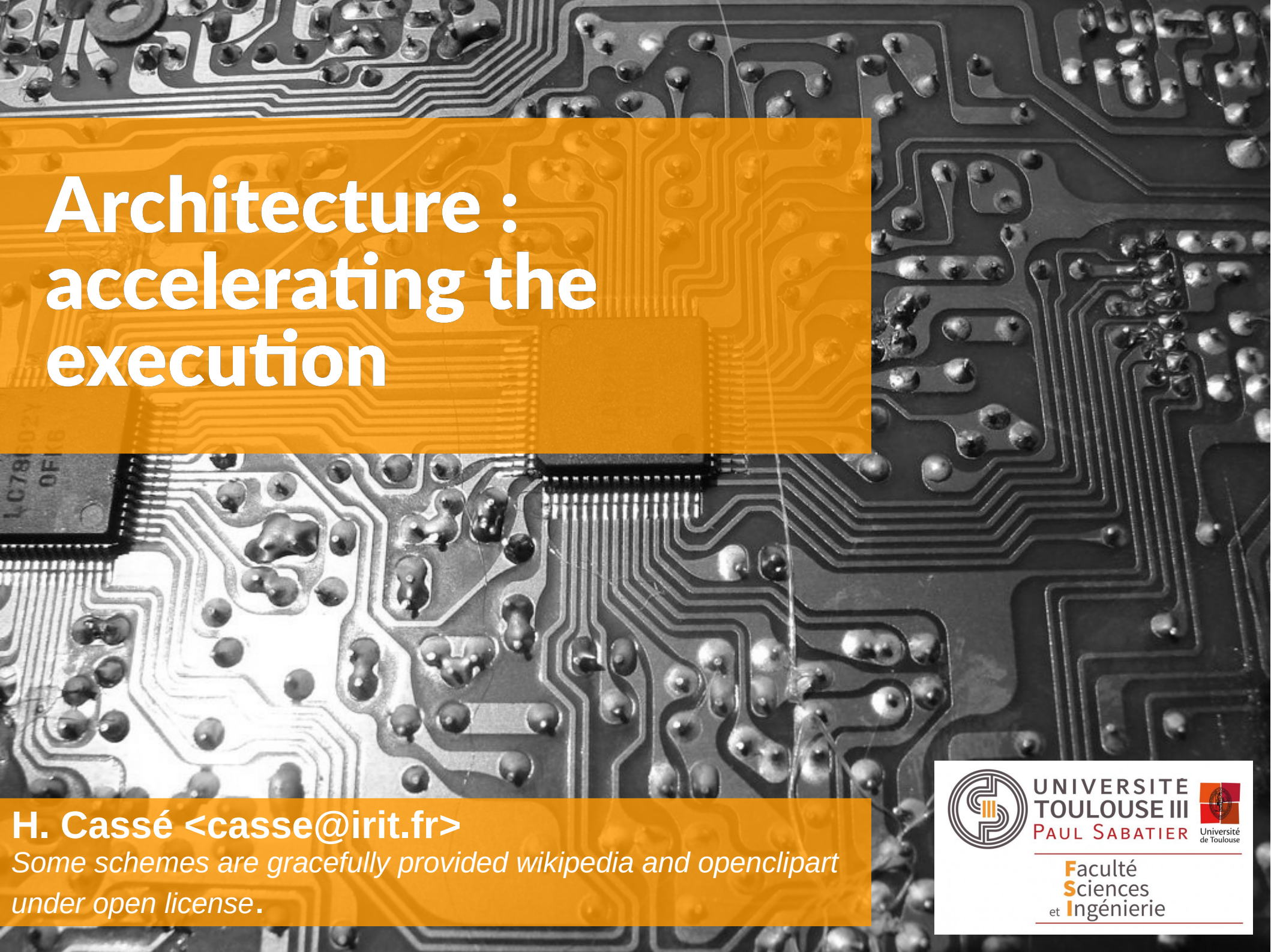
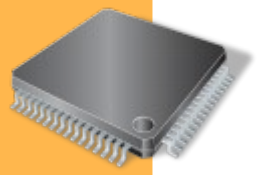


Architecture : accelerating the execution

H. Cassé <casse@irit.fr>

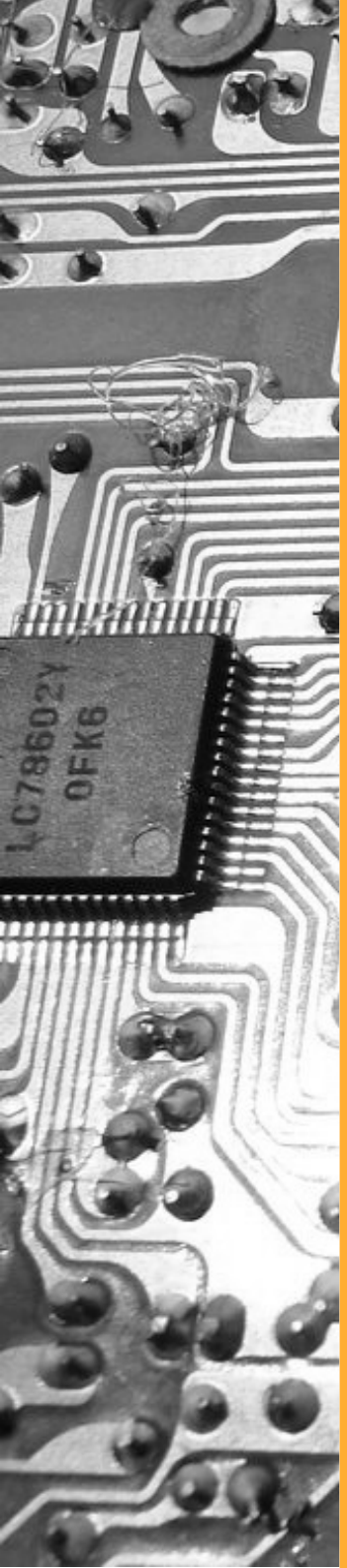
Some schemes are gracefully provided wikipedia and openclipart under open license.

UNIVERSITÉ TOULOUSE III PAUL SABATIER
Université de Toulouse
Faculté Sciences et Ingénierie



Introduction

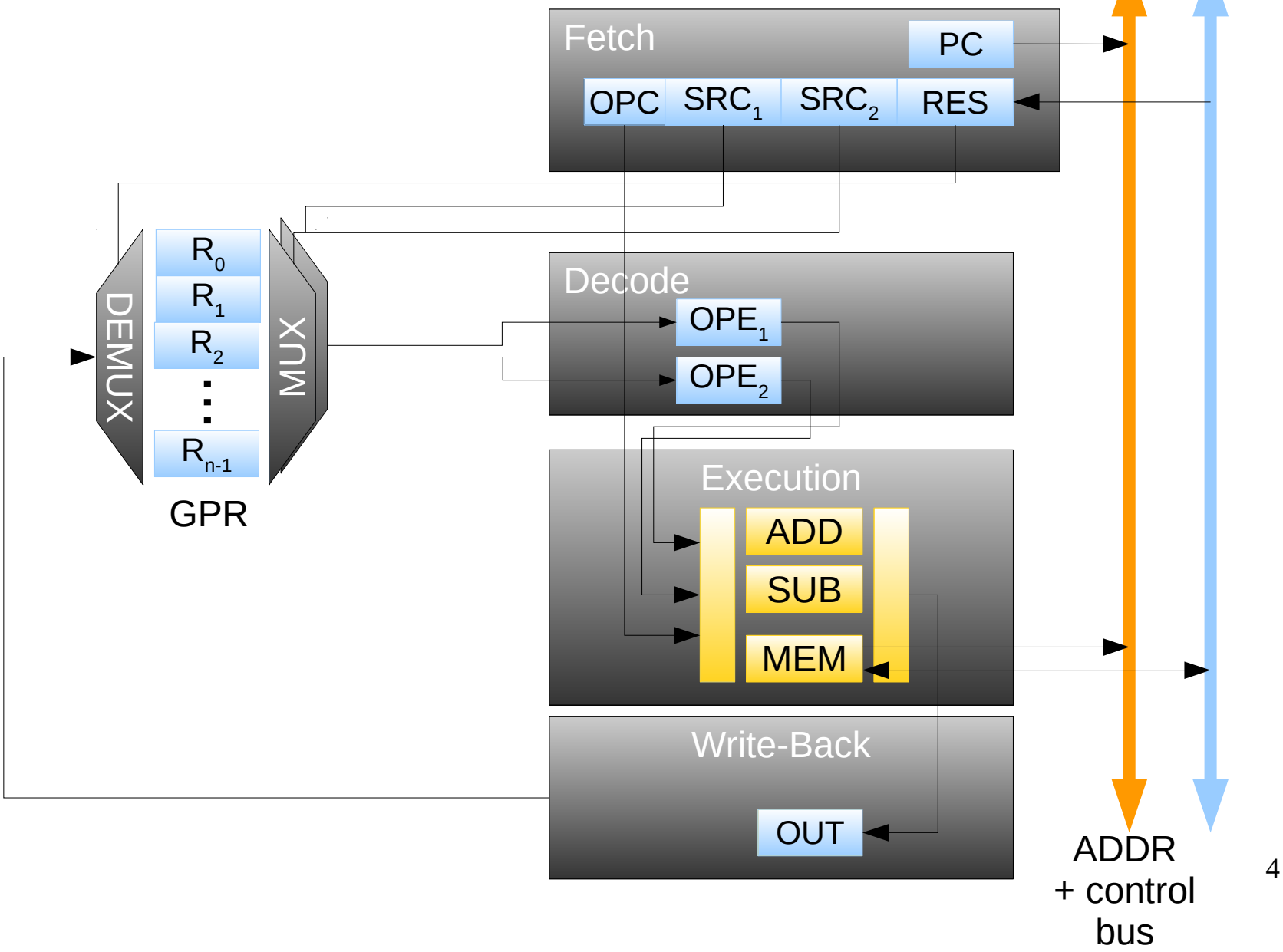
- previous lecture
  - how to make an execution core
  - how to make a memory
  - how to perform input / output
- initial Von Neuman needs acceleration
  - pipeline
  - caches (code + data)
  - speculative execution

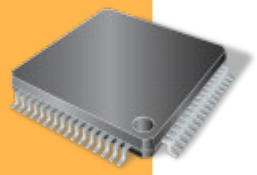


Overview

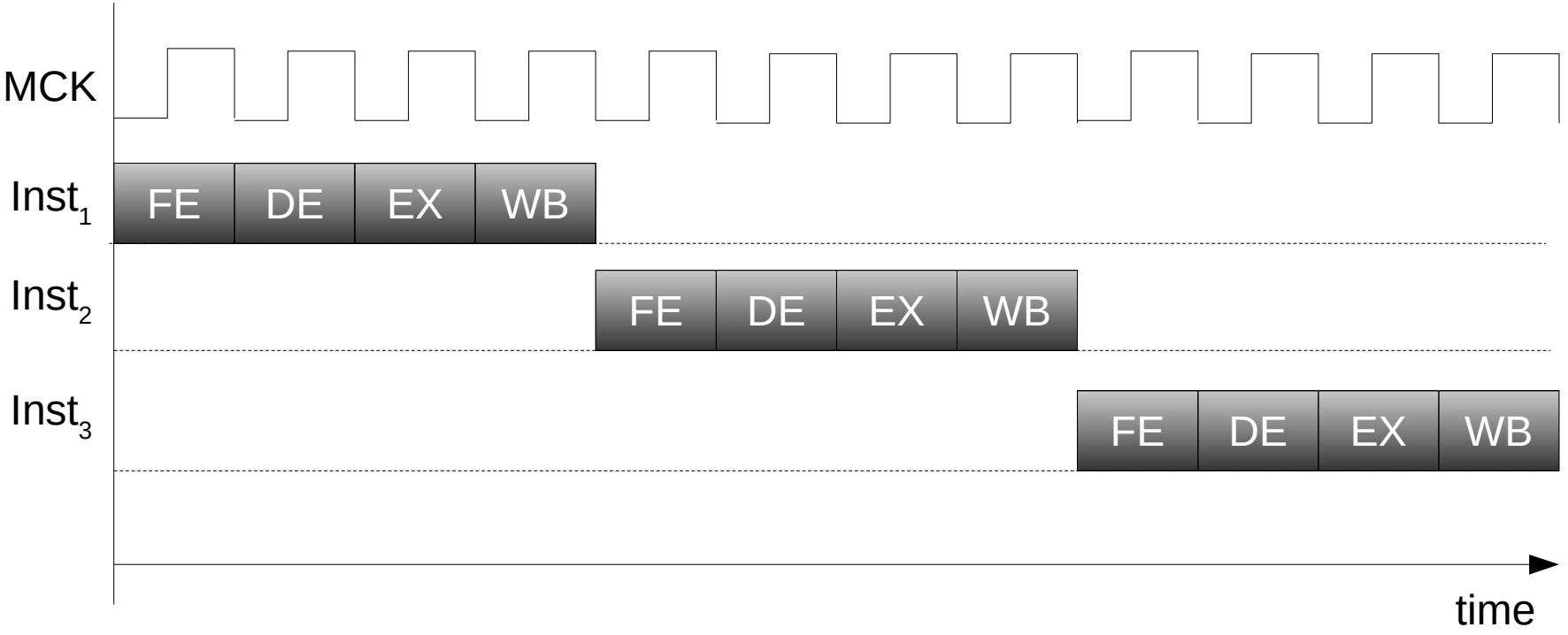
- Introduction
- **Pipeline**
- Cache memories
- Conclusion

