MMU and virtual memory

A comprehensive view

Roberto E. Vargas Caballero

Clue Technologies

2021

Table of Contents

Processes

Process Definition

Process Definition

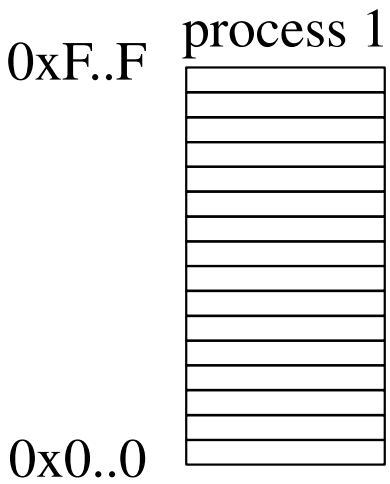
- Classical: "It is an instance of an executing program"

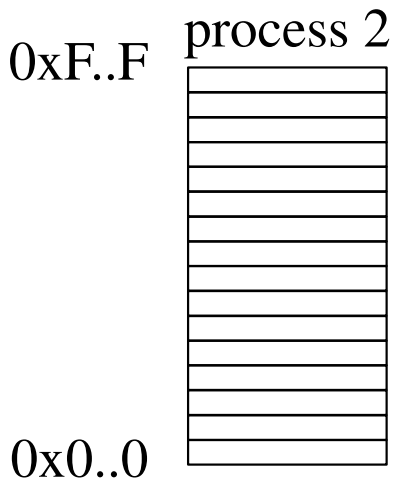
Process Definition

- Classical: "It is an instance of an executing program"
- Wikipedia: "In computing, a process is the instance of a computer program that is being executed by one or many threads."

Process Definition

- Classical: "It is an instance of an executing program"
- Wikipedia: "In computing, a process is the instance of a computer program that is being executed by one or many threads."
- Missing point:
  Every process executes in a different memory map!





Memory Map

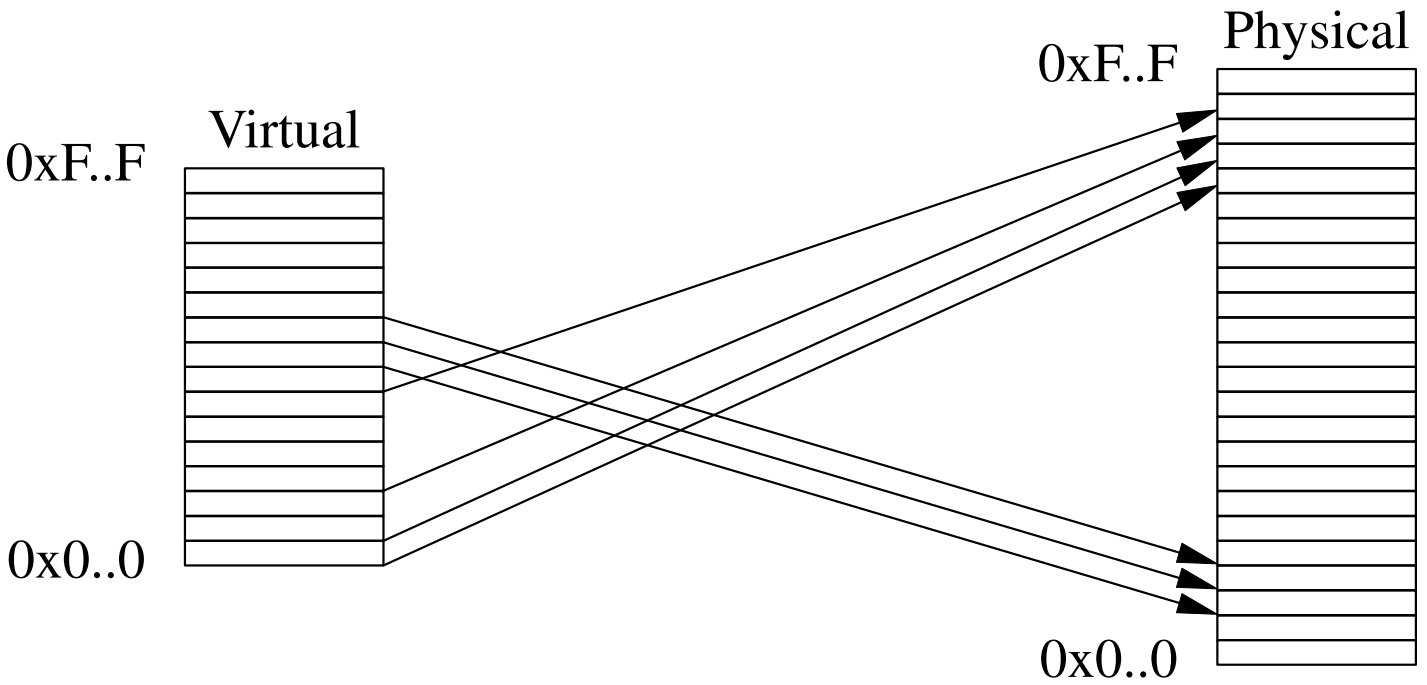
- Definition: A translation between a physical memory space and a virtual memory space

Memory Map

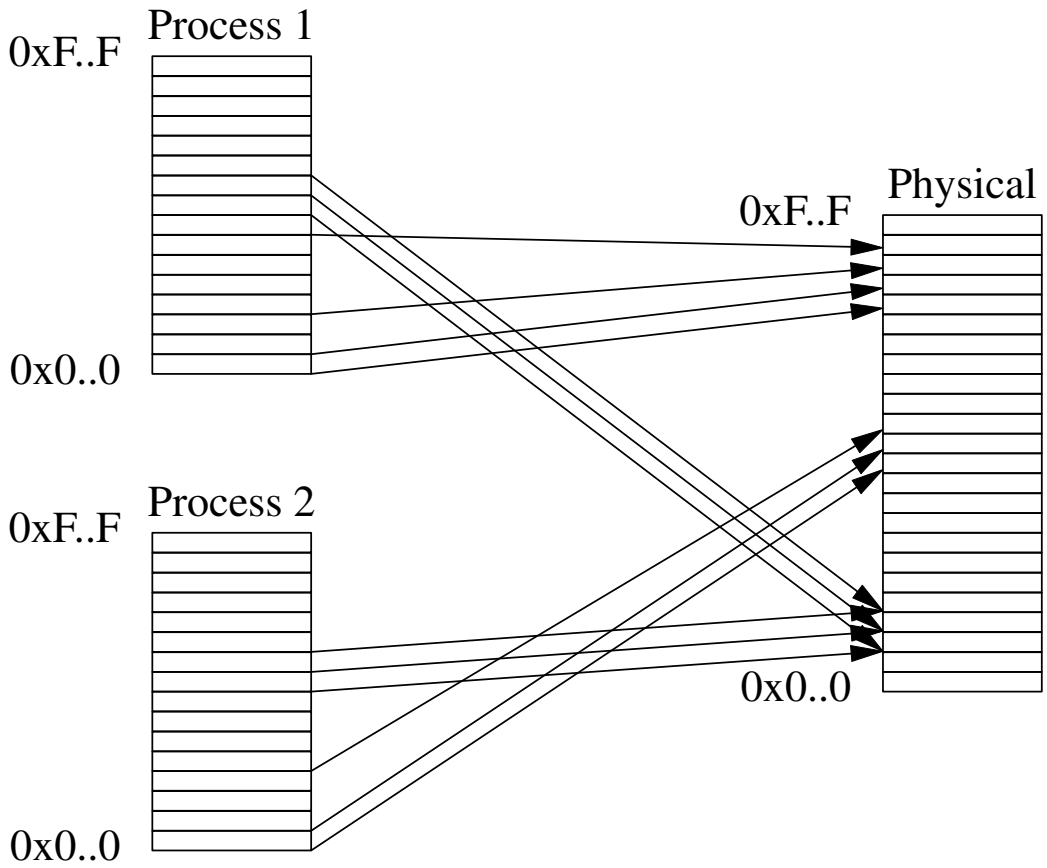
- Definition: A translation between a physical memory space and a virtual memory space
- Physical space: It is the address space used when the CPU accesses physically the memory chips.

Memory Map

- Definition: A translation between a physical memory space and a virtual memory space
- Physical space: It is the address space used when the CPU accesses physically the memory chips.
- Virtual space: It is the address space used by the process.

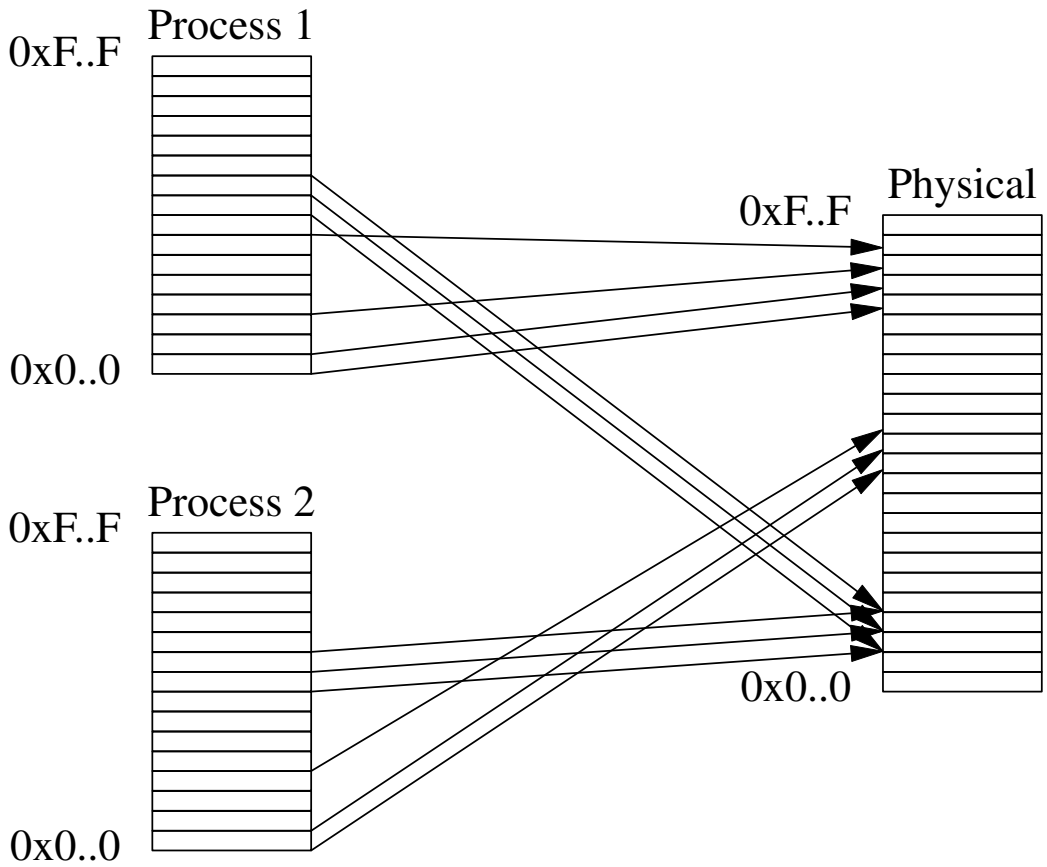


Two processes



- Every process has a different memory map, and same addresses can reference different physical memory.

Two processes



- Every process has a different memory map, and same addresses can reference different physical memory.
- Why mappings are not bijective functions? Why do they share physical regions?

Processor execution modes

Supervisor mode

- Can execute all machine instructions
- Can reference all memory locations

Processor execution modes

User mode

- Can only execute a subset of instructions
- Can only reference a subset of memory locations

Processor execution modes

How to change processor mode

- To enter in Priviledge mode

Processor execution modes

How to change processor mode

- To enter in Priviledge mode
  - Interrupt: External attention request from external device.
    - Serial output FIFO empty
    - Timer interrupt

Processor execution modes

How to change processor mode

- To enter in Priviledge mode
  - Interrupt: External attention request from external device.
    - Serial output FIFO empty
    - Timer interrupt
  - Exception: Internal attention request from program execution. Sometimes called traps.
    - Invalid memory access.
    - Invalid instruction executed.

Processor execution modes

How to change processor mode

- To enter in Priviledge mode
  - Interrupt: External attention request from external device.
    - Serial output FIFO empty
    - Timer interrupt
  - Exception: Internal attention request from program execution. Sometimes called traps.
    - Invalid memory access.
    - Invalid instruction executed.
  - syscall/trap instruction executed. Sometimes called software interrupts.

Processor execution modes

How to change processor mode

- To enter in Priviledge mode
  - Interrupt: External attention request from external device.
    - Serial output FIFO empty
    - Timer interrupt
  - Exception: Internal attention request from program execution. Sometimes called traps.
    - Invalid memory access.
    - Invalid instruction executed.
  - syscall/trap instruction executed. Sometimes called software interrupts.
- To leave Priviledge mode

Processor execution modes

How to change processor mode

- To enter in Priviledge mode
  - Interrupt: External attention request from external device.
    - Serial output FIFO empty
    - Timer interrupt
  - Exception: Internal attention request from program execution. Sometimes called traps.
    - Invalid memory access.
    - Invalid instruction executed.
  - syscall/trap instruction executed. Sometimes called software interrupts.
- To leave Priviledge mode
  - iret/eret instruction.

Processor execution modes

Steps done by the processor to

- Enter priviledge mode

Processor execution modes

Steps done by the processor to

- Enter priviledge mode
  - saves current program counter
  - changes execution mode
  - switches to kernel stack pointer
  - jumps to service routine

Processor execution modes

Steps done by the processor to

- Enter priviledge mode
  - saves current program counter
  - changes execution mode
  - switches to kernel stack pointer
  - jumps to service routine
- Leave priviledge mode

Processor execution modes

Steps done by the processor to

- Enter priviledge mode
  - saves current program counter
  - changes execution mode
  - switches to kernel stack pointer
  - jumps to service routine
- Leave priviledge mode
  - restores execution mode
  - switches to user stack pointer
  - restores pogram counter

Processor execution modes

User mode

```
...
fork();
...
```

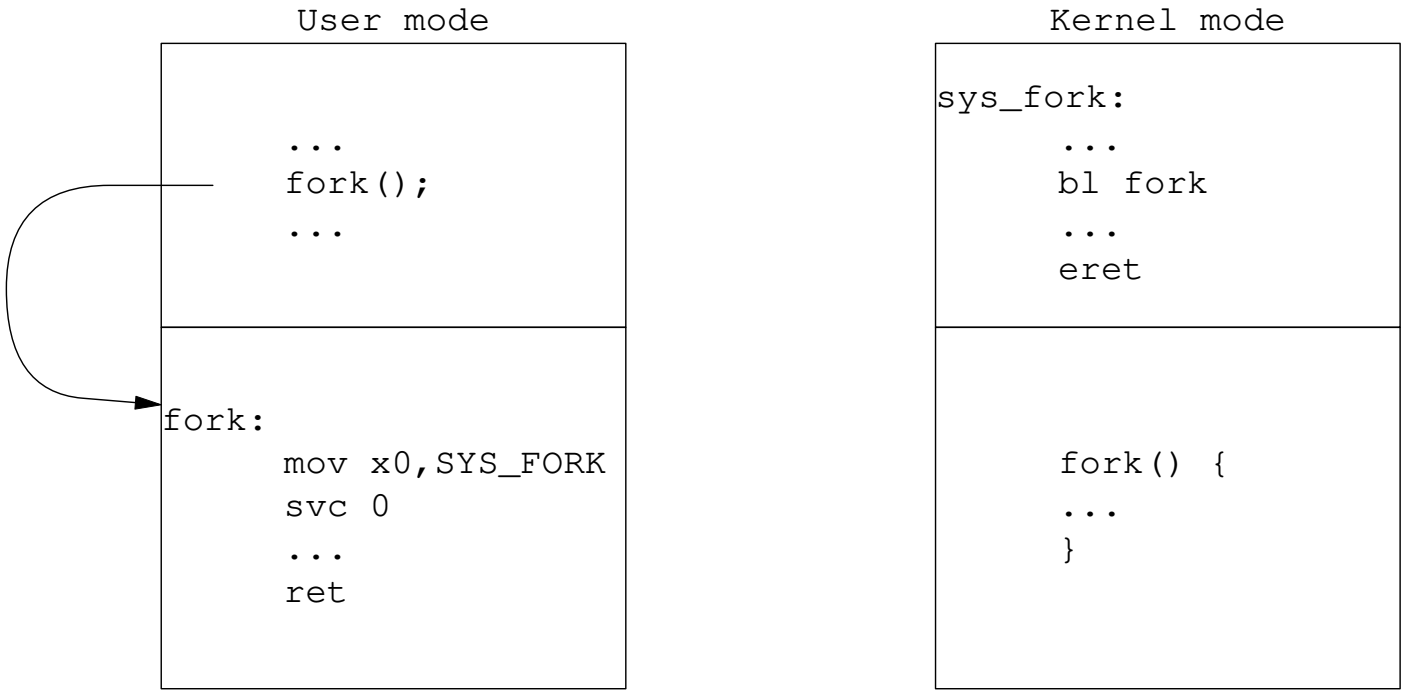
```
fork:
        mov x0,SYS_FORK
        svc 0
        ...
        ret
```

Kernel mode

```
sys_fork:
        ...
        bl fork
        ...
        eret
```

```
fork() {
...
}
```

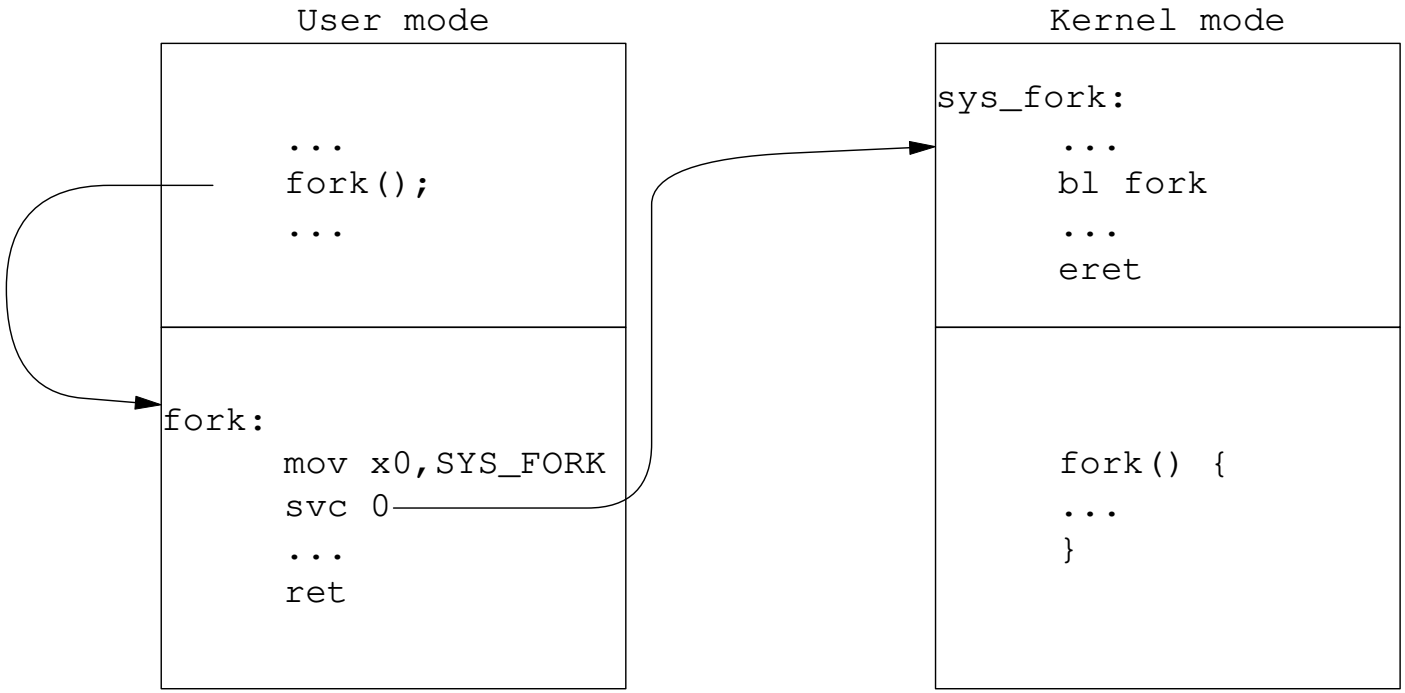
Processor execution modes



Call to syscall library wrapper

The call to *fork* jumps to a wrapper written in ARM assembly with a *SVC* instruction

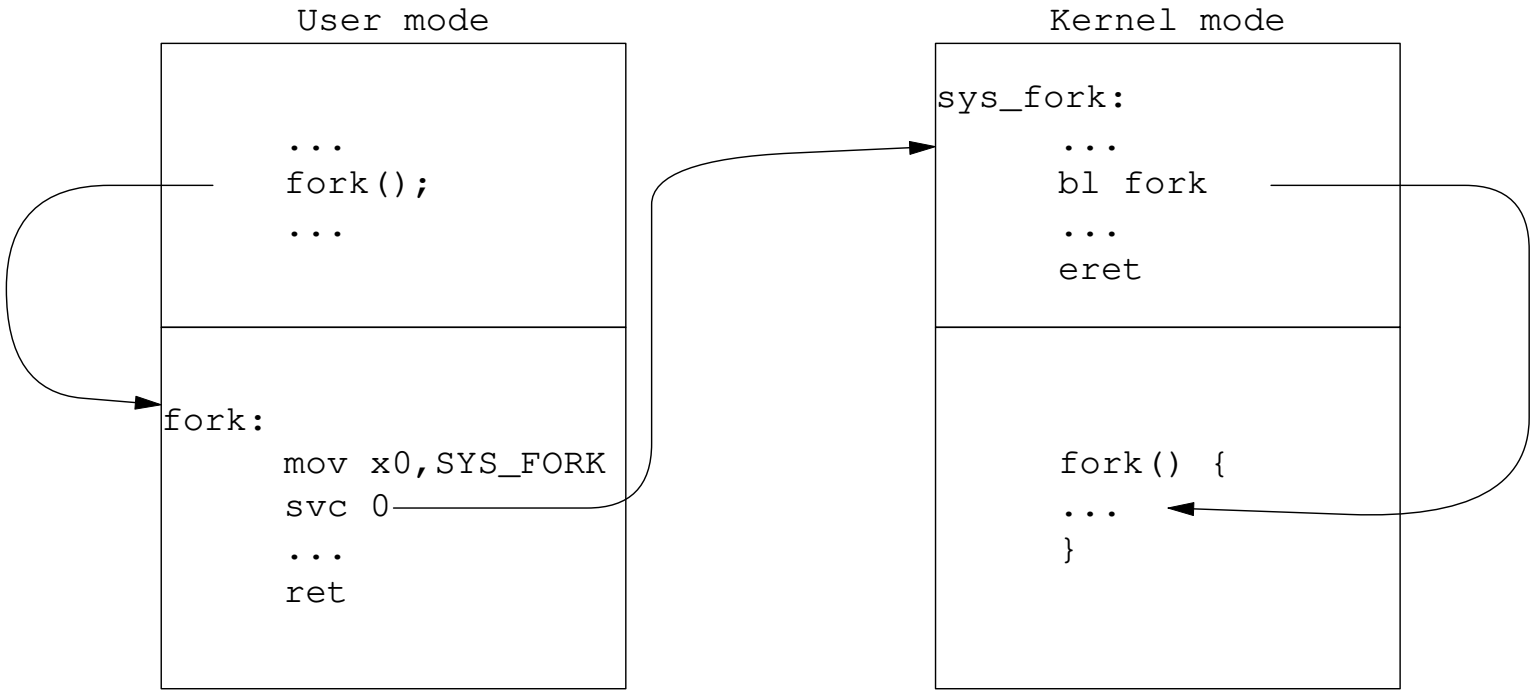
Processor execution modes



Execution of SVC instruction

it saves the current program counter, change the execution mode to Supervisor mode and jumps to specific assembly code for the fork syscall.

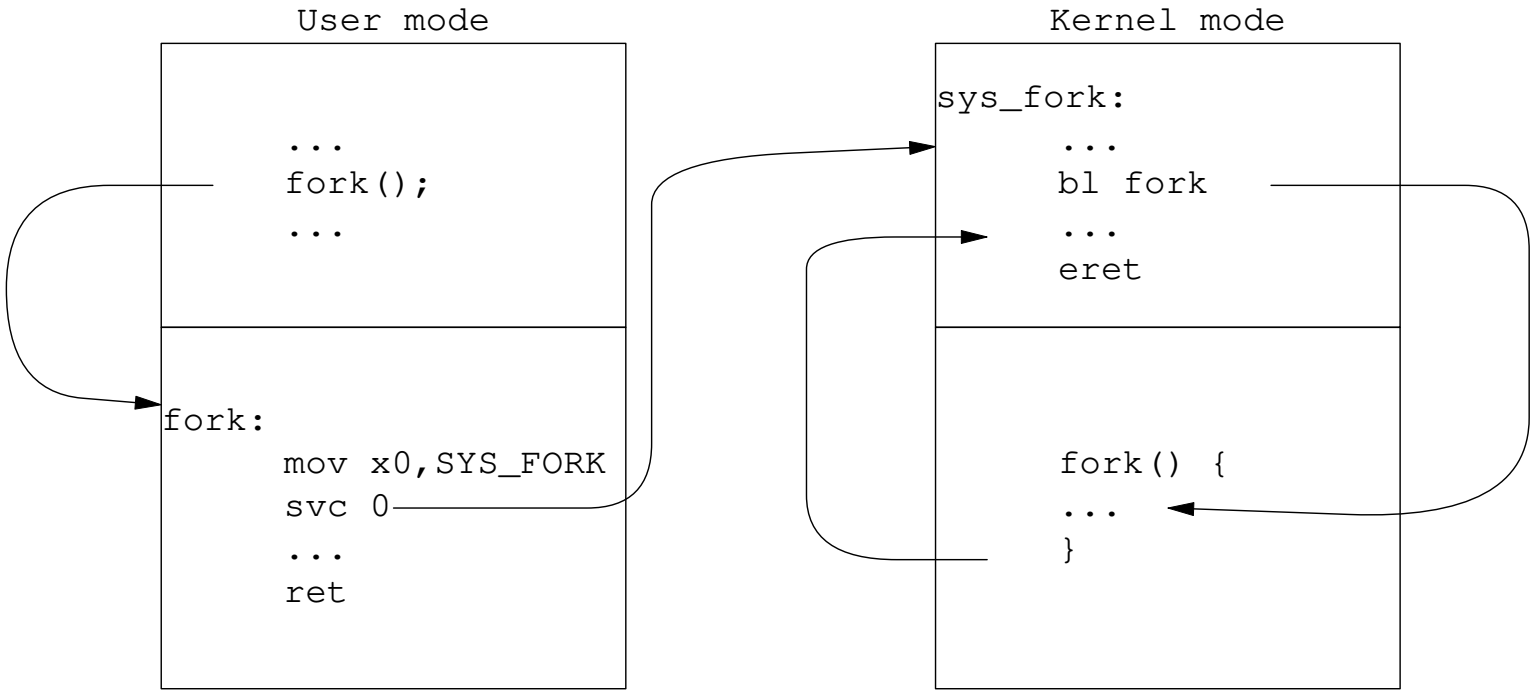
Processor execution modes



Execution of fork syscall

After saving all the registers it jumps to the C code where the *fork* syscall is implemented.

