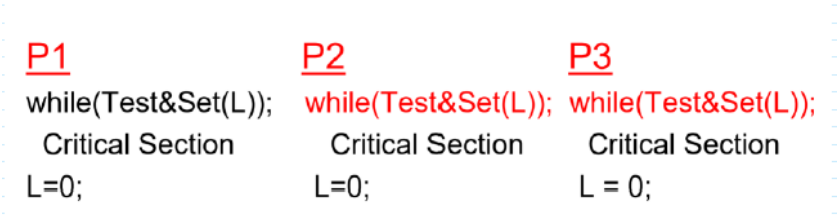
P1	P2	P3
while(Test&Set(L));	while(Test&Set(L));	while(Test&Set(L));
Critical Section	Critical Section	Critical Section
L=0;	L=0;	L = 0;

- Suppose that process P1 is in its Critical Section.
- Processes P2 & P3 are trying to acquire lock in order to enter their critical section.

P1	P2	P3
while(Test&Set(L));	while(Test&Set(L));	while(Test&Set(L));
Critical Section	Critical Section	Critical Section
L=0;	L=0;	L = 0;

- The Lock L==1 due to Test & Set (L) that was executed by P1.

- When P2 & P3 execute Test & Set (L), they overwrite the 1 and get a return value of 1.
- The situation changes when P1 exits its critical section.



More on Locks:

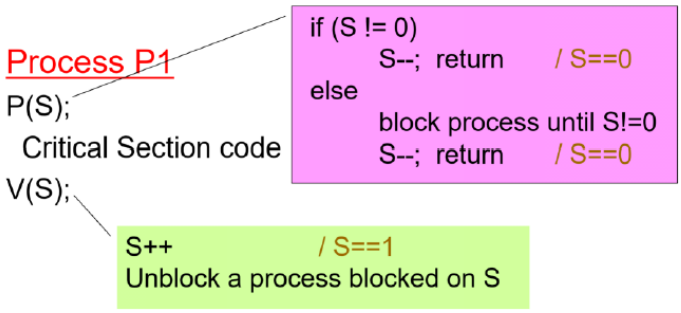
- Other names for this kind of lock:
  - Mutex
  - Mutex lock
  - Spin wait lock
  - Spin lock
  - Busy wait lock
- There are also locks where instead of busy waiting, an unsuccessful process gets blocked by the operating system.
- Block → Moved into the waiting state until the lock becomes available.

Semaphores:

- A more general synchronization mechanism than the lock.
- Operations→ P(wait) and V(signal)
- P(s)→ 's' is a semaphore—some kind of data structure
  - If 's' is non-zero, decrements 's' and returns.
  - Else the process that executed P(s) gets blocked until 's' becomes non-zero, at which time the process is restarted
    - After restarting, decrements 's' and returns.
- V(s)
  - Increments 's' by 1.
  - Subsequently, if there are any processes blocked waiting for 's', then one of them is restarted.

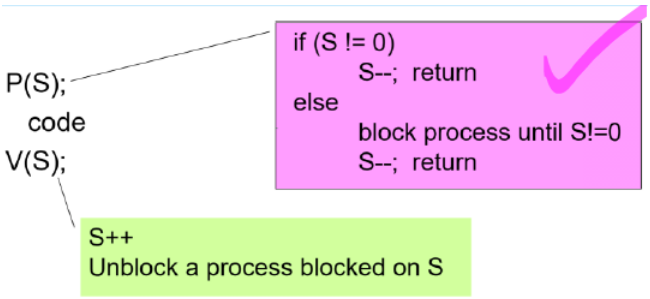
Semaphores:

- Initialize a semaphore s = 1
- Surround each critical section in the concurrent programs by calls to P(s) (before the critical section) and V(s) (after the critical section).



Semaphore Examples:

- The previous examples showed how a semaphore can be used to do the work of a mutex lock.
- Semaphores can be used for other purposes as well.
    - Example 1: initialize semaphore S = 10.
        - Suppose that processes surround code by P(s) and V(s) as with the previous example.



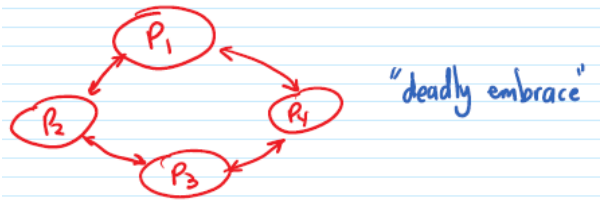
        - 10 processes will be allowed to proceed.
        - Processes beyond that will be blocked until one of the first 10 executes V(s).
    - Example 2: Consider our concurrent program where process P1 reads 2 matrices; process P2 multiplies them and process P3 outputs the product.
        - initialize semaphores S1 = 0, S2 = 0.

