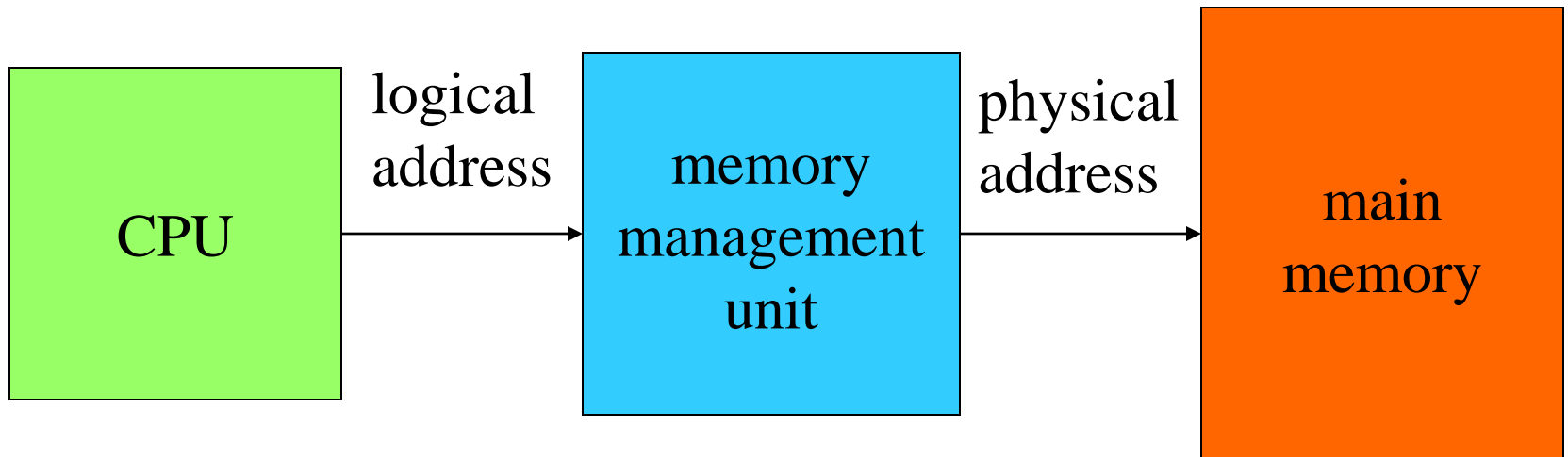


# Memory management units

⌘ Memory management unit (MMU)  
translates addresses:



# Access time comparison

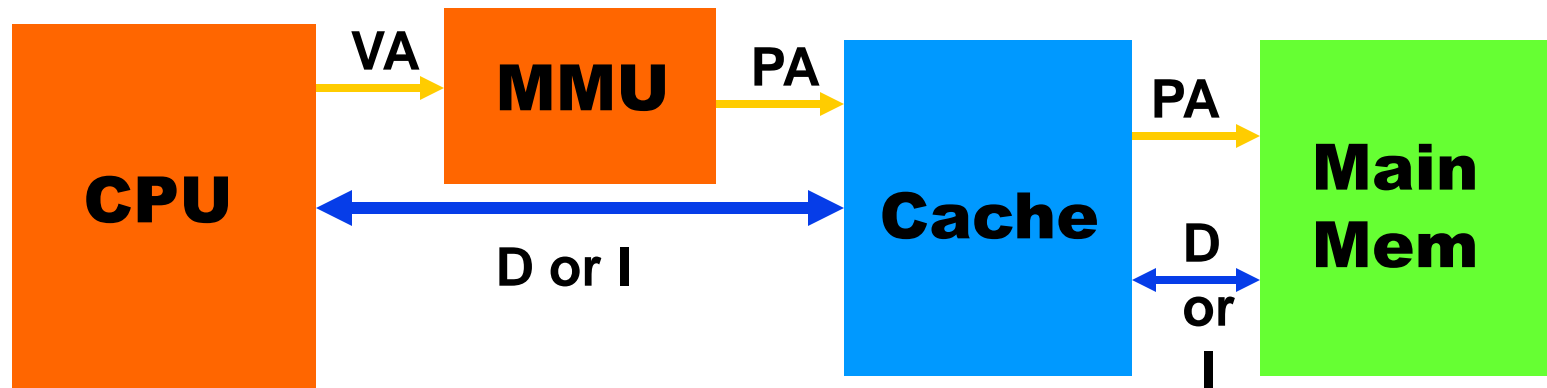
Media	Read	Write	Erase
<b>DRAM</b>	<b>60ns (2B) 2.56us (512B)</b>	<b>60ns (2B) 2.56us (512B)</b>	<b>N/A</b>
<b>NOR flash</b>	<b>150ns (2B) 14.4us (512B)</b>	<b>211us (2B) 3.53ms (512B)</b>	<b>1.2s (128KB)</b>
<b>NAND flash</b>	<b>10.2us (2B) 35.9us (512B)</b>	<b>201us (2B) 226us (512B)</b>	<b>2ms (16KB, 128K)</b>
<b>Disk</b>	<b>12.5ms (512B) (Average seek)</b>	<b>14.5ms (512B) (Average seek)</b>	<b>N/A</b>

## ⌘ Price

☑ HDD << NAND < DRAM < NOR

# MMU

- ⌘ Responsible for **VIRTUAL → PHYSICAL** address mapping
- ⌘ Sits between CPU and cache



- ⌘ Cache operates on **Physical Addresses** (mostly - some research on VA cache)

# Address translation

- ⌘ Requires some sort of register/table to allow arbitrary mappings of logical to physical addresses.
- ⌘ Two basic schemes:
  - ☑ segmented;
  - ☑ paged.
- ⌘ Segmentation and paging can be combined (x86).