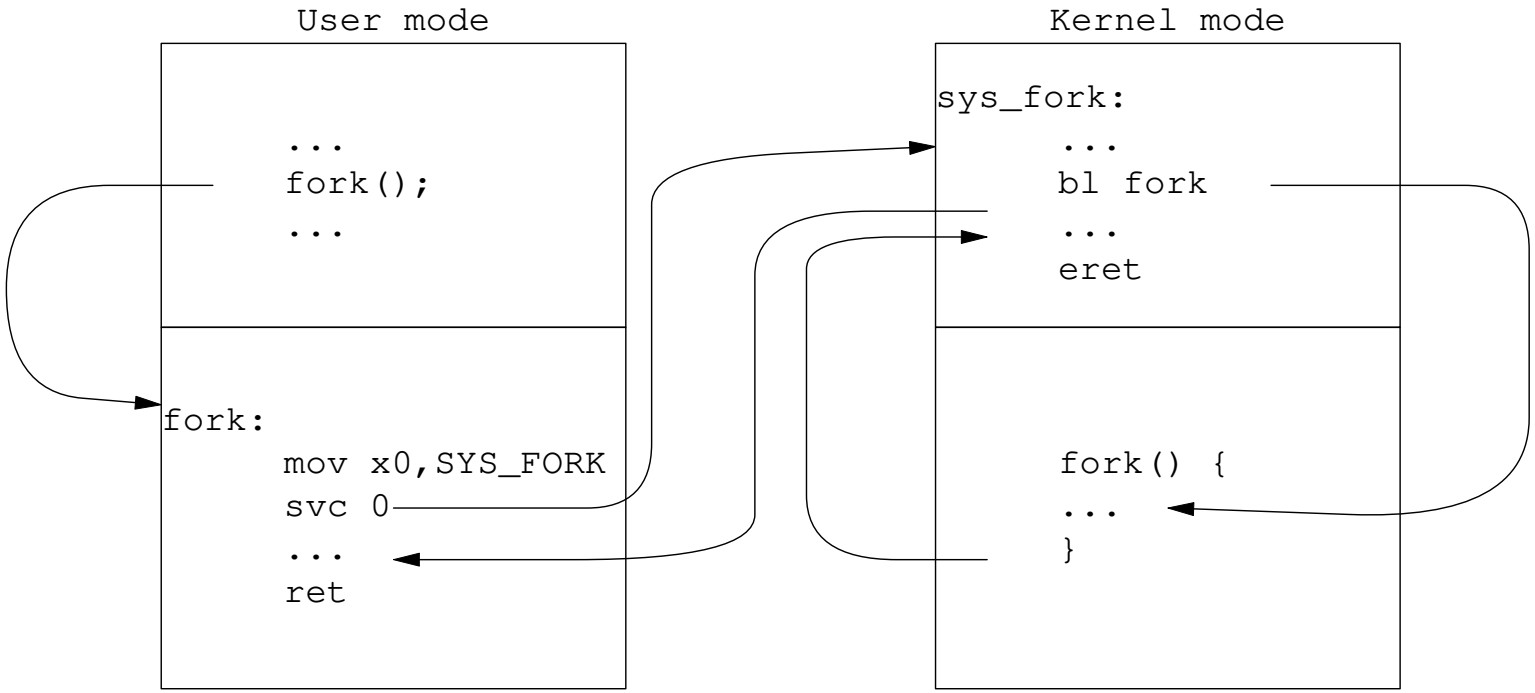
Processor execution modes



Returning from fork syscall

When the *fork* syscall finishes it returns to the assembly code where all the registers are restored.

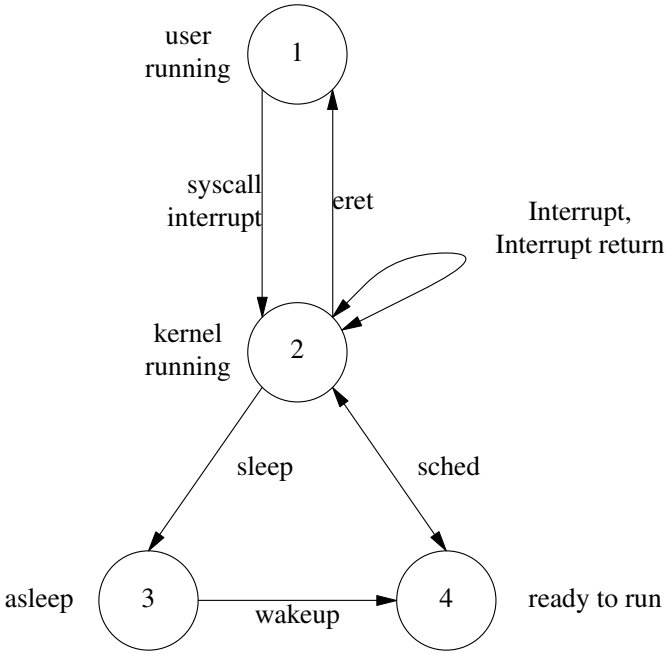
Processor execution modes



Returning to User mode

When the *iret* instruction is executed then the original program counter is restored and the processor returns to User mode.

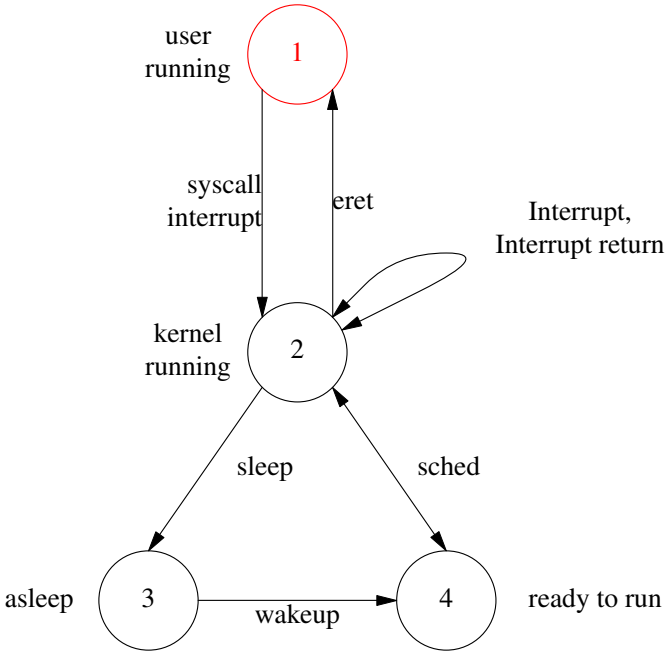
Simplified process states



Process execution

Processes change the state while its execution

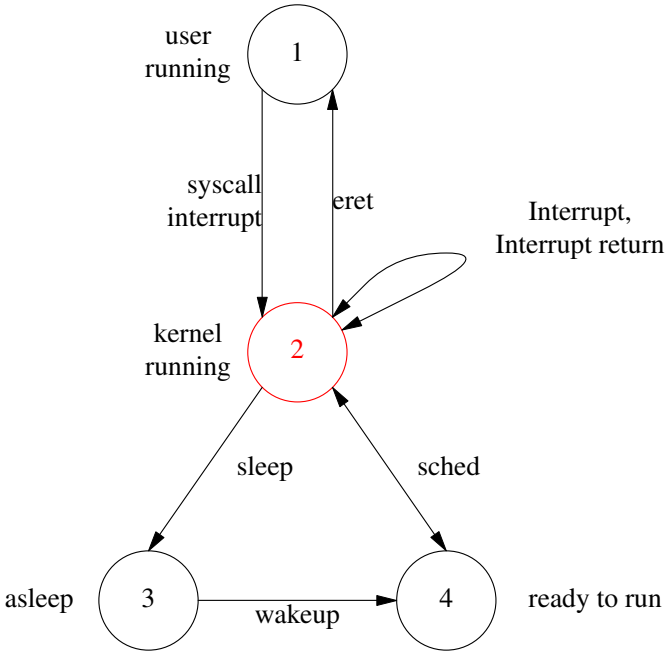
Simplified process states



User running

The process is running and executing user code and the processor is running in User mode.

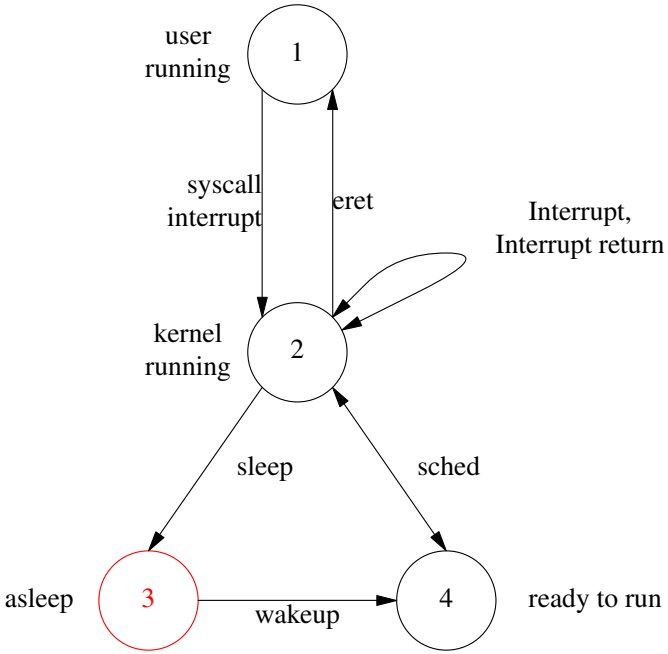
Simplified process states



Kernel running

The process is running and executing kernel code and the processor is running in Kernel mode.

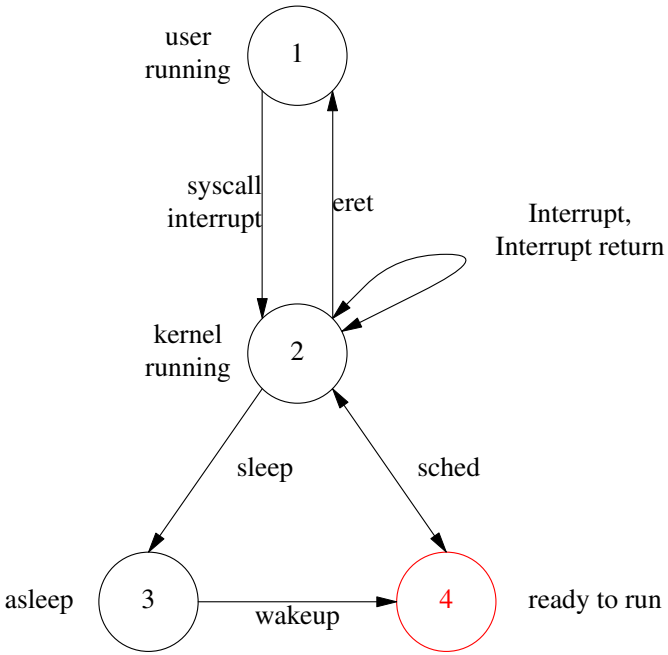
Simplified process states



asleep

The process is not running and it is waiting for some event. The program counter and the stack pointer are pointing to kernel memory.

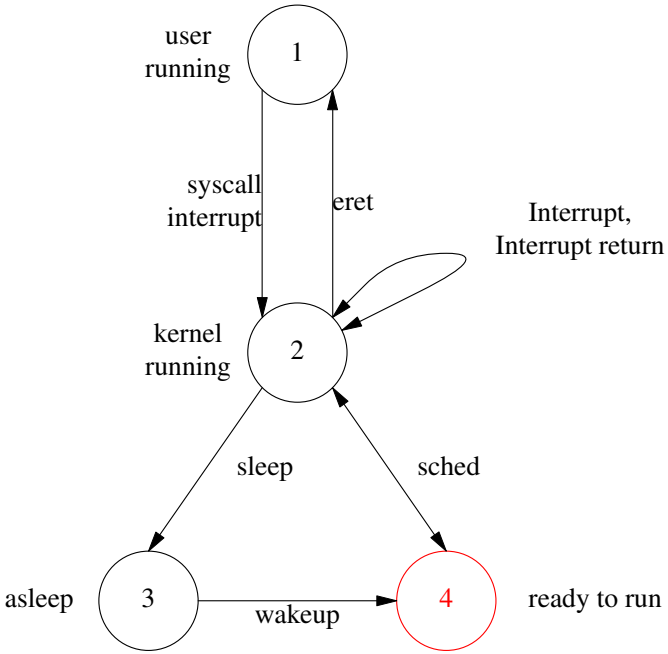
Simplified process states



Ready to run

The proecess is not running but it is ready to run and it is waiting to be scheduled. The program counter and the stack pointer are pointing to kernel memory.

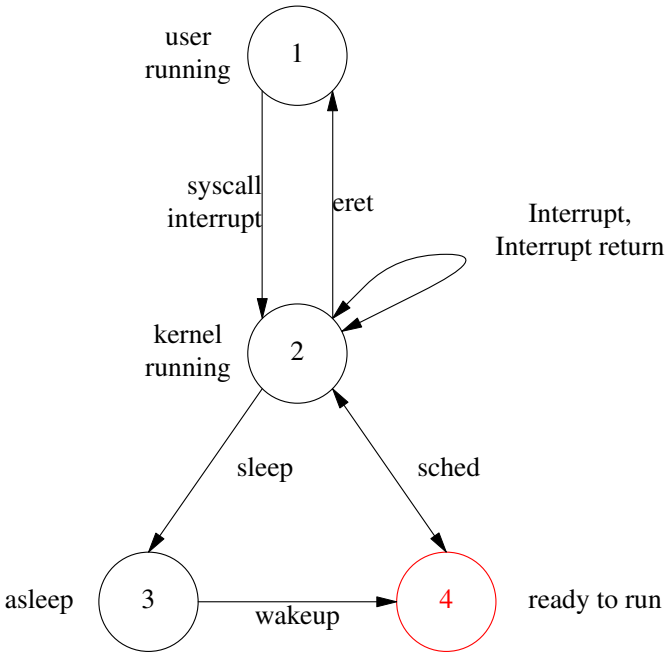
Simplified process states



What is the kernel?

- Only a state of processes.

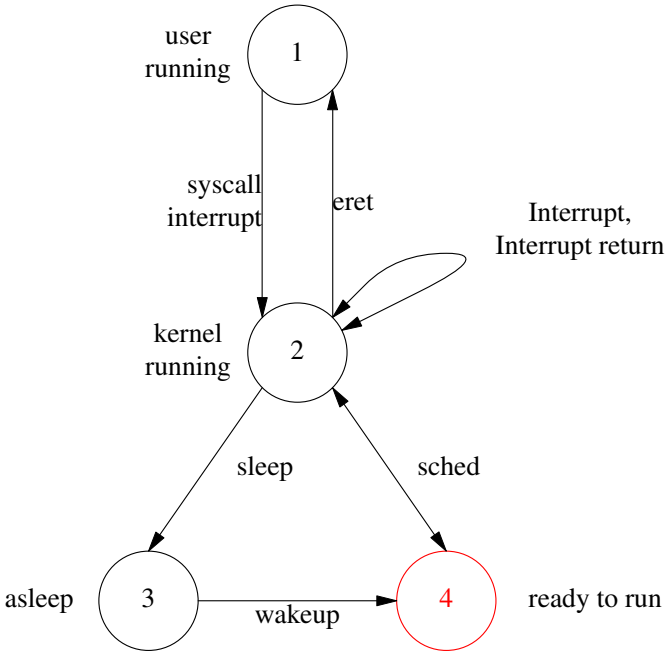
Simplified process states



What is the kernel?

- Only a state of processes.
- Ususally more of one process in kernel mode

Simplified process states



What is the kernel?

- Only a state of processes.
- Ususally more of one process in kernel mode
- But only one process running per hardware thread!.

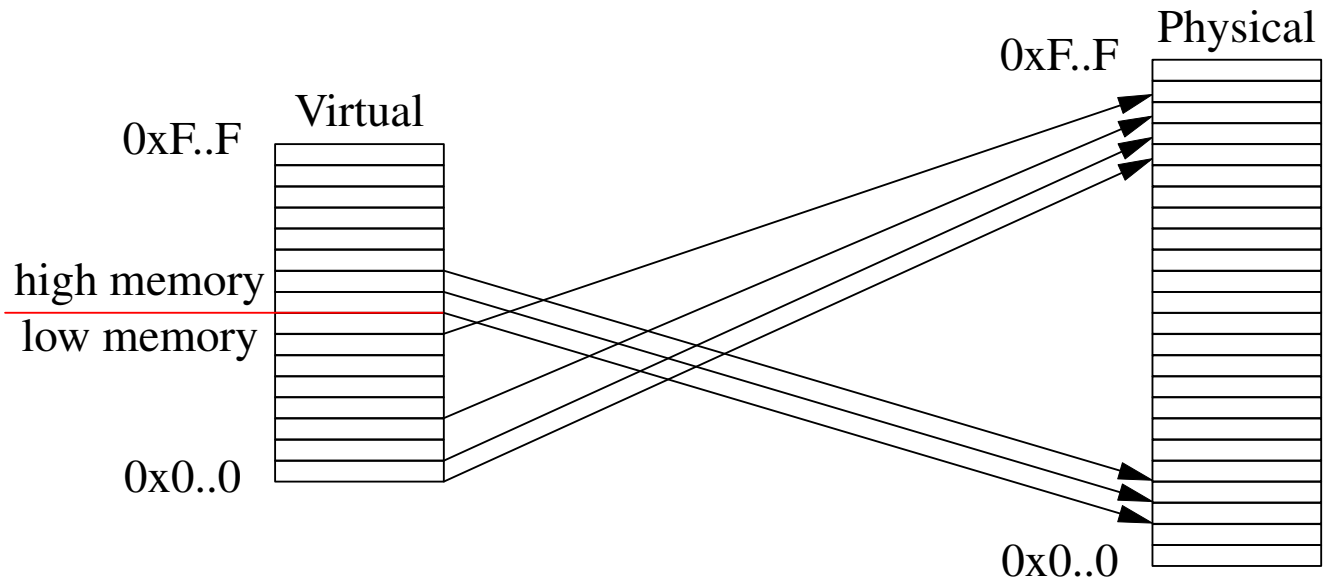
High and low memory

What happens if User mode can access the kernel memory?

High and low memory

What happens if User mode can access the kernel memory?

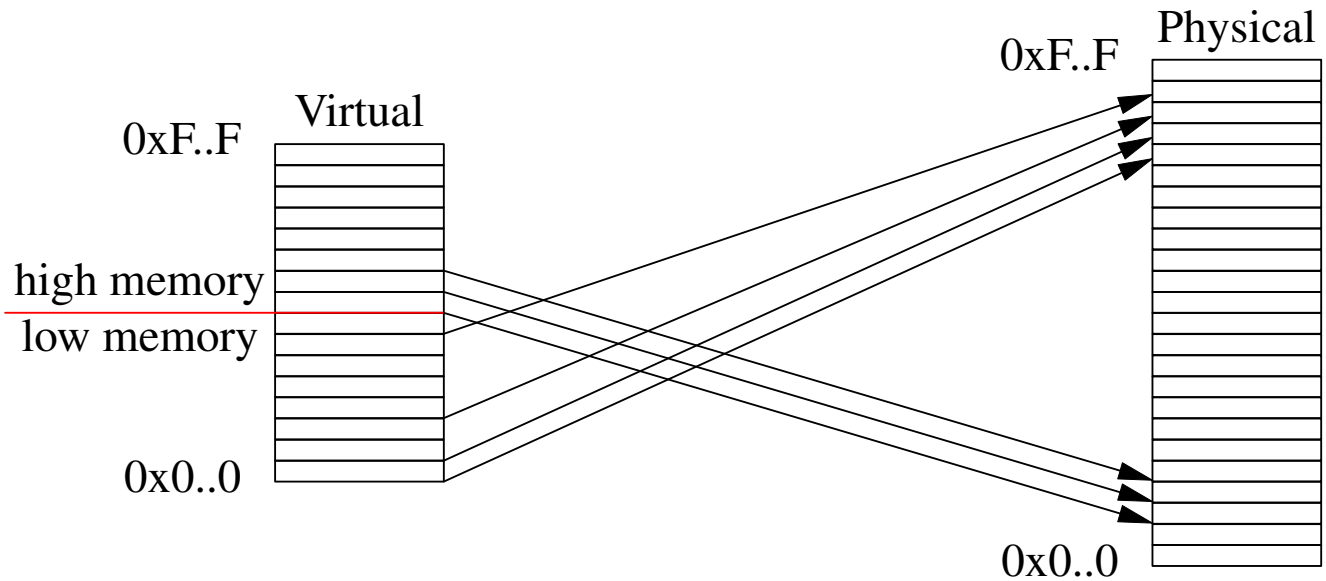
- Virtual memory space is split:
  - High memory: It only can be accessed by the kernel.
  - Low memory: It can be accessed by kernel and user.



High and low memory

What happens if User mode can access the kernel memory?

- Virtual memory space is split:
  - High memory: It only can be accessed by the kernel.
  - Low memory: It can be accessed by kernel and user.
- This mode of working, having accessible the user memory at any time, simplifies memory transfers between kernel space and user space.

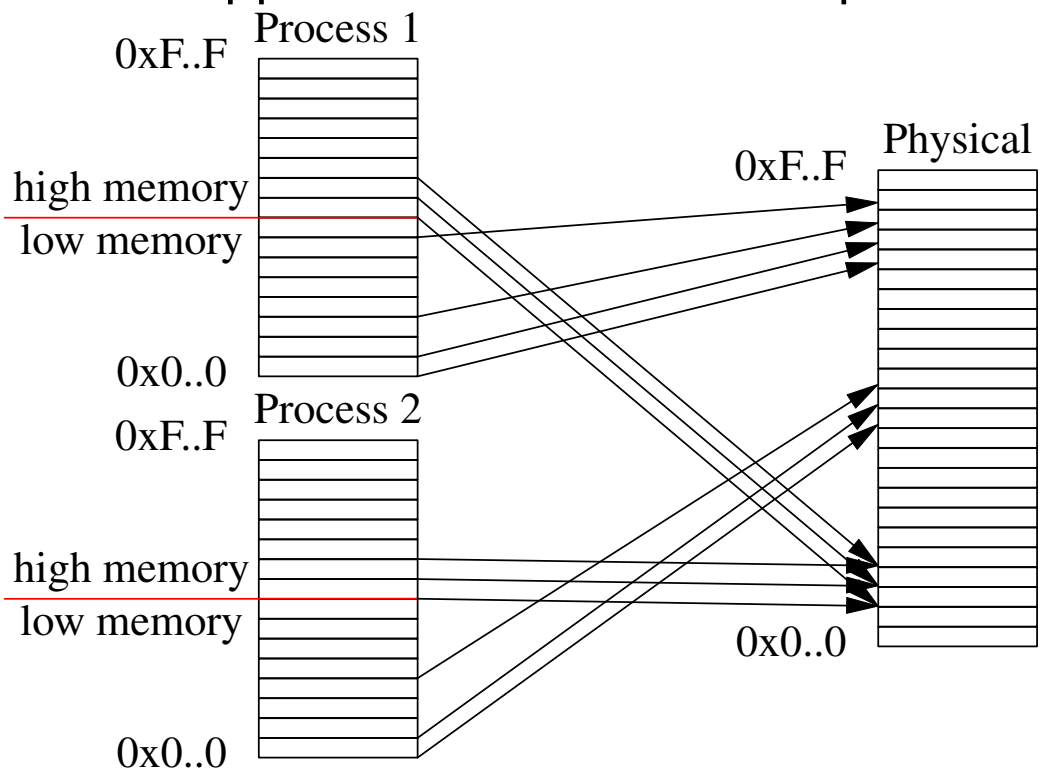


High and low memory (2)

- What happens with more of one process?