Process P1	Process P2	Process P3
	$P(S_1)$	$P(S_2)$
Read A[ ], B[ ]	C[ ] = A[ ] * B[ ]	Write C[ ]
$V(S_1)$	$V(S_2)$	

Lecture 8, Part 3: Problems with Concurrent Programming

Deadlock:

- P1: AcquireLock(L); AcquireLock(L);
    - Suppose that the first AcquireLock(L) succeeds
        - P1 is then waiting for something (release of lock that it is holding) that will never happen.
        - This is a simple example of a general problem called deadlock.
            - Deadlock is caused by a cycle of processes waiting for resources help by others while holding resources needed by others.



Classical Problems:

- Producers-Consumer problem or Bounded Buffer problem.
    - Producer process makes things and puts them into a fixed size shared buffer(array).
    - Consumer process takes things out of shared buffer and uses them.
- Problem:
    - Must ensure that producer doesn't put into full buffer or consumer takes out of an empty buffer.
    - Must also treat buffer as a critical section.

Pseudo-Code Solutions:

```
sharedBuffer [0....., N-1]
Producer: repeatedly
    Produce x;  ①if (buffer is full) wait for consumption
    Buffer[i++] = x  ③ signal consumer
```

```
Consumer: repeatedly
    ② If (buffer is empty) wait for production
      y = Buffer[--i]
      Consume y  ④ signal producer
```

Dining Philosopher's Problem:

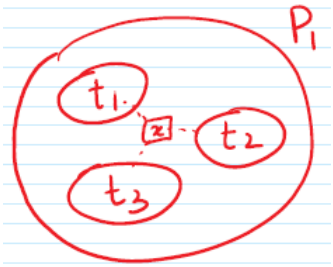
- N philosophers are sitting around a circular table with a plate of food in front of each philosopher and a fork between each two philosophers.
- Philosopher does: repeatedly
  - Eat (using 2 forks)
  - Think
- Problem: avoid deadlock, be fair

Threads:

- Thread of control in a process
  - "light weight" process
- Concept of weight is related to:
  - Time for creation
  - Time for contact switch
  - Size of context
- Recall: Process as a Data Structure: Process context:
  - Test, Data Stack, Heap
  - Data stored in a hardware
  - Other information maintained by the OS
    - Process, parent and user identifiers
    - Memory management information: Page Table
    - File related info: Open files, File pointers

Threads vs Processes:

- Thread context
  - Thread ID
  - Stack
  - Stack pointer, PC, GPR values
- So thread context switching can be much faster than process context switch.
- Question: Where are the text, data and heap associated with a thread are going to come from?
- Many threads in the same process share parts of that process context.
  - Virtual address space (other than the stack).
- Therefore, picture looks like this



- So, threads in the same process share variables that are not stack allowed.

Thread Implementation:

- Could either be supported in the operating system or by a library.
- Pthreads→ POSIX thread library
    - Int pthread_create = fork
    - Pthread_attr
    - Pthread_join
    - Pthread_exit
    - Pthread_detach

Synchronization Primitives:

- Mutex locks:
    - Int pthread_mutex_lock(pthread_mutex_t * mutex)
    - Pthread_mutex_unlock
- Semaphores:
    - Sem_init
    - Sem_wait→ P
    - Sem_post→ V

Lecture 9: Pipelining in Details

Lecture 9, Part 1: Review of Pipelining with More Details

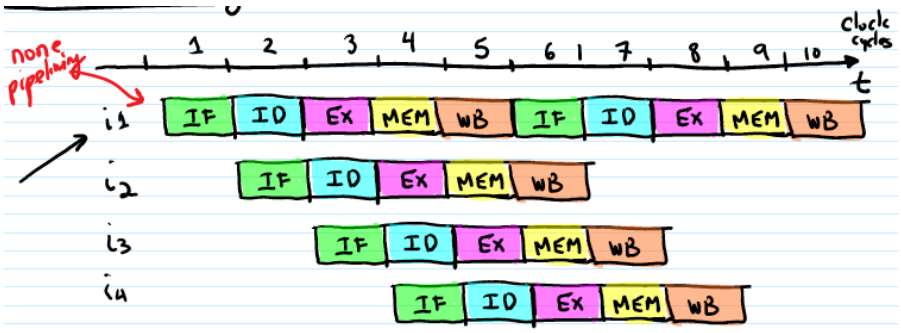
- Any kind of microcontroller will have the same set of instruction cycles.
- IF→ instruction fails
- ID→ instruction decode
- EX→ execution (ALU)
- MEM→ memory access
- WB→ write block

Each of these has a dedicated hardware

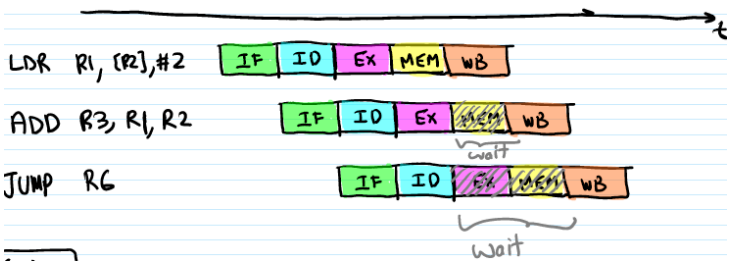
Performance of a Processor:

- Two ways:
  - Execution time of a single instruction (3, 4, 5 cycles)
  - Throughput of instruction execution.
- Cycles Per Instruction (CPI):
  - Average # of instructions per cycle
    - Therefore, CPI between 3—5 → without pipelining
- Pipelining:
  - Why keep Fetch hardware idle while we are decoding.