# Segmented vs paged



⌘ Two types of address translation

⌘ Segmenting

- ☑ A large, arbitrarily sized region of memory

- ☑ A segment: a start address, (nonuniform) size

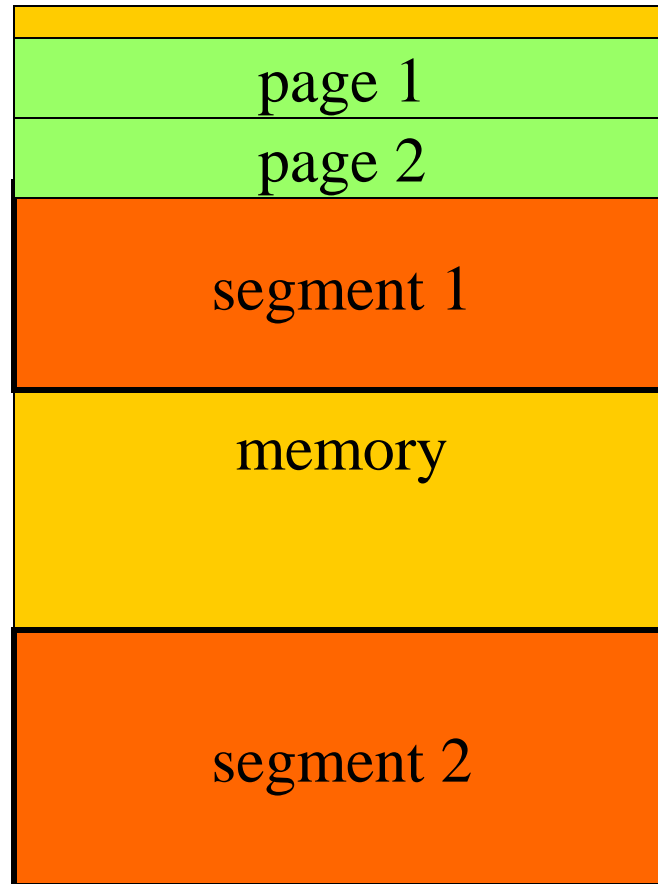
⌘ Paging

- ☑ Support small, equal sized region of memory

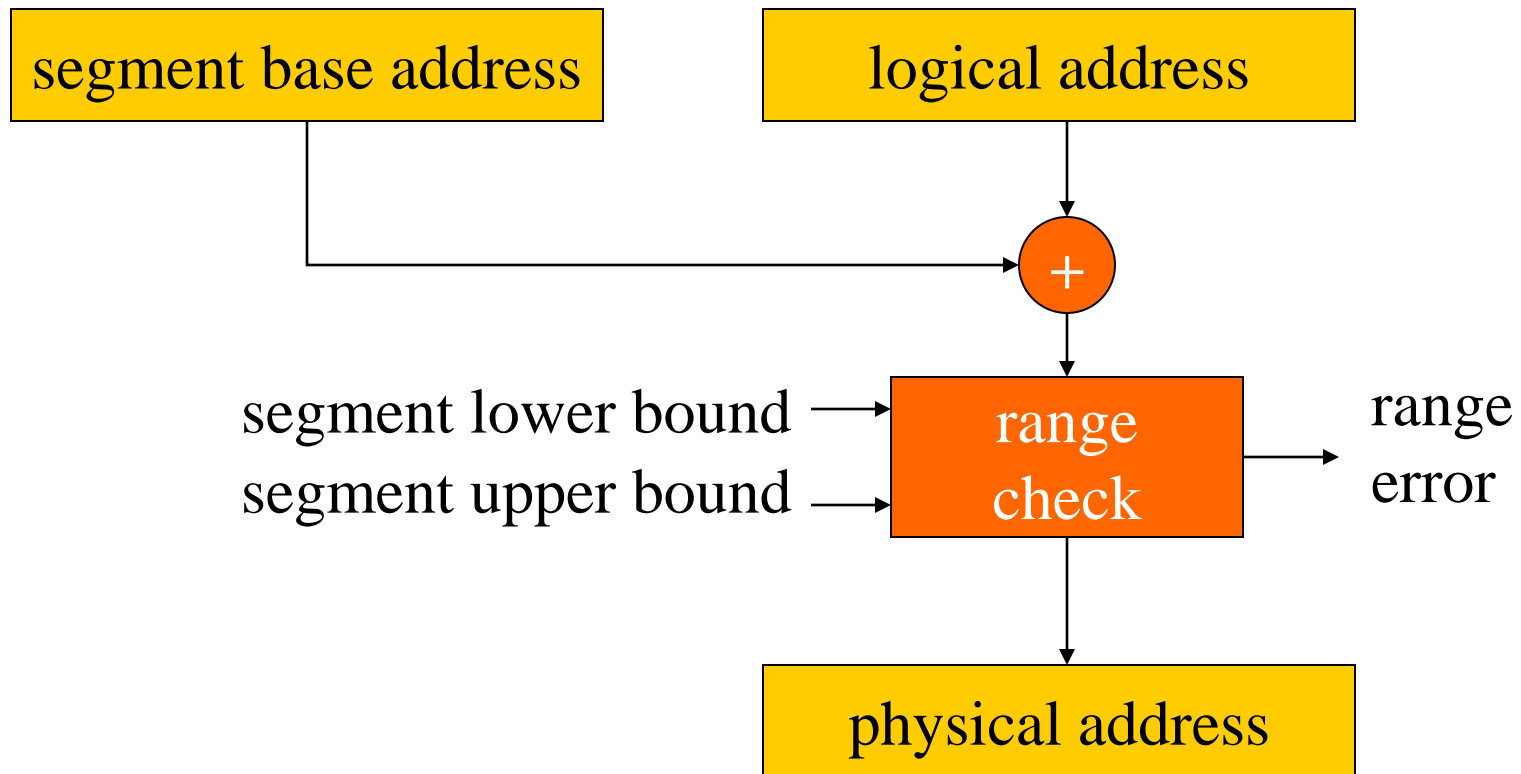
- ☑ A page: a start address

⌘ Paged segment: fragmentation

# Segments and pages



# Segment address translation



# Memory management tasks

- ⌘ Allows programs to move in physical memory during execution (on-demand).
- ⌘ Allows **virtual memory**:
  - ☑ memory images kept in secondary storage;
  - ☑ images returned to main memory on demand during execution.
- ⌘ **Page fault**: request for location not resident in memory.