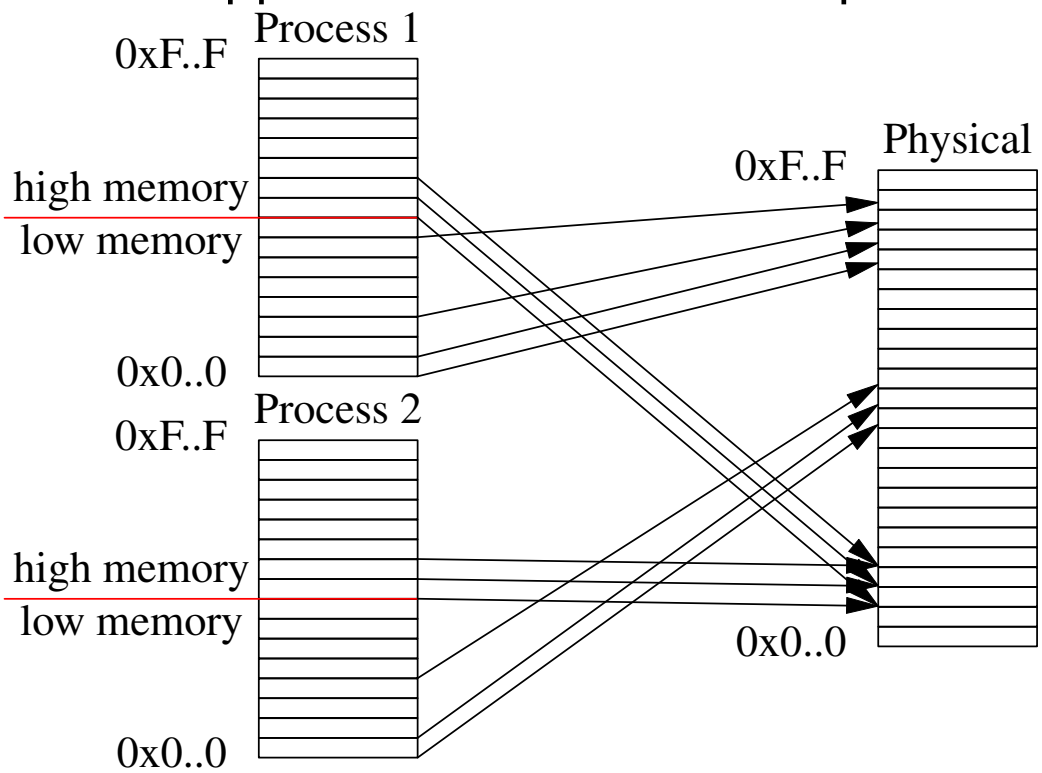
High and low memory (2)

- What happens with more of one process?



High and low memory (2)

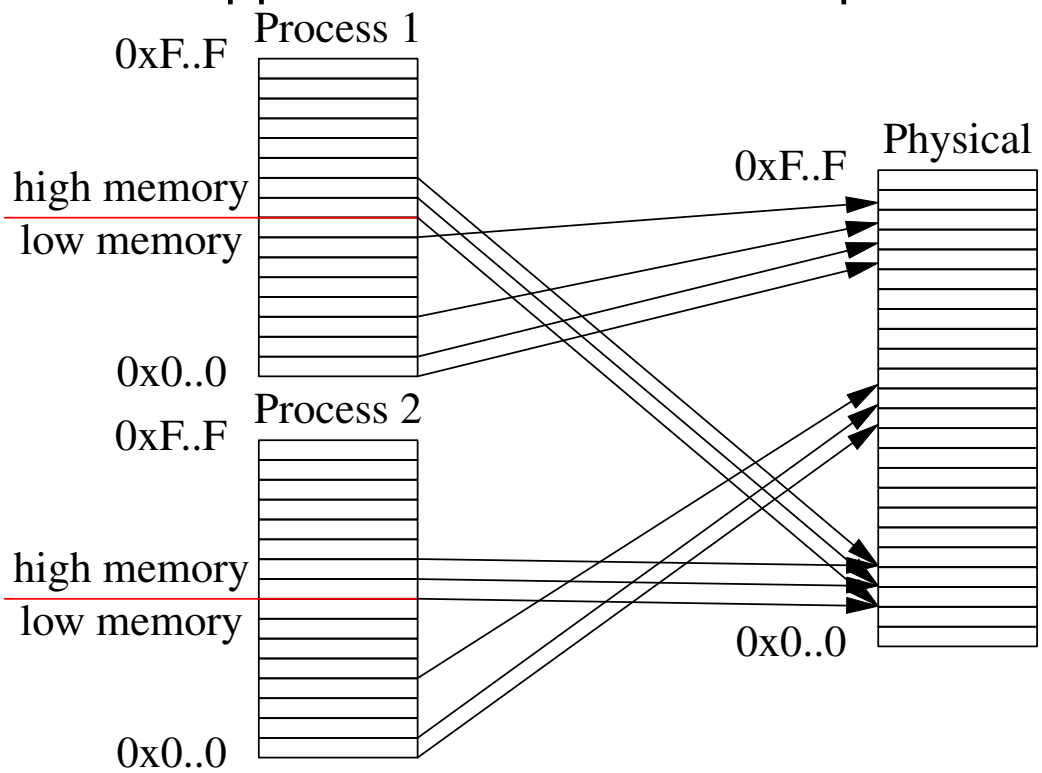
- What happens with more of one process?



  - The mapping of the high memory is shared between all the process!

High and low memory (2)

- What happens with more of one process?



  - The mapping of the high memory is shared between all the process!
  - All the processes may be in kernel mode at some moment.

Implementing processes

- Context:

Implementing processes

- Context:
  - Sections / Software segments: Contiguous block of memory.

Implementing processes

- Context:
  - Sections / Software segments: Contiguous block of memory.
    - text: Executable code

Implementing processes

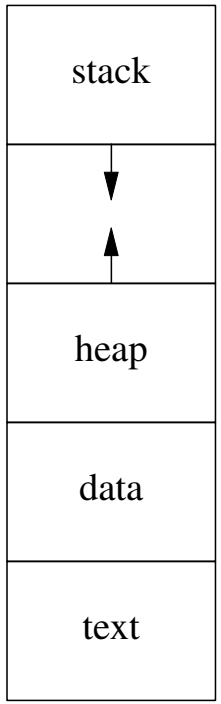
- Context:
  - Sections / Software segments: Contiguous block of memory.
    - text: Executable code
    - data: Static data

Implementing processes

- Context:
  - Sections / Software segments: Contiguous block of memory.
    - text: Executable code
    - data: Static data
    - heap: Dynamic (growing) data

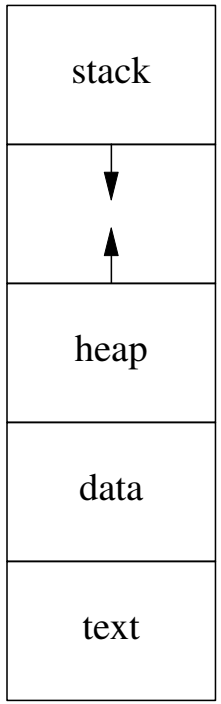
Implementing processes

- Context:
  - Sections / Software segments: Contiguous block of memory.
    - text: Executable code
    - data: Static data
    - heap: Dynamic (growing) data
    - stack: Temporary storage used by the processor.



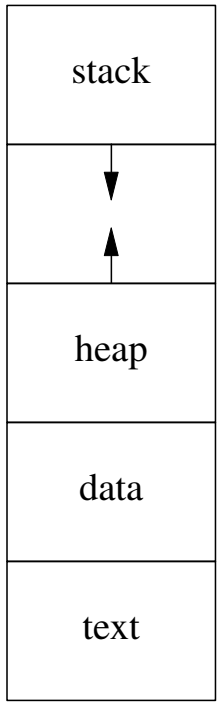
Implementing processes

- Context:
  - Sections / Software segments: Contiguous block of memory.
    - text: Executable code
    - data: Static data
    - heap: Dynamic (growing) data
    - stack: Temporary storage used by the processor.
  - Machine status
    - General purpose registers



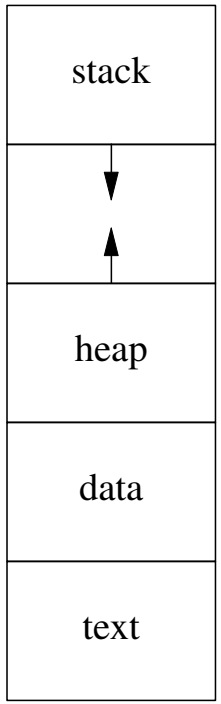
Implementing processes

- Context:
  - Sections / Software segments: Contiguous block of memory.
    - text: Executable code
    - data: Static data
    - heap: Dynamic (growing) data
    - stack: Temporary storage used by the processor.
  - Machine status
    - General purpose registers
    - Floating point registers



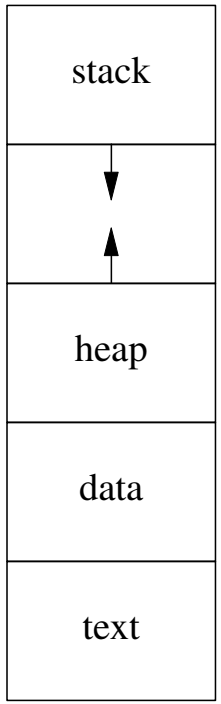
Implementing processes

- Context:
  - Sections / Software segments: Contiguous block of memory.
    - text: Executable code
    - data: Static data
    - heap: Dynamic (growing) data
    - stack: Temporary storage used by the processor.
  - Machine status
    - General purpose registers
    - Floating point registers
    - User program counter register



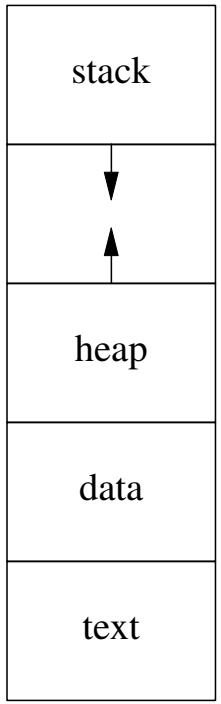
Implementing processes

- Context:
  - Sections / Software segments: Contiguous block of memory.
    - text: Executable code
    - data: Static data
    - heap: Dynamic (growing) data
    - stack: Temporary storage used by the processor.
  - Machine status
    - General purpose registers
    - Floating point registers
    - User program counter register
    - User stack register



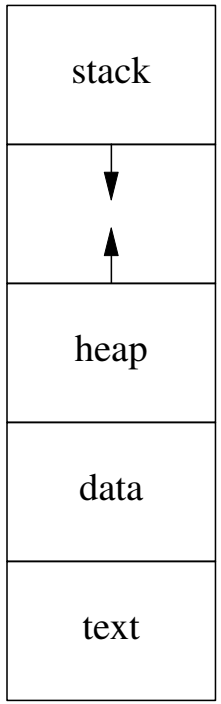
Implementing processes

- Context:
  - Sections / Software segments: Contiguous block of memory.
    - text: Executable code
    - data: Static data
    - heap: Dynamic (growing) data
    - stack: Temporary storage used by the processor.
  - Machine status
    - General purpose registers
    - Floating point registers
    - User program counter register
    - User stack register
    - Kernel program counter register



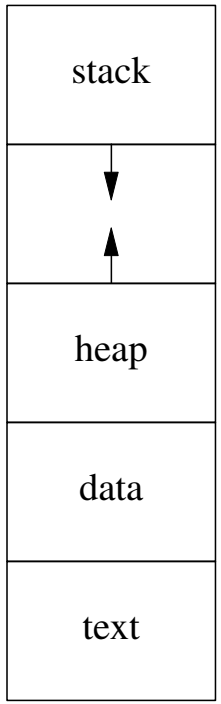
Implementing processes

- Context:
  - Sections / Software segments: Contiguous block of memory.
    - text: Executable code
    - data: Static data
    - heap: Dynamic (growing) data
    - stack: Temporary storage used by the processor.
  - Machine status
    - General purpose registers
    - Floating point registers
    - User program counter register
    - User stack register
    - Kernel program counter register
    - Kernel stack register



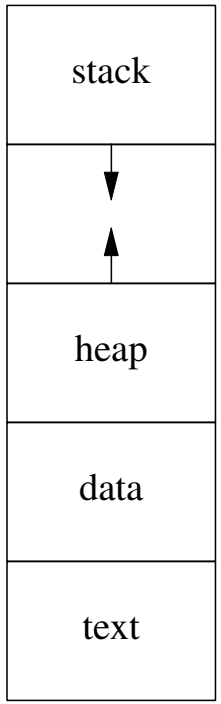
Implementing processes

- Context:
  - Sections / Software segments: Contiguous block of memory.
    - text: Executable code
    - data: Static data
    - heap: Dynamic (growing) data
    - stack: Temporary storage used by the processor.
  - Machine status
    - General purpose registers
    - Floating point registers
    - User program counter register
    - User stack register
    - Kernel program counter register
    - Kernel stack register
    - Processes sleeps in kernel mode while they are executing a syscall in user space!



Implementing processes

- Context:
  - Sections / Software segments: Contiguous block of memory.
    - text: Executable code
    - data: Static data
    - heap: Dynamic (growing) data
    - stack: Temporary storage used by the processor.
  - Machine status
    - General purpose registers
    - Floating point registers
    - User program counter register
    - User stack register
    - Kernel program counter register
    - Kernel stack register
    - Processes sleeps in kernel mode while they are executing a syscall in user space!
  - Process status (open files, signal mask, ...).



Context switch

- Wikipedia: It refers to the process of storing the system state for one task, so that task can be paused and another task resumed.

Context switch

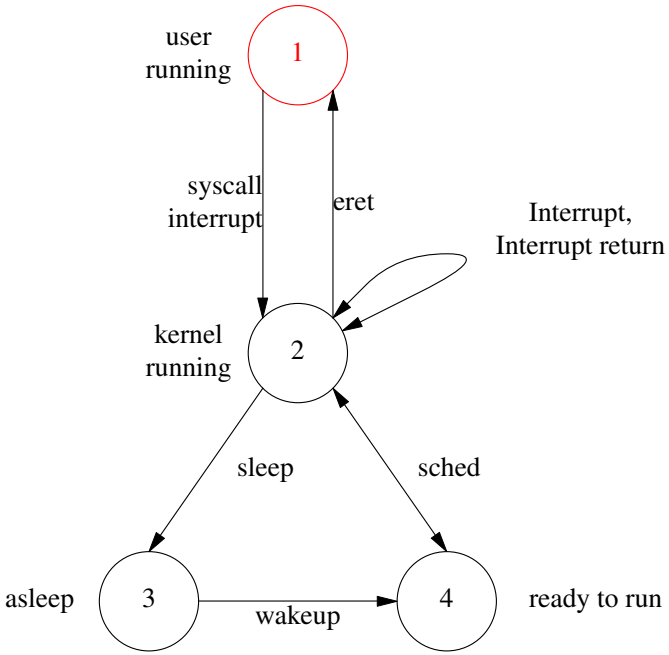
- Wikipedia: It refers to the process of storing the system state for one task, so that task can be paused and another task resumed.
- Can be the scheduler a process?

Context switch

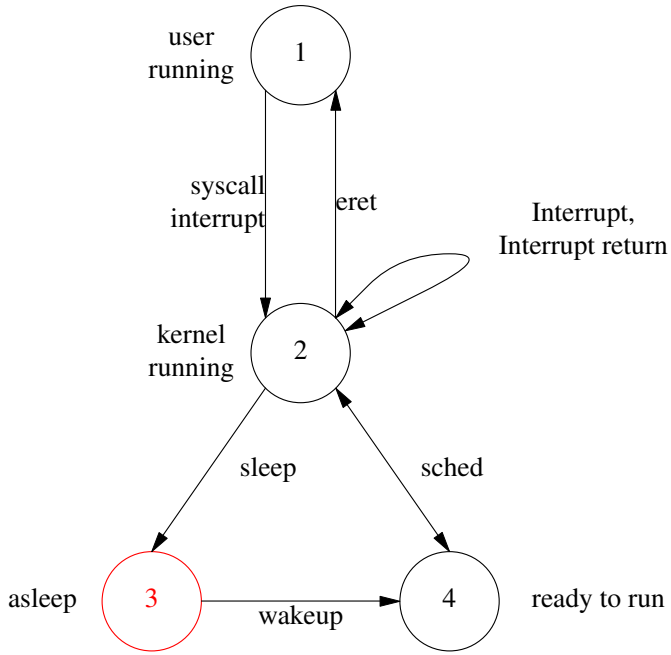
- Wikipedia: It refers to the process of storing the system state for one task, so that task can be paused and another task resumed.
- Can be the scheduler a process?
  - If the scheduler is a processs who schedule the scheduler to schedule the other processes?

Scheduling a process

PROCESS 1



PROCESS 2

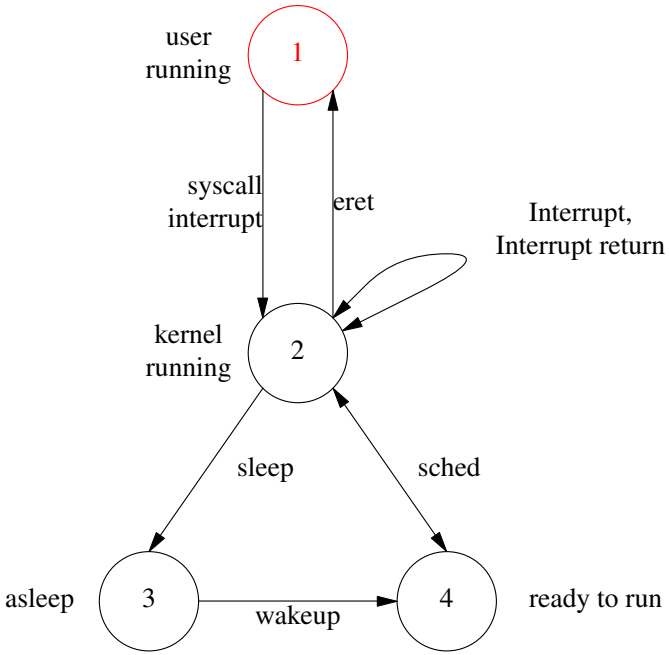


Assumptions

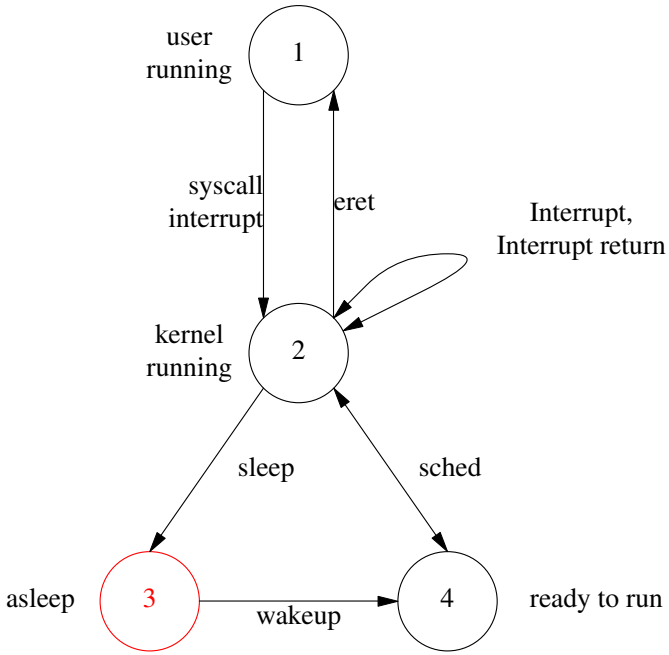
We assume only 1 processor with only 1 core with only 1 hardware thread.

Scheduling a process

PROCESS 1



PROCESS 2

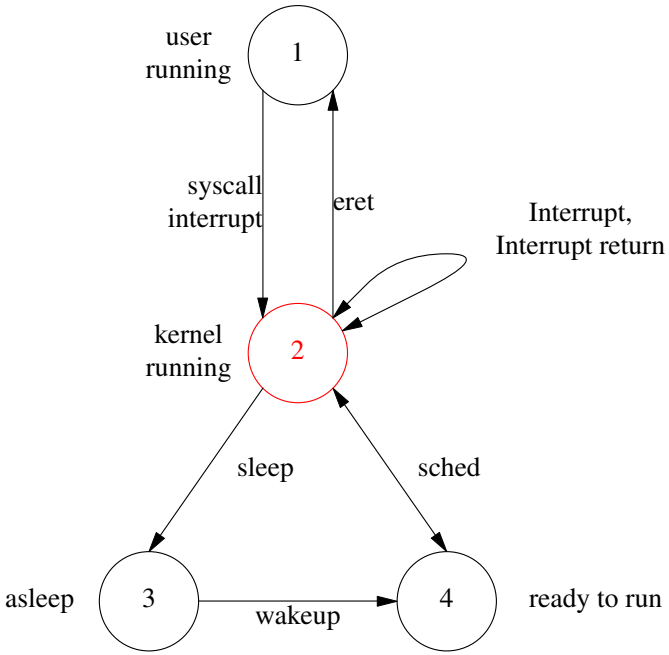


Initial state

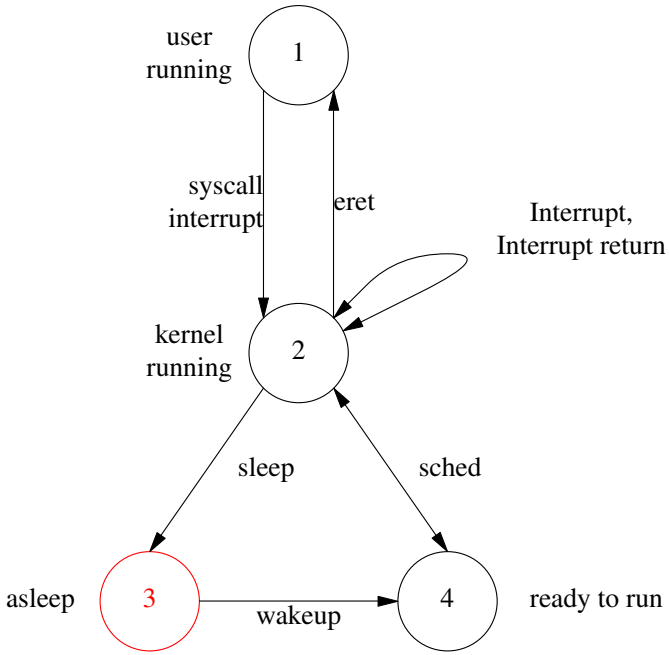
- Process 1 is in User running
- Process 2 is in asleep

Scheduling a process

PROCESS 1



PROCESS 2

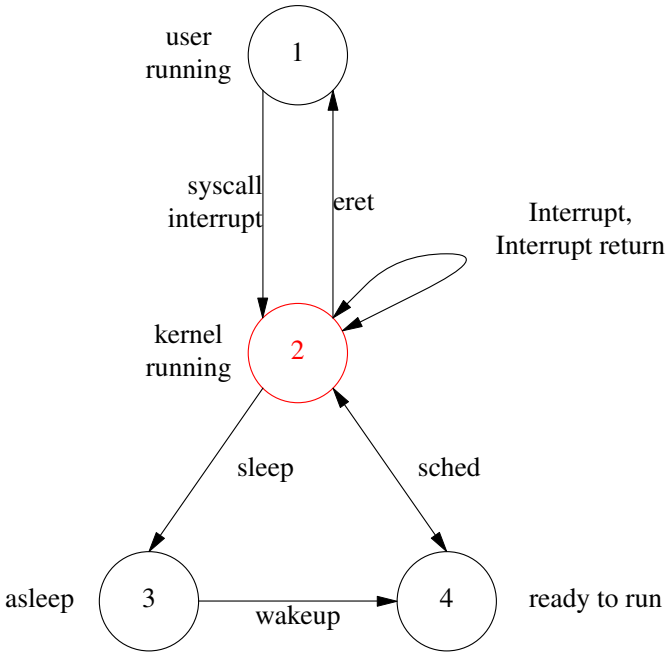


Process 1 does a syscall

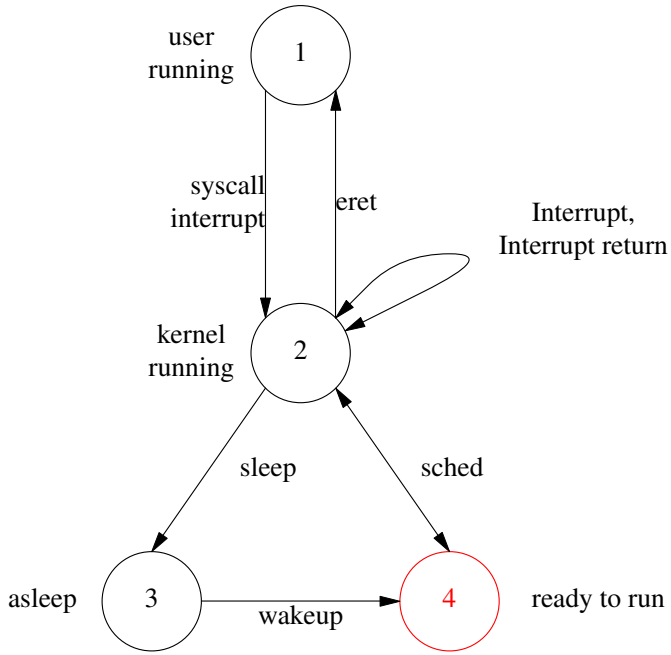
- Process 1 is in Kernel running
- Process 2 is in asleep