Processor Pipelining:

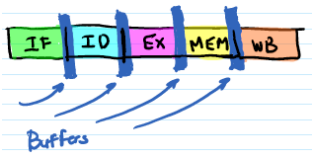


- Execution time of each instruction is still 5 cycles, but the throughput is 1 instruction per cycle.
  - Initial pipeline fill time of 4 cycles
- As we said, not every instruction is 5 cycles.



Buffering:

- We need to save the state of all instruction hardware for pipelining.



- Pipeline stages→ IF, ID, EX, MEM, WB
  - 5 stage pipeline or a pipeline of depth 6
- Assume that the time delay through each stage is the same→ 1 cycle
- Pipeline speedup = $time_{non\text{-}pipelined}/time_{pipelined}$

- Calculate how much time it takes to run a program involving 'n' executions.
  - Non-pipelined: 5n cycles
  - Pipelined processor: 4 + n cycles
  - Speed up = 5n/(4+n)
  - $\lim_{n\to\infty}(\frac{5n}{4+n}) = 5$

Pipeline Speedup:

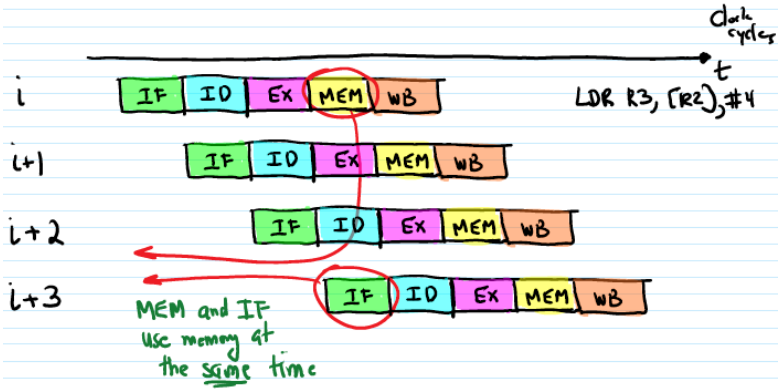
- A pipeline with 'p' stages could give a speed up of 'p' component to a non-pipelined processor that takes 'p' cycles for each instruction.
  - Assuming on every 'p' clock cycles, an instruction completes execution.

Lecture 9, Part 2: Pipeline Hazards

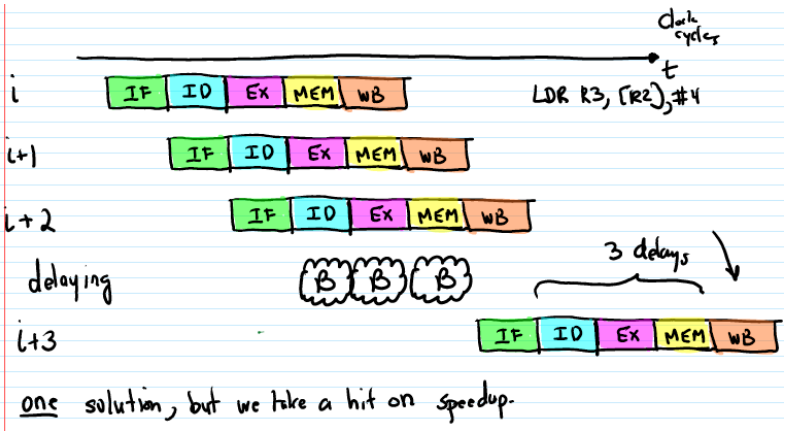
- Hazard→ a dangerous situation
- Pipeline hazard→ a situation where an instruction cannot proceed through the pipeline as it should (or as we have shown it).

1. Structural Hazards→ When two or more instructions in the pipeline need to use the same resource of the same time.
2. Data Hazard→ When an instruction depends on the data result of a prior instruction that is still in the pipeline.
3. Control Hazard→ A hazard that arises due to control transfer instructions.