Register Bank

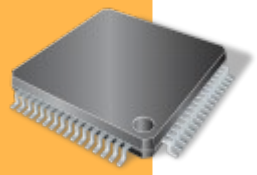




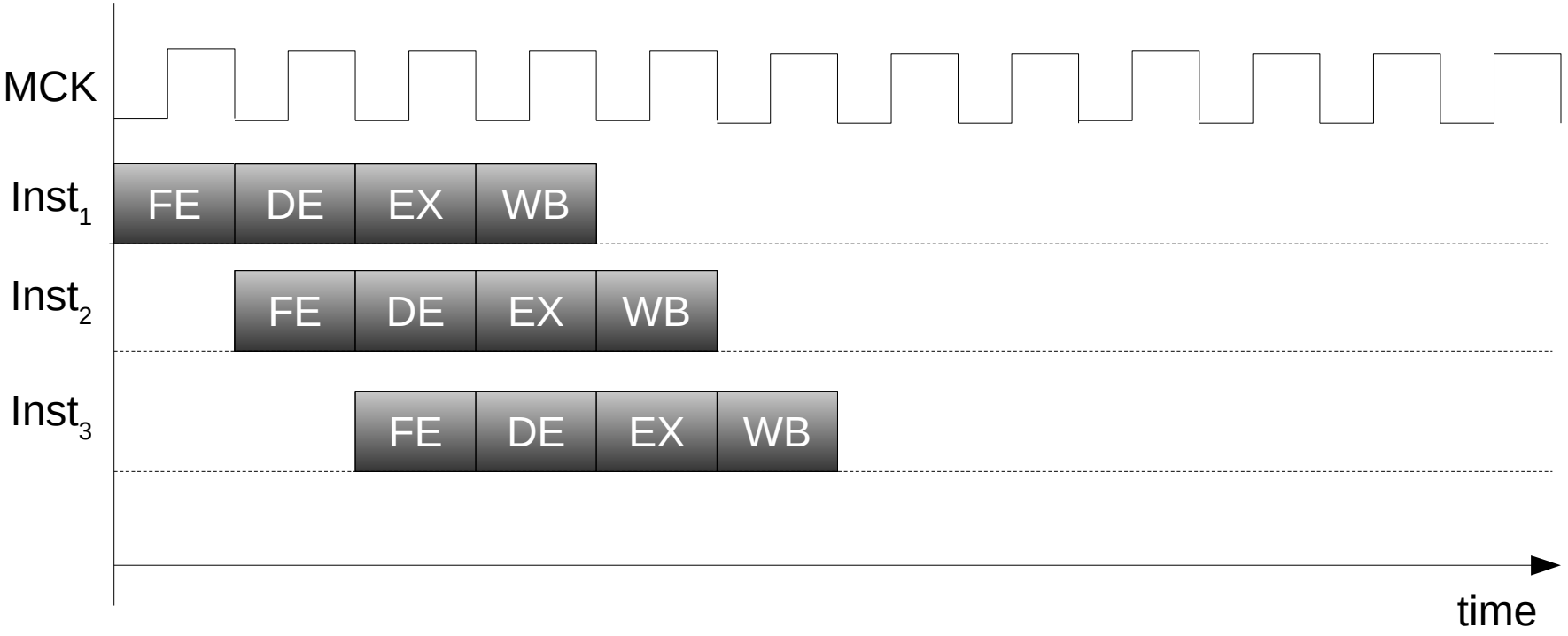
Execution profile



1 instruction takes 4 cycles to be executed
throughput = 1 instruction / 4 cycle
MCK = 100MHz → 25,000,000 instructions / second
FE, DE, EX, WB works 1 / 4 cycle → 3 / 4 cycle lazy

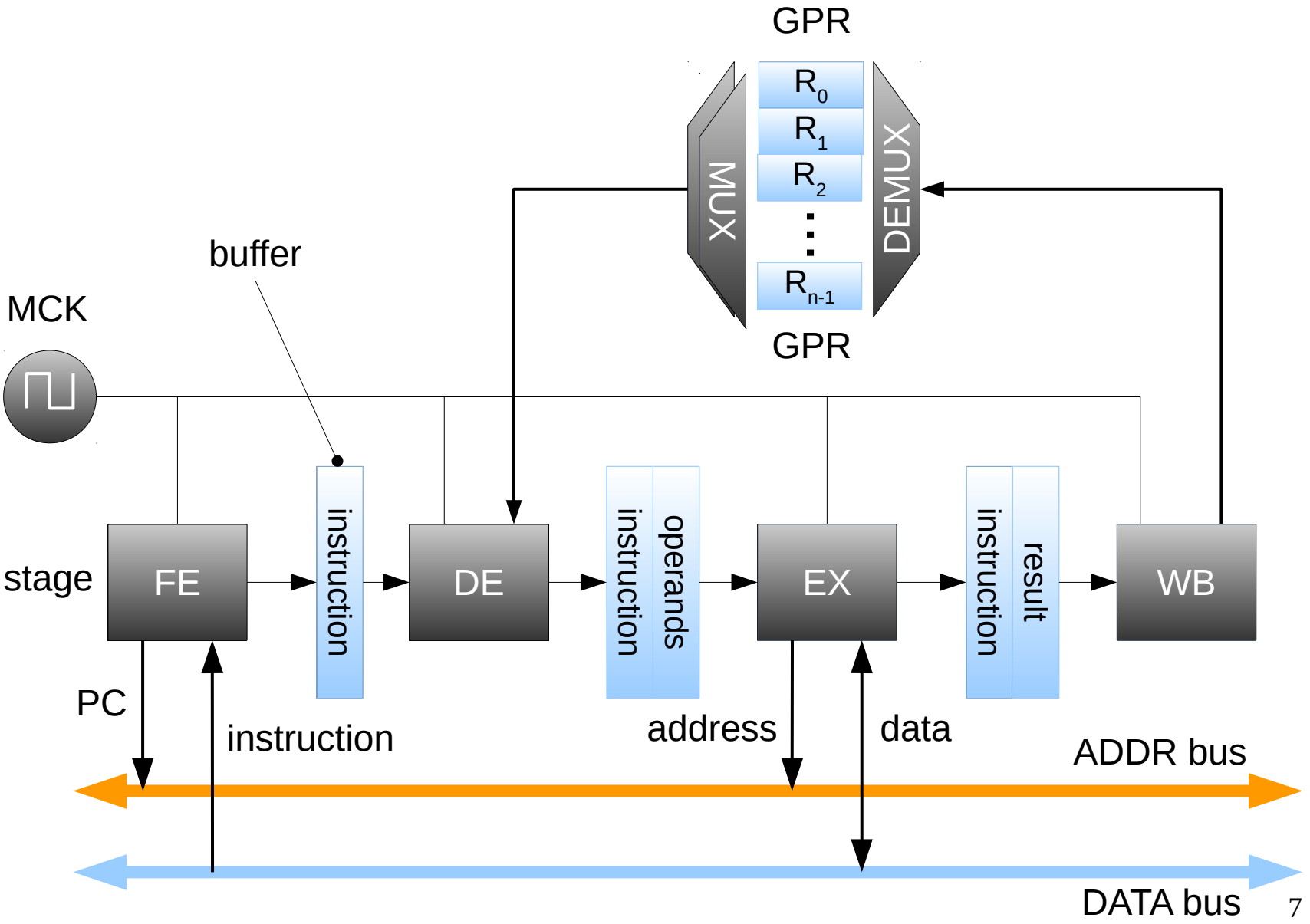


Pipeline execution

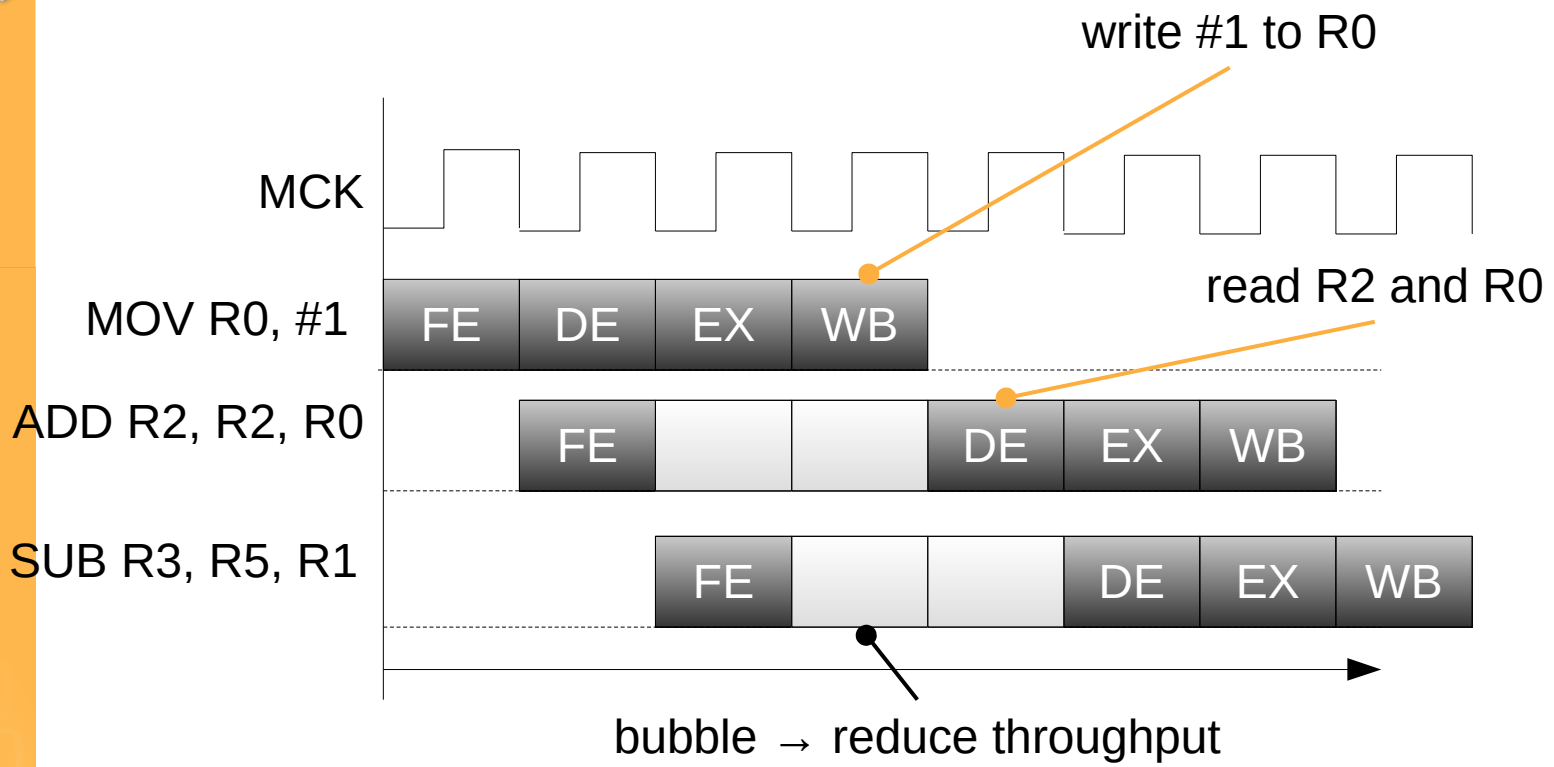


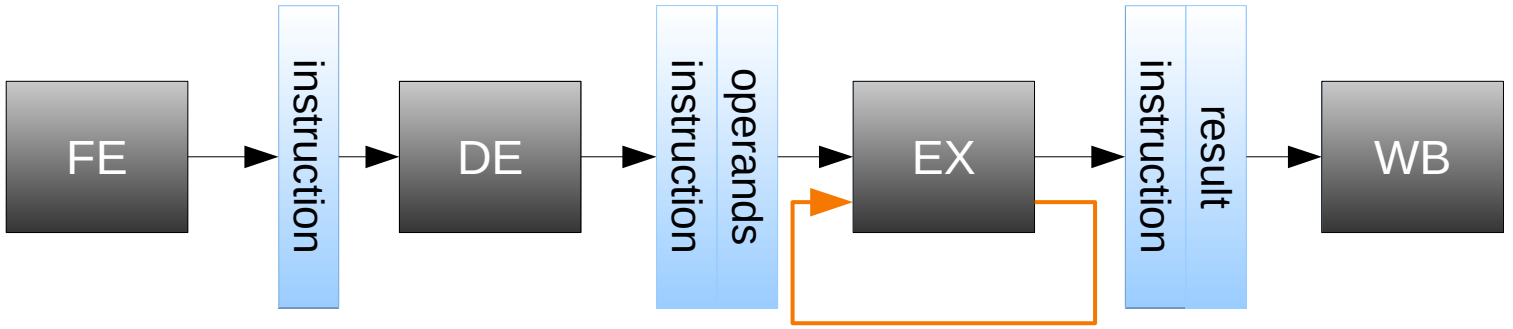
1 instruction takes 4 cycles to be executed
while $Inst_1$ is decoded, $Inst_2$ is fetched, ...
throughput = 1 instruction / 1 cycle (1 IPC)
MCK = 100MHz → 100,000,000 instructions / second
FE, DE, EX, CM used 100% of time

How to implement it?