# Page table



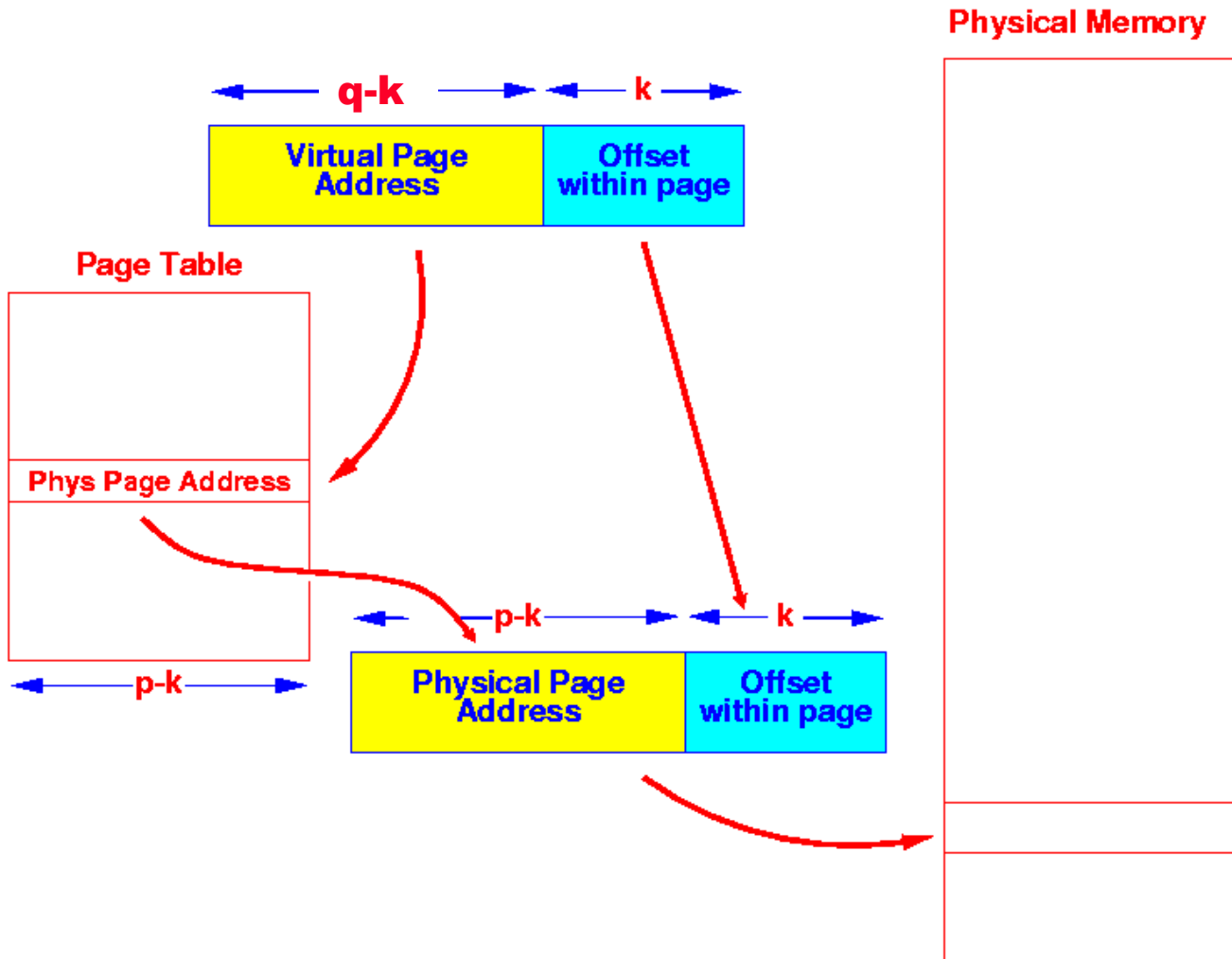
- ⌘ Operating System allocates pages of physical memory to users
- ⌘ OS constructs **page tables**
  - ☑ *one for each user*
- ⌘ Obtain page address from memory address to select a page table entry

# Page table



- ⌘ Page table entry contains physical page address if not a page fault
  - ☑ If a page fault, ?
- ⌘ Page fault is an exception
- ⌘ Page fault handler
  - ☑ Read the requested data from disc to main memory
  - ☑ Update the MMU's page table

# Paging – address translation



# Virtual memory space

- ⌘ Page Table Entries can also point to disc blocks
  - ☑ If Valid bit is **set**, page in memory (address is physical page address);
  - ☑ If **cleared**, page “swapped out” (address is disc block address)
  - ☑ MMU hardware generates **page fault** when swapped out page is requested
- ⌘ Allows virtual memory space to be larger than physical memory
  - ☑ Only “**working set**” is in physical memory
  - ☑ Remainder on paging disc