Structural Hazards:

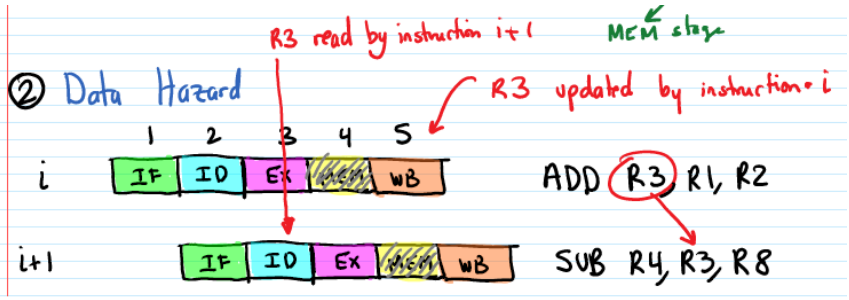


- Solution 1: Insert delays between the instruction cycles

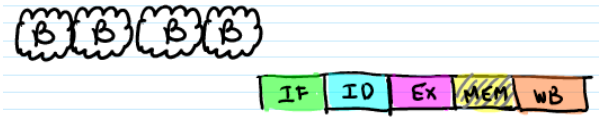


- Solution 2:
  - Design main memory so that it can handle 2 memory requests at the same time.
  - Double ported memory.
  - Separate instruction cache (for use by the IF pipeline stage) and data cache (MEM stage).

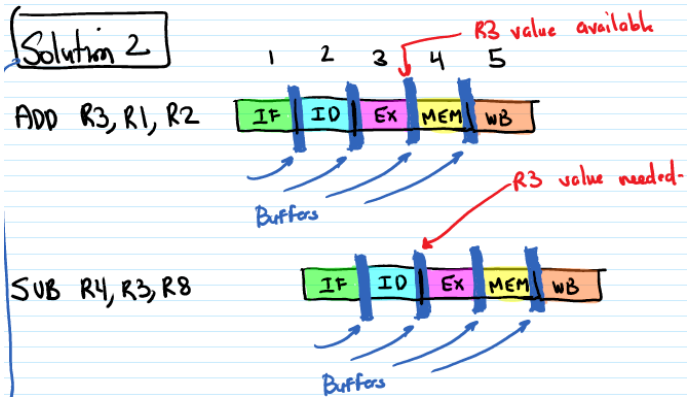
Data Hazards:



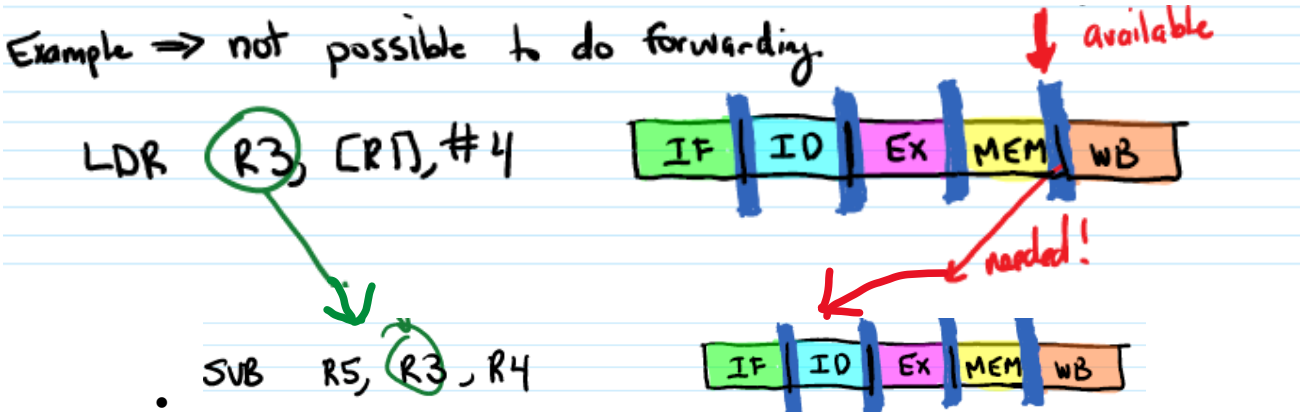
- Solution 1:



  - Delay or stall the progress of instruction 'i+1' through the pipeline until the data is available.
  - Interlock→ hardware that is included in the processor to detect data dependency and stall the dependent instruction.
- Solution 2: Forwarding or Bypassing→ forward the result to EX as soon as it is available anywhere in the pipeline.



  - Therefore, if ALU output was available then we can give ALU the subtract instruction value before it needs it.
    - Not always possible



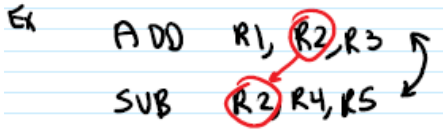
- Solution 3:
  - Load delay slot
    - Build the hardware to assume that an instruction that uses a load value is separated from the load instruction.
- Solution 4:
  - Instruction Scheduling
    - Reorder the instructions of the program so that dependent instructions are far enough apart from each other.
    - Two ways:
      1. Static Instruction Scheduling
      2. Dynamic Instruction Scheduling

Lecture 10: Pipelining Issues

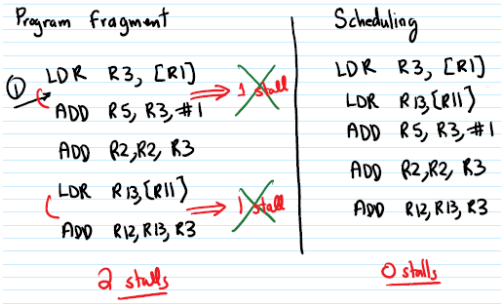
Lecture 10, Part 1: Instruction Scheduling

Static Instruction Scheduling:

- Reorder the instructions of the program to eliminate data hazards.
    - Or speed execution time.
- Reordering must be safe.