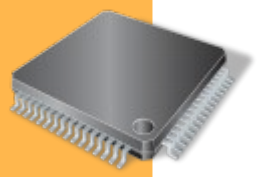
Issue 1: data dependency





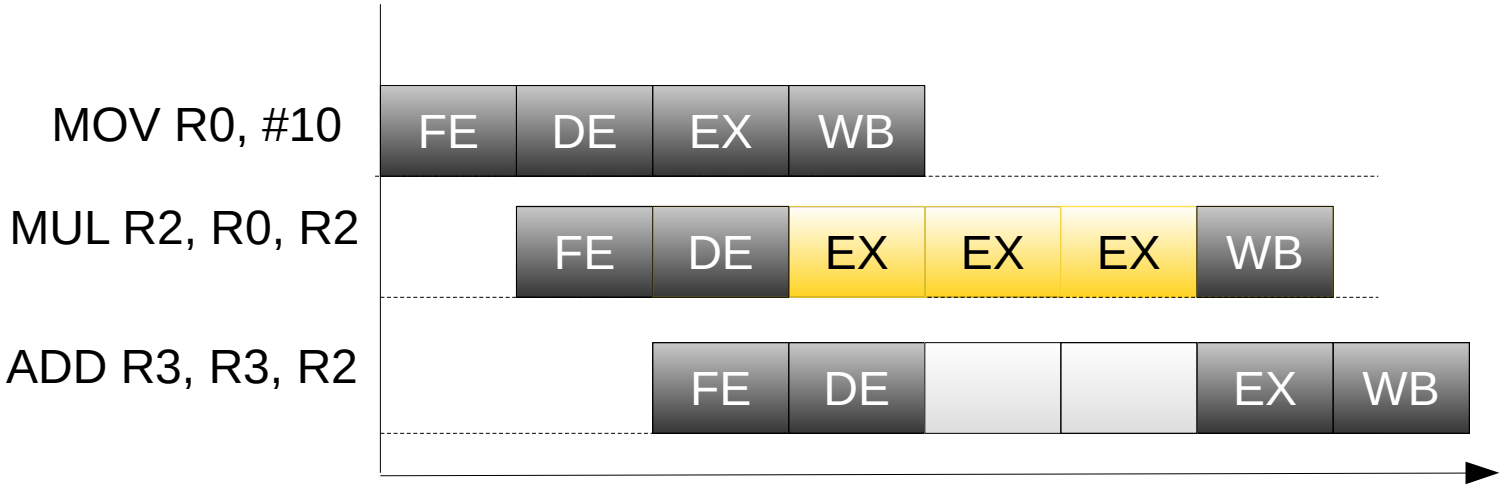
solution: by-pass

re-used result
is preserved in EX

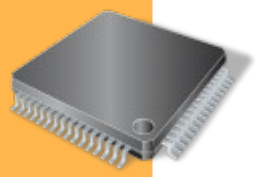


Issue 2: long-time instructions

- cycle time = longest operation
- time of multiplication, division, floating-point computation = 4-20 × addition, subtraction, …
- but long-time execution are rare!
- extend cycle time → waste of computation power
- solution: stay in DE several cycle / instruction type

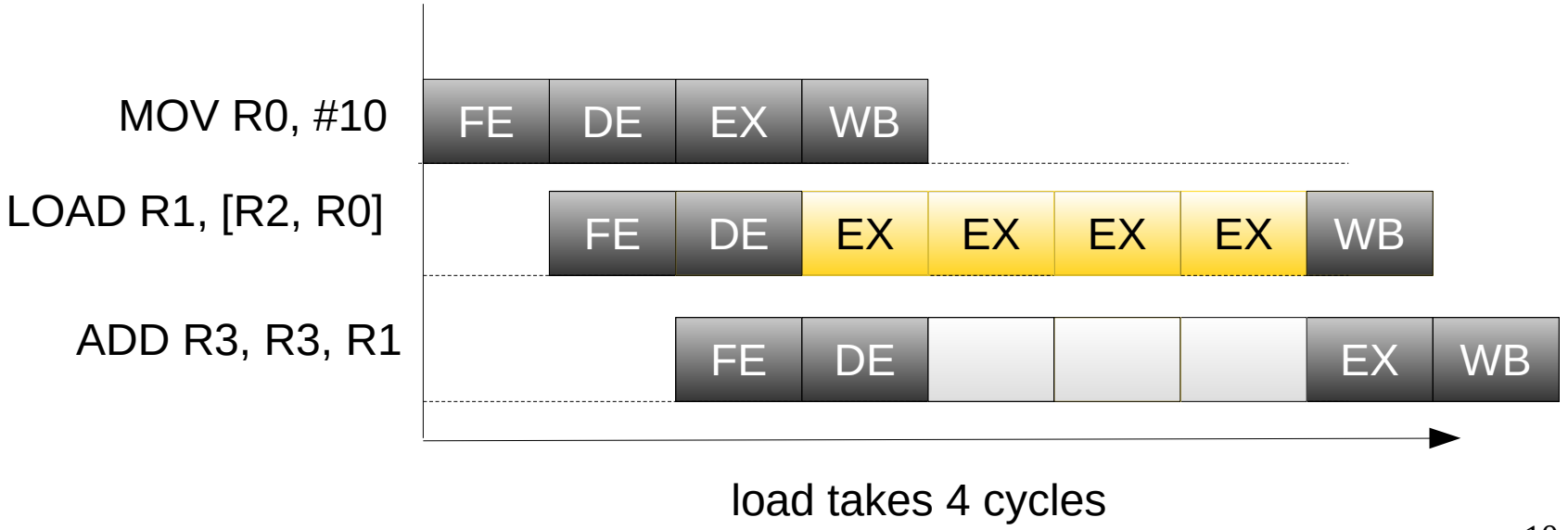


multiplication takes 3 cycles → insertion of bubbles

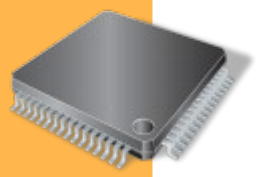


Issue 2 (b): memory accesses

- memory access – access time depends on memory
  - SRAM – 1-4 cycles
  - DRAM – 10-20 cycles
  - I/O – 1-100 cycles
  - DRAM + refresh – 10-2000 cycles
- solution: wait for acknowledge in DE stage

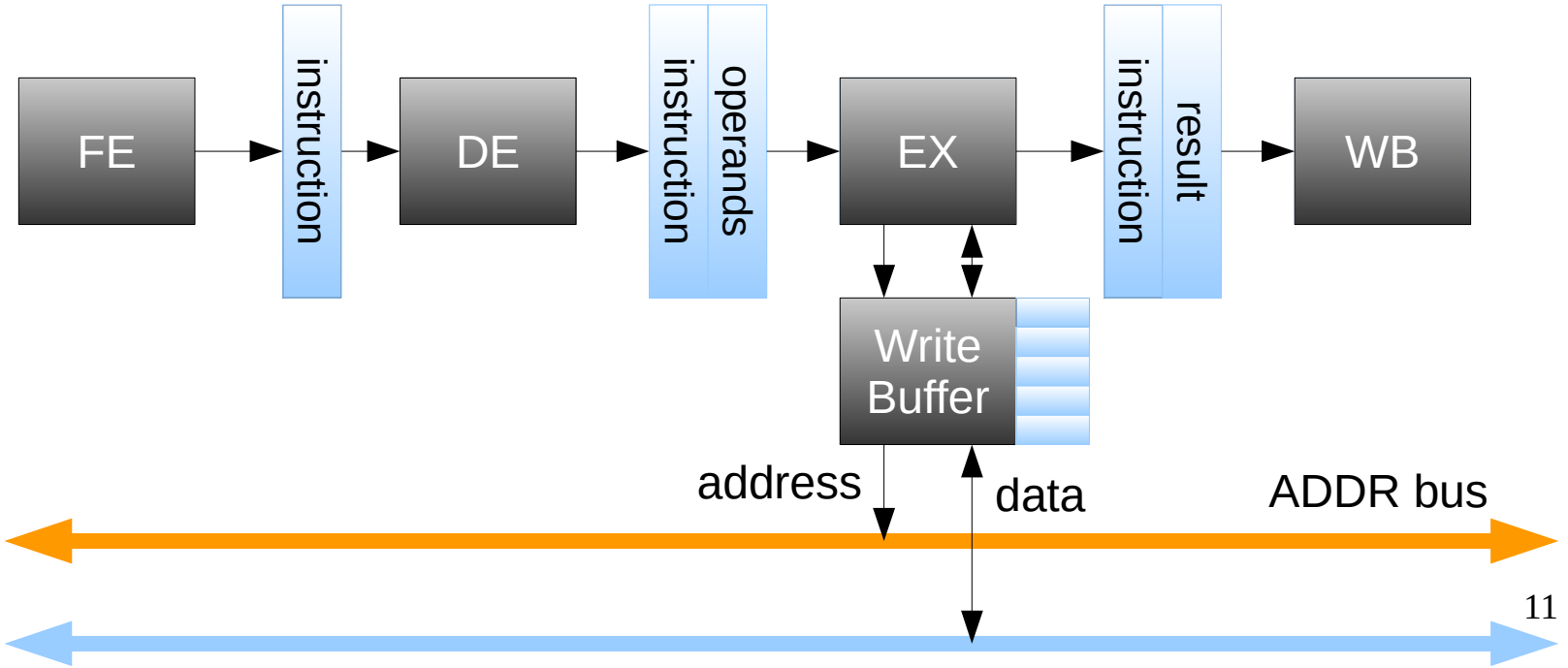


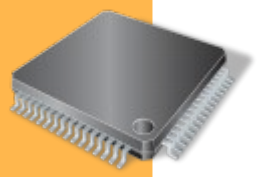
load takes 4 cycles



Issue 2 (c): store buffer

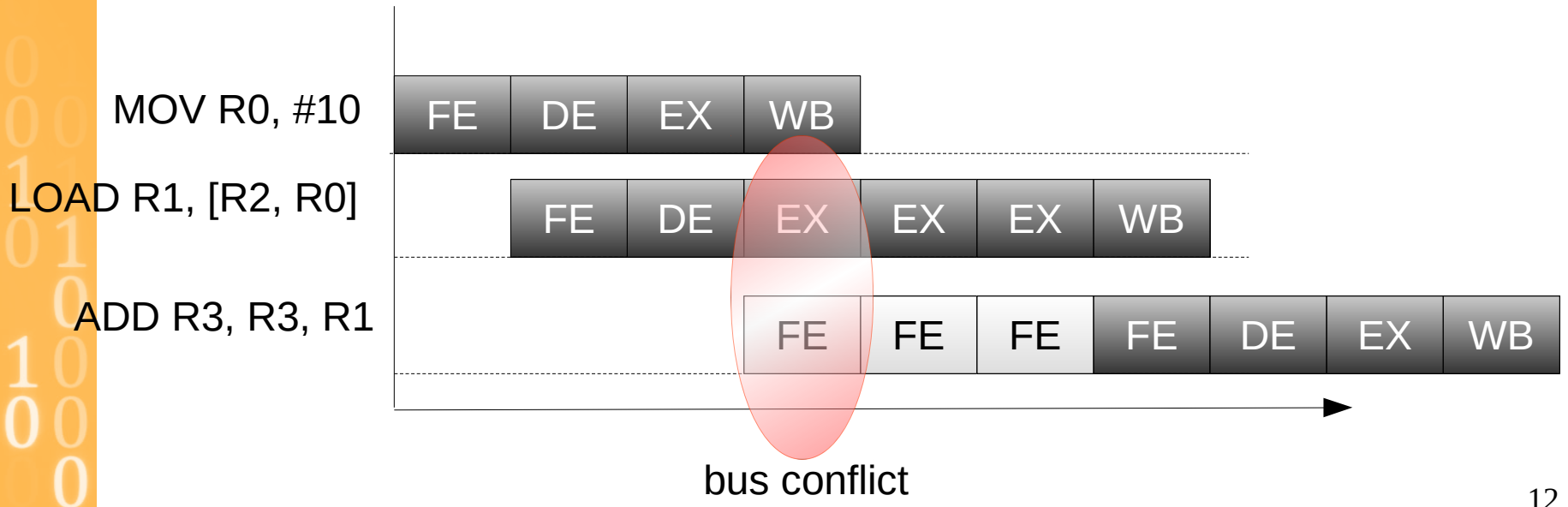
- load → wait for memory answer for use it in next instructions
- store → no wait needed → no pipeline stop needed → manage it asynchronously with the memory