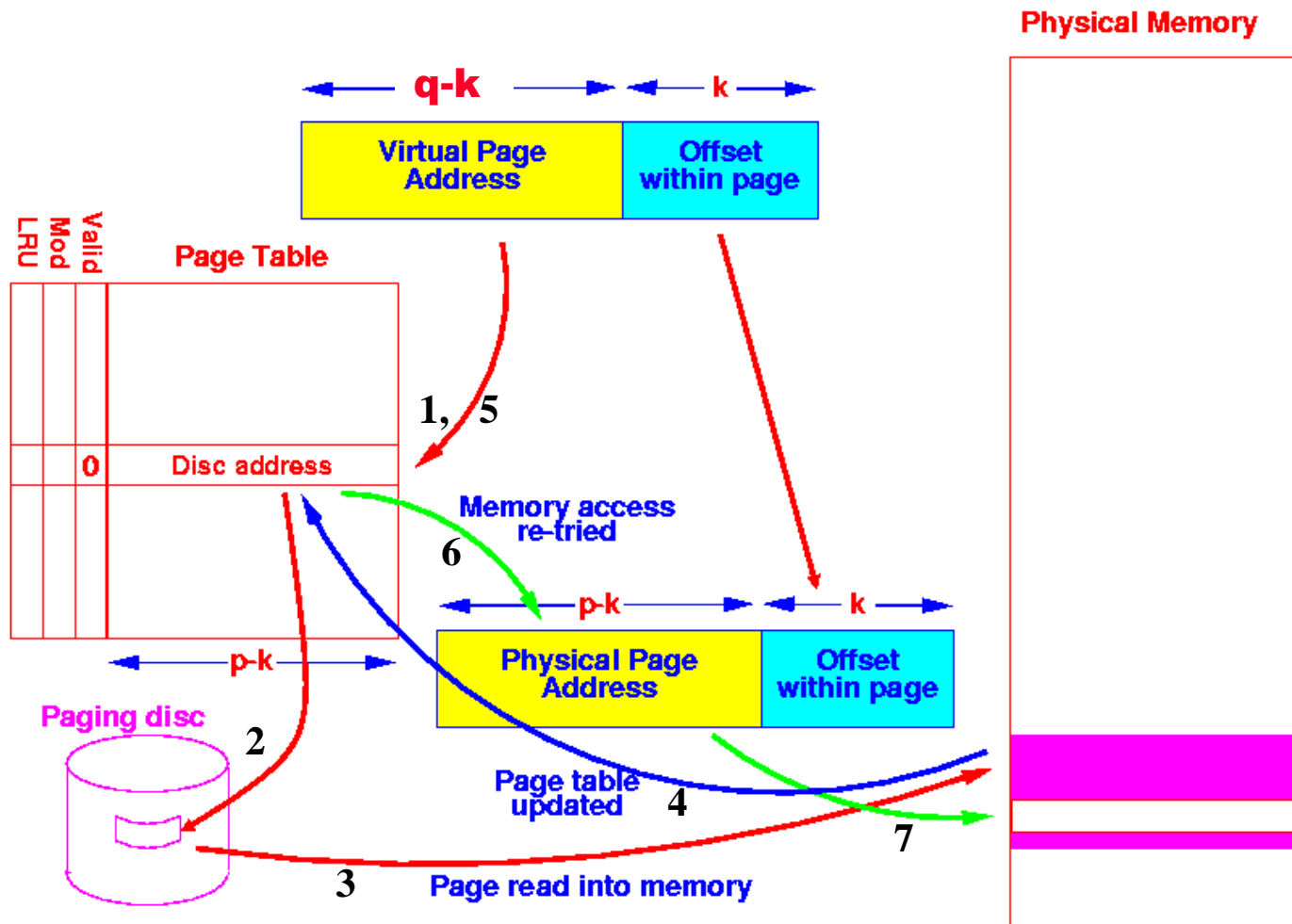
# Page fault handler



⌘ Page Fault Handler: part of OS kernel

- ☑ Read page table (assumed to be in memory)
- ☑ Finds usable physical page (limited resource)
  - ☒ LRU algorithm
- ☑ Writes it back to disc if modified
- ☑ Reads requested page from paging disc
- ☑ Adjusts page table entries
- ☑ Memory access re-tried

# Page Fault



# Page table - practicalities

## ⌘ Page size

⊞ 8 kbyte pages  $\Rightarrow k = 13$

⊞  $q = 32$ ,  $q - k = 19$

⊞ So page table size

⊞  $2^{19} \approx 0.5 \times 10^6$  entries

⊞ Each entry 4 bytes

$\Rightarrow 0.5 \times 10^6 \times 4 \approx 2$  Mbytes!

## ⌘ Page tables can take a lot of memory!

⊞ Larger page sizes reduce page table size  
*but* can waste space (fragmentation)

# Page table - practicalities



- ⌘ Page tables are stored in main memory
  - ☒ They're too large to be in smaller memories!
  - ☒ MMU needs to read page table for address translation
- ∴ Address translation can require additional memory accesses!



# Page Fault can be a never ending story

⌘ Can be an expensive process!

☒ Usual to allow page tables to be swapped out too!

⇒ Page fault can be generated on the page tables!

# MMU - Protection



## ⌘ Page table entries

- ☑ Extra bits are added to specify access rights

  - ☒ Set by OS (software)

*but*

  - ☒ Checked by MMU hardware!

- ☑ Access control bits

  - ☒ Read

  - ☒ Write

  - ☒ Read/Write

  - ☒ Execute only

# Alternative Page Table Styles

## ⌘ Inverted Page tables

☒ One page table entry (PTE) / page of physical memory

☒ MMU has to search for correct VA entry

∴ PowerPC hashes VA → PTE address

- PTE address =  $h(\text{VA})$

- $h$  – hash function

☒ Hashing ⇒ collisions

## ⌘ Hash functions in hardware

☒ “hash” of  $n$  bits to produce  $m$  bits (Usually  $m \ll n$ )

☒ Fewer bits reduces information content

# Inverted page table

## ⌘ Hash functions in hardware

⊞ “Fewer bits reduces information content

⊞ There are only  $2^m$  distinct values now!