Scheduling a process

PROCESS 1



PROCESS 2

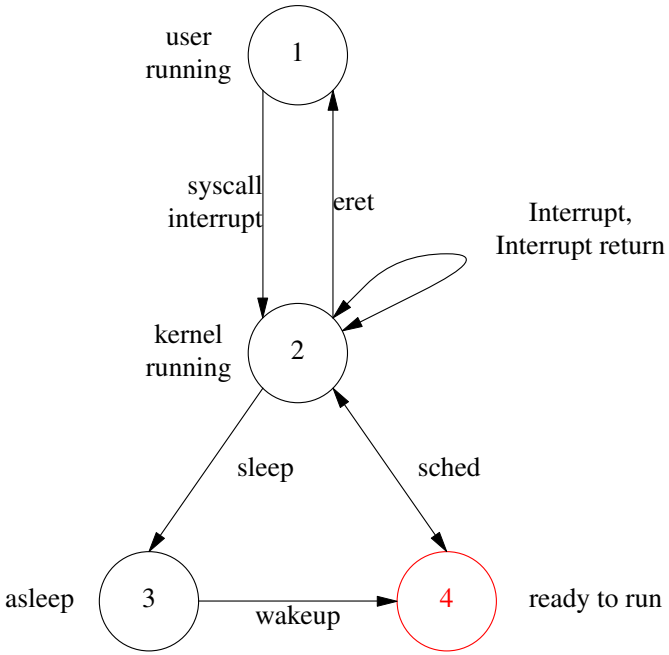


Process 1 wakeup Process 2

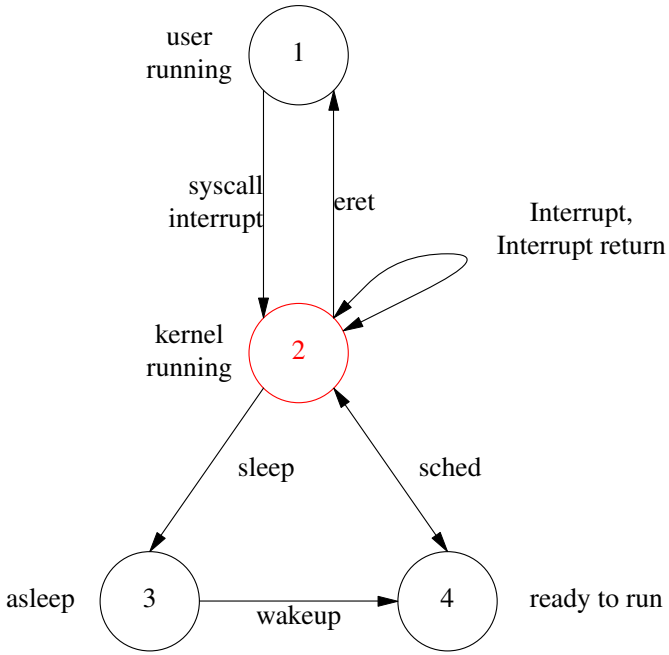
- Process 1 is in Kernel running
- Process 2 is ready to run

Scheduling a process

PROCESS 1



PROCESS 2

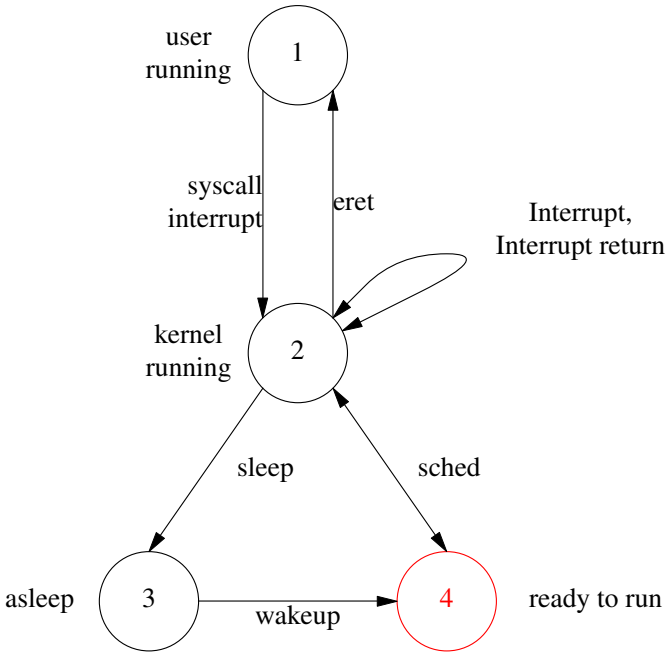


Process 1 calls the sched function

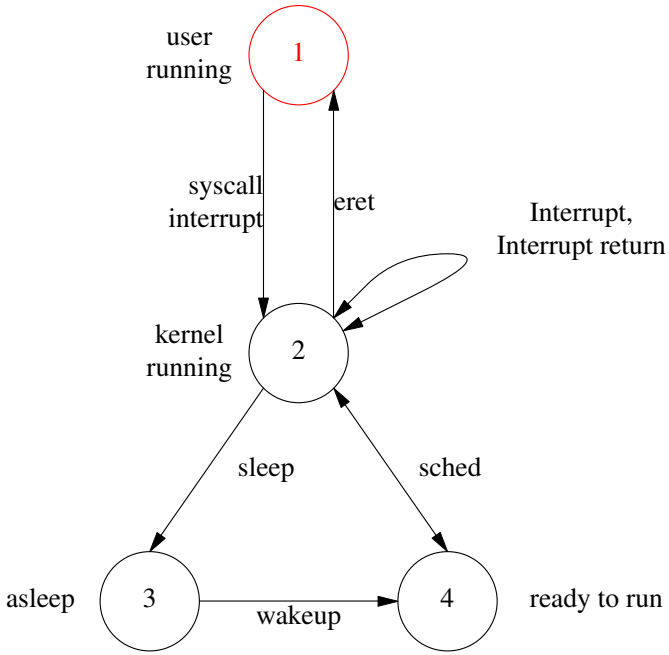
- Process 1 is in ready to run
- Process 2 is kernel running

Scheduling a process

PROCESS 1



PROCESS 2



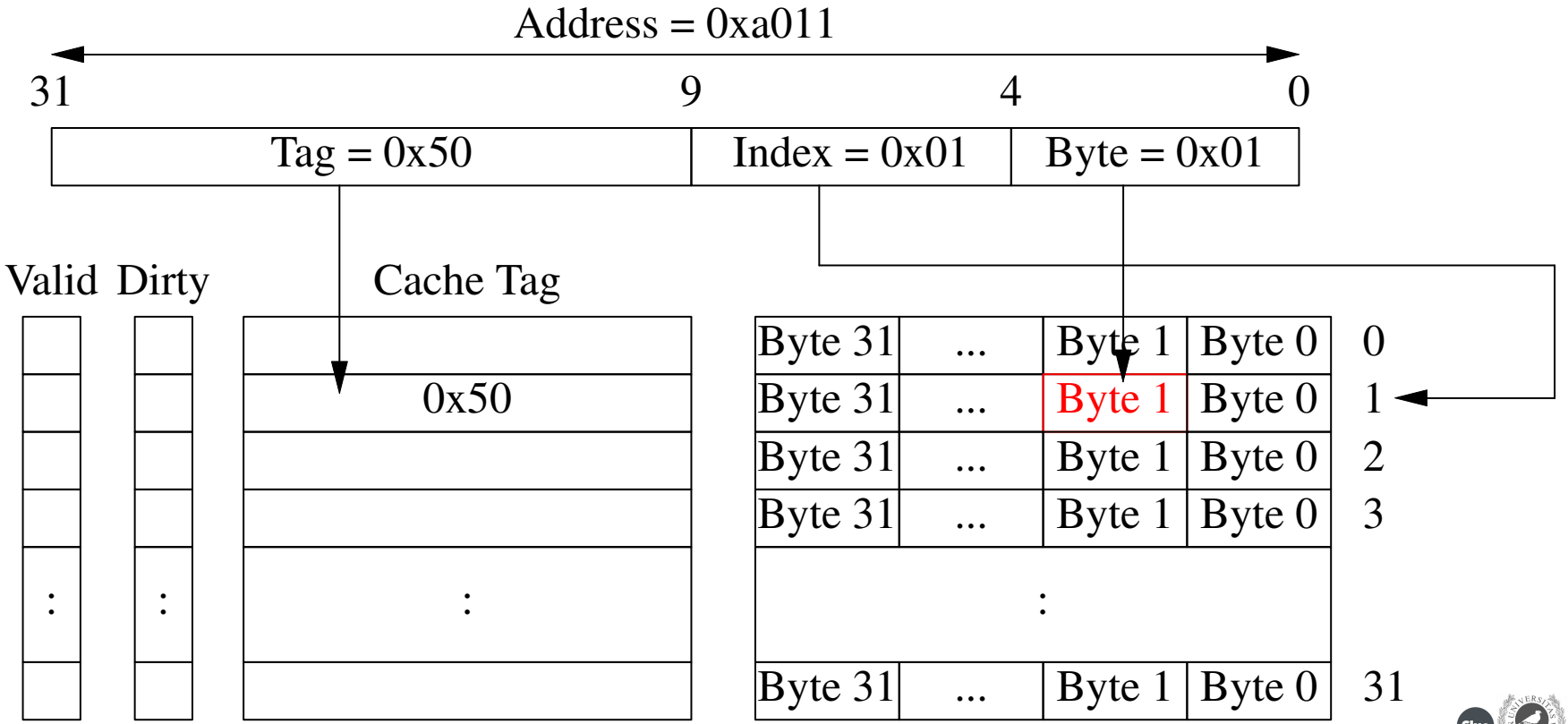
Process 2 executes eret

- Process 1 is in ready to run
- Process 2 is user running

Memory system

Direct mapping, 1 KB, 32Bytes/line

- Capacity = 2^N bytes; Line Size = 2^M:
  - Bits [M-1:0] select the Byte within the Line (ByteInLine, BIL)
  - Bits [N-1:M] are the Index into the cache (a.k.a. "set")
  - Bits [31:N] are the tag



Cache concepts

- Hit: it means that the tag of the address is in the cache

Cache concepts

- Hit: it means that the tag of the address is in the cache
- Miss: it means that the tag of the address is not in the cache

Cache concepts

- Hit: it means that the tag of the address is in the cache
- Miss: it means that the tag of the address is not in the cache
- Must take advantage of the locality principle
  - In computer science, locality of reference, also known as the principle of locality, is the tendency of a processor to access the same set of memory locations repetitively over a short period of time.
    - Temporal locality
    - Spatial locality.

Cache concepts

- Hit: it means that the tag of the address is in the cache
- Miss: it means that the tag of the address is not in the cache
- Must take advantage of the locality principle
  - In computer science, locality of reference, also known as the principle of locality, is the tendency of a processor to access the same set of memory locations repetitively over a short period of time.
    - Temporal locality
    - Spatial locality.
- Evict: Replace a line with other with a different TAG.

Multi level cache

- We can use the locality principle to improve performance

Multi level cache

- We can use the locality principle to improve performance
  - Put the most recent data used in a fast cache inside of the core (L1)

Multi level cache

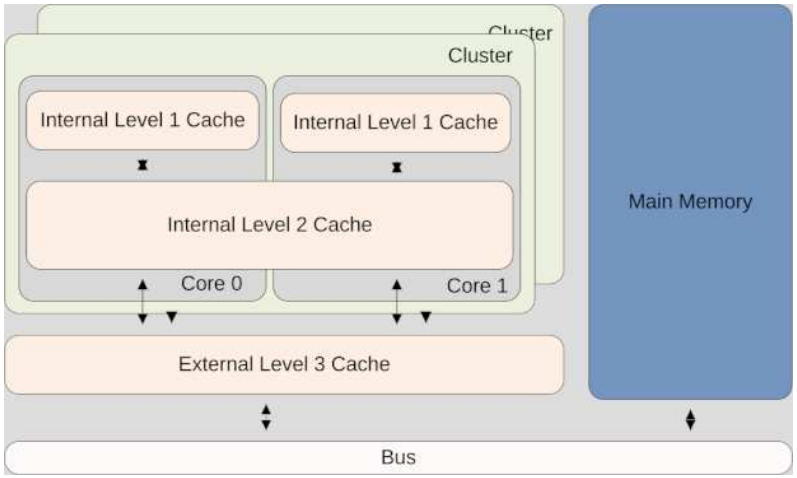
- We can use the locality principle to improve performance
  - Put the most recent data used in a fast cache inside of the core (L1)
  - Put recent data used in a slower but bigger cache inside of the cluster (L2)

Multi level cache

- We can use the locality principle to improve performance
  - Put the most recent data used in a fast cache inside of the core (L1)
  - Put recent data used in a slower but bigger cache inside of the cluster (L2)
  - Put data used in a slower bug bigger cache inside of the SoC or out of the processor (L3)

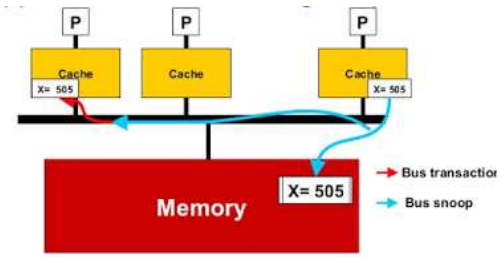
Multi level cache

- We can use the locality principle to improve performance
  - Put the most recent data used in a fast cache inside of the core (L1)
  - Put recent data used in a slower but bigger cache inside of the cluster (L2)
  - Put data used in a slower bug bigger cache inside of the SoC or out of the processor (L3)

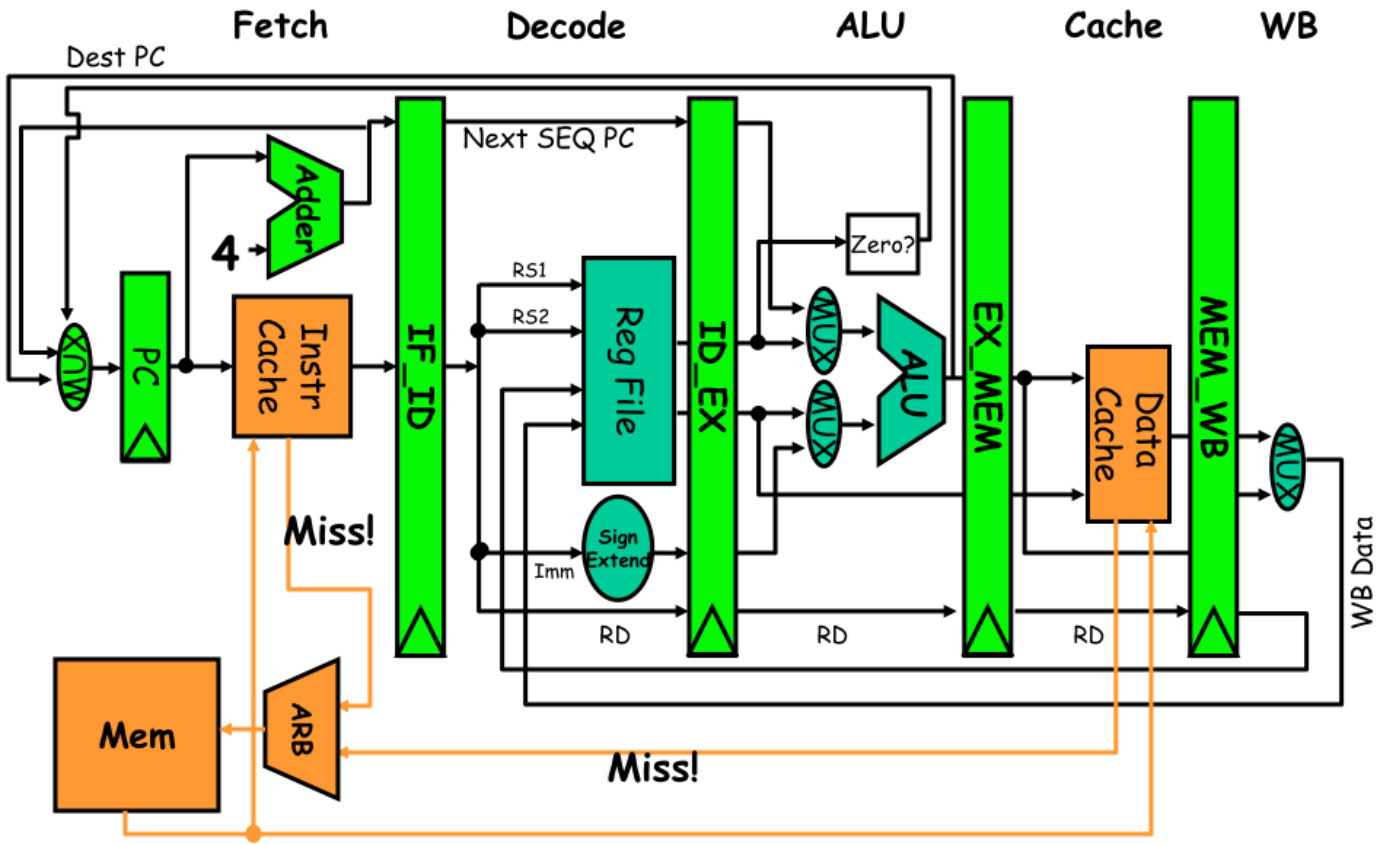


Cache coherence

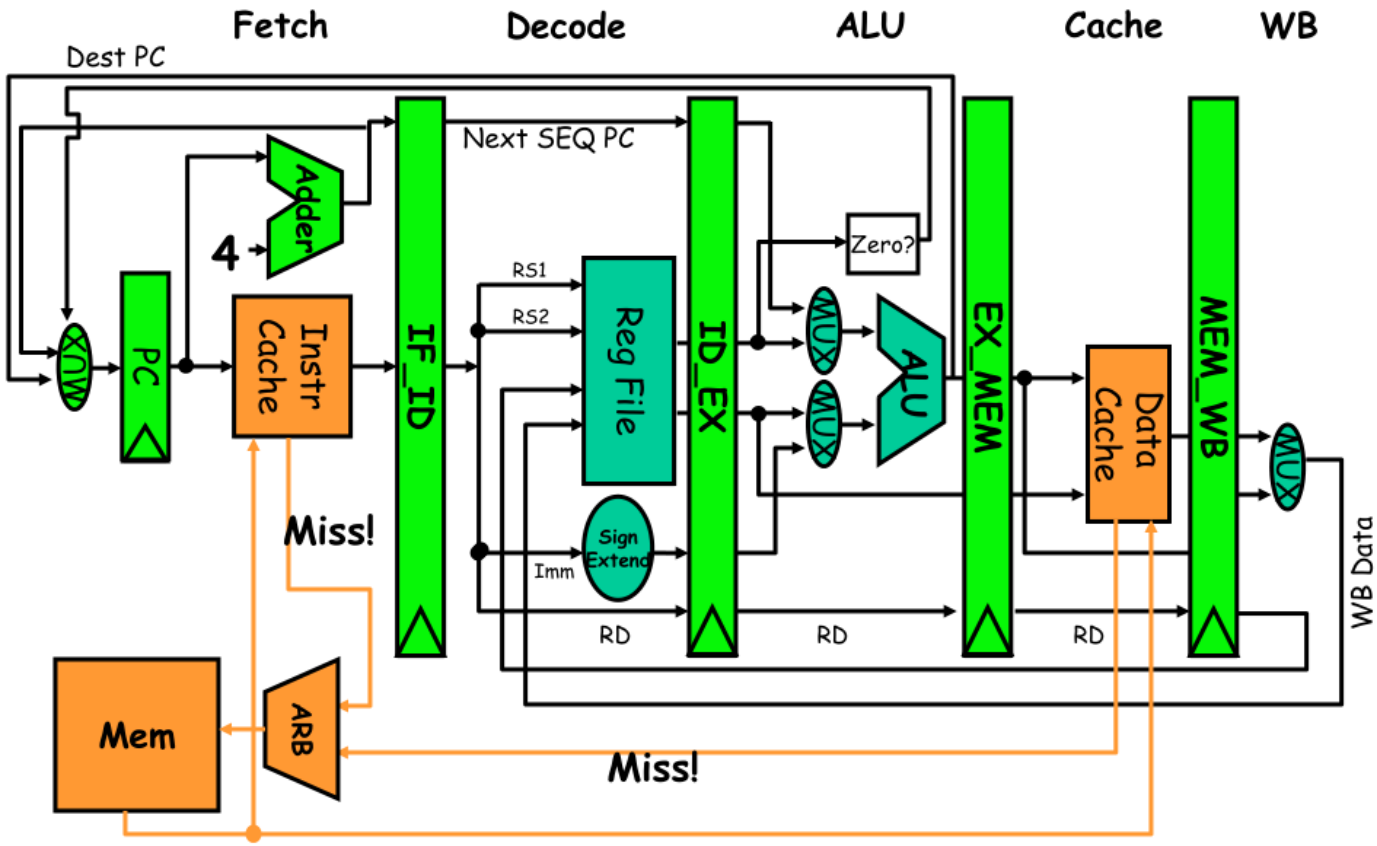
- Bus snooping: A mechanism to ensure cache coherence between multiple caches
- Cache coherence:
  - Is the uniformity of shared resource data that ends up stored in multiple local caches
- Each cache constantly snoop on the bus
- There are multiple protocols to keep the coherence:
  - MSI
  - MESI
  - MOESI



Basic 5 stage pipeline



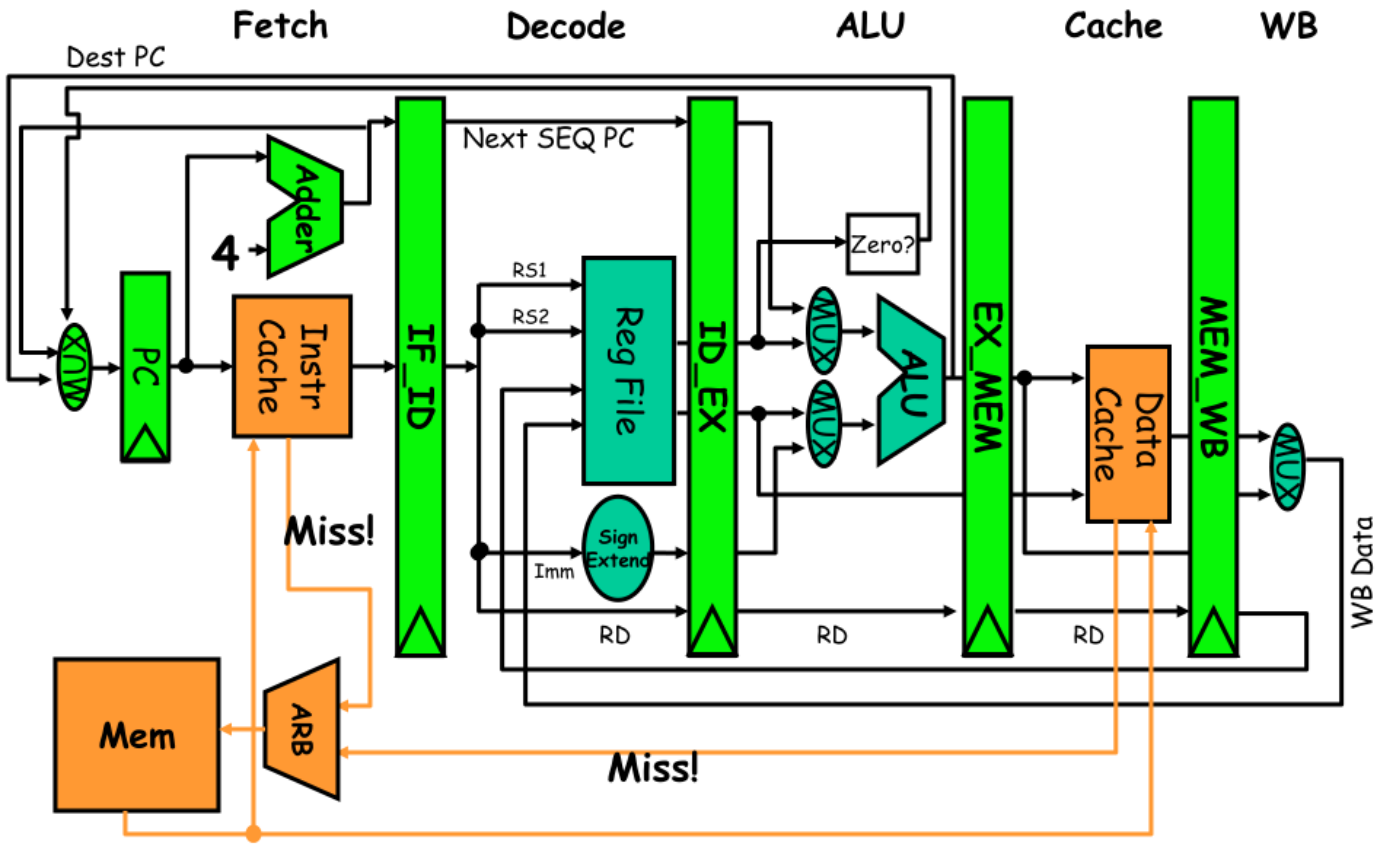
Basic 5 stage pipeline



Fetch stage

It fetchs the next instruction from memory

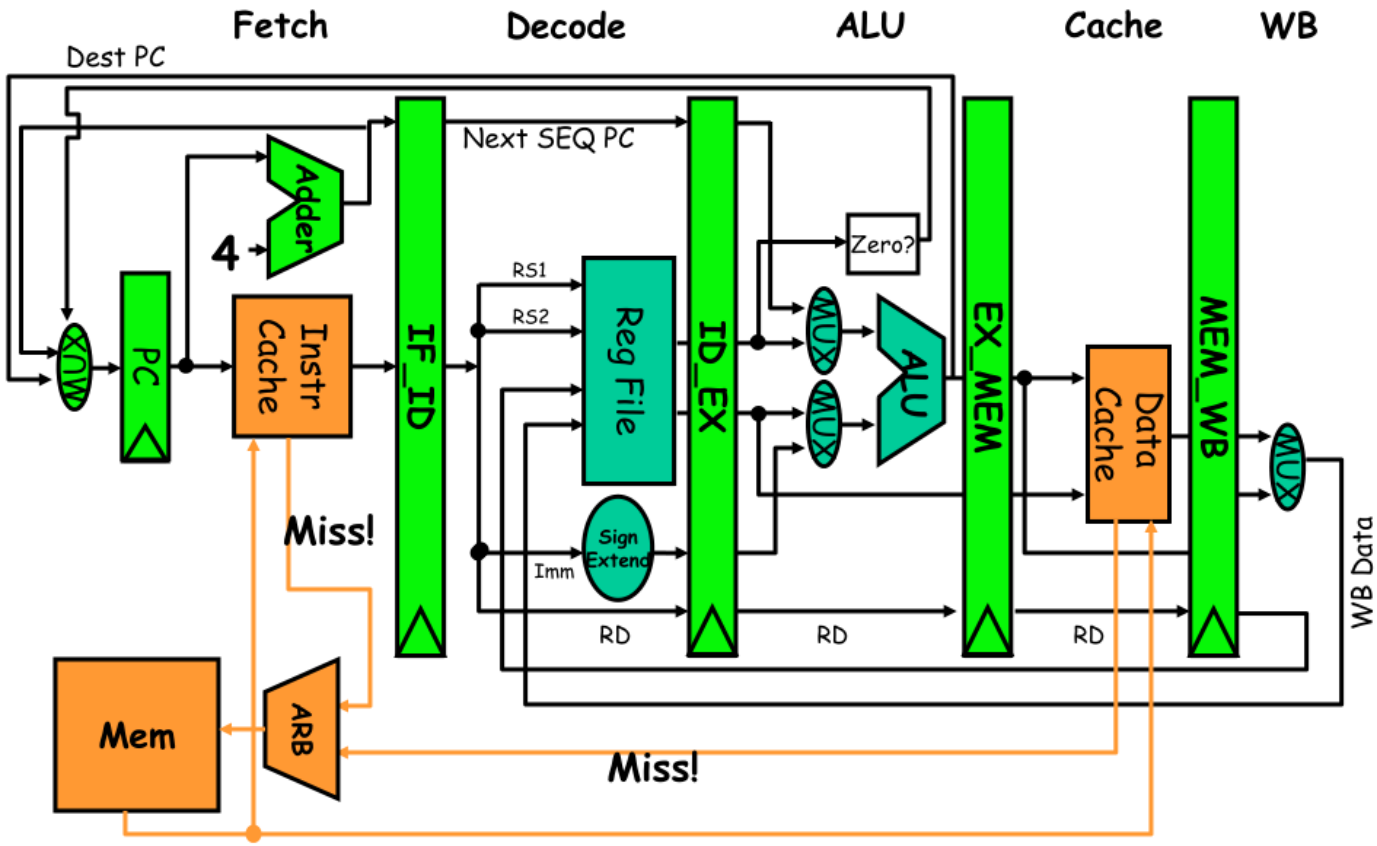
Basic 5 stage pipeline



Decode

It decodes the next instruction and generates all the signals needed by next stages