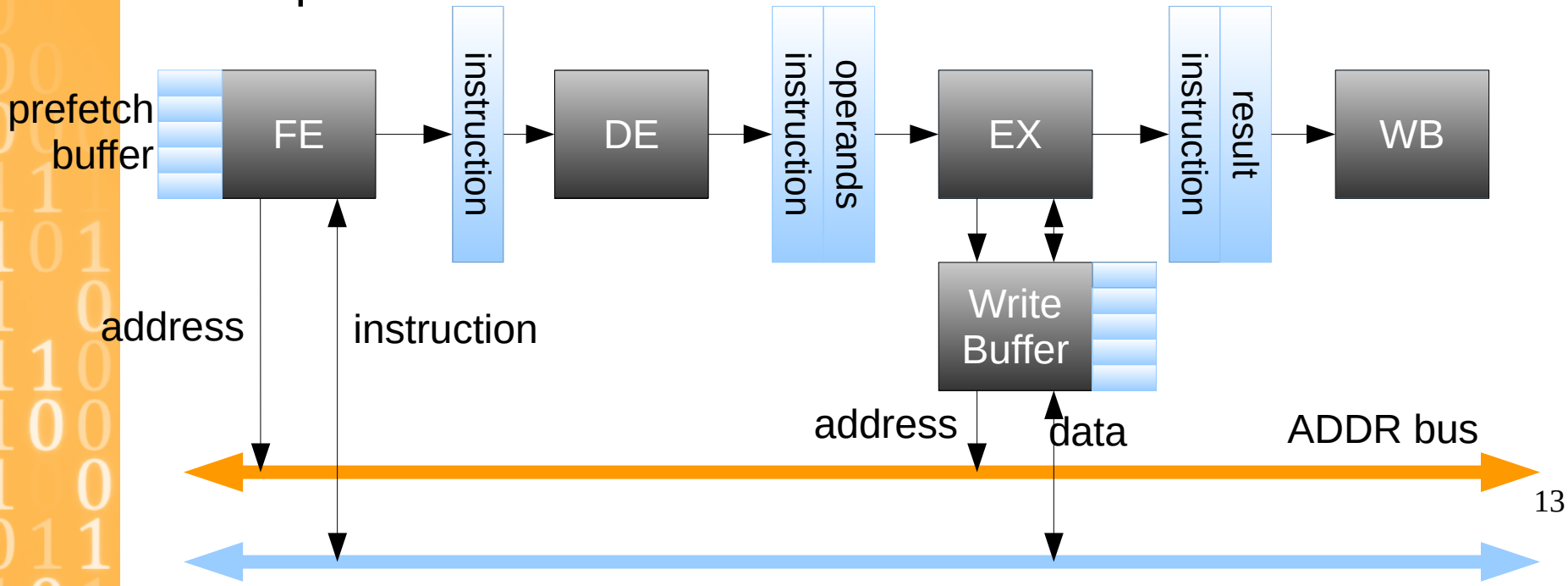
Issue 3: FE / DE competition for bus

- each cycle → read an instruction
- sometimes → load/store read/write data to memory
- which one ?
  - usually DE has priority
  - else it will block FE also

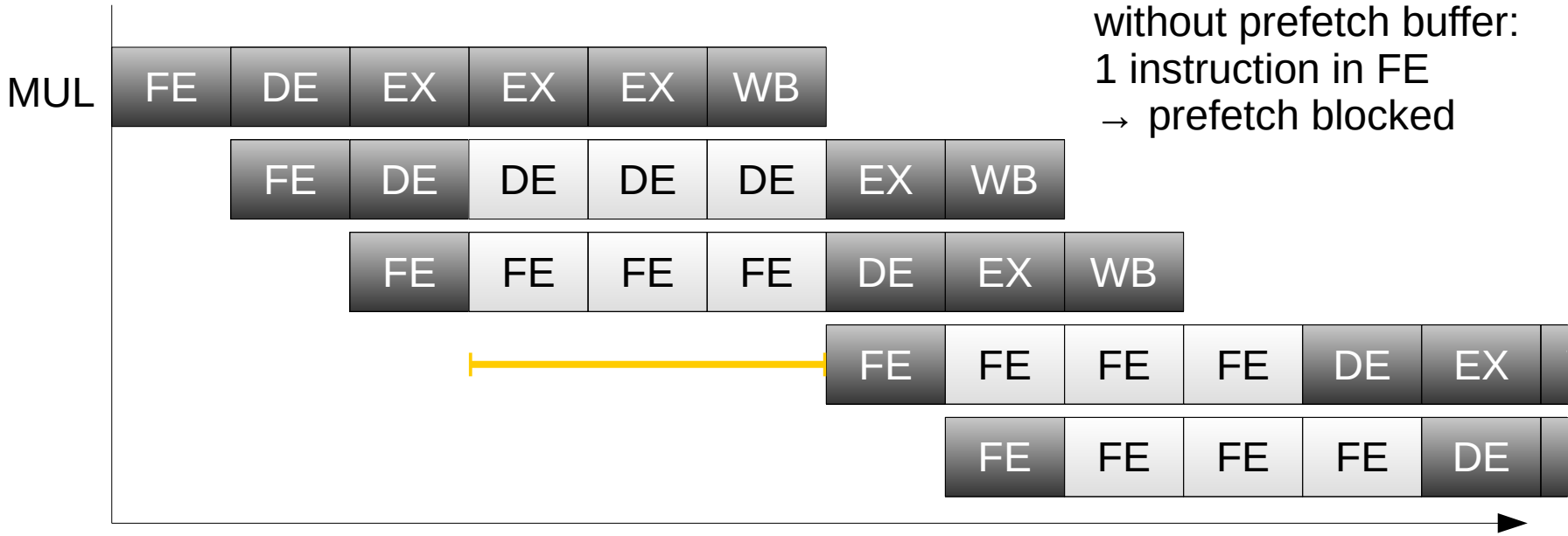


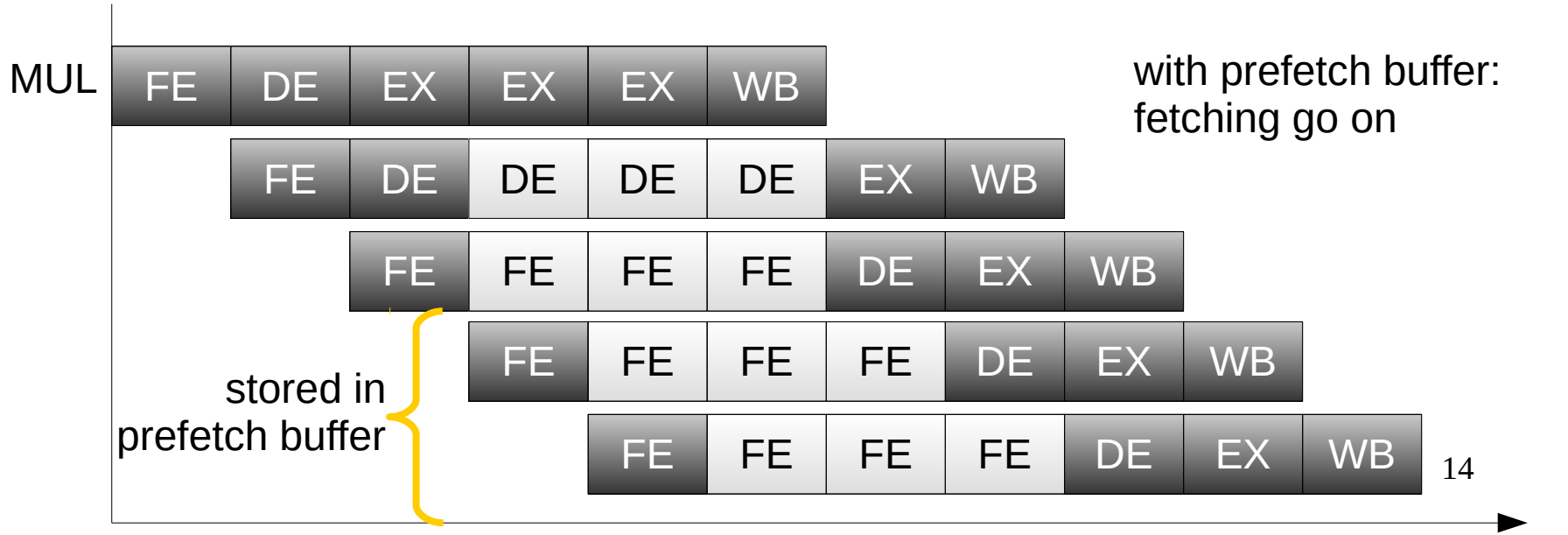
Issue 3 (b): prefetch buffer

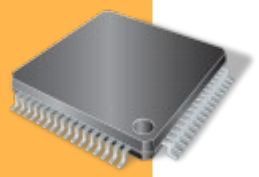
- during a long-time instruction execution in EX
- FE still fetching instruction from memory
- store them in prefetch buffer
- when a bus conflict occurs → instruction are extracted from the prefetch buffer



Issue (3): prefetch buffer work

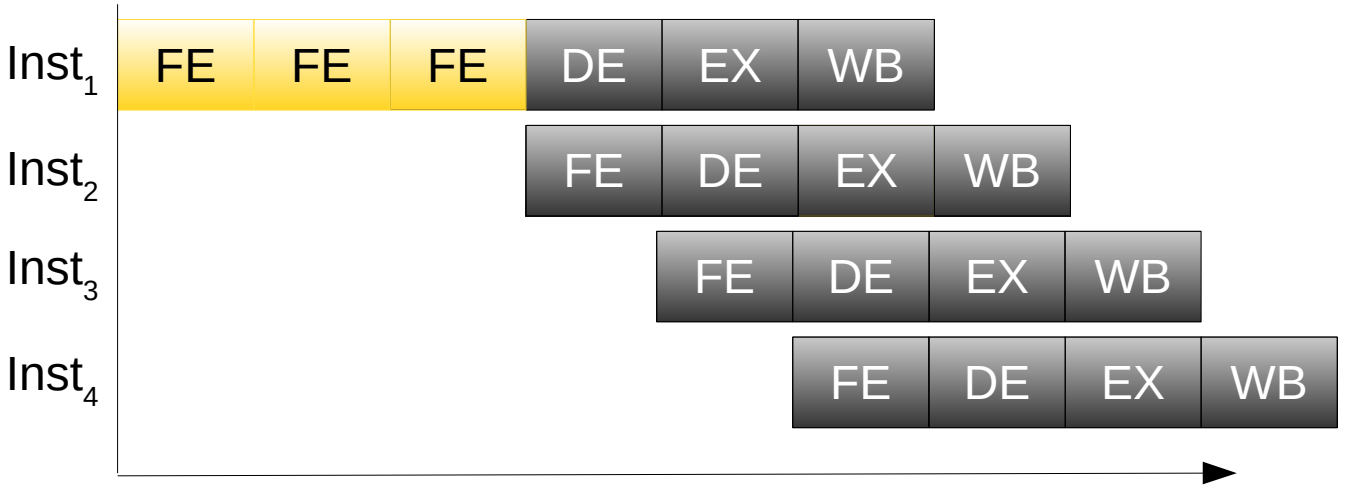




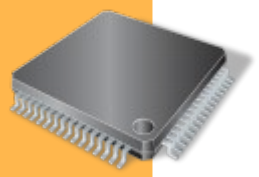


Issue 4: instruction read

- burst-mode memory access
  - first access → access time
  - next accesses in sequence → 1 instruction / cycle
- instruction cache

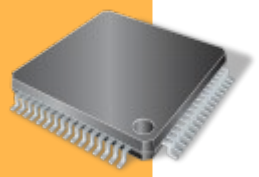


load takes 4 cycles



Industrial microprocessors

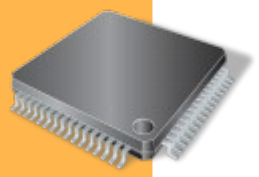
- frequency
  - laptop / desktop – 2-4 GHz (10^9 IPC)
  - embedded – n × 10MHz – n × 100MHz
- pipeline
  - 3 stages (ARM7)
  - most frequent 5 stages (ARM9, Leon, etc)
  - 6-7 stages (PowerPC)
  - super-pipeline – 10 stages (no more used)



Exercise (1)

- ADD, SUB – 1 cycle EX
- MUL – 3 cycles EX
- STR – store, 1 cycle (write buffer)
- LDR – load, 4 cycles
- no prefetch buffer
- 4 cycles for instruction access time