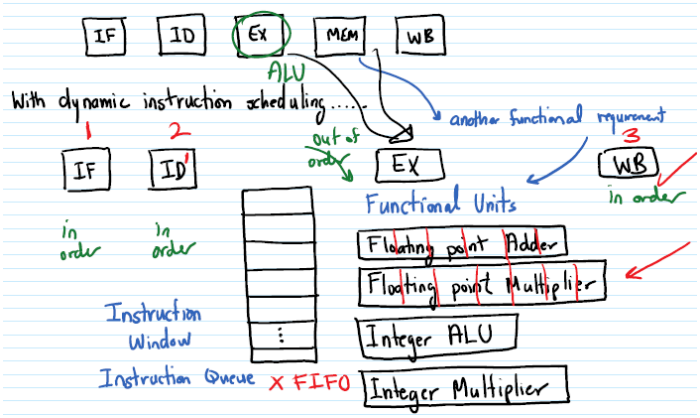
- Two instructions can be exchanged if they are independent of each other.
- Example: Static Instruction Scheduling



Kinds of Dependence:

- True dependence:
    - ADD R1, R2, R3
    - SUB R4, R1, R5
- Anti-dependence:
    - ADD R1, R2, R3
    - SUB R2, R4, R5
- Output Dependence:
    - ADD R1, R2, R3
    - SUB R1, R4, R5

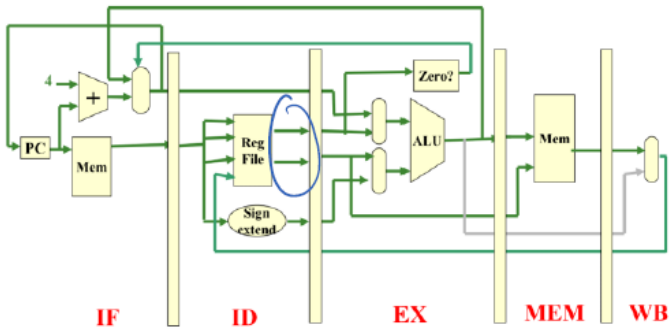
Dynamic Instruction Scheduling:



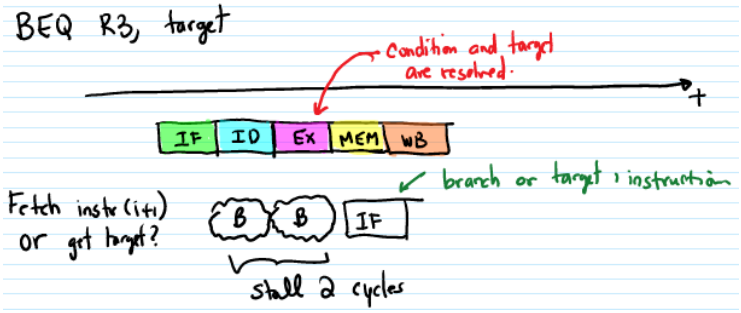
- The hardware dynamically schedules instructions from the Instruction Windows for execution on the functional units.
- The instructions could execute in an order that is different from that specified by the program.
    - With the same result.
- Such processors are called “out of order” processors.

Lecture 10, Part 2: Control Hazards

- Recall: Control Hazard: A hazard that arises due to control transfer instructions.
- BEQ R1, target;



- BEQ R3, target;



- Observation→ since the branch is resolved in the EX stage. There must be 2 stall cycles after every conditional branch instruction.

Reducing Impact of Branch Stall:

- The execution of a conditional branch involves 2 activities:
  - Evaluate the branch condition
  - Compute the branch target address
- To reduce branch stall effect, we could:
  - Evaluate the condition earlier.
  - Compute the target address earlier.