⊞ Some  $n$ -bit patterns will reduce to the same  $m$ -bit patterns

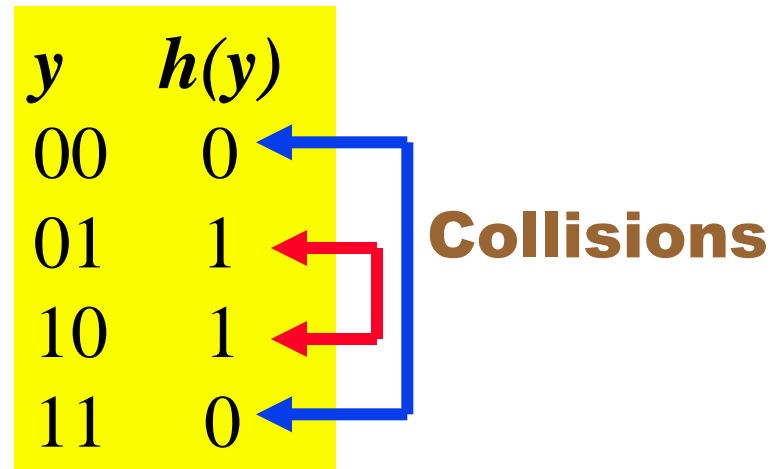
⊞ Trivial example

⊞ 2-bits  $\rightarrow$  1-bit with xor

⊞  $h(x_1 x_0) = x_1 \text{ xor } x_0$

$y$	$h(y)$
00	0
01	1
10	1
11	0

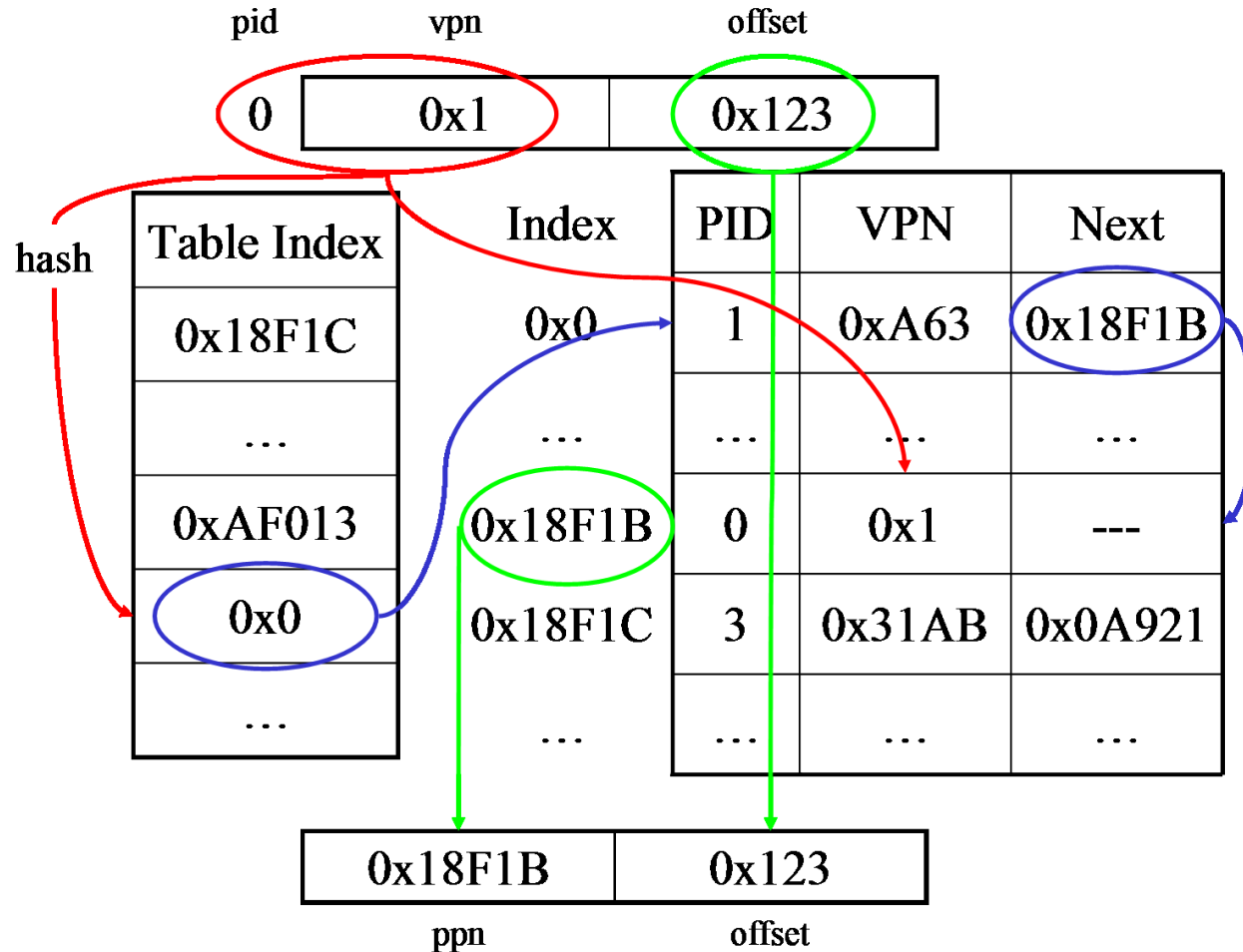
**Collisions**



# Inverted page table

- ⌘ One page table entry per physical page
- ⌘ MMU has to search for correct VA entry
  - ⊞ PowerPC **hashes** VA → PTE address
  - ⊞ Hashing ⇒ collisions
  - ⊞ PTEs are linked together
    - ⊞ PTE contains **tags** (like cache) and link bits
    - ⊞ MMU searches linked list to find correct entry
- ⌘ Smaller Page Tables / Longer searches

# Inverted page table



# Fast Address Translation

⌘ two+ memory accesses for each datum?

☑ Page table 1 - 3 (single - 3 level tables)

☑ Actual data 1

☑ *system can be slowed down*

⌘ Translation Look-Aside Buffer

- Acronym: TLB or TLAB
- Small **cache** of recently-used **page table entries**
- Usually fully-associative
- Can be quite small!

# TLB - Examples

## ⌘ TLB sizes

☑ MIPS R10000            1996        64 entries

☑ Pentium 4 (Prescott) 2006    64 entries

- One page table entry / **page** of data

- *Locality of reference*

- Programs spend a lot of time in same memory region

⇒ TLB hit rates tend to be very high

- 98%

⇒ Compensate for cost of a miss

(many memory accesses –

but for only 2% of references to memory!)