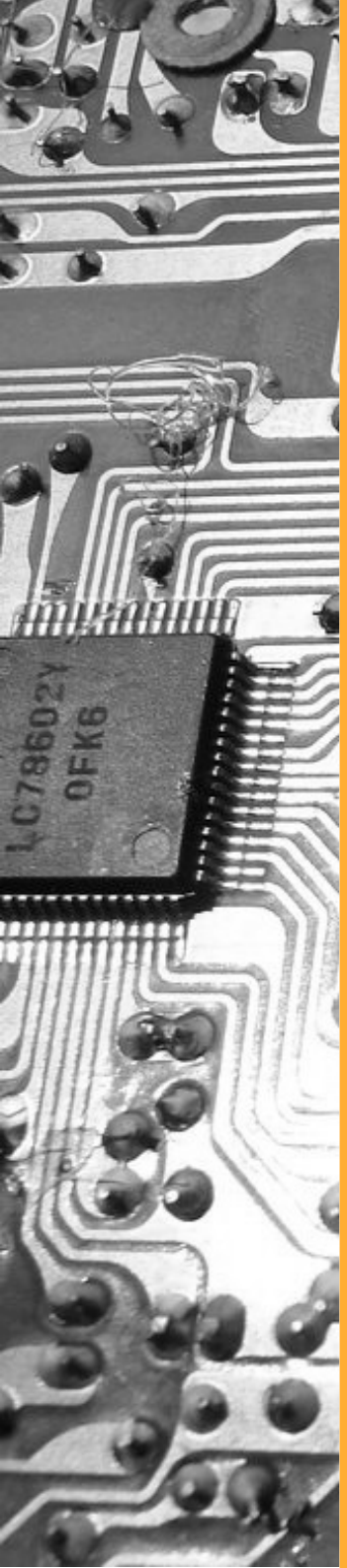
- draw pipeline use of
  a) ADD R0, R0, R1
  b) SUB R1, R2, R1
  c) MUL R1, R2, R1
  d) STR R1, [R3]
  e) ADD R0, R3, #4
  f) LDR R2, [R3]
  g) ADD R2, R2, #1



Exercise (2)

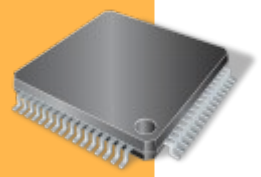
- ADD, SUB – 1 cycle EX
- MUL – 3 cycles EX
- STR – store, 1 cycle (write buffer)
- LDR – load, 4 cycles
- with prefetch buffer
- 4 cycles for instruction access time

- draw pipeline use of
  a) MUL R0, R0, R1
  b) SUB R1, R2, R1
  c) ADD R1, R2, R1
  d) ADD R1, R3, #2
  e) SUB R2, R2, R2
  f) LDR R2, [R3]
  g) ADD R2, R2, R2

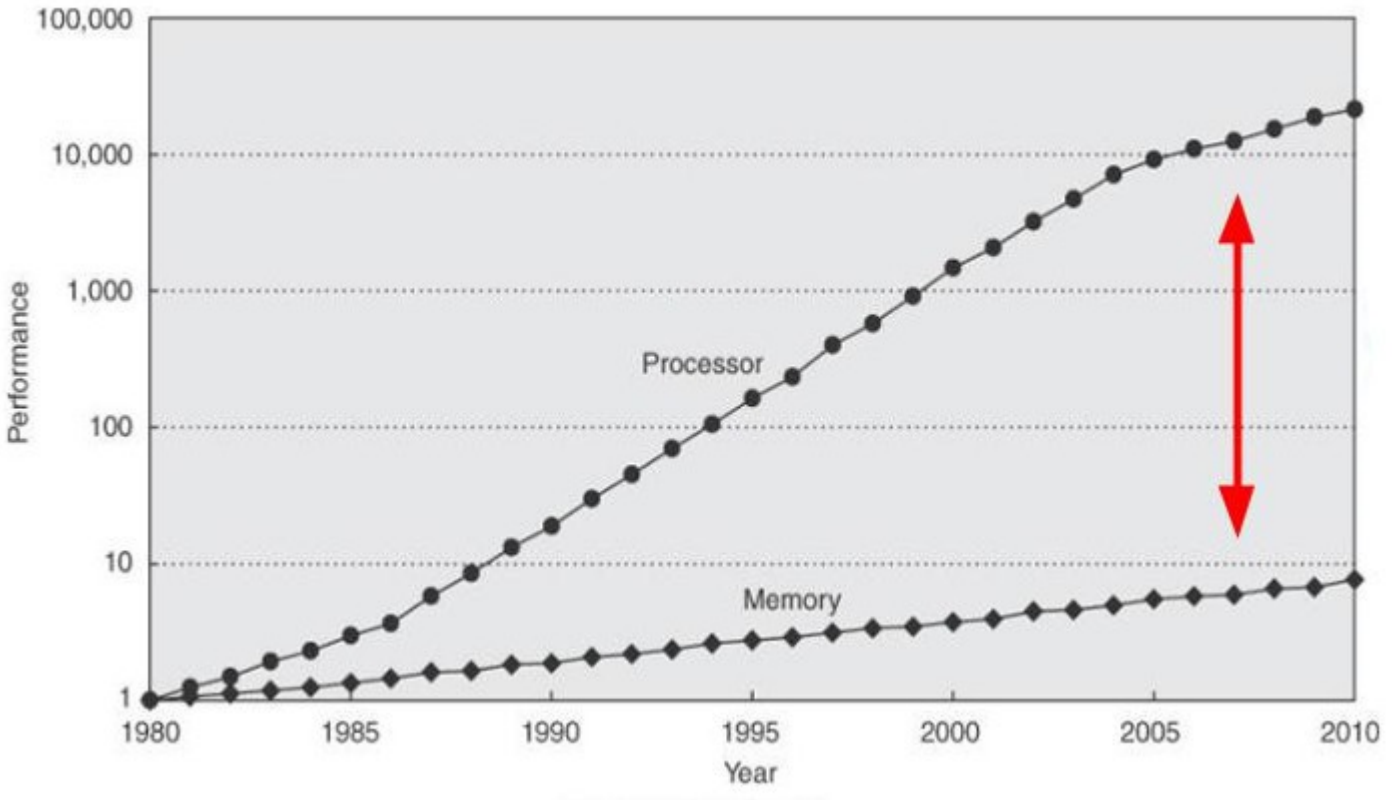


Overview

- Introduction
- Pipeline
- **Cache memories**
- Conclusion



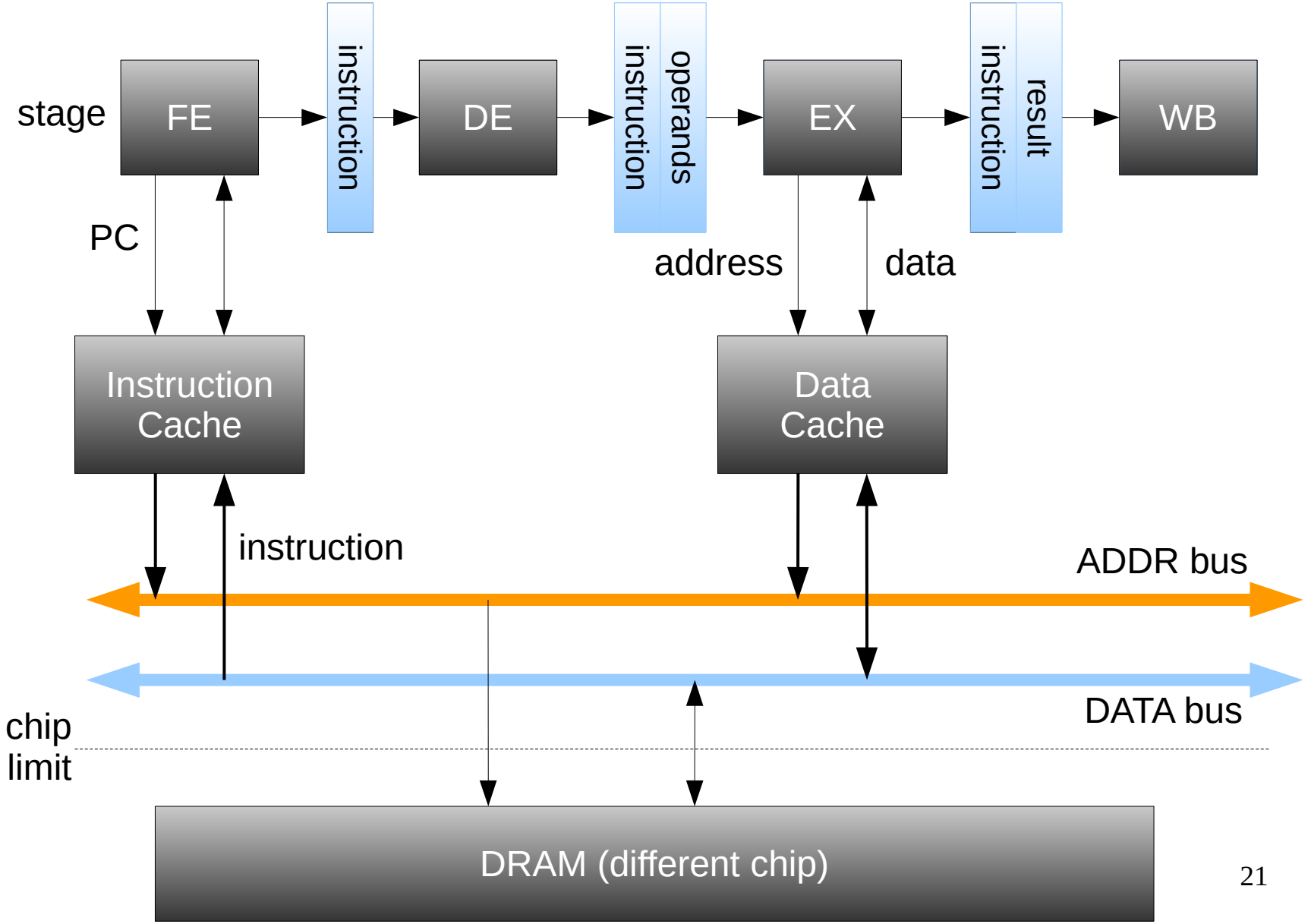
Performance Memory Gap

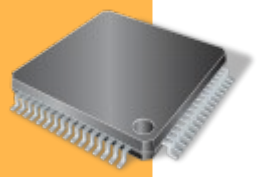


memory = DRAM → 90% of memory on laptop / desktop
gap reduced for embedded systems

- 50% of memory is flash (same latency as DRAM)
- microprocessors are less powerful

Inserting cache memories

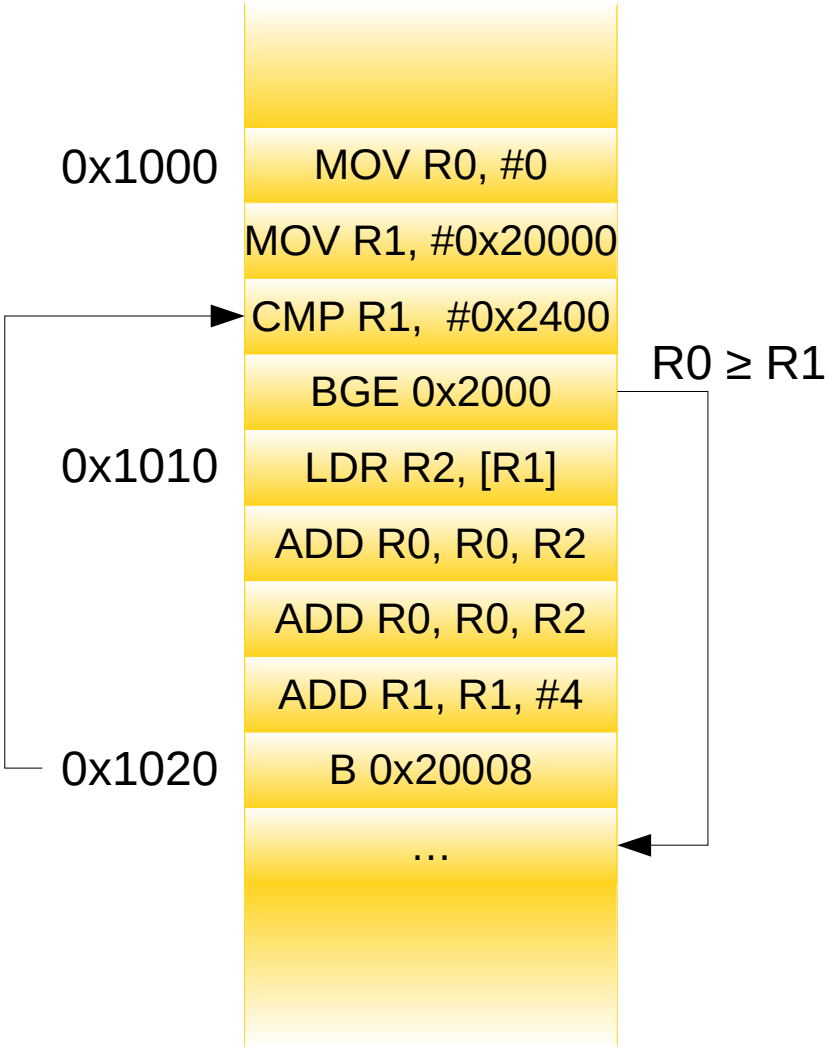




Temporal Locality for Instructions

- when I execute an instruction
- high probability to re-execute it in a close future
- loops, interrupts (90% of the program)