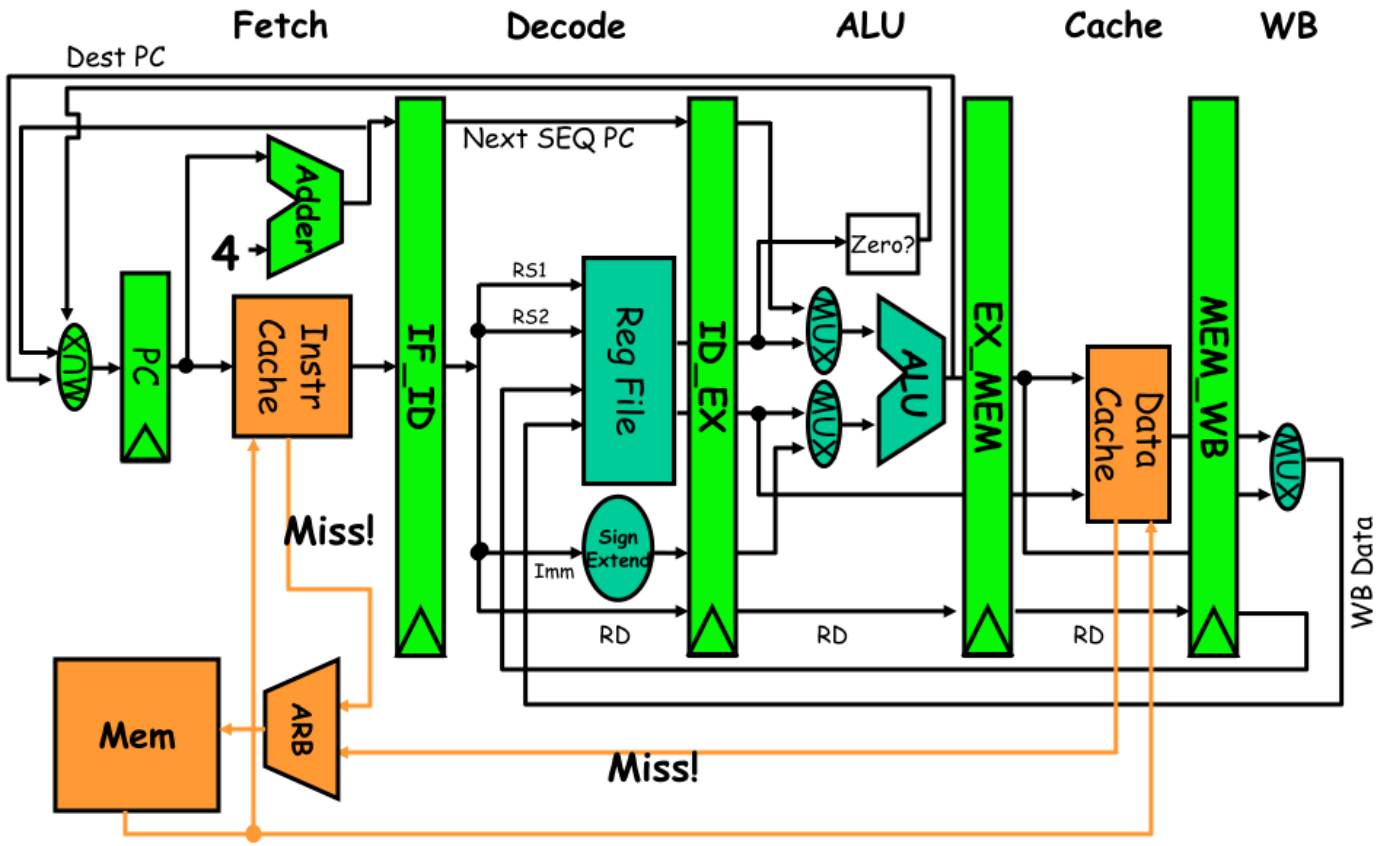
Basic 5 stage pipeline



ALU

It executes the arithmetic or logic operations needed by the instruction

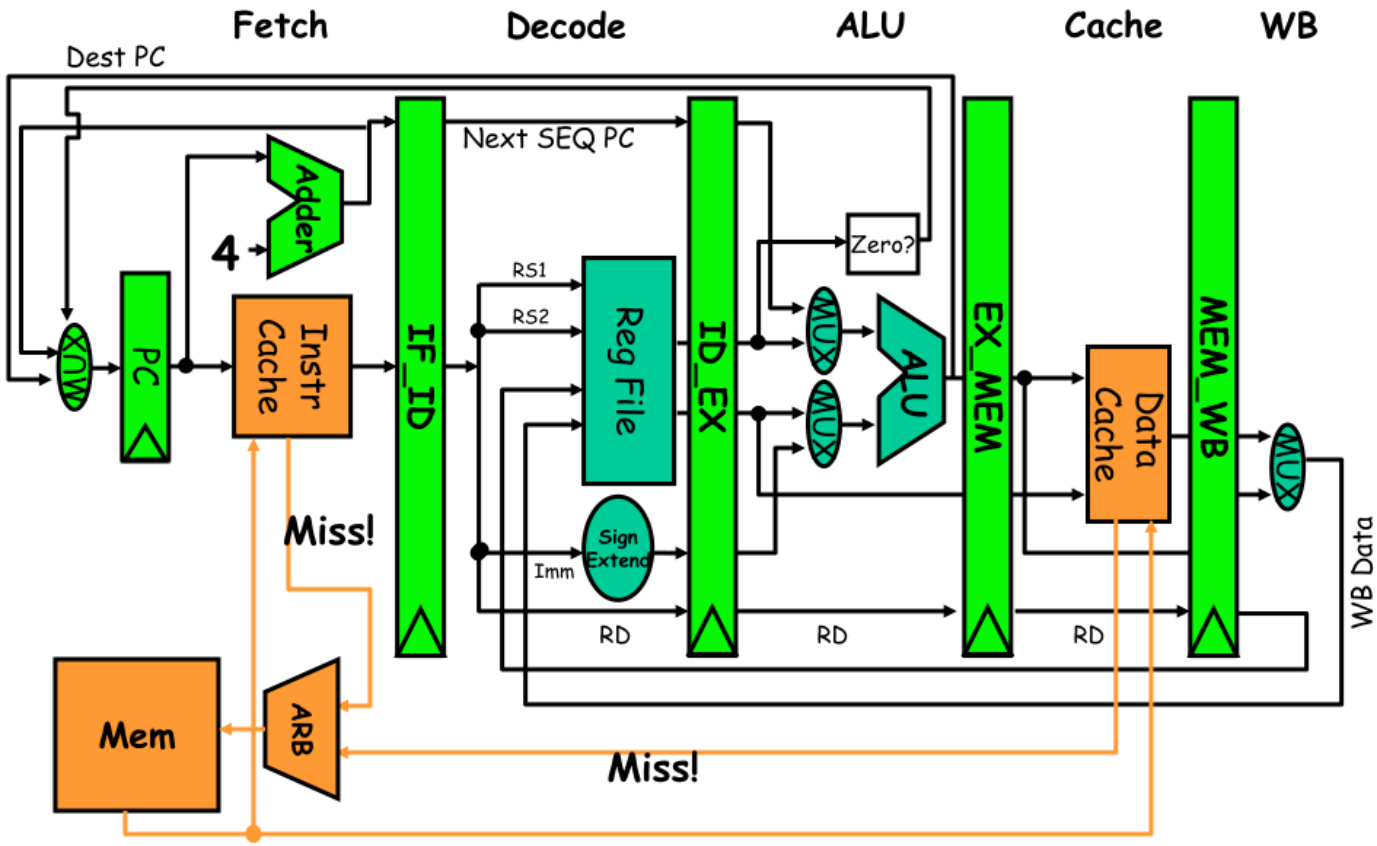
Basic 5 stage pipeline



Cache

It performs any memory read/write needed by the by the instruction

Basic 5 stage pipeline



WB

It writes the result back to the register file in case of being needed.

Instruction cache

- In case of Hit

Instruction cache

- In case of Hit
  - No changes to existing control

Instruction cache

- In case of Hit
  - No changes to existing control
- In case of a miss

Instruction cache

- In case of Hit
  - No changes to existing control
- In case of a miss
  - Hold the PC to avoid losing its value

Instruction cache

- In case of Hit
  - No changes to existing control
- In case of a miss
  - Hold the PC to avoid losing its value
  - Keep sending down a "nop" instruction to decode

Instruction cache

- In case of Hit
  - No changes to existing control
- In case of a miss
  - Hold the PC to avoid losing its value
  - Keep sending down a "nop" instruction to decode
  - Send read request (PC) to the ARB

Instruction cache

- In case of Hit
  - No changes to existing control
- In case of a miss
  - Hold the PC to avoid losing its value
  - Keep sending down a "nop" instruction to decode
  - Send read request (PC) to the ARB
  - Wait until main memory responds with FILL

Data cache

- In case of Hit,

Data cache

- In case of Hit,
  - No changes to existing control

Data cache

- In case of Hit,
  - No changes to existing control
- If we have a miss

Data cache

- In case of Hit,
  - No changes to existing control
- If we have a miss
  - Stall the pipeline

Data cache

- In case of Hit,
  - No changes to existing control
- If we have a miss
  - Stall the pipeline
  - Send read request to the ARB

Data cache

- In case of Hit,
  - No changes to existing control
- If we have a miss
  - Stall the pipeline
  - Send read request to the ARB
  - Wait until memory responds

Data cache

- In case of Hit,
  - No changes to existing control
- If we have a miss
  - Stall the pipeline
  - Send read request to the ARB
  - Wait until memory responds
  - Wait until main memory responds with FILL

Data cache

- In case of Hit,
  - No changes to existing control
- If we have a miss
  - Stall the pipeline
  - Send read request to the ARB
  - Wait until memory responds
  - Wait until main memory responds with FILL
  - Unstall the pipeline

Issues with our current pipeline

- Problem 1
  - For loads&fetches, TAG&data can be accesed in parallel

Issues with our current pipeline

- Problem 1
  - For loads&fetches, TAG&data can be accesed in parallel
  - For stores

Issues with our current pipeline

- Problem 1
  - For loads&fetches, TAG&data can be accesed in parallel
  - For stores
    - This would be a big mistake!

Issues with our current pipeline

- Problem 1
  - For loads&fetches, TAG&data can be accesed in parallel
  - For stores
    - This would be a big mistake!
    - If we did in parallel, we might write the wrong line!

Issues with our current pipeline

- Problem 1
  - For loads&fetches, TAG&data can be accesed in parallel
  - For stores
    - This would be a big mistake!
    - If we did in parallel, we might write the wrong line!
    - First, we must check TAGs

Issues with our current pipeline

- Problem 1
  - For loads&fetches, TAG&data can be accesed in parallel
  - For stores
    - This would be a big mistake!
    - If we did in parallel, we might write the wrong line!
    - First, we must check TAGs
    - Only in HIT, then we can write in the cache

Issues with our current pipeline

- Problem 1
  - For loads&fetches, TAG&data can be accesed in parallel
  - For stores
    - This would be a big mistake!
    - If we did in parallel, we might write the wrong line!
    - First, we must check TAGs
    - Only in HIT, then we can write in the cache
- Problem 2
  - When we need to EVICT a cache line that has been modified

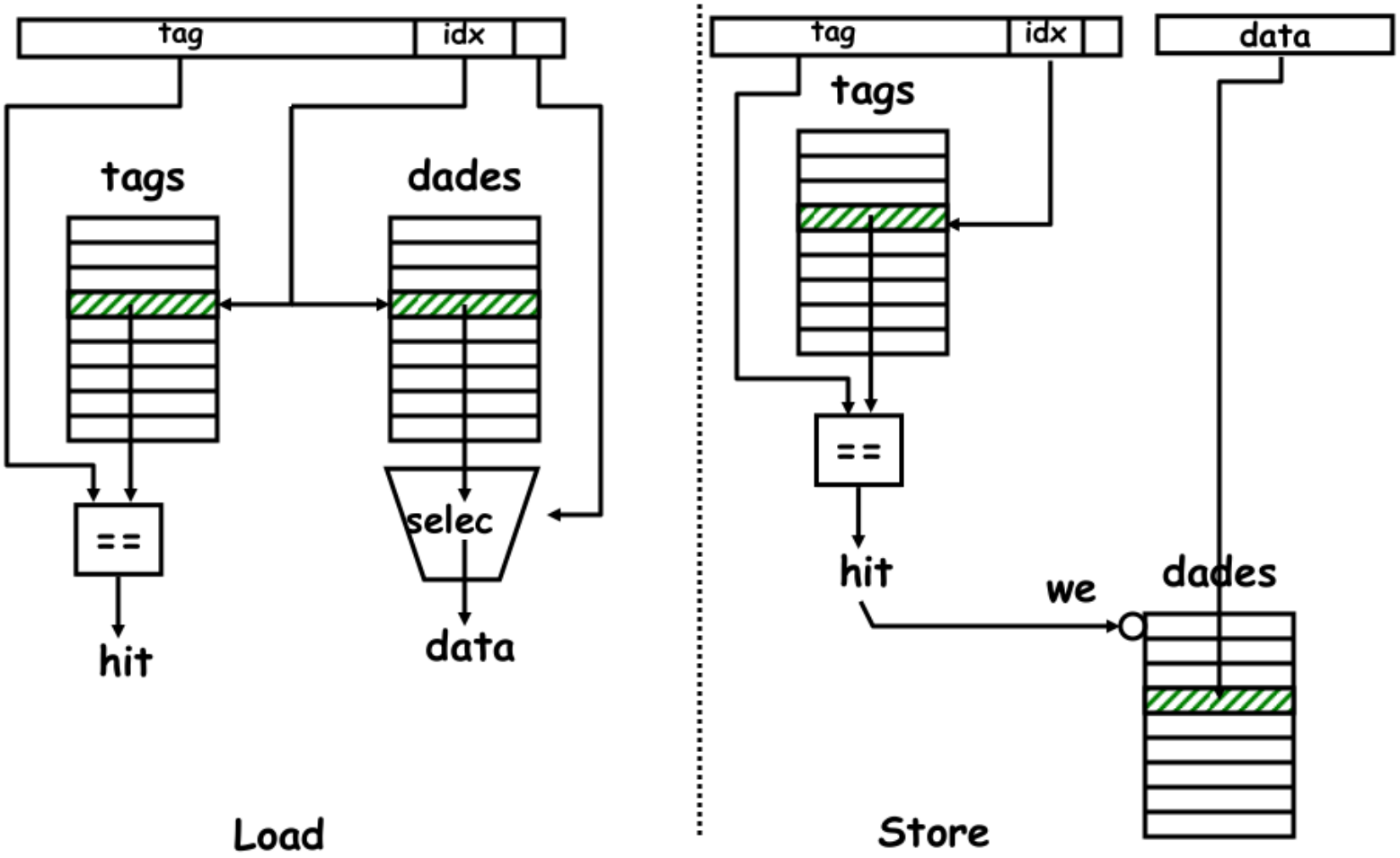
Issues with our current pipeline

- Problem 1
  - For loads&fetches, TAG&data can be accesed in parallel
  - For stores
    - This would be a big mistake!
    - If we did in parallel, we might write the wrong line!
    - First, we must check TAGs
    - Only in HIT, then we can write in the cache
- Problem 2
  - When we need to EVICT a cache line that has been modified
    - First, we need to send it to memory

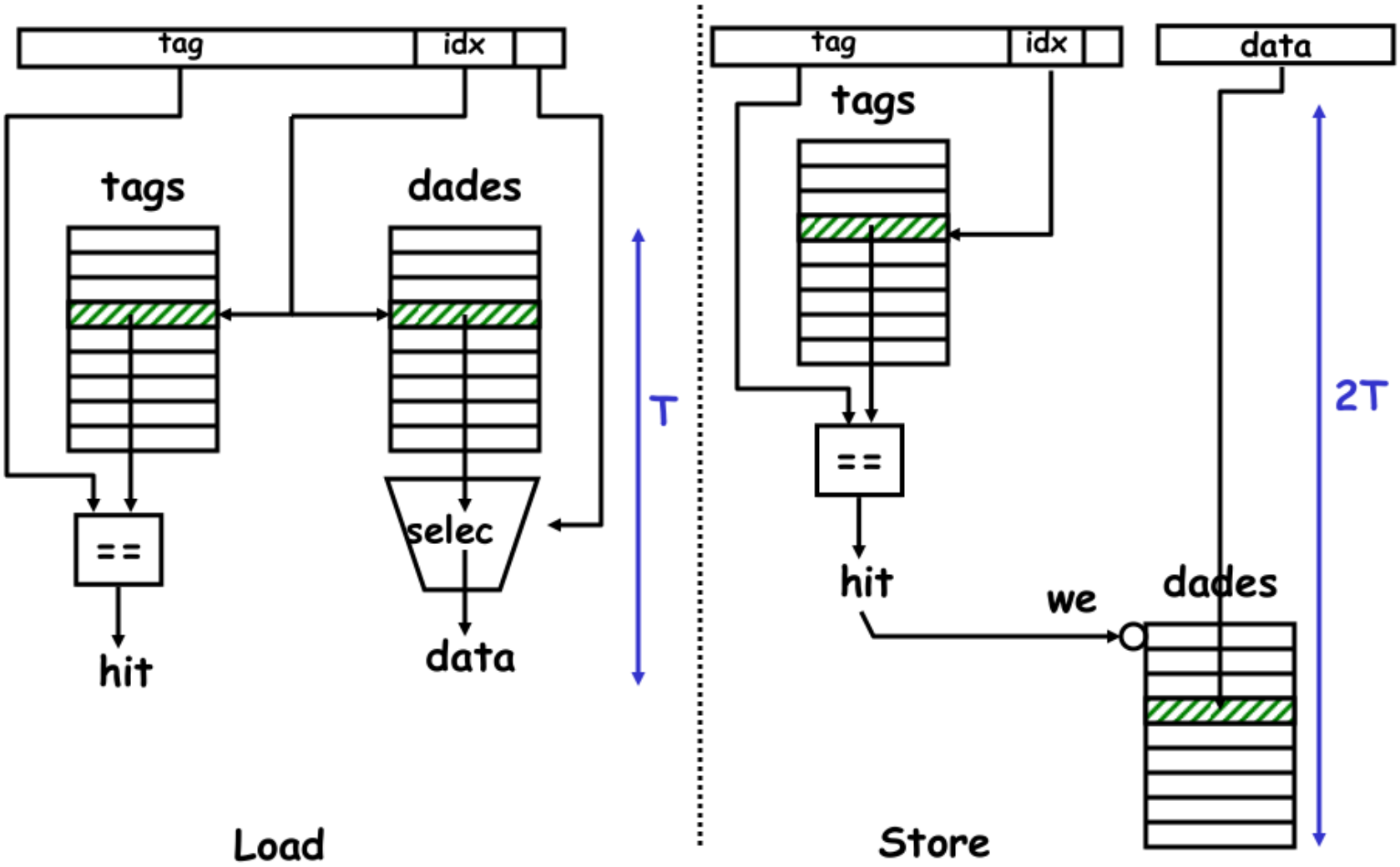
Issues with our current pipeline

- Problem 1
  - For loads&fetches, TAG&data can be accesed in parallel
  - For stores
    - This would be a big mistake!
    - If we did in parallel, we might write the wrong line!
    - First, we must check TAGs
    - Only in HIT, then we can write in the cache
- Problem 2
  - When we need to EVICT a cache line that has been modified
    - First, we need to send it to memory
    - AKA "Dirty replacement" or "Dirty Eviction"

Basic flow in the cache



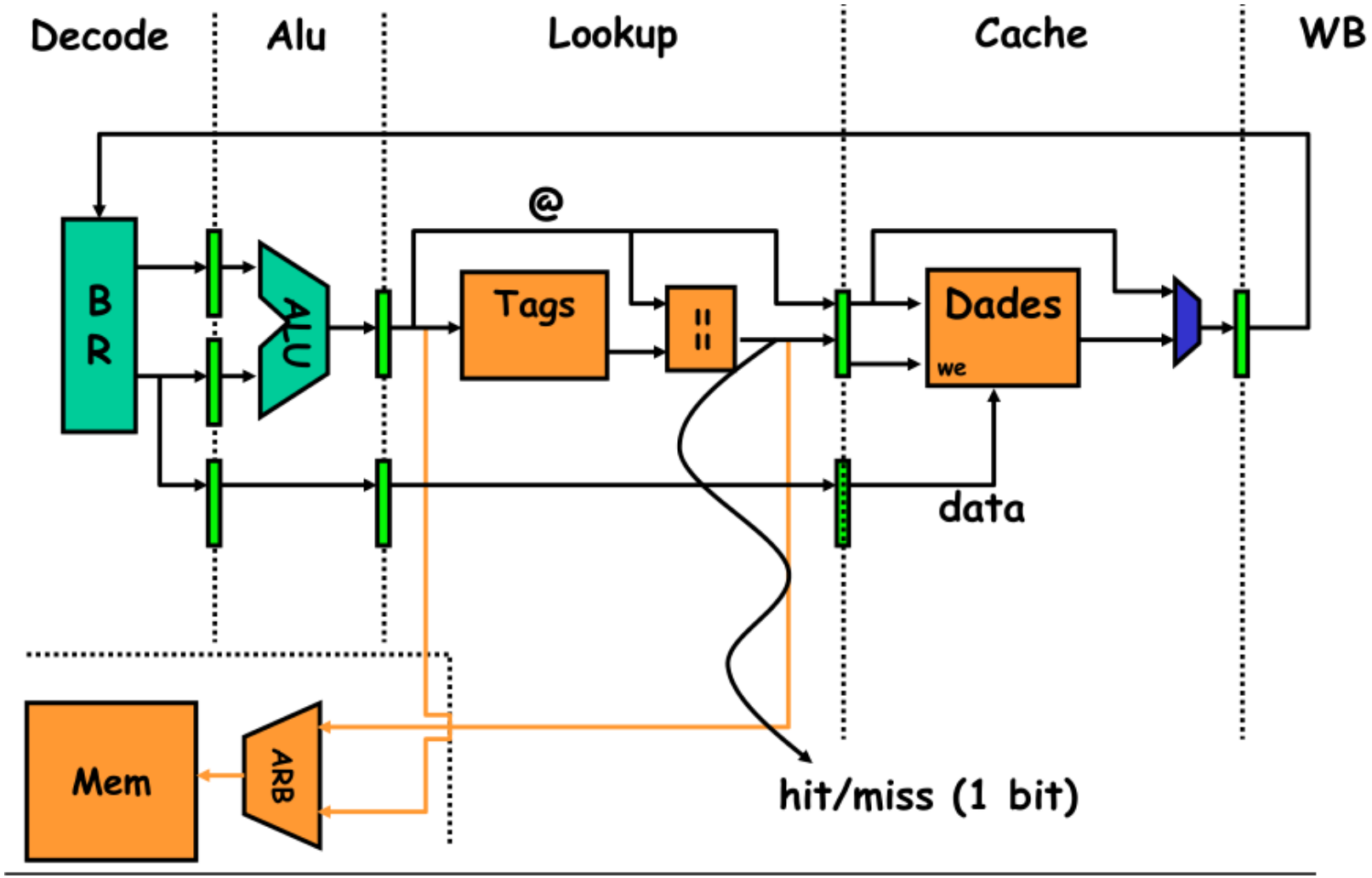
Problem 1: Can we fit in a single cycle?



Proposal 1: Add a new pipe stage

- Add new pipe stage: “tag lookup”
- ”Lookup” stage
  - Index the tag array with the index bits from @
  - Compare the tag output with the @
  - Pass on the hit/miss indication to next stage
  - In case of miss, send miss request to ARB
- Cache stage
  - If load & HIT, read data and pass on to WB stage
  - If store & HIT, write data into data array
  - Else... Do nothing

New Pipeline



Dirty Eviction

- Eviction:

Dirty Eviction

- Eviction:
  - Replacing a line in the cache by another line

Dirty Eviction

- Eviction:
  - Replacing a line in the cache by another line
- Dirty:

Dirty Eviction

- Eviction:
  - Replacing a line in the cache by another line
- Dirty:
  - A line is dirty if it has been modified by the processor
- Assume we have a load that misses

Dirty Eviction

- Eviction:
  - Replacing a line in the cache by another line
- Dirty:
  - A line is dirty if it has been modified by the processor
- Assume we have a load that misses
  - Load accesses line 12 in the cache
  - What happens if line 12 is dirty?
  - Can we just fill on top of line 12?