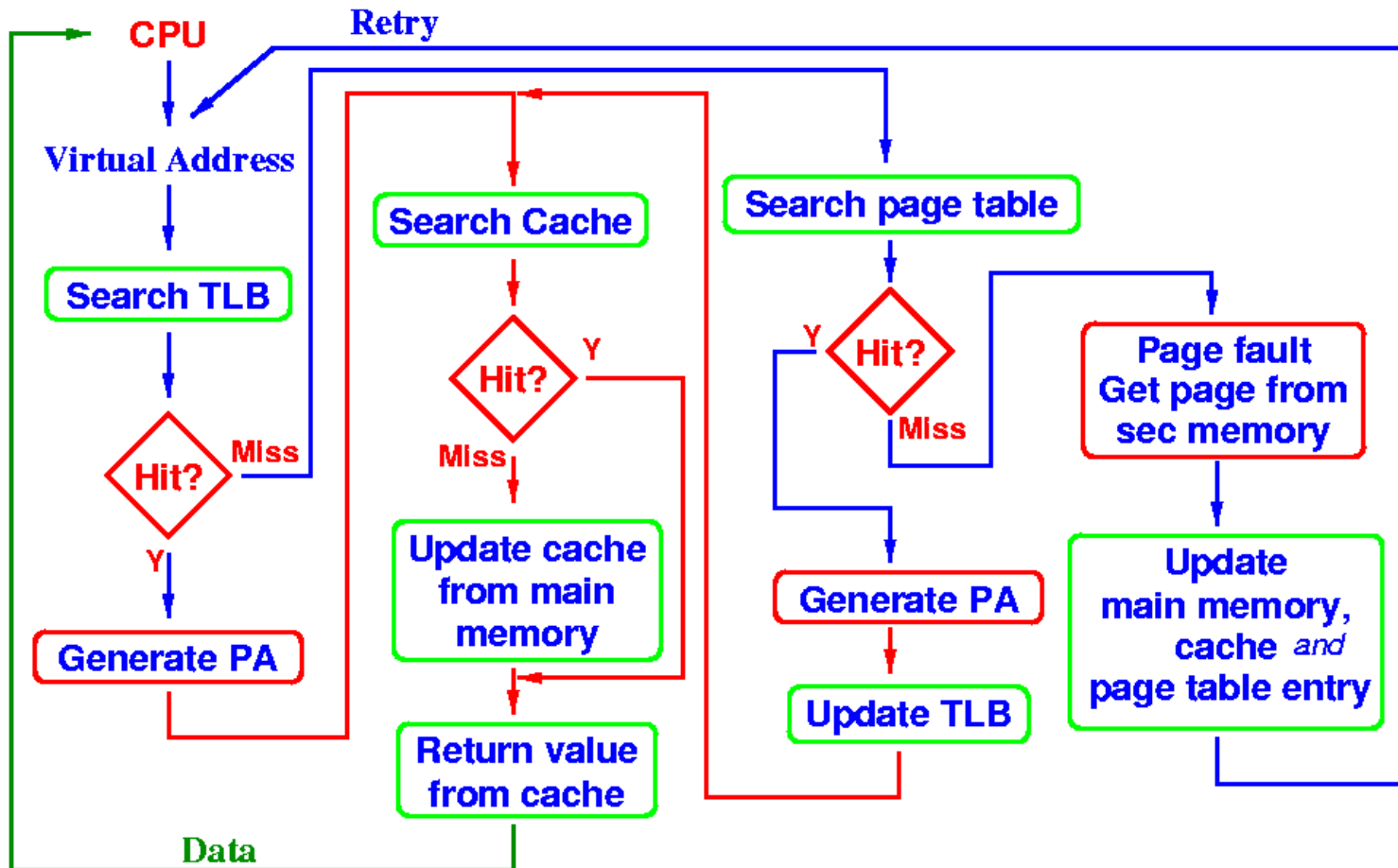


# TLB – Sequential access

- ⌘ Luckily, sequential access is fine!
- ⌘ Example: large (several MByte) matrix of doubles (8 bytes floating point values)
  - ☑ 8kbyte pages => 1024 doubles/page
- ⌘ Sequential access, *eg* sum all values:

```
for (j=0; j<n; j++)  
    sum = sum + x[j]
```

# Memory Hierarchy - Operation



# ARM memory management

⌘ Memory region types:

☑ section: 1 Mbyte block;

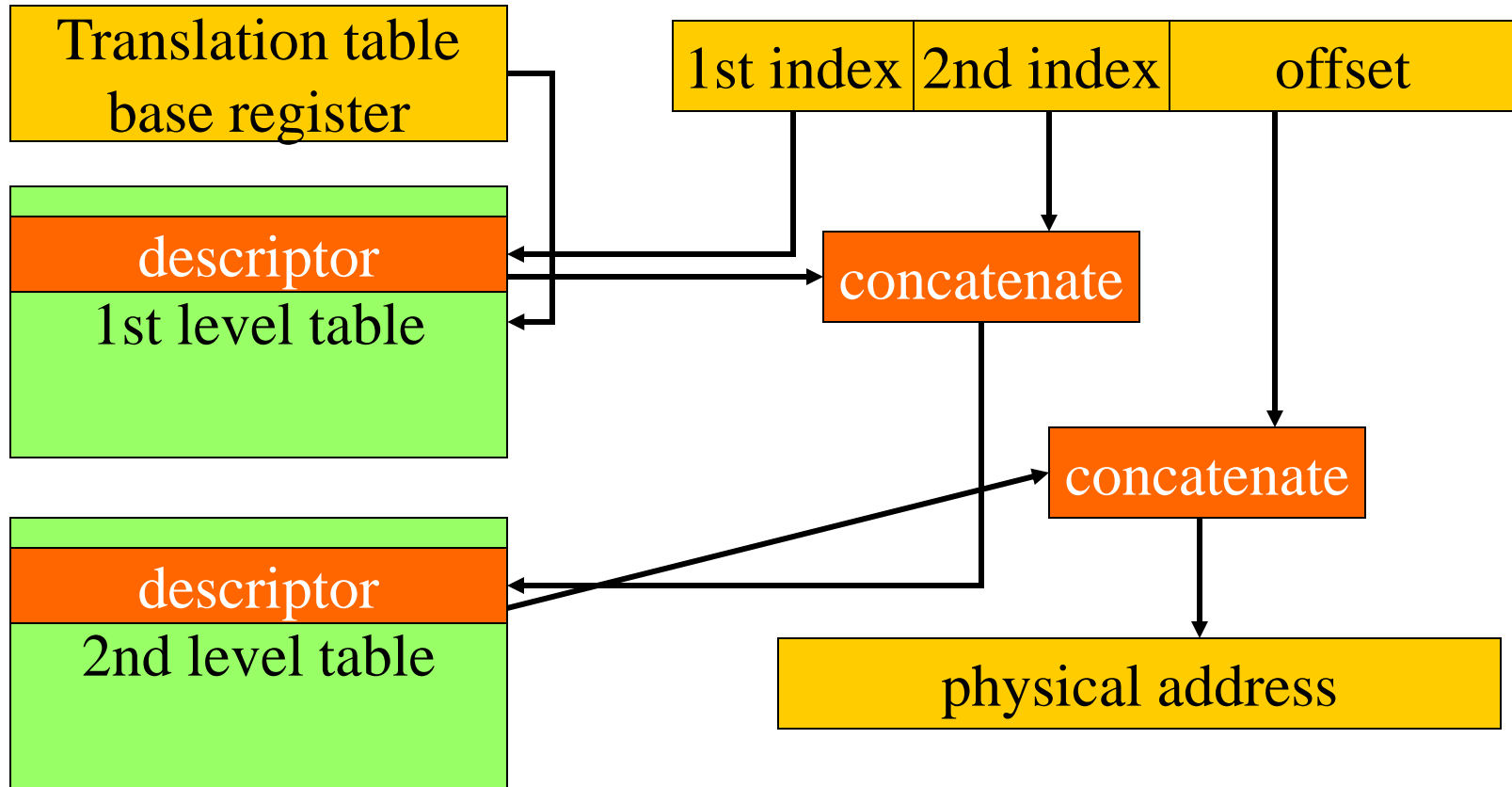
☑ large page: 64 kbytes;

☑ small page: 4 kbytes.