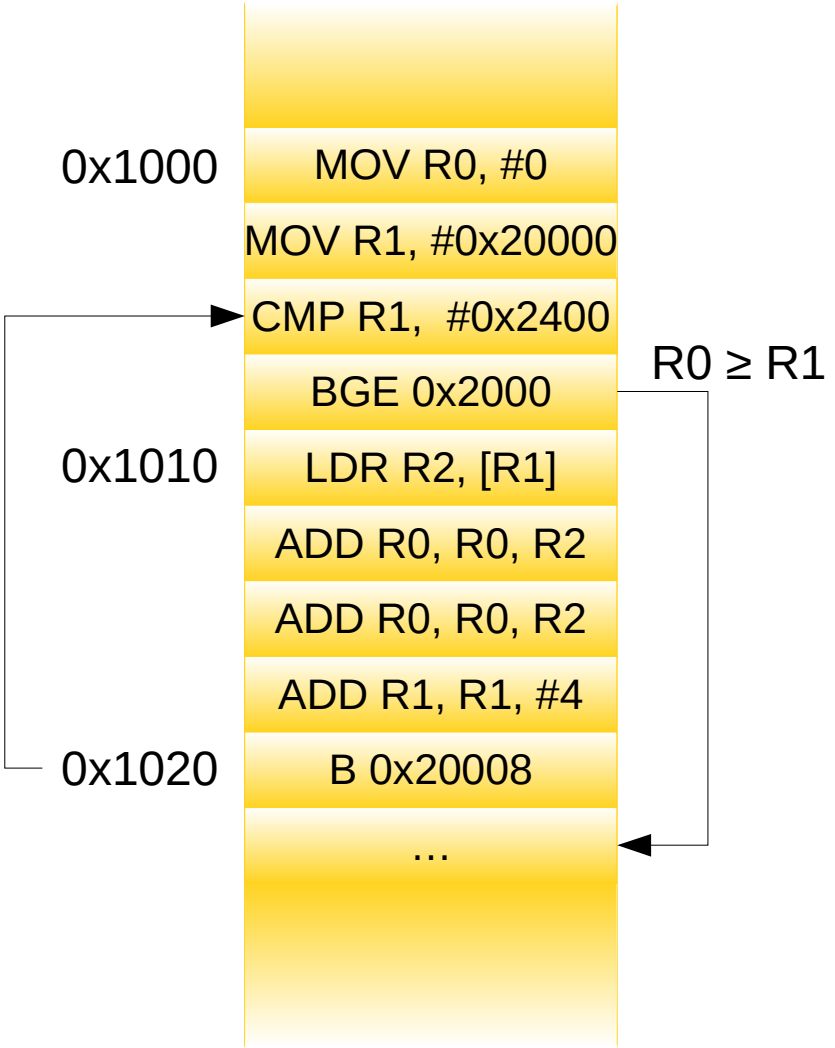
```
int t[256];
s = 0;
for(i = 0; i < 256; i++)
        s = s + t[i];
```



Temporal Locality for Data

- in a particular segment of code
- some data are used and re-used a lot
- case for
  - local variables
  - some global variables

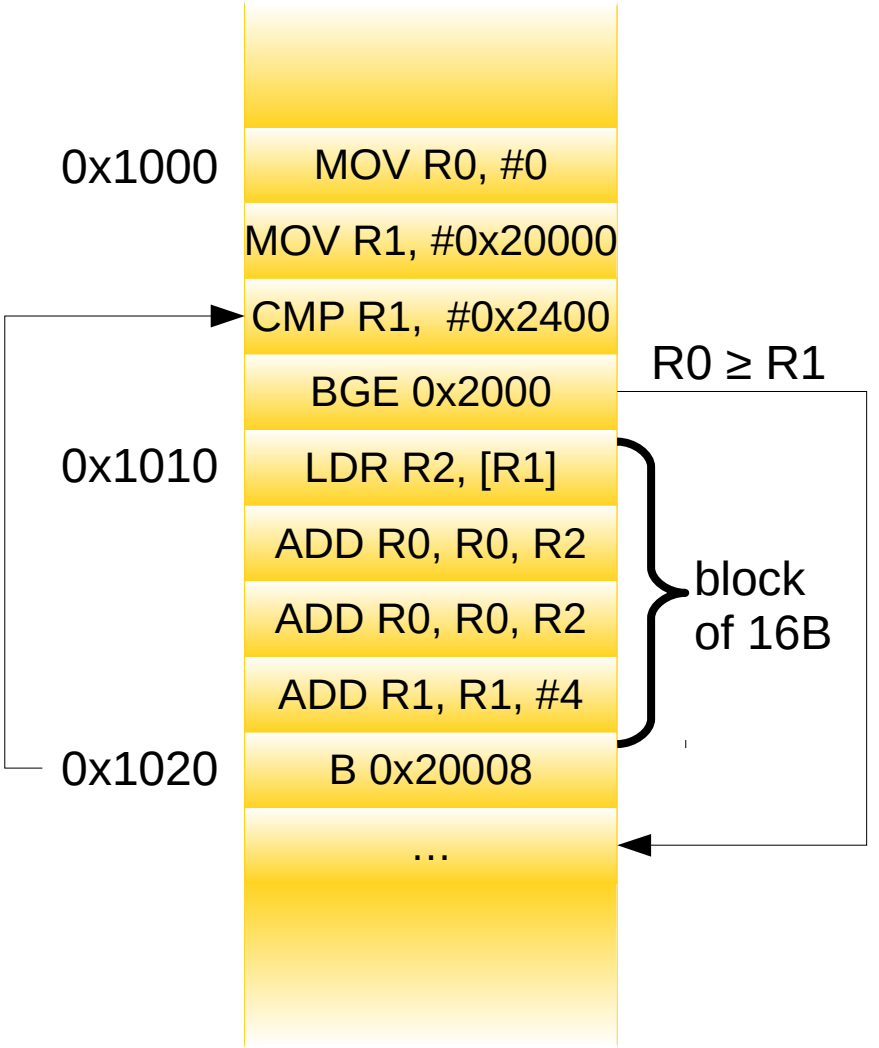
```
int t[256];
s = 0;
for(i = 0; i < 256; i++)
        s = s + t[i];
```

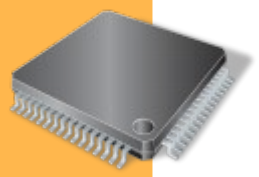


Spatial Locality for Instructions

- when I execute an instruction
- high probability to execute the next instruction
- load several instructions at time to fill the cache → group them by block

```
int t[256];
s = 0;
for(i = 0; i < 256; i++)
        s = s + t[i];
```

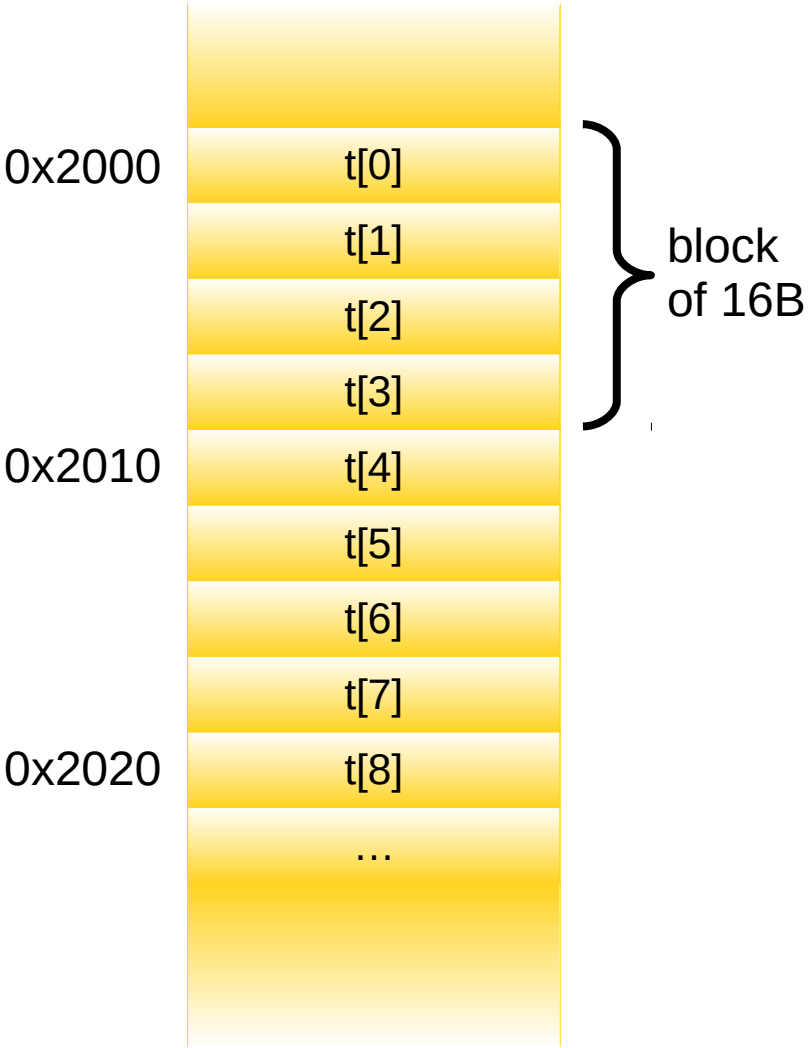




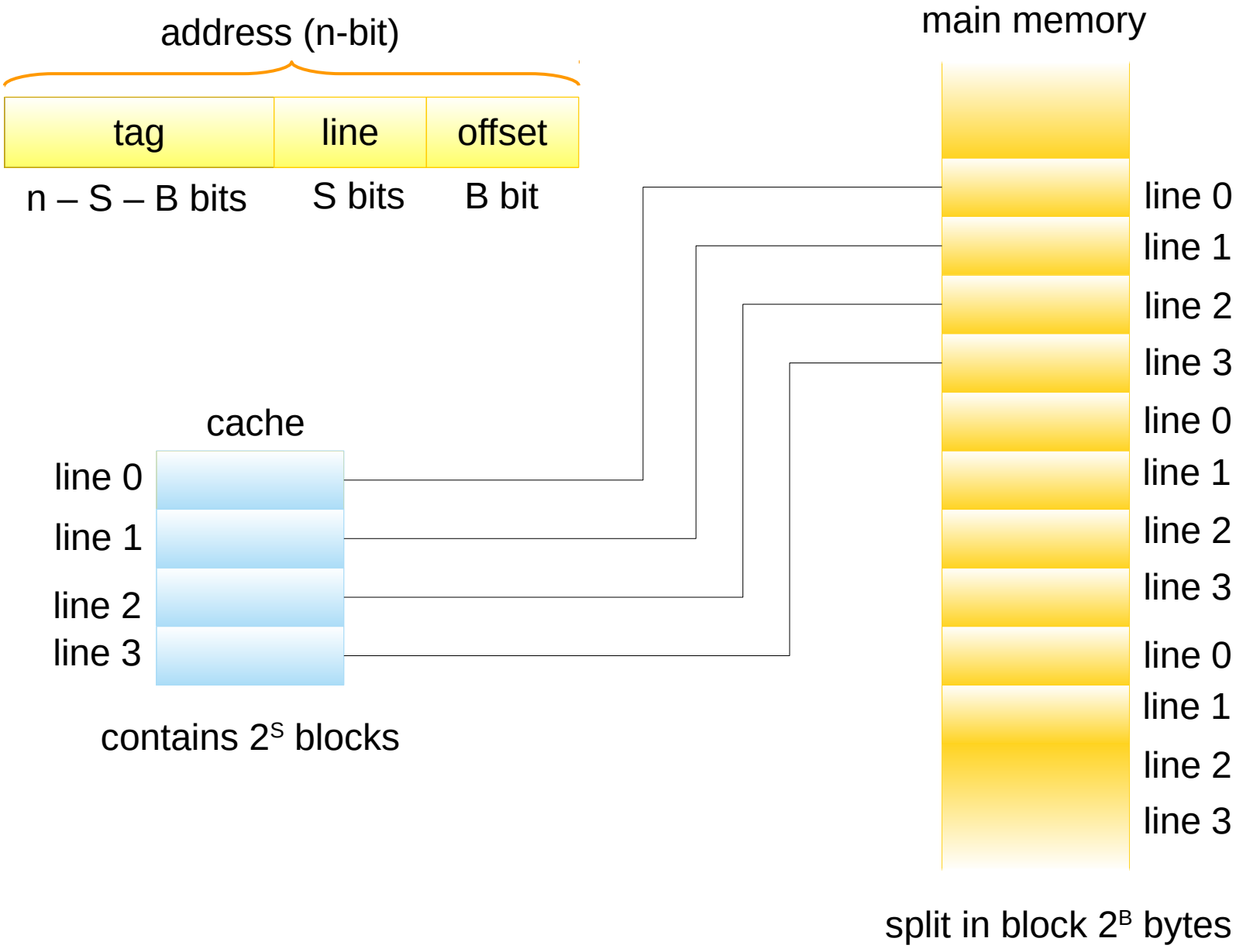
Spatial Locality for Data

- when I work with an array (frequent)
- when I access t[i] in a loop (frequent)
- I will often access t[i+1] or t[I-1], i.e. close data in memory
- group and load several data at time

```
int t[256];
s = 0;
for(i = 0; i < 256; i++)
        s = s + t[i];
```