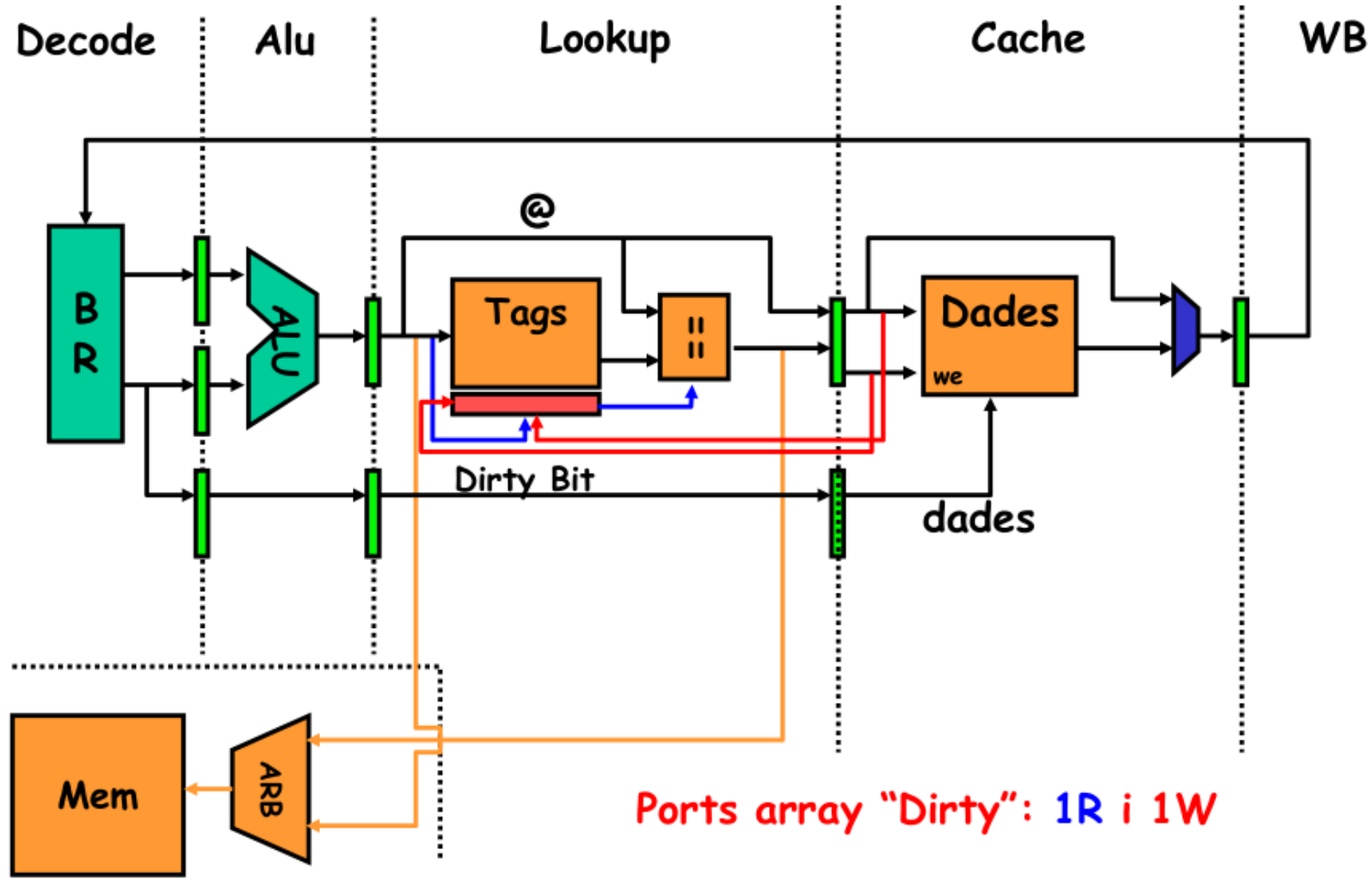
Dirty Eviction

- Eviction:
  - Replacing a line in the cache by another line
- Dirty:
  - A line is dirty if it has been modified by the processor
- Assume we have a load that misses
  - Load accesses line 12 in the cache
  - What happens if line 12 is dirty?
  - Can we just fill on top of line 12?
    - NOOOO!!

Dirty Eviction

- Eviction:
  - Replacing a line in the cache by another line
- Dirty:
  - A line is dirty if it has been modified by the processor
- Assume we have a load that misses
  - Load accesses line 12 in the cache
  - What happens if line 12 is dirty?
  - Can we just fill on top of line 12?
    - NOOOO!!
    - First we must send line 12 to memory
    - Only then can we go fetch the new line and fill it into the cache

Use of the "dirty bit"



Changes to the miss logic

- In the lookup stage, we read the dirty array
- If line is dirty and we have a miss

Changes to the miss logic

- In the lookup stage, we read the dirty array
- If line is dirty and we have a miss
  - Stall pipeline (F, D, A)

Changes to the miss logic

- In the lookup stage, we read the dirty array
- If line is dirty and we have a miss
  - Stall pipeline (F, D, A)
  - C and WB must keep going

Changes to the miss logic

- In the lookup stage, we read the dirty array
- If line is dirty and we have a miss
  - Stall pipeline (F, D, A)
  - C and WB must keep going
  - Send dirty line to memory

Changes to the miss logic

- In the lookup stage, we read the dirty array
- If line is dirty and we have a miss
  - Stall pipeline (F, D, A)
  - C and WB must keep going
  - Send dirty line to memory
  - Wait for ACK

Changes to the miss logic

- In the lookup stage, we read the dirty array
- If line is dirty and we have a miss
  - Stall pipeline (F, D, A)
  - C and WB must keep going
  - Send dirty line to memory
  - Wait for ACK
  - Request new line

Changes to the miss logic

- In the lookup stage, we read the dirty array
- If line is dirty and we have a miss
  - Stall pipeline (F, D, A)
  - C and WB must keep going
  - Send dirty line to memory
  - Wait for ACK
  - Request new line
  - Fill the new data into the cache

Changes to the miss logic

- In the lookup stage, we read the dirty array
- If line is dirty and we have a miss
  - Stall pipeline (F, D, A)
  - C and WB must keep going
  - Send dirty line to memory
  - Wait for ACK
  - Request new line
  - Fill the new data into the cache
  - Unlock pipeline

What about loads

What about loads

- Loads don't really need this extra lookup stage
  - So, what should we do with them?

What about loads

- Loads don't really need this extra lookup stage
  - So, what should we do with them?
- We like uniform pipelines

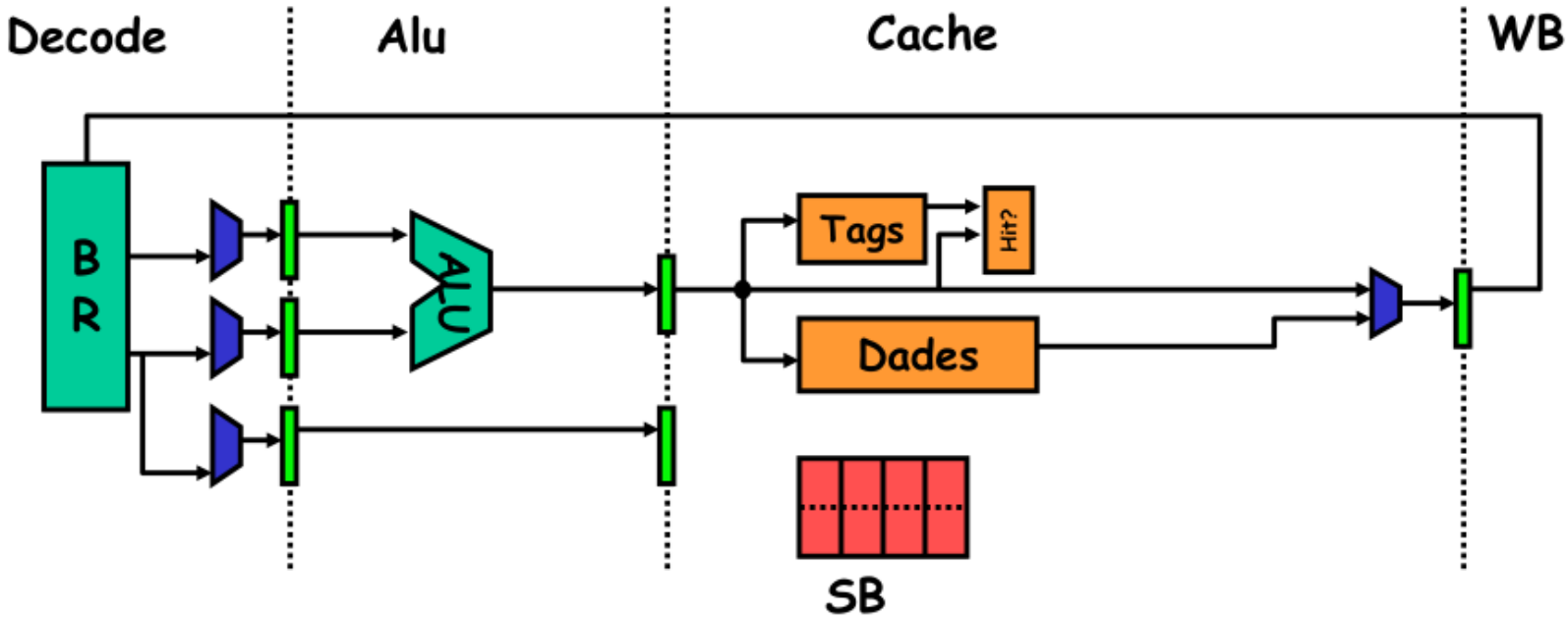
What about loads

- Loads don't really need this extra lookup stage
  - So, what should we do with them?
- We like uniform pipelines
  - Loads will also use the lookup and c stages
  - Problem

What about loads

- Loads don't really need this extra lookup stage
  - So, what should we do with them?
- We like uniform pipelines
  - Loads will also use the lookup and c stages
  - Problem
    - The MEM to ALU stall will be longer (lower perf)

Proposal 2: Pipeline with store buffer



Store buffer

- Holds stores that have not yet completed
- When the C stage is free, then we copy the data from the store buffer into the data array

Store buffer

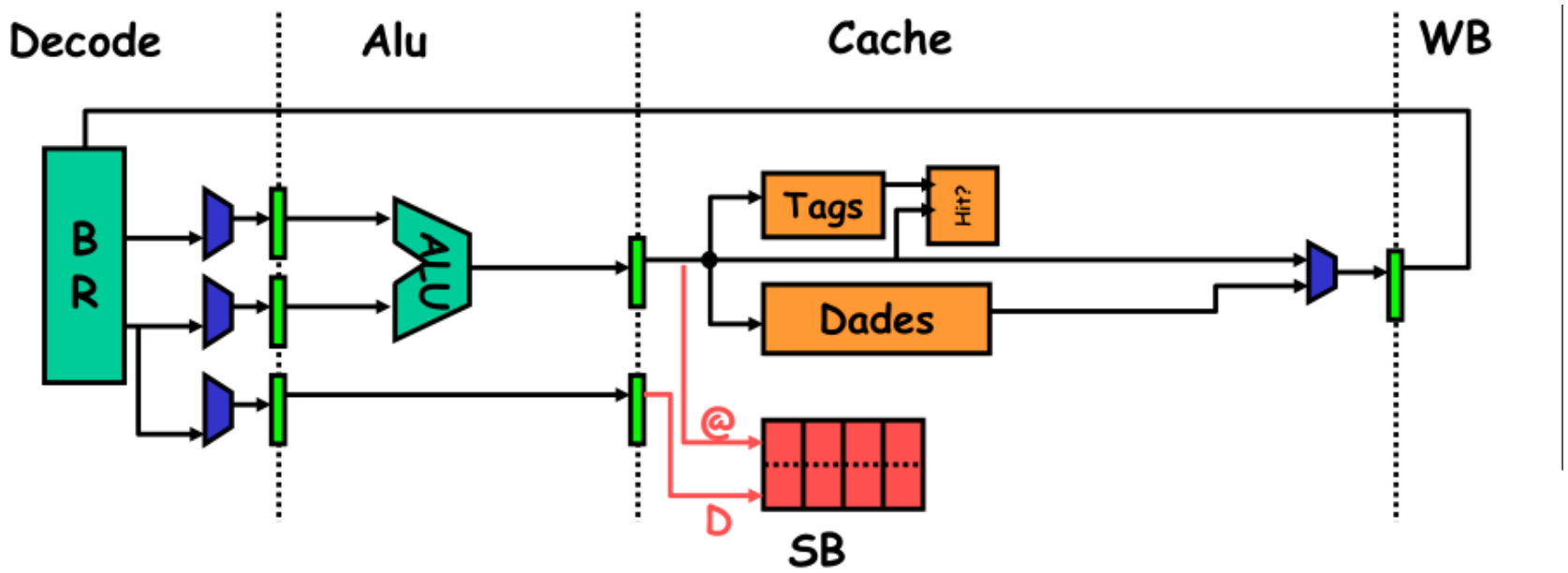
- Holds stores that have not yet completed
- When the C stage is free, then we copy the data from the store buffer into the data array
- Now loads need to read from the store buffer

Store buffer

- Holds stores that have not yet completed
- When the C stage is free, then we copy the data from the store buffer into the data array
- Now loads need to read from the store buffer
  - To get the most recent data

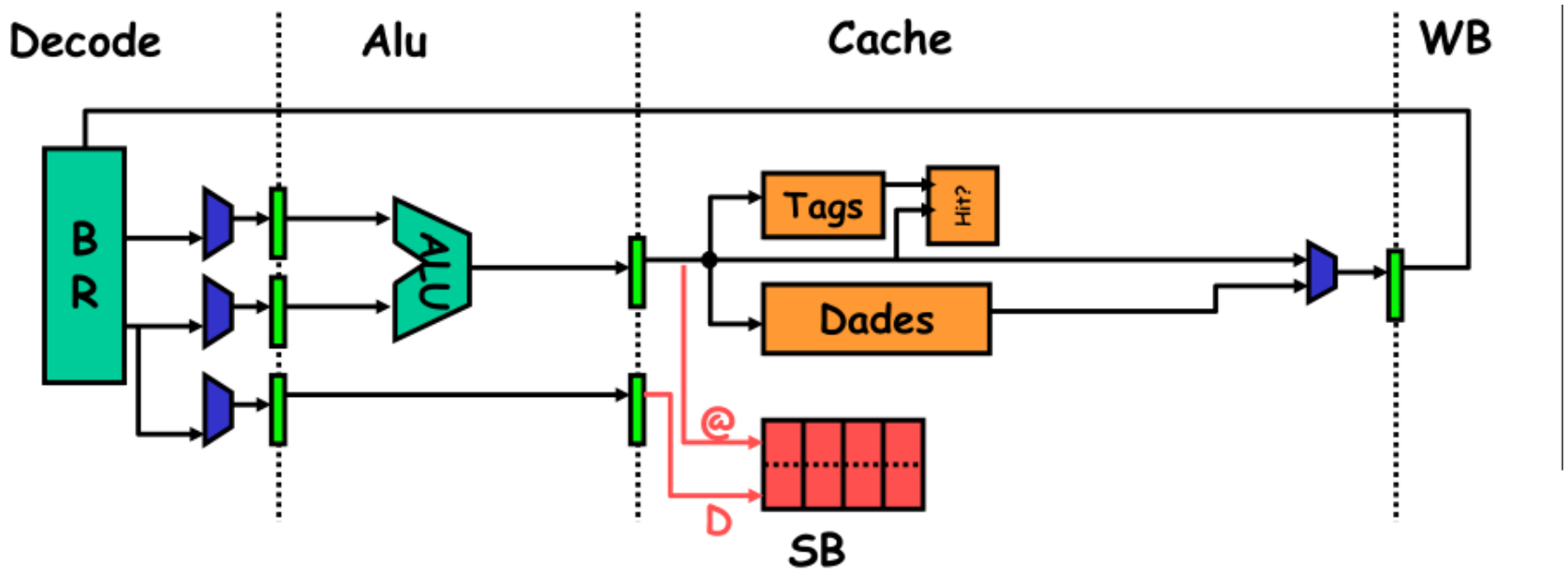
Control for a store in the "C" stage

- Read tags and check for hit/miss
- If miss, block pipeline and send req to ARB
- Else, save the @ and the data into the SB
  - We have not written anything into the data array!!!



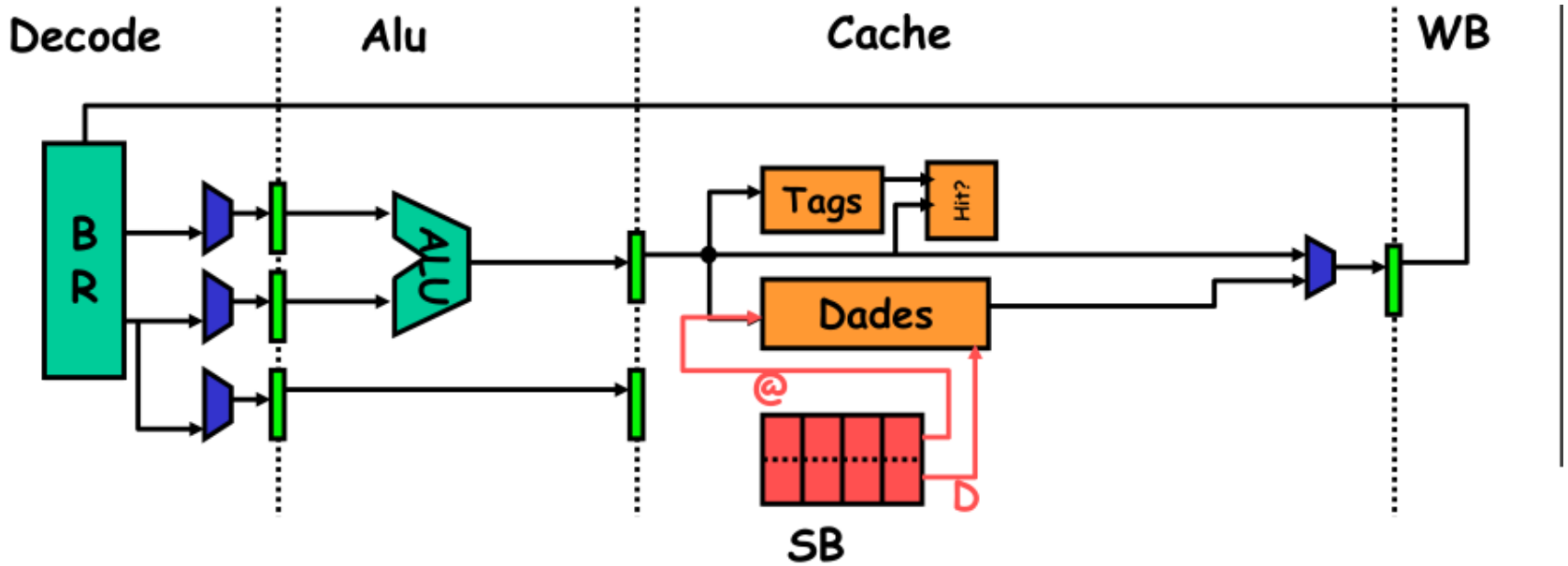
Control for a store in the "C" stage

- Read tags and check for hit/miss
- If miss, block pipeline and send req to ARB
- Else, save the @ and the data into the SB
  - We have not written anything into the data array!!!
  - What happens with multi cores?



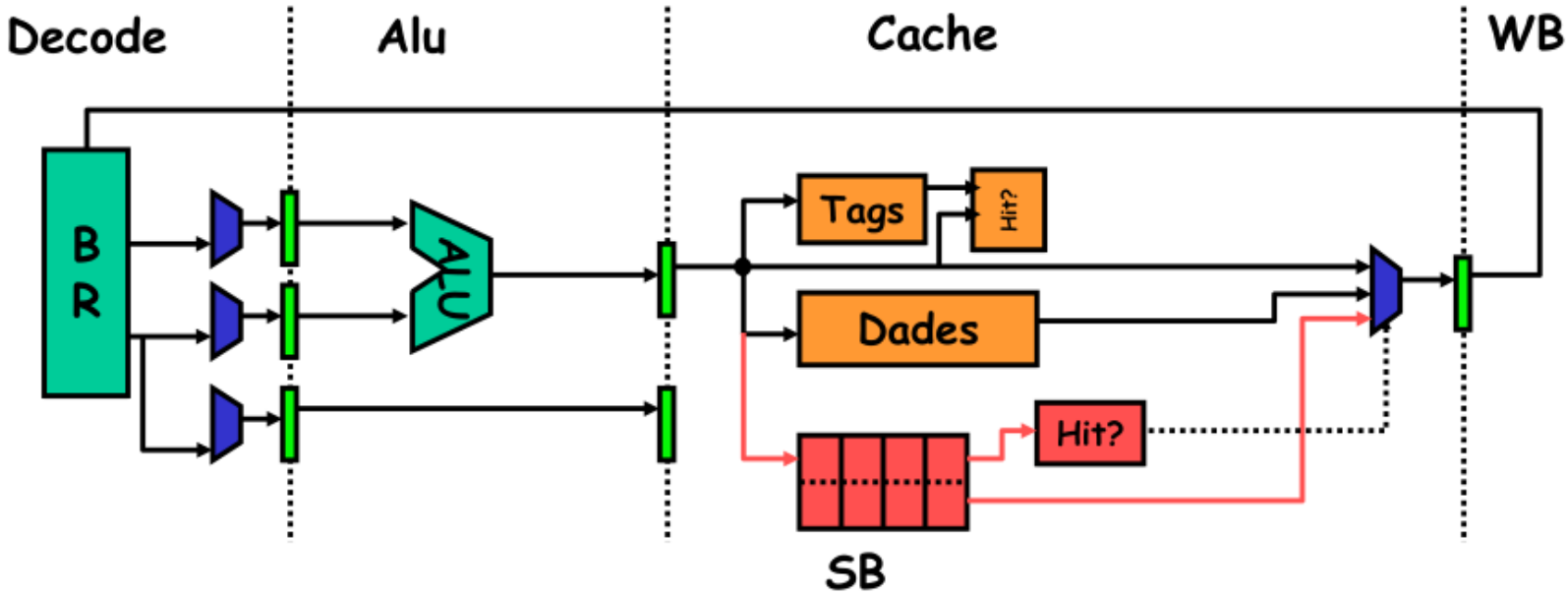
Control for the c stage for an ALU op

- ALU result flows through to the WB stage
- If SB is NOT empty
  - Take the oldest store from the SB
  - Write it into the Cache



C stage control for a "load"

- Index the tag array and determine hit/miss
- Index the data array and read data (I'm feeling lucky)
- Compare the load @ with all the @s in the SB
  - If hit, use the data from the SB
  - If miss (and we did hit in the cache) use the data from the cache
  - If miss in both places, go ask main memory



Load miss

- Let idx(L) be the index of the load into the cache
- Let idx(SBi) be the index of store buffer entry "i"
- What happens if we have idx(L) == idx(SBi)?

Load miss

- Let idx(L) be the index of the load into the cache
- Let idx(SBi) be the index of store buffer entry "i"
- What happens if we have idx(L) == idx(SBi)?
  - We have a pending write in the SB

Load miss

- Let idx(L) be the index of the load into the cache
- Let idx(SBi) be the index of store buffer entry "i"
- What happens if we have idx(L) == idx(SBi)?
  - We have a pending write in the SB
- If we make a blind request to mem

Load miss

- Let idx(L) be the index of the load into the cache
- Let idx(SBi) be the index of store buffer entry "i"
- What happens if we have idx(L) == idx(SBi)?
  - We have a pending write in the SB
- If we make a blind request to mem
  - We fill the requested data into the data array
  - We will be "stepping on" the cache line that was pending to be written by SBi and we will be causing a functional failure
- Solution
  - Stall the pipeline

Load miss

- Let idx(L) be the index of the load into the cache
- Let idx(SBi) be the index of store buffer entry "i"
- What happens if we have idx(L) == idx(SBi)?
  - We have a pending write in the SB
- If we make a blind request to mem
  - We fill the requested data into the data array
  - We will be "stepping on" the cache line that was pending to be written by SBi and we will be causing a functional failure
- Solution
  - Stall the pipeline
  - Drain the SB into the cache

Load miss

- Let idx(L) be the index of the load into the cache
- Let idx(SBi) be the index of store buffer entry "i"
- What happens if we have idx(L) == idx(SBi)?
  - We have a pending write in the SB
- If we make a blind request to mem
  - We fill the requested data into the data array
  - We will be "stepping on" the cache line that was pending to be written by SBi and we will be causing a functional failure
- Solution
  - Stall the pipeline
  - Drain the SB into the cache
  - Unstall the pipeline

SB Full?

- If a store finds the SB full we then need to

SB Full?

- If a store finds the SB full we then need to
  - Stall pipeline

SB Full?

- If a store finds the SB full we then need to
  - Stall pipeline
  - At least drain 1 entry from the SB into the cache to leave one free entry for the current store

SB Full?

- If a store finds the SB full we then need to
  - Stall pipeline
  - At least drain 1 entry from the SB into the cache to leave one free entry for the current store
  - Unstall pipeline

SB Full?

- If a store finds the SB full we then need to
  - Stall pipeline
  - At least drain 1 entry from the SB into the cache to leave one free entry for the current store
  - Unstall pipeline
- It's proably a good idea to flush the full SB

SB Full?

- If a store finds the SB full we then need to
  - Stall pipeline
  - At least drain 1 entry from the SB into the cache to leave one free entry for the current store
  - Unstall pipeline
- It's proably a good idea to flush the full SB
- Guidance for the compiler writer?

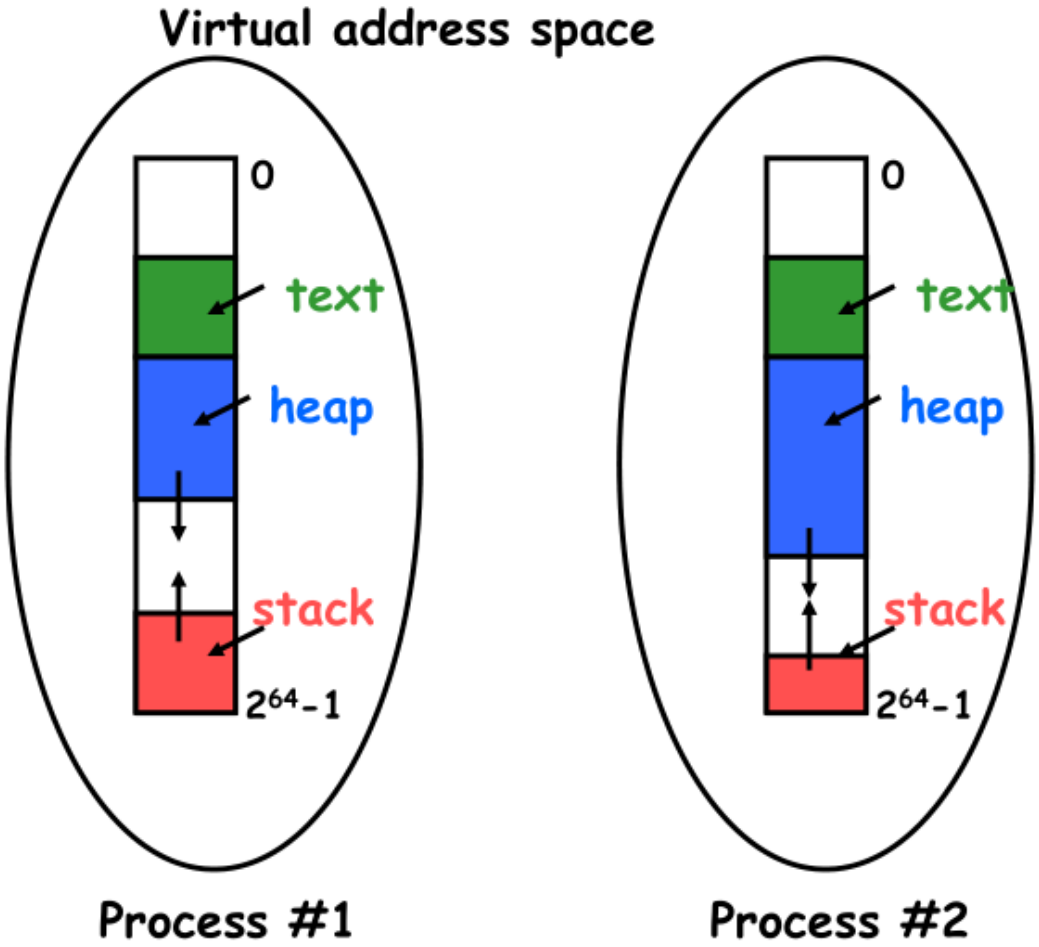
SB Full?

- If a store finds the SB full we then need to
  - Stall pipeline
  - At least drain 1 entry from the SB into the cache to leave one free entry for the current store
  - Unstall pipeline
- It's proably a good idea to flush the full SB
- Guidance for the compiler writer?
  - Interleave stores with arithmetic ops

Virtual memory

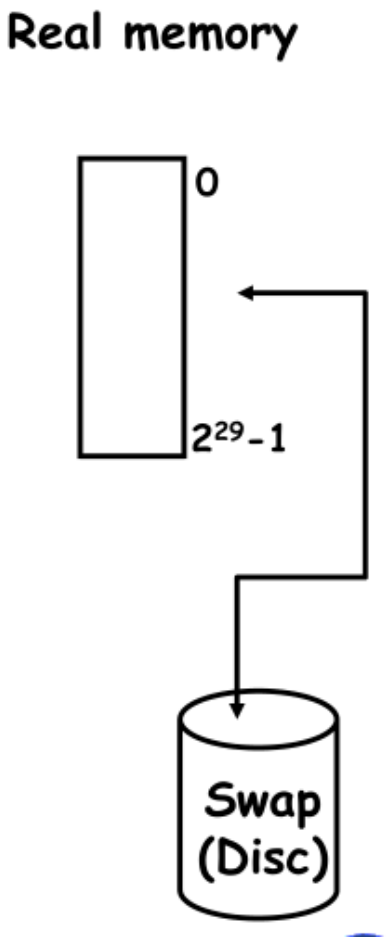
- Operating systems present a virtual memory address space to each process running on the system.
- Each and every one of the processes believes he is living in a machine with 2^n bytes, where "n" is the machine address width
  - Alpha: n=64, virtual address space $0..2^{64} - 1$
  - 386: n=32, virtual address space $0..2^{32} - 1$
  - amd64 (x86_64): n=64, virtual address space $0..2^{64} - 1$
  - Arm aarch32: n=32, virtual address space $0..2^{32} - 1$
  - Arm aarch64: n=64, virtual address space $0..2^{64} - 1$
- The OS controls access to physical memory and creates the illusion that all processes have access to all memory

A process view of memory

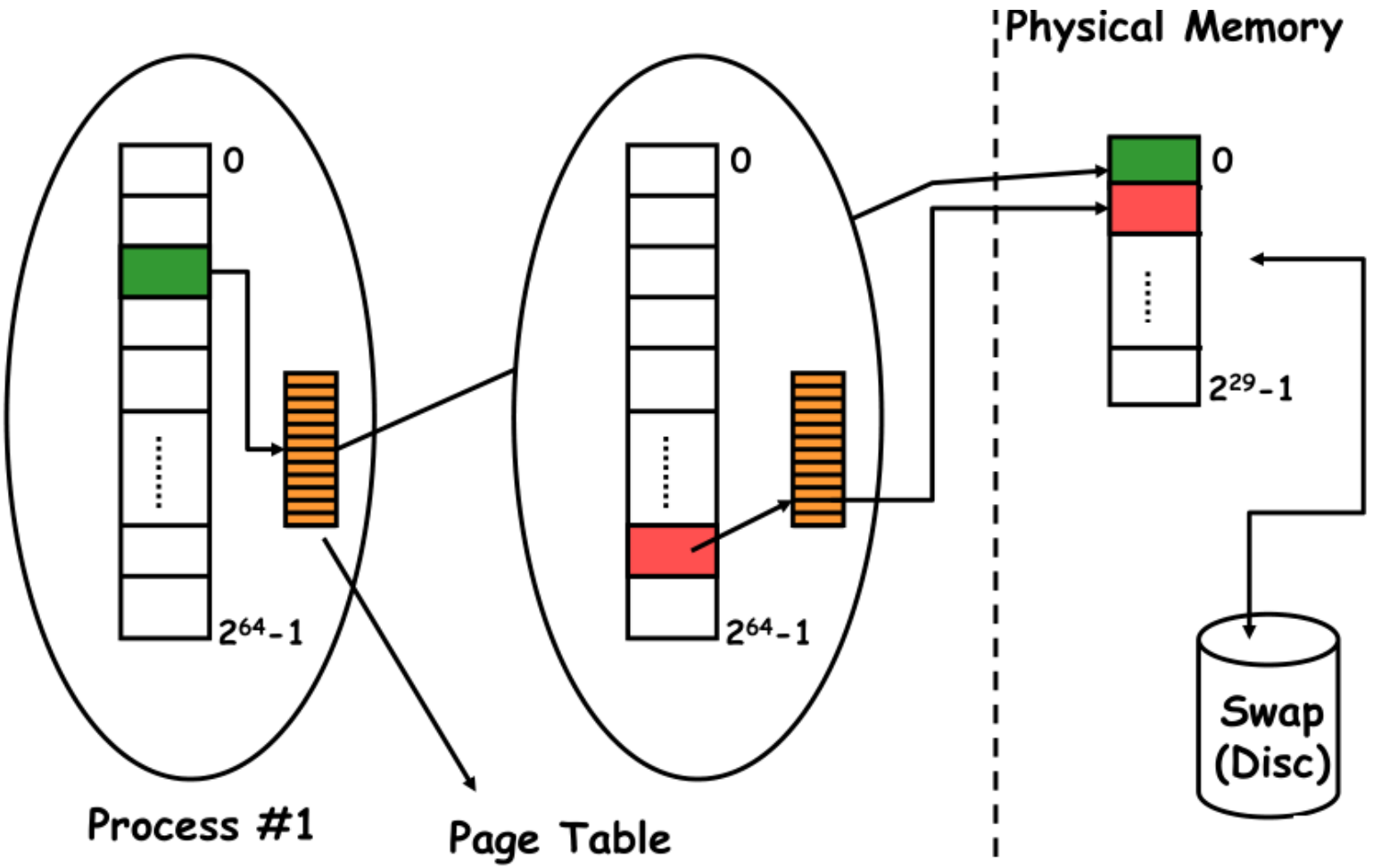


Real (or Physical) Memory

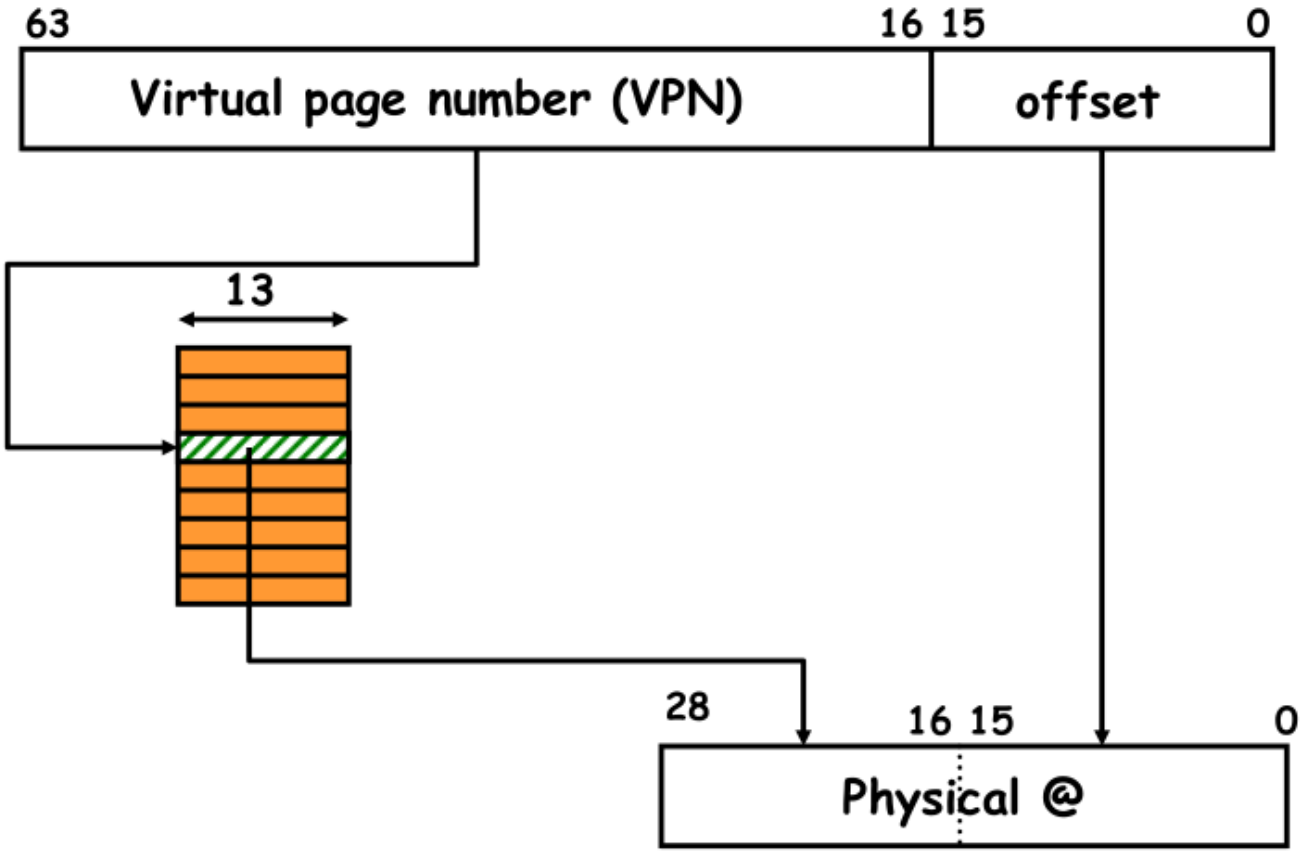
- Much smaller than virtual memory
  - 4 Gbytes in a typical PC actual (2^{32} bytes)
- Managed by the operating system
  - Creates the illusion that each process has access to 2^{64} bytes of memory
  - When we run out of physical memory, the OS copies back and forth into a swap disk
  - Virtual memory implementations
    - Segmented
    - Paged into same-sized blocks of memory



Virtual to Physical Mapping



Adress translation



Page table

- One page table for each Unix process
  - Holds, for each virtual page
    - The virtual to physical mapping
    - The page protections (R, W, X)
    - Some other stuff?

Page table

- One page table for each Unix process
  - Holds, for each virtual page
    - The virtual to physical mapping
    - The page protections (R, W, X)
    - Some other stuff?
    - We can mark this page to not go to the Store Buffer

Page table

- One page table for each Unix process
  - Holds, for each virtual page
    - The virtual to physical mapping
    - The page protections (R, W, X)
    - Some other stuff?
    - We can mark this page to not go to the Store Buffer
    - We can mark this page as not cacheable

Page table

- One page table for each Unix process
  - Holds, for each virtual page
    - The virtual to physical mapping
    - The page protections (R, W, X)
    - Some other stuff?
    - We can mark this page to not go to the Store Buffer
    - We can mark this page as not cacheable
- The alpha memory system
  - 64 bit Virtual address space (2^{64} bytes of virtual memory per process)
  - 8 Kbytes pages (2^{13} bytes)

Page table

- One page table for each Unix process
  - Holds, for each virtual page
    - The virtual to physical mapping
    - The page protections (R, W, X)
    - Some other stuff?
    - We can mark this page to not go to the Store Buffer
    - We can mark this page as not cacheable
- The alpha memory system
  - 64 bit Virtual address space (2^{64} bytes of virtual memory per process)
  - 8 Kbytes pages (2^{13} bytes)
- What is the size of the page table itself?

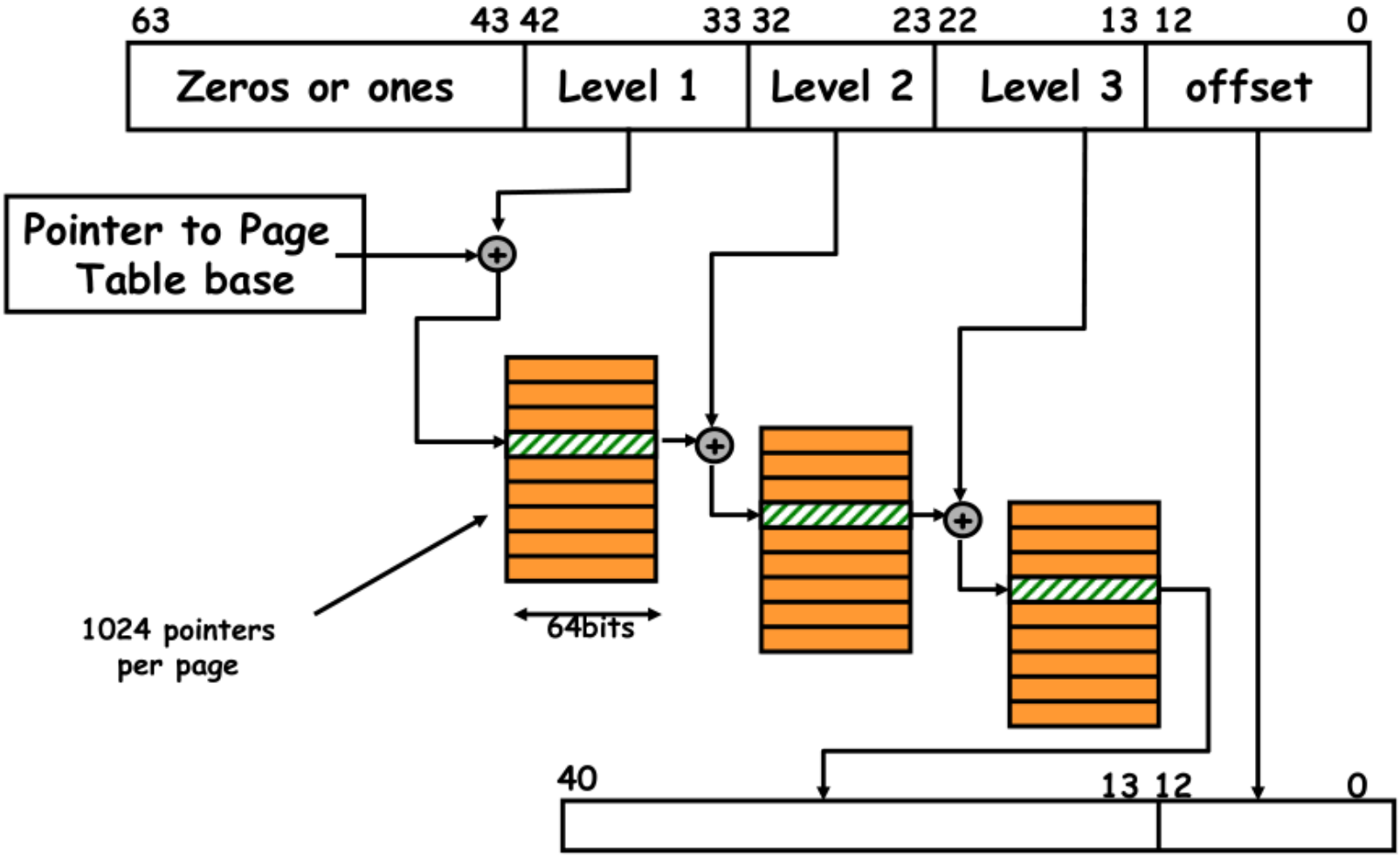
Page table

- One page table for each Unix process
  - Holds, for each virtual page
    - The virtual to physical mapping
    - The page protections (R, W, X)
    - Some other stuff?
    - We can mark this page to not go to the Store Buffer
    - We can mark this page as not cacheable
- The alpha memory system
  - 64 bit Virtual address space (2^{64} bytes of virtual memory per process)
  - 8 Kbytes pages (2^{13} bytes)
- What is the size of the page table itself?
  - Entries $= 2^{64} bytes / 2^{13} bytes = 2^{51}$
  - Each entry needs 13 bits (but let's round it up to16 bits)
  - Page table size $= 2^{51} x 2 bytes = 2^{52} bytes$
  - IMPOSSIBLE!!

Page table

- One page table for each Unix process
  - Holds, for each virtual page
    - The virtual to physical mapping
    - The page protections (R, W, X)
    - Some other stuff?
    - We can mark this page to not go to the Store Buffer
    - We can mark this page as not cacheable
- The alpha memory system
  - 64 bit Virtual address space (2^{64} bytes of virtual memory per process)
  - 8 Kbytes pages (2^{13} bytes)
- What is the size of the page table itself?
  - Entries $= 2^{64} bytes / 2^{13} bytes = 2^{51}$
  - Each entry needs 13 bits (but let's round it up to16 bits)
  - Page table size $= 2^{51} x 2 bytes = 2^{52} bytes$
  - IMPOSSIBLE!!
- Solution: Multi-level page table

Alpha 21264 page table



How do we do the translation in HW?

- 1 load could result in 3+1 accesses to memory

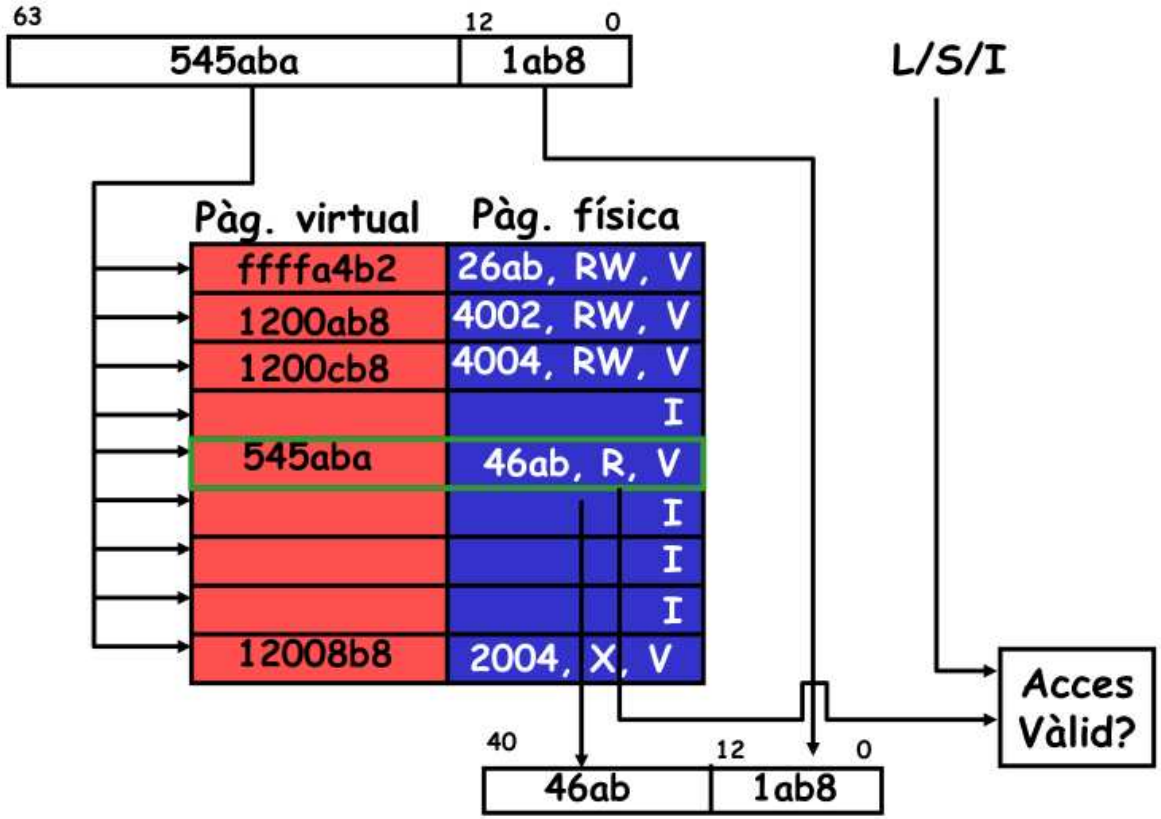
How do we do the translation in HW?

- 1 load could result in 3+1 accesses to memory
- Solution
  - Must take advantage of the locality principle
  - In computer science, locality of reference, also known as the principle of locality, is the tendency of a processor to access the same set of memory locations repetitively over a short period of time. There are two basic types of reference locality, temporal and spatial locality.

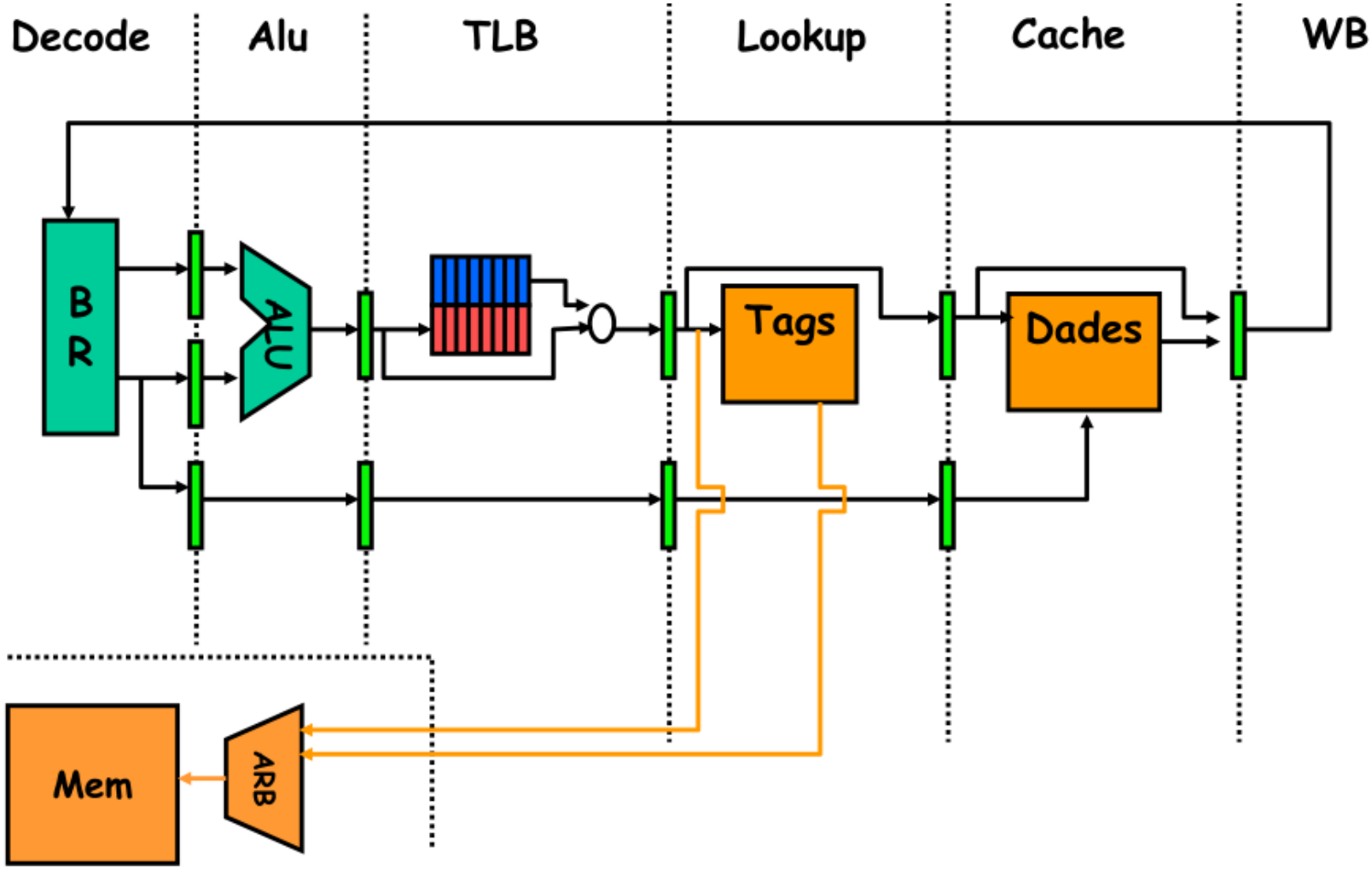
How do we do the translation in HW?

- 1 load could result in 3+1 accesses to memory
- Solution
  - Must take advantage of the locality principle
  - In computer science, locality of reference, also known as the principle of locality, is the tendency of a processor to access the same set of memory locations repetitively over a short period of time. There are two basic types of reference locality, temporal and spatial locality.
  - We will have a cache for "translations"
    - Translation Lookaside Buffer (TLB) or
    - Translation Buffer (TB)
  - Before accessing the cache
    - We will lookup the TLB to see if we have a translation
    - If we have a miss we will ask the OS for help

TLB (CAM)

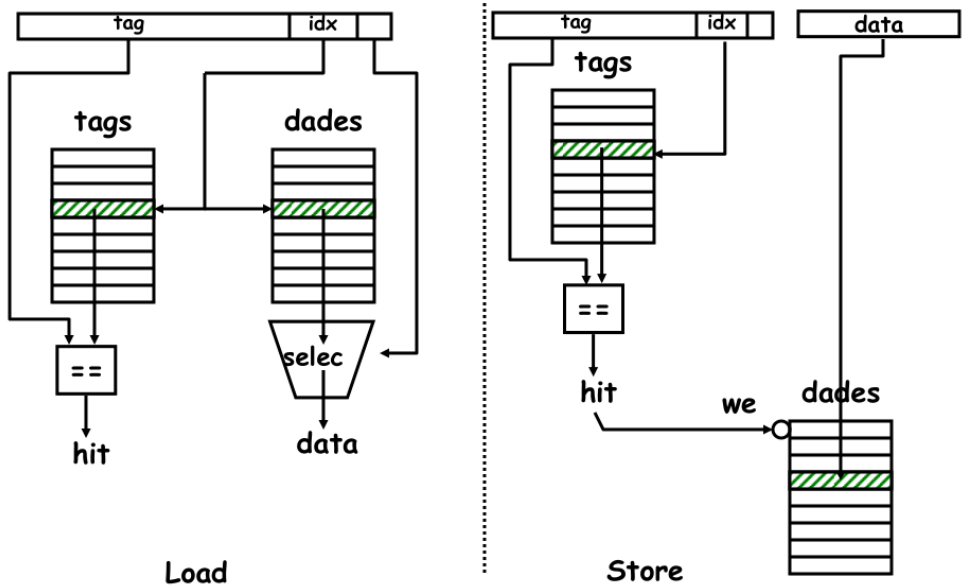

63
12
0
545aba
1ab8
L/S/I
Pàg. virtual
Pàg. física
ffffa4b2
26ab, RW, V
1200ab8
4002, RW, V
1200cb8
4004, RW, V
I
545aba
46ab, R, V
I
I
I
12008b8
2004, X, V
Acces
Vàlid?
40
12
0
46ab
1ab8

New Pipeline



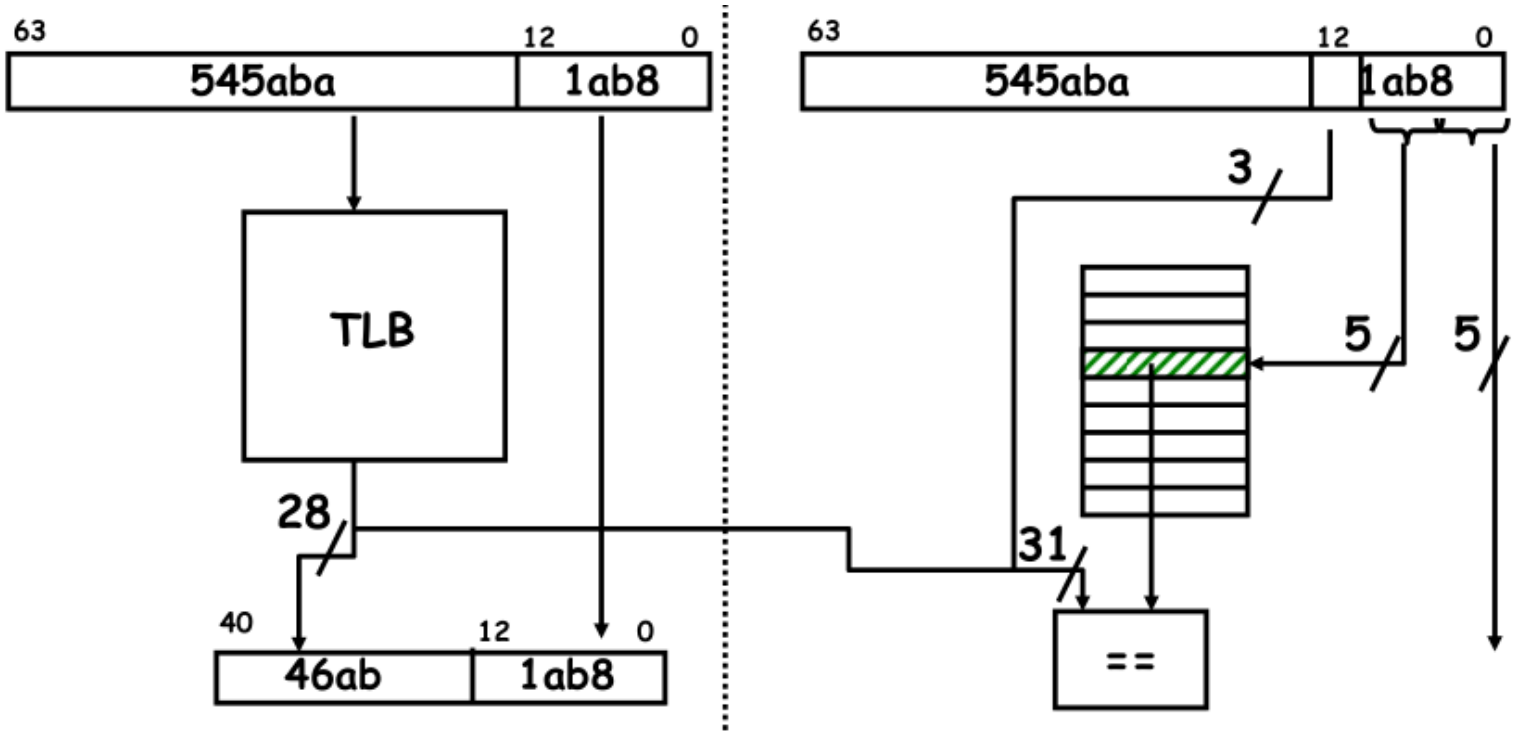
Do we really need an extra stage?

- Recall how we access the cache using direct maping
- How many bits do we need for "idx"?
  - C = cache size (bytes)
  - L = line size (bytes)
  - $Bits(idx) = log_2(\frac{C}{L})$



Virtual Index withPhysical Tag

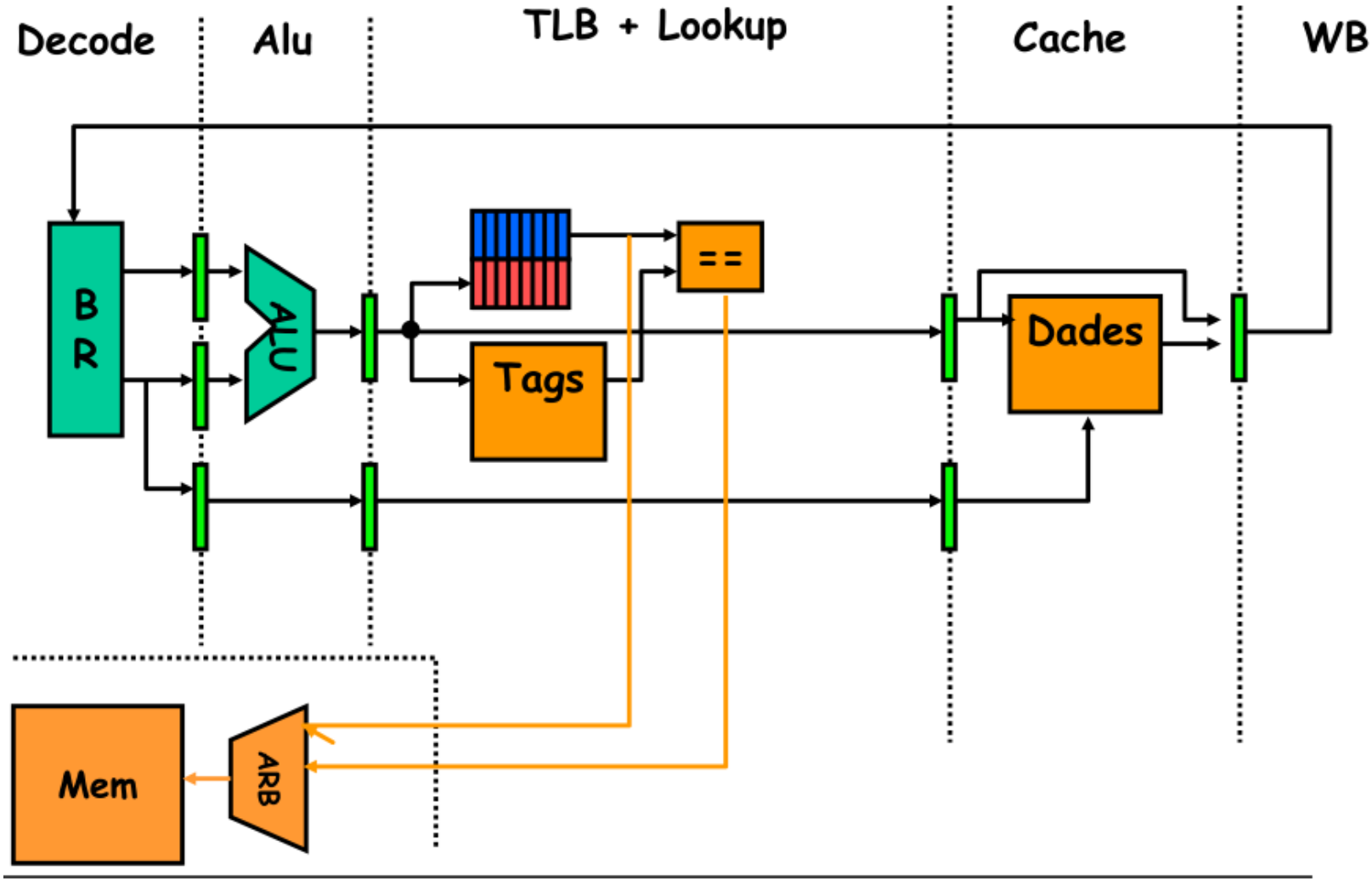
- If cache size ¡= page size
  - Example: Cache = 1KB, Line = 32B, Page = 8 KB



Could we tag the cache with virtual addresses?

- The answer is “yes”, but it is not easy
- The reasons to NOT do so are
  - Even if we use a physical tag for the cache, we still need a TLB to establish the validity of an access relative to its page (i.e., can’t write into a read-only page)
  - If two processes share the same cache and cache their private memory using the same virtual address we have a “synonim”!
    - If not taken care of, this can lead to one process using the wrong data
    - Possible solutions: flush cache on context switch

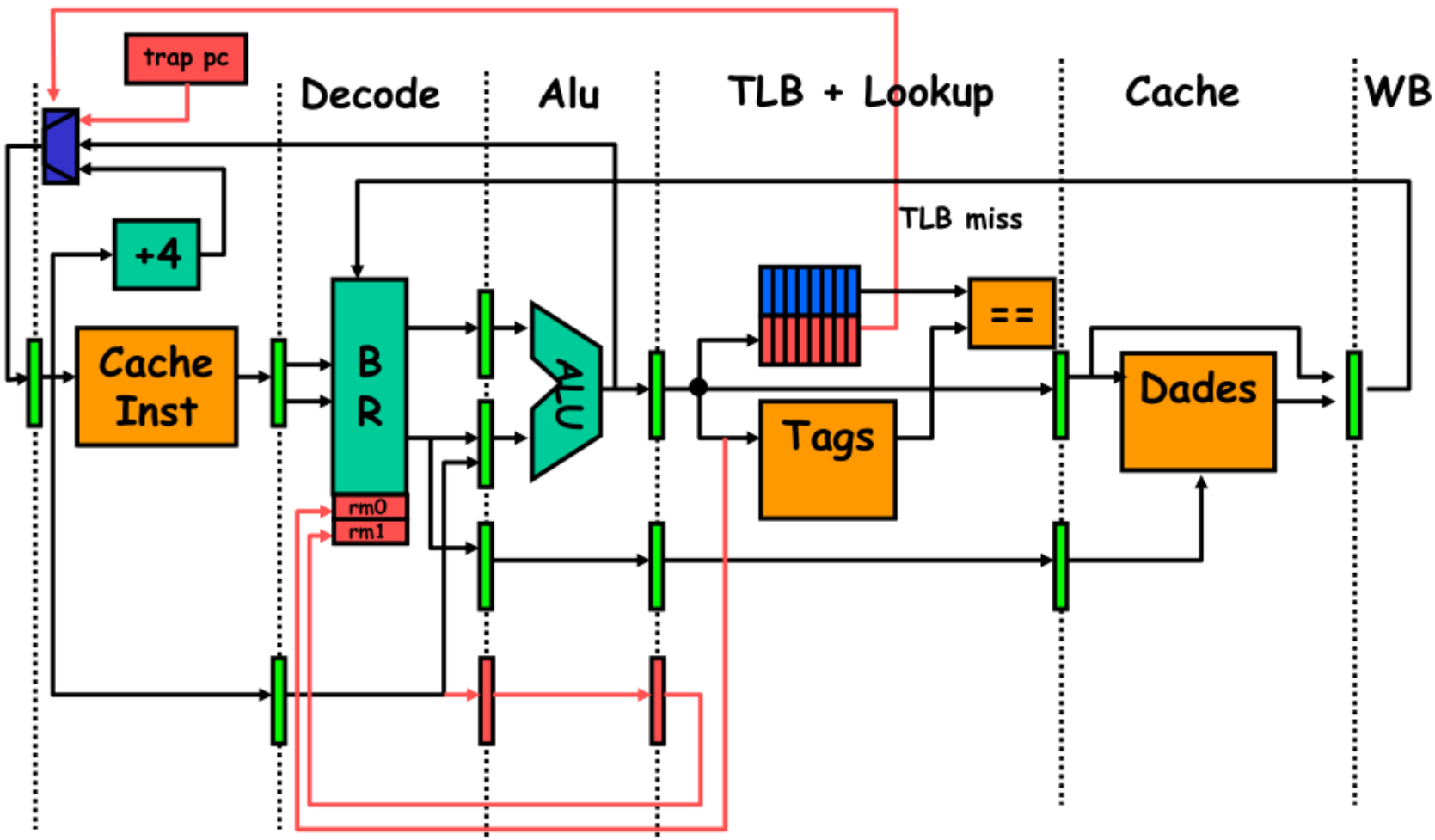
New Pipeline



On a TLB Miss...

- We must invoke the OS ...
  - We need to kill all the instruction younger than the offending instruction
  - We must alter the PC register to jump into OS code that handles tlb misses
  - We need to communicate info to the OS
    - @ that incurred in the TLB miss (i.e., lack of translation)
    - PC of the instruction that experienced the TLB miss
- The OS then needs to
  - Search in the page table the translation for "@"
  - Insert the ¡@, translation¿ tuple into the TLB
    - Possibly removing some other cached translation
  - Jump back to the PC of the instruction that had the tlb miss

TLB miss: additional datapath

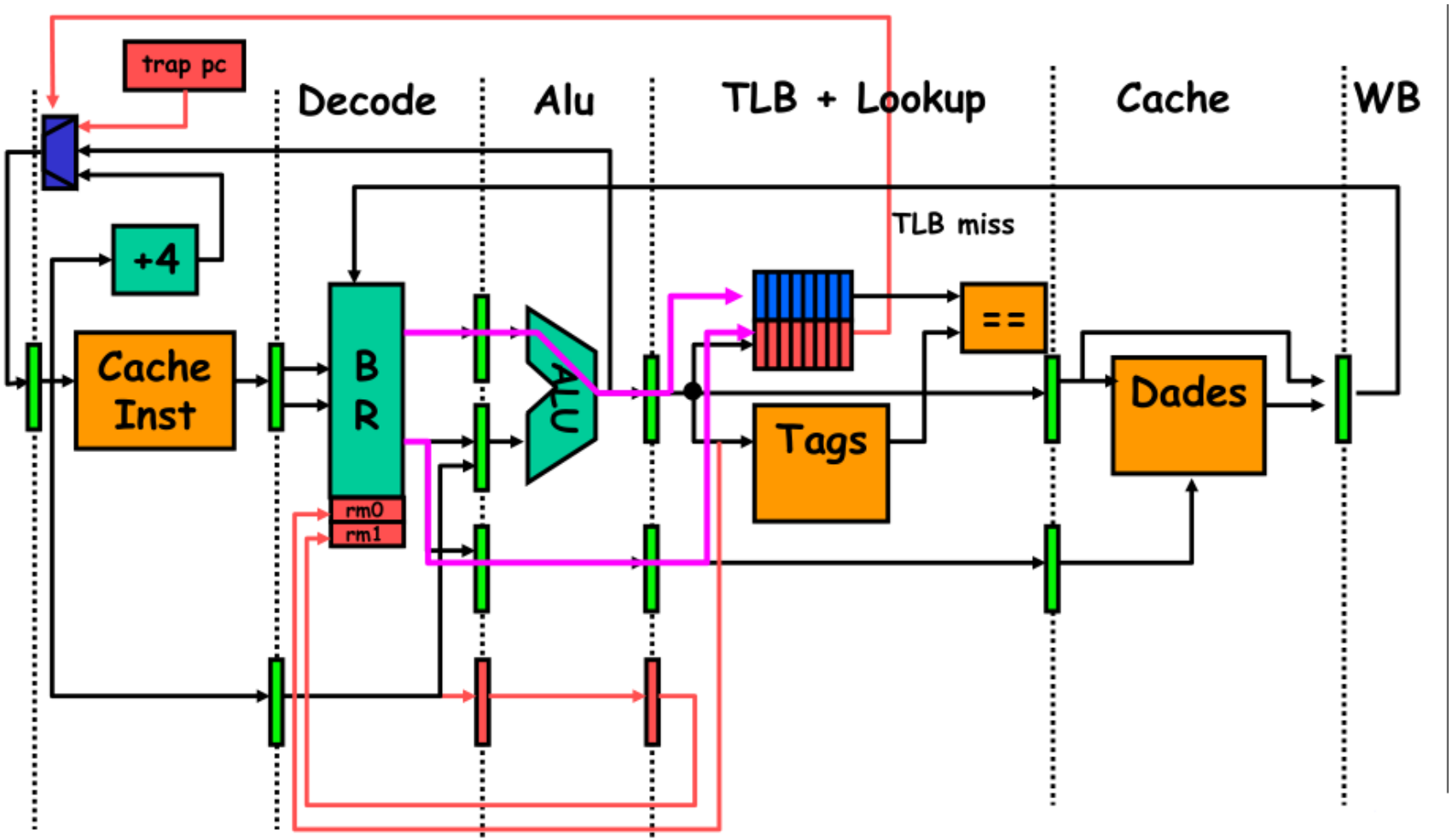


TLB Miss handling code (simplified)

```
store           r0,0x8000       // save process state
store           r1,0x8004       // save process state
store           r2,0x8008       // save process state
movctrl         rm0,r0          // copy miss PC
movctrl         rm1,r1          // copy miss @
store           r0,-4(sp)       // make r1 our "return address"
call            _tlbmiss        // returns translation in r2
tlbwrite        r0,r2           // install <virtual,physical> mapping into TLB
load            0x8000,r0       // restore process state
load            0x8004,r1       // restore process state
load            0x8008,r2       // restore process state
iret                            // will jump back to the address located in -4(sp), which is the address
                                // of the faulting instruction. Also, iret will lower the privilege mode
                                // back to \normal"
```

- First, save the process state
- Then read the info captured by the HW using special "movctrl rmX" instructions
  - These instructions only work in privileged mode
- The _tlbmiss routine
  - Walks the page table and finds the translation for the failing address and returns the translation
- Then we need a new instruction to insert the ¡Virtual,Physical¿ mapping into the TLB
  - tlbwrite r_x, r_y

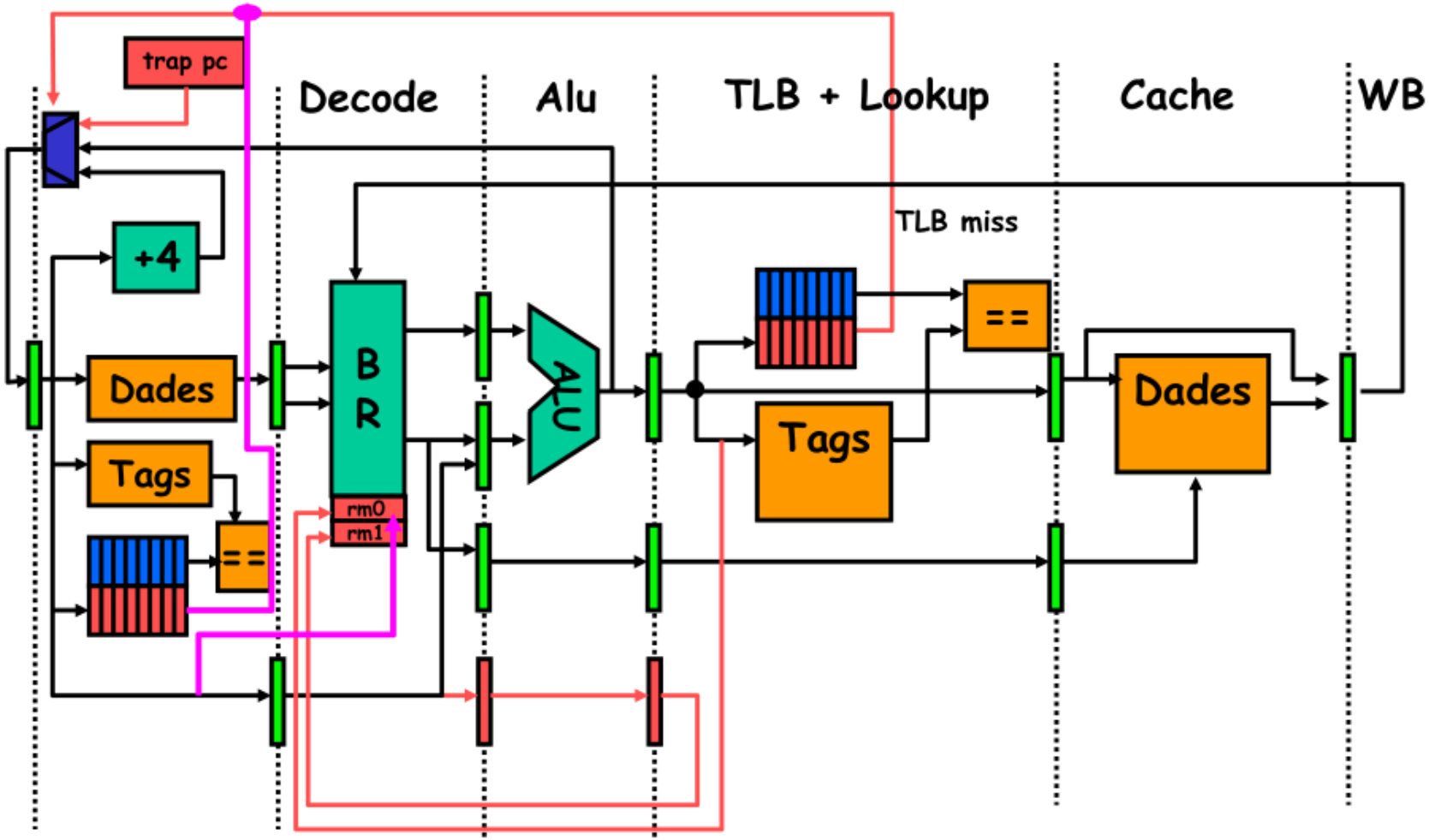
tlbwrite r4,r5



Instruction TLB

- A program's PC is *also* in virtual space
- Hence, we must ALSO translate the PC before we can index the instruction cache
- As we did for the data cache, if the size of the icache is ¡= page size then we can do the translation in parallel

iTLB


trap pc
Decode
Alu
TLB + Lookup
Cache
WB
TLB miss
+4
Dades
Tags
B
R
rm0
rm1
==
ALU
==
Tags
Dades

ARM Virtual memory system

Arm documentation

- ARM Architecture Reference Manual Armv8, for Armv8-A architecture profiles
  - `https://developer.arm.com/documentation/ddi0487/fc/`
- Cortex-A Series Programmer's guide for ARMv8
  - `https://developer.arm.com/documentation/den0024/a/`
- Technical Reference manuals
  - `https://developer.arm.com/documentation/ddi0500/j/`
  - `https://developer.arm.com/documentation/100095/0001`
  - `https://developer.arm.com/documentation/100023/0002`
  - ...

Arm introducction

- ARMv8 Overview
  - `ARMv8A-overview.pdf`
- Basics about ARM memory model
  - `ARMv8-VMSA.pdf`
- ARMv8 Memory management
  - `ARMv8-Memory.pdf`

Os9 description