} ID & Hardware

Control Hazard Solutions:

1. Static Branch Prediction

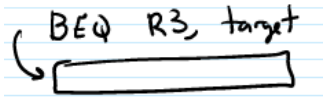
Predict before program starts executing

Prediction and Correctness:

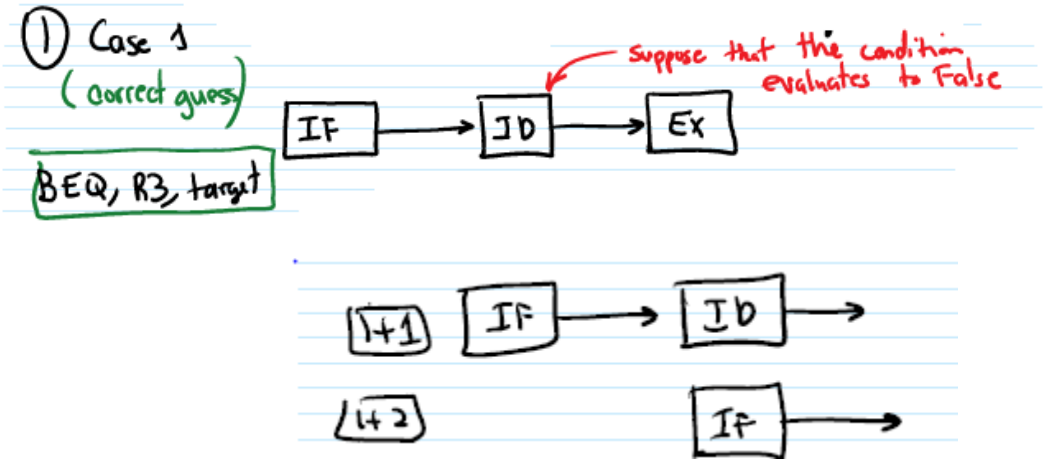
- Prediction→ guessing what is going to happen
- What if the guess in incorrect?
    - The pipelined processor hardware must be built to detect the misprediction and take corrective actions.

Guess Policies:

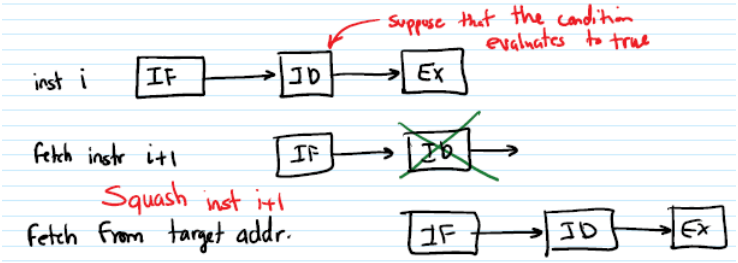
- Static Not-Taken Policy
    - Hardware built to fetch next instruction.



    - After ID stage, if it is found that the branch condition (hardware) is false (i.e. not taken/we were correct) →continue→ 0 stalls
        - Otherwise squash the fetched instruction and re-fetch the branch target address→ 1 stall
- Case 1:



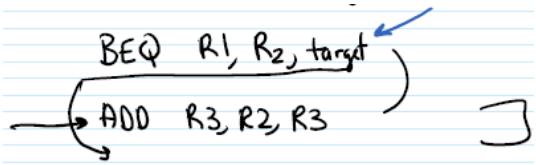
- Case 2:



  - Therefore, the average shall branch penalty < 1.

2. **Delayed Branching**

- Design hardware so that control transfer takes place after a few of the following instructions:



- Delay slots→ following instructions that are executed whether or not the branch is taken.

Delayed Branching→ Filling Delay Slots:

- Instructions that do not affect the branching condition can be put in the delay slot.
  - By the compiler.
- Where to get instructions to fill delay slots?

1. From the branch target address.
   - Beneficial only if branch is taken.

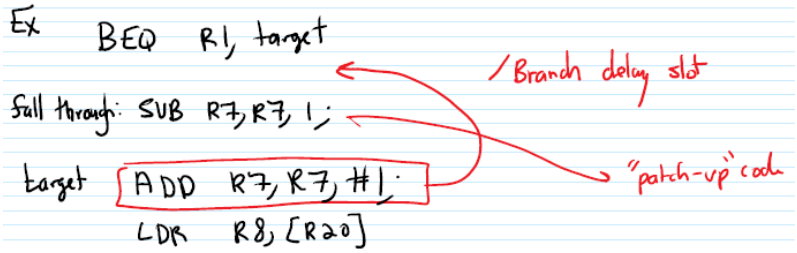
2. From the fall through (branch not taken path).
   - Beneficial when the branch is not taken.

3. From before the branch.
   - Beneficial either way.

- All 3 possibilities are valid.

Patch-up Code:

If you use idea 1 or 2 and you are wrong, then you have to reverse the process.



Lecture 11: Cache Memory

Lecture 11, Part 1: Concept of Cache Memory

Recall: In discussing the pipeline, we assumed that memory latency will be hidden so it appears to operate at processor speed.