Direct-Mapped Cache



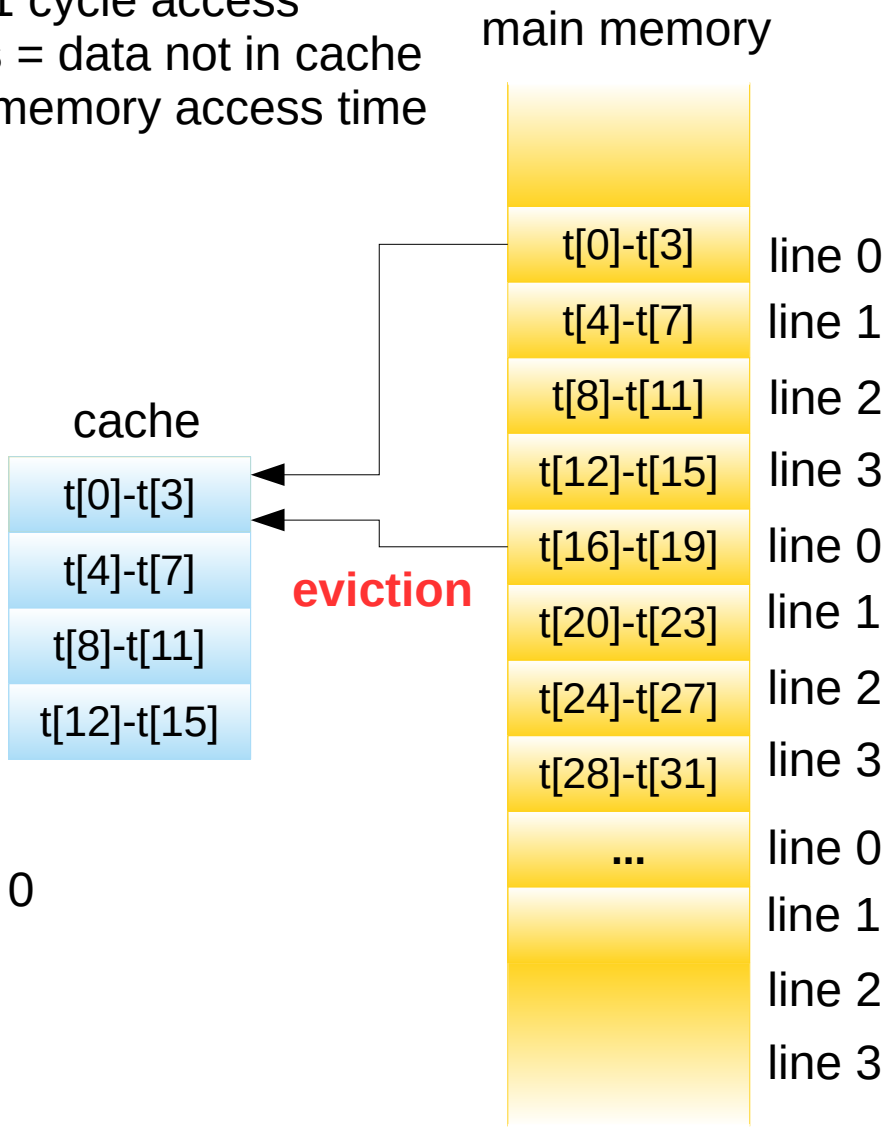
Example with an array

```
int t[256];
s = 0;
for(i = 0; i < 256; i++)
      s = s + t[i];
```

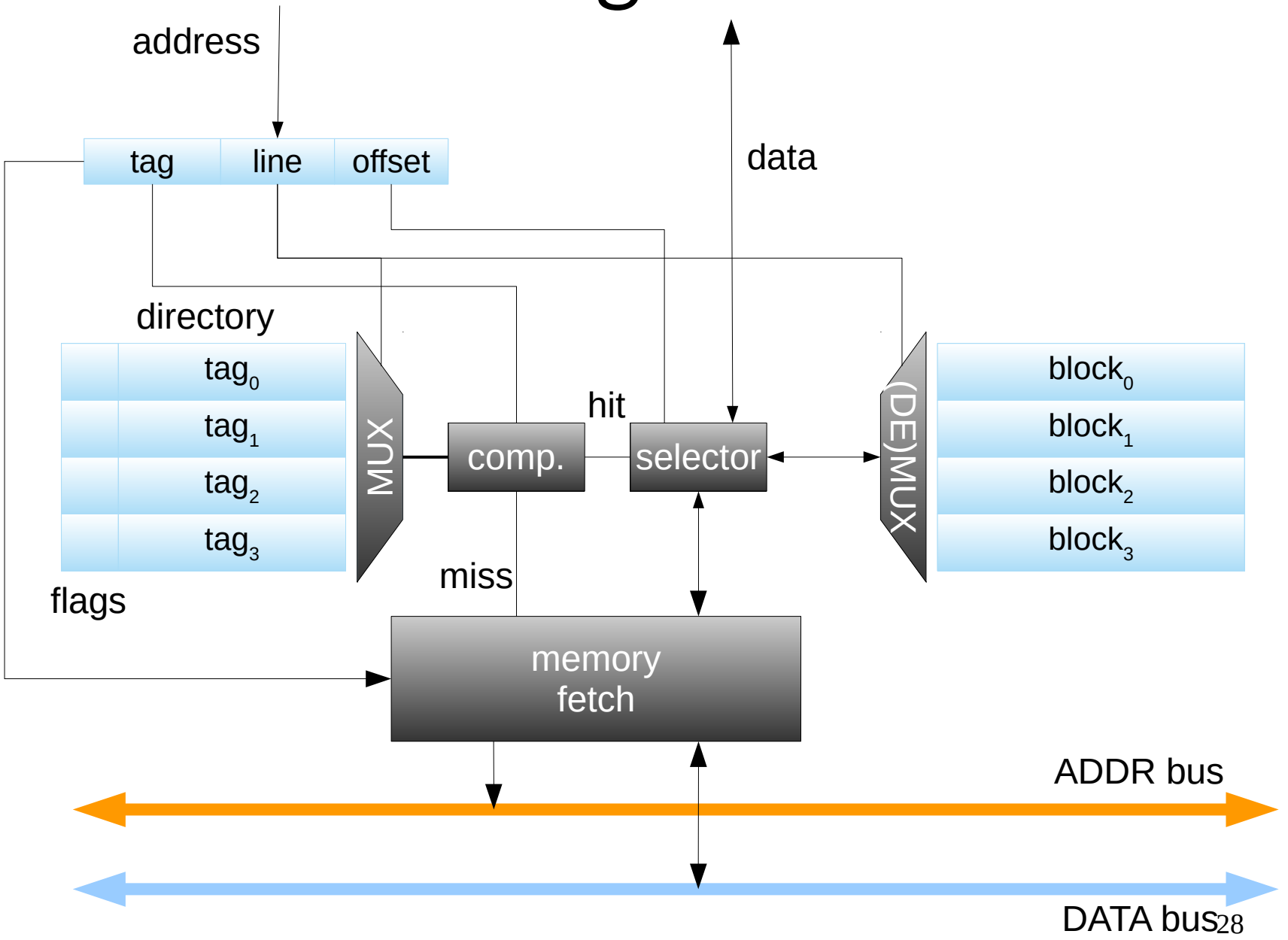
hit = data in cache,
1 cycle access
miss = data not in cache
memory access time

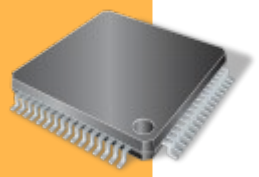
B = 4 → 16B blocks
S = 2 → 4 lines

t[0] – miss, load in line 0
t[1], t[2], t[3] – hits
t[4] – miss, load in line 1
t[5], t[6], t[7] – hits
t[8] – miss, load in line 2
t[9], t[10], t[11] – hits
t[12] – miss, load in line 3
t[13], t[14], t[15] – hits
t[16] – miss, remove block in 0
and load new block in 0
t[17], t[18], t[19] – hits
…
hit rate = 3 / 4 access = 75%



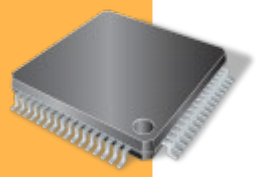
Building a cache





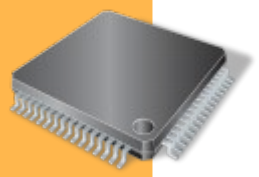
Writing to Data Cache (a)

- difference with Instruction Cache
  - store operations modify the content of cache
  - write block back in the memory
- Write-Back
  - Dirty bit added to directory
  - set when a write is performed
  - write-back to memory at eviction if Dirty is set
  - coalesce several stores until write-back
- + Write-Allocate
  - data blocks are loaded in cache
  - on a load
  - on a store
- + Write-Buffer



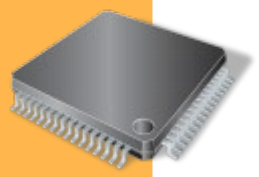
Writing to Data Cache (b)

- Write-Through
  - update the cache block
  - and update the memory also
- + Write-No-Allocate
  - no block allocate if written block is not in the cache
  - may cost a miss for uncached data
- + Write Buffer
  - no more miss on write



Benefits of cache for pipeline

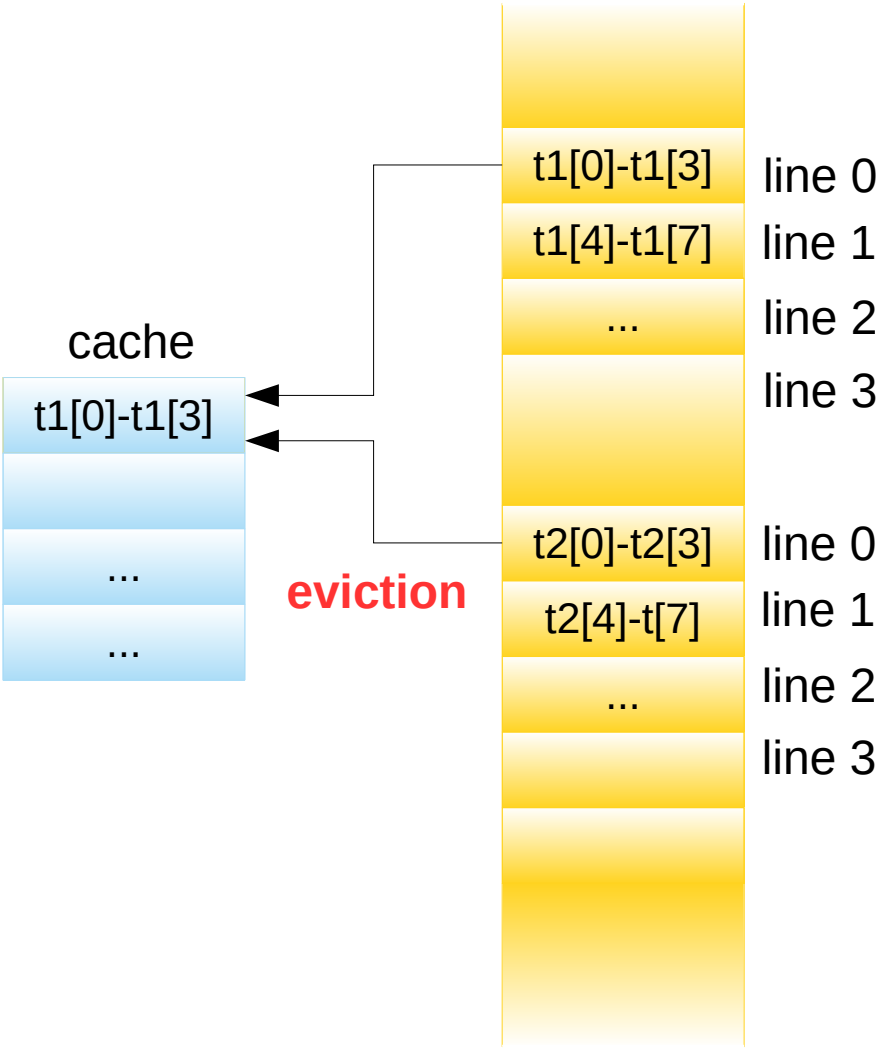
- EX stage (Data Cache)
  - 1 cycle memory access for a hit
  - unchanged for a miss
  - may be coupled with a Write Buffer
- FE stage (Instruction Cache)
  - no more fetching time for a hit
  - fetching unchanged for a miss
  - speculative execution cost reduced with cache hits



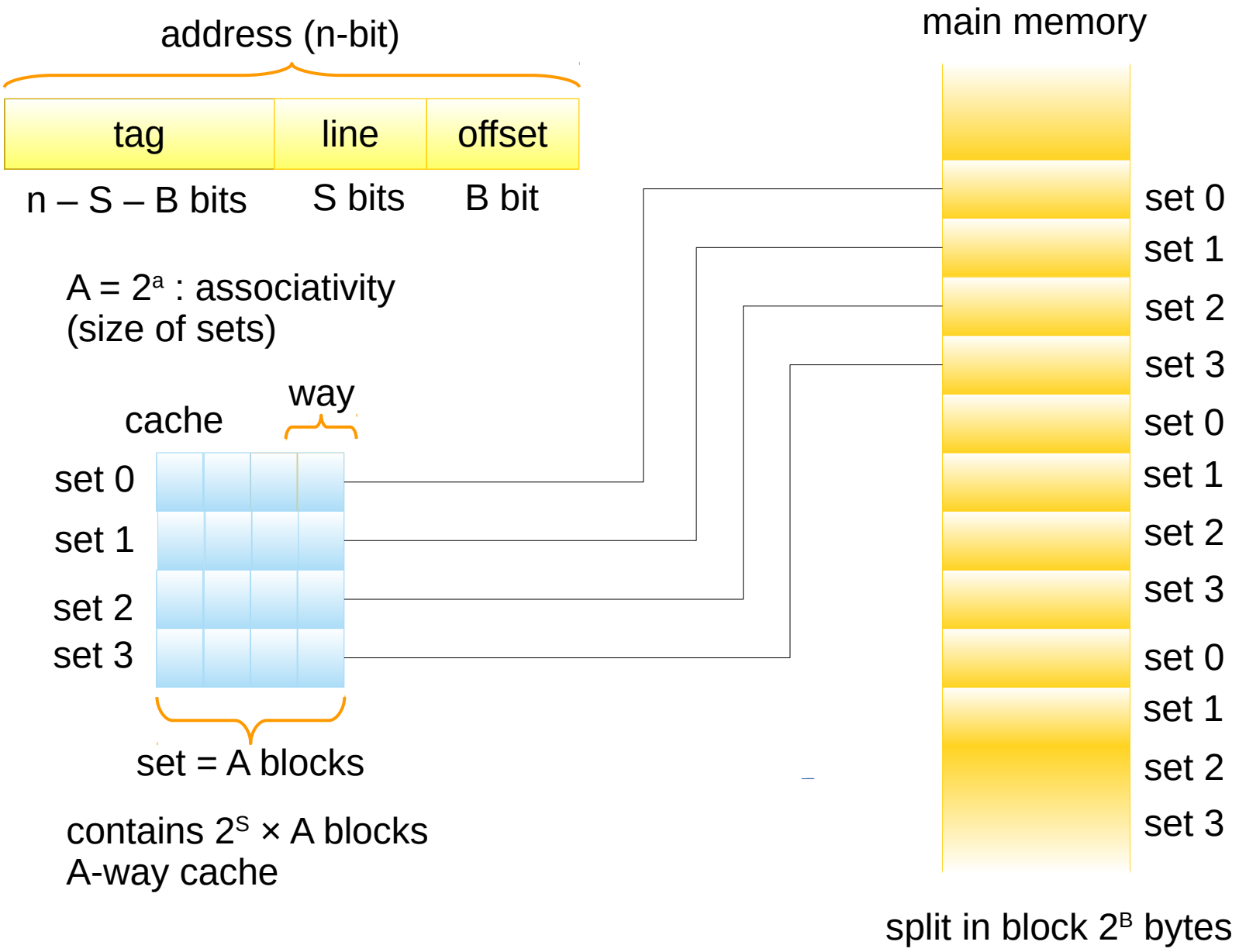
Replacement problem

```
int t1[256], t2[256];
s = 0;
for(i = 0; i < 256; i++)
        s = s + t1[i] * t2[i];
```

read t1[0] → miss
block t1[0]-t1[3] loaded in 0
read t2[0] → miss
t1[0]-t1[3] evicted from 0
t2[0]-t2[3] loaded in 0
read t1[1] → miss
t2[0]-t2[3] evicted from 0
t1[0]-t1[3] loaded in 0
read t2[1] → miss
t1[0]-t1[3] evicted from 0
t2[0]-t2[3] loaded in 0
…
hit rate = 0%!



Associative Cache

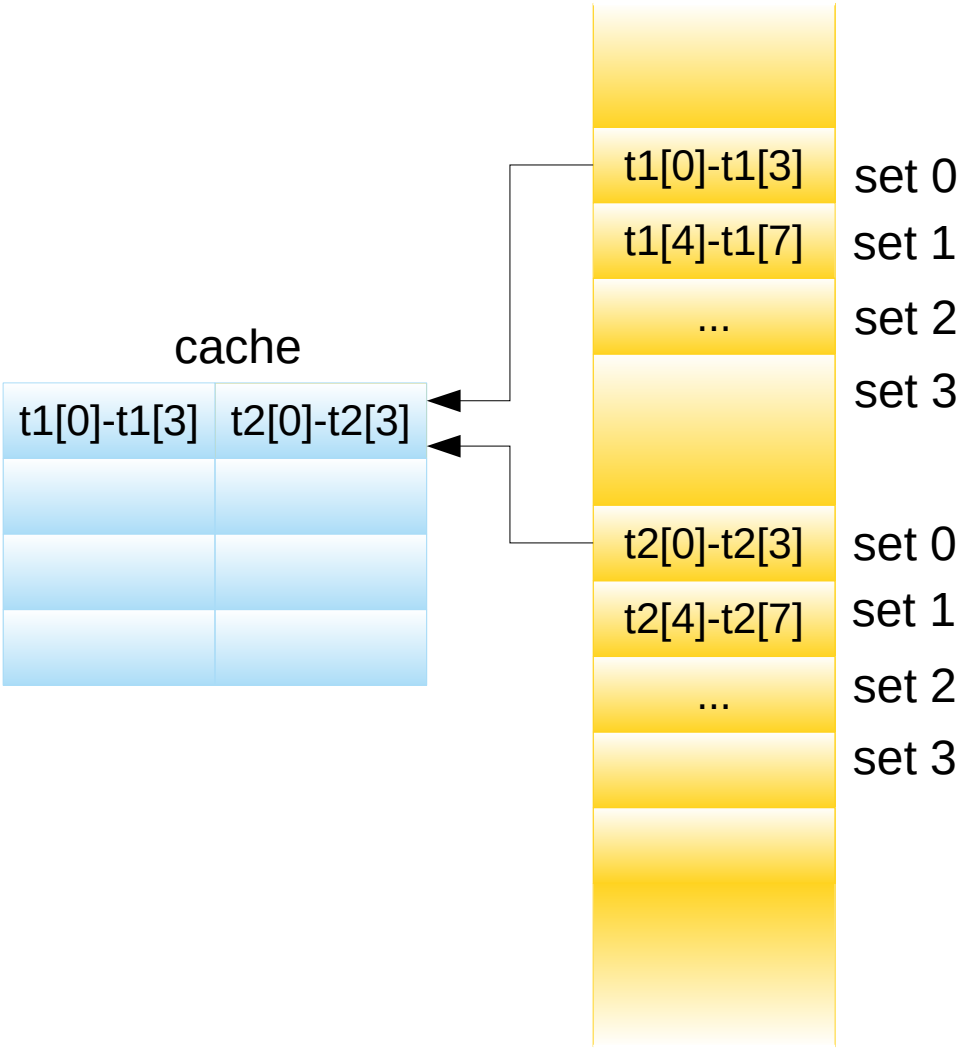


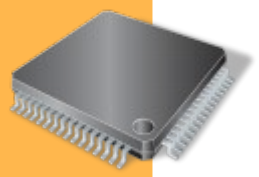
Replacement problem solution

```
int t1[256], t2[256];
s = 0;
for(i = 0; i < 256; i++)
        s = s + t1[i] * t2[i];
```

A = 2

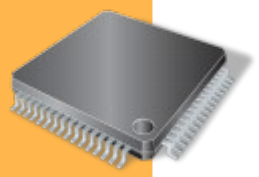
read t1[0] → miss
t1[0]-t1[3] loaded in 0
read t2[0] → miss
t2[0]-t2[3] loaded in 0
read t1[1] → hit
read t2[1] → hit
read t1[2] → hit
read t2[2] → hit
…
hit rate = 75%





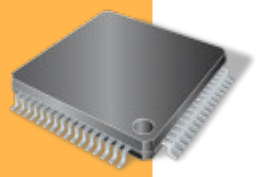
Replacement policies

- when the set is full → evict one block
  - which one?
  - replacement policy
- Common policies
  - random – evict any block
  - FIFO – set = queue, evict the last one
  - LRU (Least Recently Used) – assign at each block in set, accessed block age → 0, other block age + 1, evict the oldest block (variant Pseudo LRU)
  - MRU (Most Recently Used) – evict the youngest one
- Performances – all performs equally well



Industrial Caches

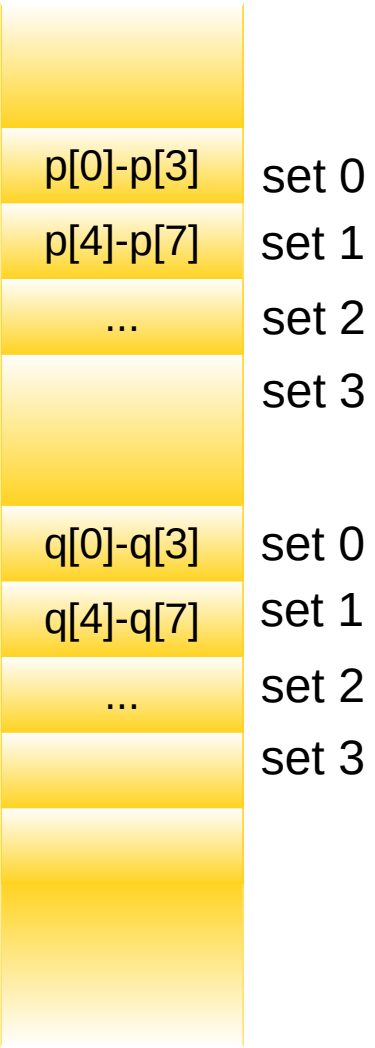
- architecture
  - separated ≠ unified
  - several levels of cache
    L1 (close to core, fast), L2, L3 (close to memory, slow)
- dimension
  - size – 16-64 KB (L1), 256KB-2MB (L2, L3)
  - associativity A = 2-8
  - block size B = 16-64B

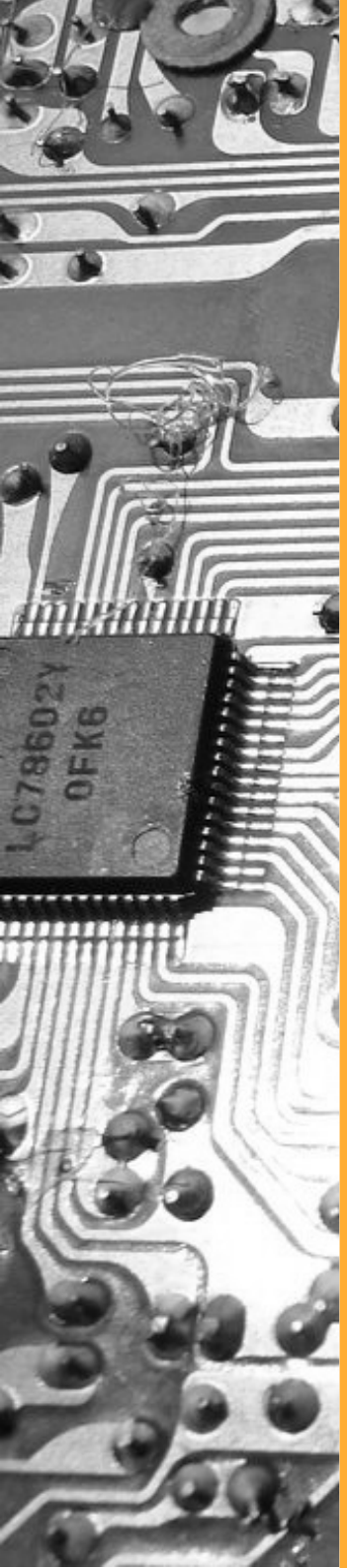


Exercise

```
void copy(int p[ ], int q[ ], int n) {
        int i;
        for(i = 0; i < 4 * n; i++)
                p[i] = q[i];
}
```

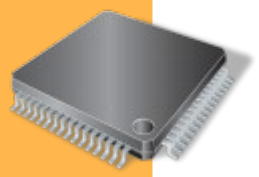
- cache (a)
  - A=1, block 16B
  - hit rate?
- cache (b)
  - A=2, block 16B, LRU
  - hit rate?





Overview

- Introduction
- Pipeline
- Cache memories
- **Conclusion**



Conclusion

- done in this course
  - transistor → microprocessor / computer
- improving IPC – more than 1 IPC
  - superscalar + out-of-order execution
  - VLIW (Very Long Instruction Word) – 1 word = n instruction
- improving branch delays – branch predictors
- even more caches – unified caches, multi-level caches
- even more parallelism → physics limit on transistor size
  - bigger chips → several execution cores embedded
  - embedded, laptop, desktop – 2-8 cores (multi-core)
  - computation servers – 64-256 cores (many-core)