⌘ An address is marked as section-mapped or page-mapped.

⌘ Two-level translation scheme.

# ARM address translation



# Elements of CPU performance



- ⌘ Cycle time.
- ⌘ CPU pipeline.
- ⌘ Memory system.

# Pipelining



- ⌘ Several instructions are executed simultaneously at different stages of completion.
- ⌘ Various conditions can cause **pipeline bubbles** that reduce utilization:
  - ☒ branches;
  - ☒ memory system delays;
  - ☒ etc.

# Performance measures

⌘ **Latency**: time it takes for an instruction to get through the pipeline.

- ☒ CPI (cycle per instruction)

- ☒ Clock cycle

⌘ **Throughput**: number of instructions executed per time period.

- ☒ IPC (instruction per cycle)

- ☒ Frequency

⌘ Pipelining increases throughput without reducing latency.

# ARM7 pipeline



⌘ ARM 7 has 3-stage pipe:

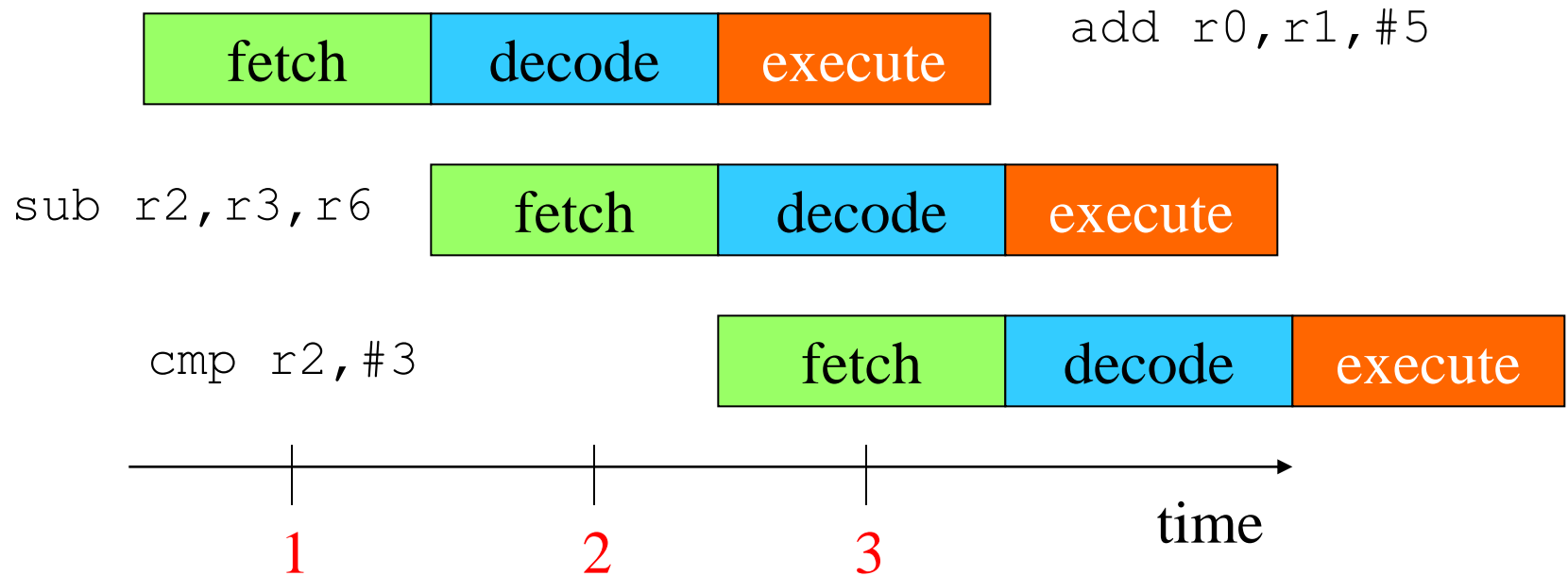
☑ **fetch** instruction from memory;

☑ **decode** opcode and operands;

☑ **execute**.



# ARM pipeline execution



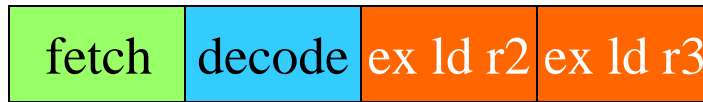
# Pipeline stalls



- ⌘ If every step cannot be completed in the same amount of time, pipeline stalls.
- ⌘ Bubbles introduced by stall increase latency, reduce throughput.