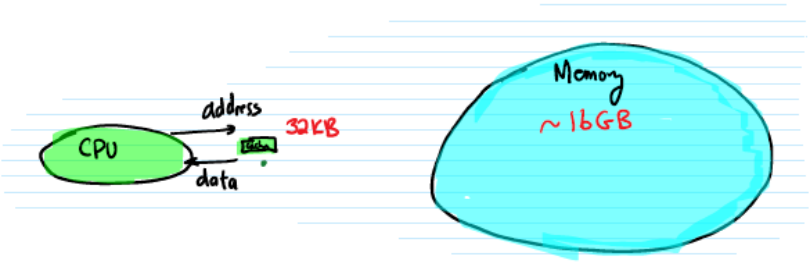
- Cache Memory: Hardware that makes this happen.
- Design principle→ Locality of Reference
- Temporal Locality: Least recently used objects are least likely to be referenced in the near future.
- Spatial Locality: Neighbors of recently referenced locations are likely to be referenced in the future.

Cache Memory→ Exploits locality of reference

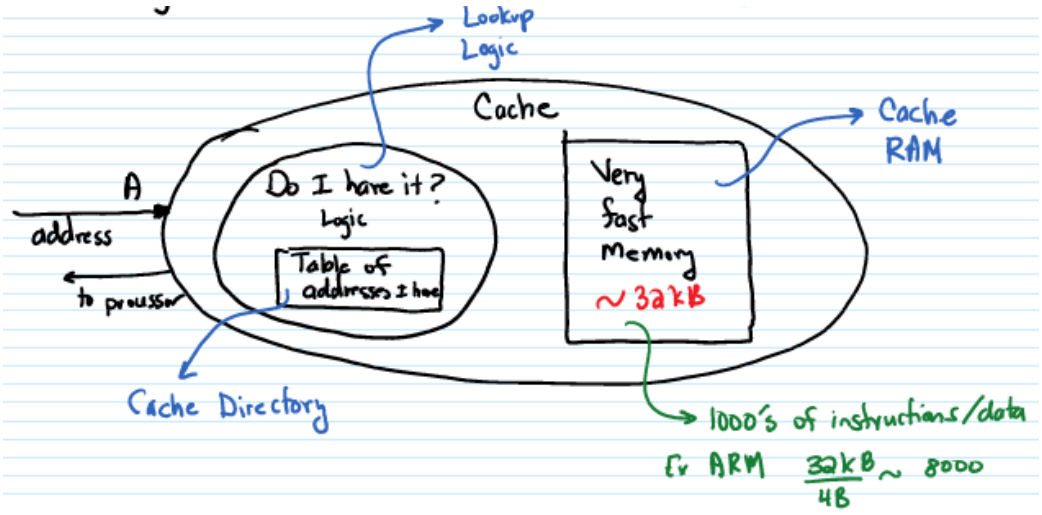
Cache → Hardware structure that provides memory objects (data instructions) that the processor references.

Directly (most of the time)

Fast ~ processor speed



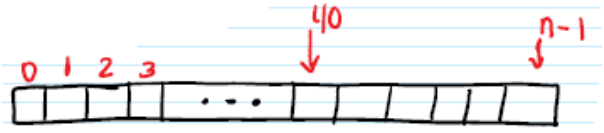
Cache Design:



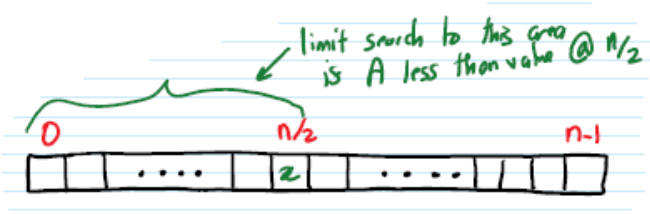
- How to do fast Cache lookup?
- →Searching
    - Techniques to search for a specific value from a large collection.
        - Example: Searching for the word “phrase” in a large text file.
        - Example: Searching for the number 10 in a large integer array.
    - Our specific search→ looking for the address ‘A’ among the 1000s of addresses in the code directory.

Search Algorithms:

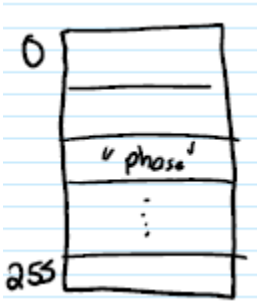
1. Linear Search



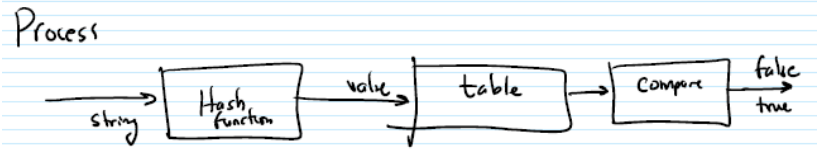
    - Compare address ‘A’ with the first address in the cache directory.
    - Else compare ‘A’ with 2nd, 3rd, etc until you find a match or reach the end.
2. Binary Search