# ARM multi-cycle LDMIA instruction

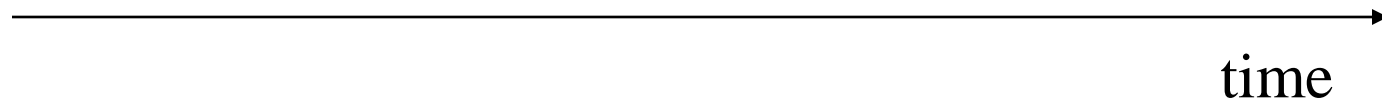
`ldmia r0, {r2, r3}`



`sub r2, r3, r6`



`cmp r2, #3`

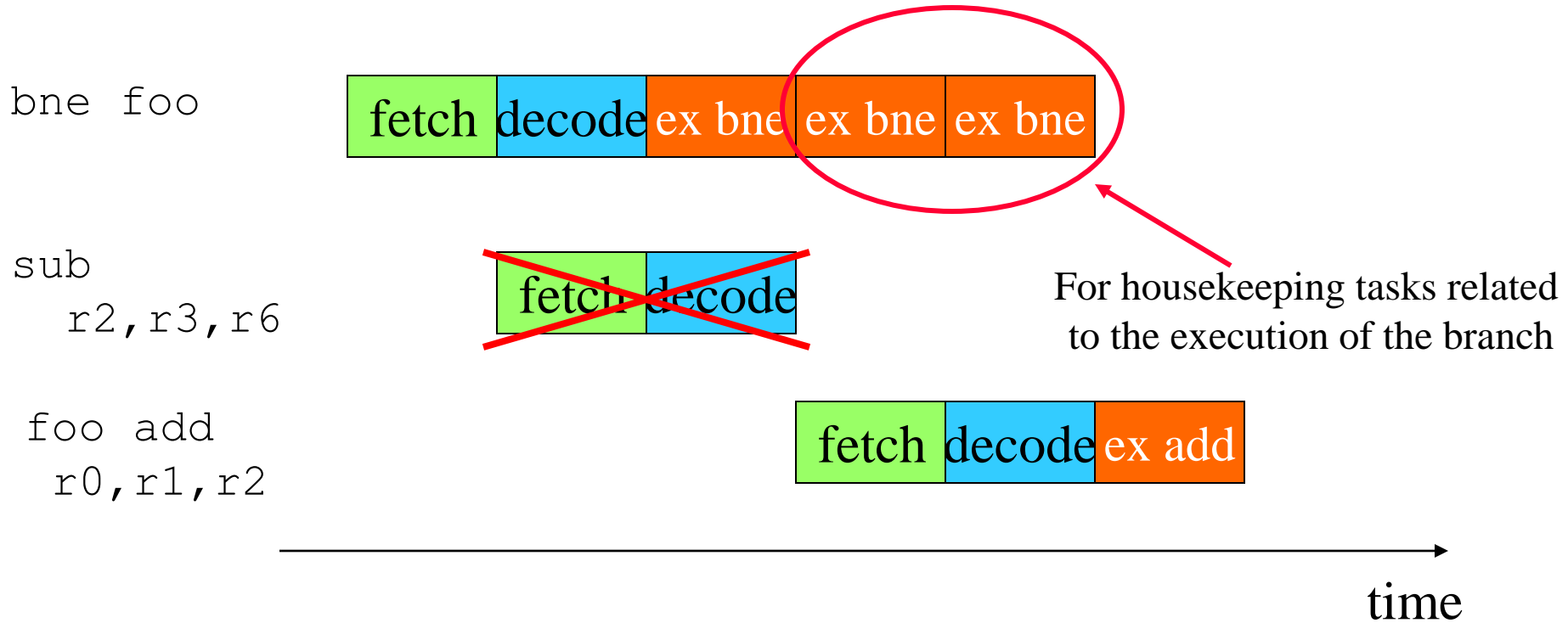


# Control stalls



- ⌘ Branches often introduce stalls (branch penalty).
  - ☑ Stall time may depend on whether branch is taken.
- ⌘ May have to squash instructions that already started executing.
- ⌘ Don't know what to fetch until condition is evaluated.

# ARM pipelined branch



# Delayed branch



- ⌘ A solution to reduce branch penalty
- ⌘ To increase pipeline efficiency, delayed branch mechanism requires  $n$  ( $1 \sim 2$ ) instructions after branch **always** executed whether branch is executed or not.

# Example: ARM execution time

⌘ Determine execution time of FIR filter:

```
for (i=0; i<N; i++)
```

```
    f = f + c[i]*x[i];
```

⌘ Only branch in loop test may take more than one cycle.

⏏ **BLT loop** takes 1 cycle best case, 3 worst case.

# FIR filter ARM code

```

; loop initiation code
MOV r0,#0 ; use r0 for i, set to 0
MOV r8,#0 ; use an index for arrays
ADR r2,N ; get address for N
LDR r1,[r2] ; get value of N
MOV r2,#0 ; use r2 for f, set to 0
ADR r3,c ; load r3 with C base
ADR r5,x ; load r5 with x base

; loop body
Loop { LDR r4,[r3,r8] ; get value of c[i]
      LDR r6,[r5,r8] ; get value of x[i]
      MUL r4,r4,r6 ; compute c[i]*x[i]
      ADD r2,r2,r4 ; add into running sum
      ; update loop counter and array index
      ADD r8,r8,#4 ; add one to array index
      ADD r0,r0,#1 ; add 1 to i
      ; test for exit
      CMP r0,r1
      BLT loop ; if i < N, continue loop
loopend ...

```



# FIR filter performance by block

Block	Variable	# instructions	# cycles
Initialization	$t_{\text{init}}$	7	7
Body	$t_{\text{body}}$	4	4
Update	$t_{\text{update}}$	2	2
Test	$t_{\text{test}}$	2	[2,4]

$$t_{\text{loop}} = t_{\text{init}} + N(t_{\text{body}} + t_{\text{update}}) + (N-1)t_{\text{test,worst}} + t_{\text{test,best}}$$

Loop test succeeds is worst case

Loop test fails is best case

# Memory system performance



⌘ Caches introduce indeterminacy in execution time.

☑ Depends on order of execution.

⌘ **Cache miss penalty**: added time due to a cache miss.

# CPU power consumption

- ⌘ Most modern CPUs are designed with power consumption in mind to some degree.
- ⌘ Power vs. energy:
  - ☑ heat depends on power consumption;
  - ☑ battery life depends on energy consumption.

# CMOS power consumption

- ⌘ **Voltage drops**: power consumption proportional to  $V^2$ .
- ⌘ **Toggling**: more activity means more power.
- ⌘ **Leakage**: basic circuit characteristics; can be eliminated by disconnecting power.

# CPU power-saving strategies



- ⌘ Reduce power supply voltage.
- ⌘ Run at lower clock frequency.
- ⌘ Disable function units with control signals when not in use.
- ⌘ Disconnect parts from power supply when not in use.

# Power management styles

- ⌘ **Static power management**: does not depend on CPU activity.
  - ☑ Example: user-activated power-down mode.
- ⌘ **Dynamic power management**: based on CPU activity.
  - ☑ Example: disabling off function units.