    - Sort the array of data items, say in increasing order.
    - So compare ‘A’ with the middle element value.
    - If they match, the search is successful.
    - Else repeat for the appropriate half of data
    - Log_2 (n)
3. Hash Searching
    - May typically take just 1 comparison independent of ‘n’.
    - The number of comparisons required doesn’t depend on the number of data values that we are searching among.
    - Hashing→ A search technique that uses a hash table indexed into a hash function.
    - Hash function→ A function computed on the search string.
    - Hash table example:
    - Example search for the word “phase”.
    - Searching for a string of characters S_0, S_1, S_2,….., S_{n-1}
    - Hash function: $\sum_{i=0}^{n-1} \frac{S_i}{n}$, S_i → ASCII

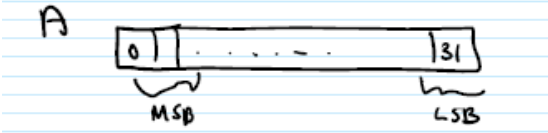


➔ "phase", "shape" both have the same hash index.
➔ This is called a hash collision.
➔ Hash Searching Process:



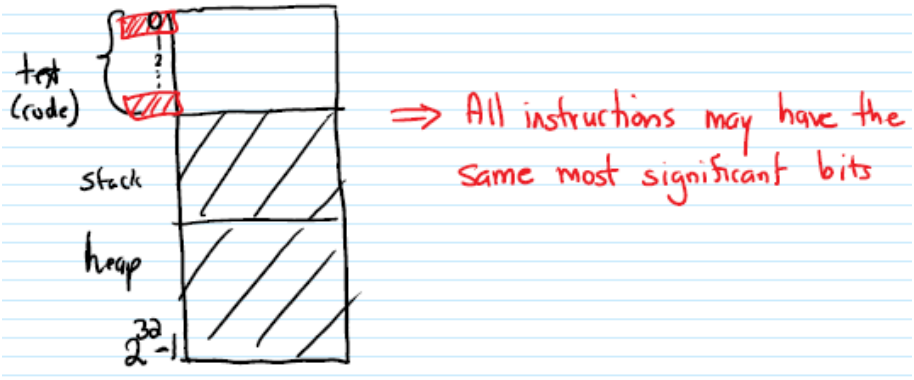
Specifically, in our Application:

- In the cache situation → cache lookup hardware is doing a search for address 'A'.
- Simple hash function?
    - Select some of the bits of the address 'A'.



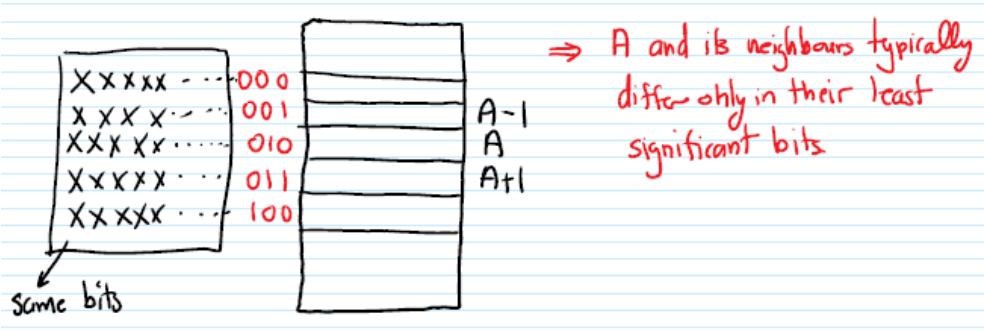
    - Which bits of address 'A' to use?
    - Few possibilities:
        - Most Significant Bits (MSB)
        - Least Significant Bits (LSB)
        - Middle bits

Most Significant Bits (MSB):

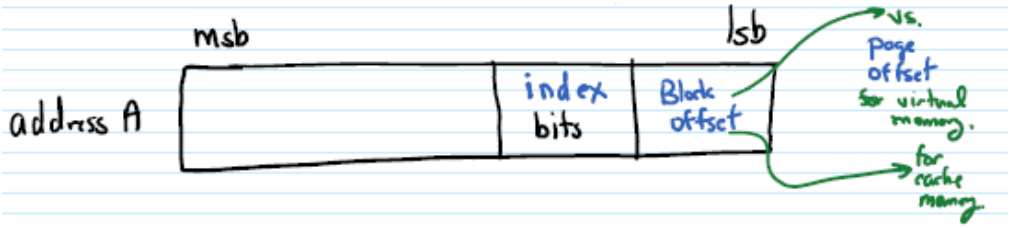


- For a small program, everything would index into the same place in the hash table→ collisions.

Least Significant Bits (LSB):



- 'A' and its neighbors possibly differ only in these bits but they should be treated as one unit, not hashing into different hash table entries.
- Therefore, using the least significant address bits for hashing is not a good idea.



- Block: serves the same purpose in cache memory as the Page does in Virtual Memory.
  - Reduces translation table size.
  - Exploits Spatial Locality of reference.
- The Cache contains one entry for each block, just like the Page Table contains one entry for each virtual page.

Summary:

- A Cache is organized in terms of blocks → memory locations that share the same address bits other than LSB.
- Main memory is also organized in terms of blocks.
- The cache hardware views the address as: