EECE 2322: Fundamentals of Digital Design and Computer Organization

Lecture 15_1: Microarchitecture

Xiaolin Xu
Department of ECE
Northeastern University

Review: Exceptions

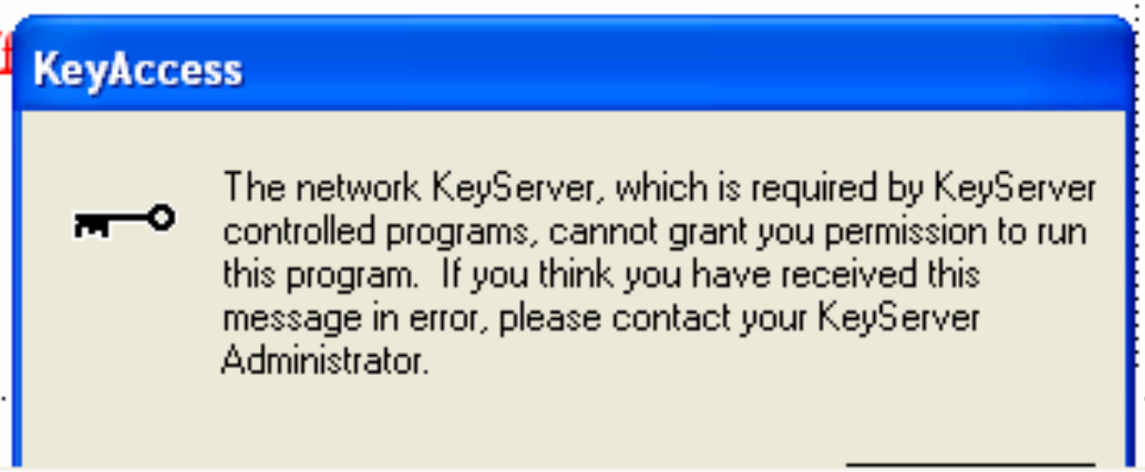
- Unscheduled function call to *exception handler*
- Caused by:
  - Hardware, also called an *interrupt*, e.g. keyboard
  - Software, also called *traps*, e.g. undefined instruction
- When exception occurs, the processor:
  - Records cause of exception (`Cause` register)
  - Jumps to exception handler (0x80000180)
  - Returns to program (`EPC` register)

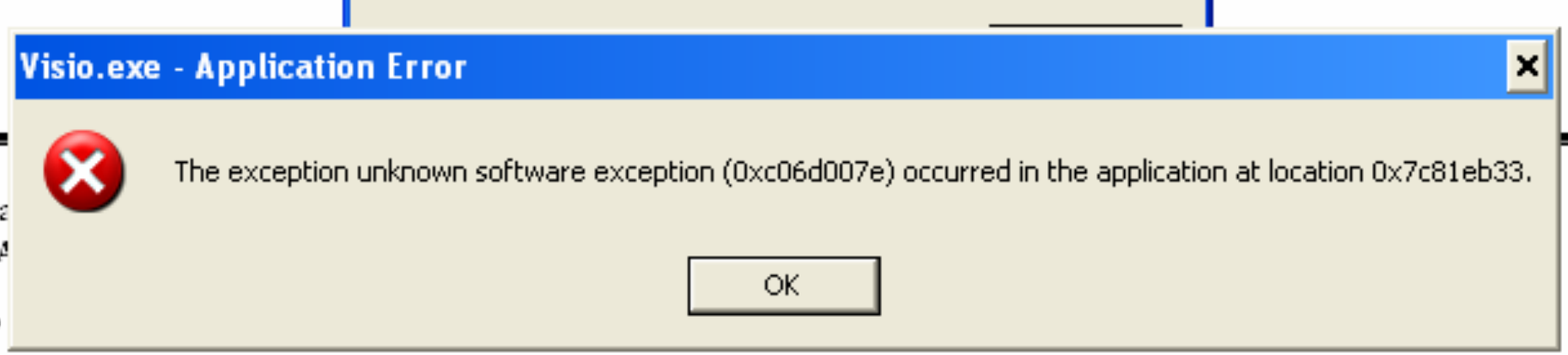
Example Exception

sequential circuits.¶

Can we design a spiff

Figure 2.1 shows a
inputs, A and B, and on
box indicates that it is
this case, the function is





Harris a
— 26 A

CHAP

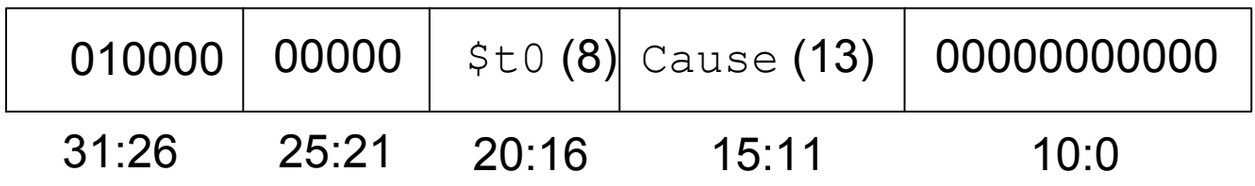
words, we say the output Y is a function of the two inputs A and B where the function performed is A OR B.¶

The *implementation* of the combinational circuit is independent of its functionality. Figure 2.1 and Figure 2.2 show two possible implementa-

Exception Registers

- Not part of register file
  - `Cause`
    - Records cause of exception
    - Coprocessor 0 register 13
  - `EPC` (Exception PC)
    - Records PC where exception occurred
    - Coprocessor 0 register 14
- Move from Coprocessor 0
  - `mfc0 $t0, Cause`
  - Moves contents of `Cause` into `$t0`

mfc0

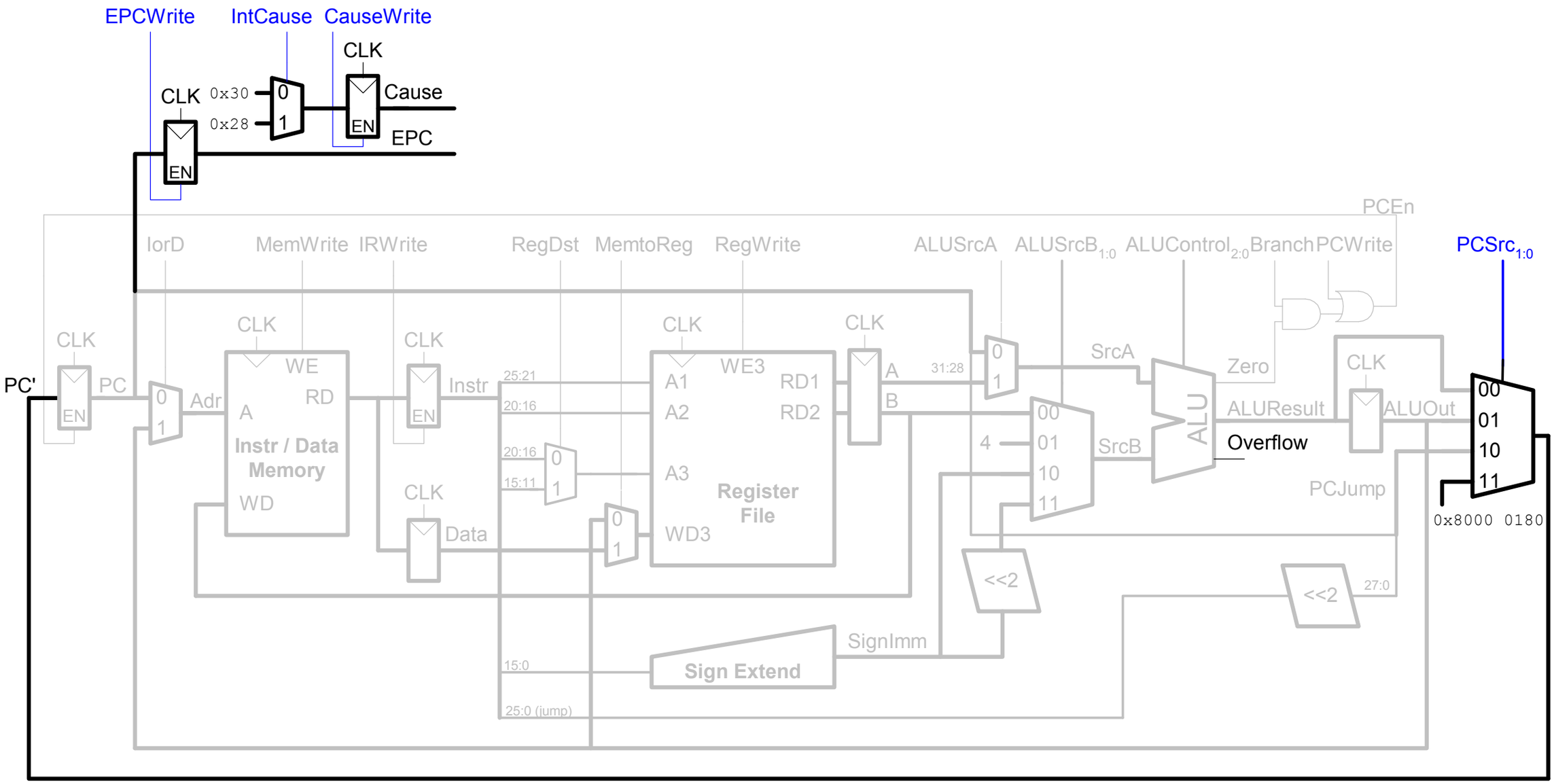


Exception Causes

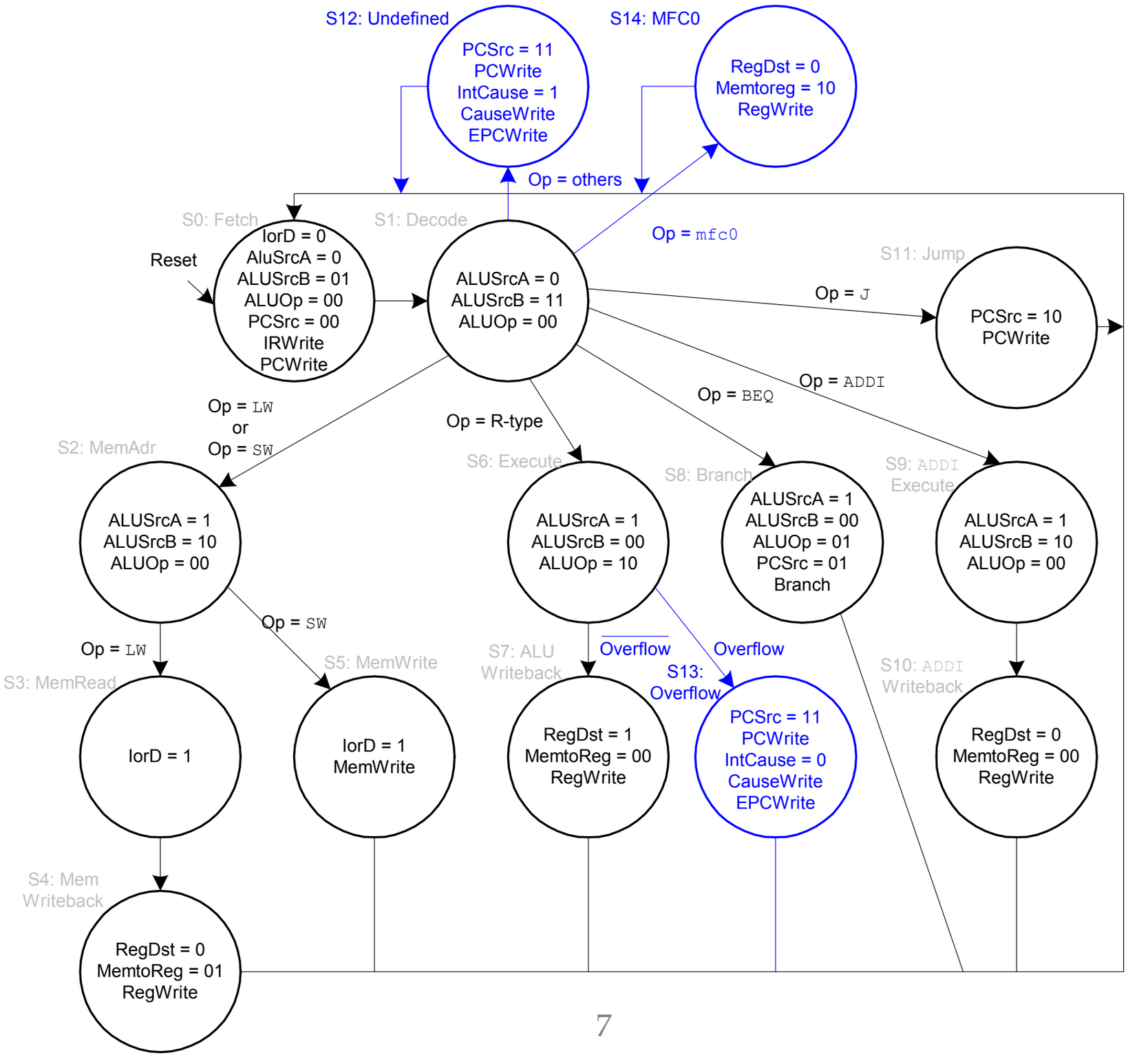
Exception	Cause
Hardware Interrupt	0x00000000
System Call	0x00000020
Breakpoint / Divide by 0	0x00000024
Undefined Instruction	0x00000028
Arithmetic Overflow	0x00000030

Extend multicycle MIPS processor to handle last two types of exceptions

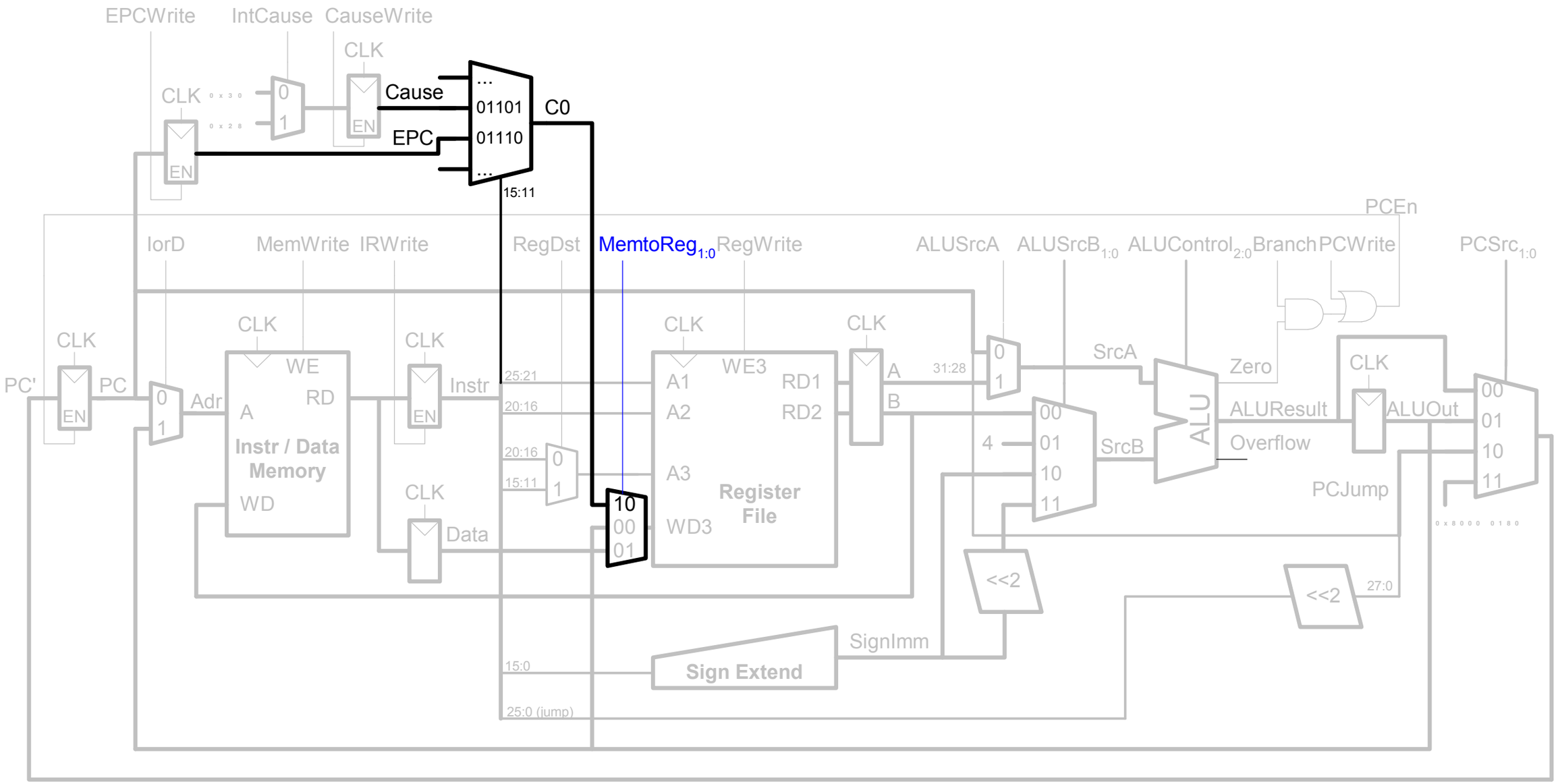
Exception Hardware: EPC & Cause



Control FSM with Exceptions



Exception Hardware: mfc0



Advanced Microarchitecture

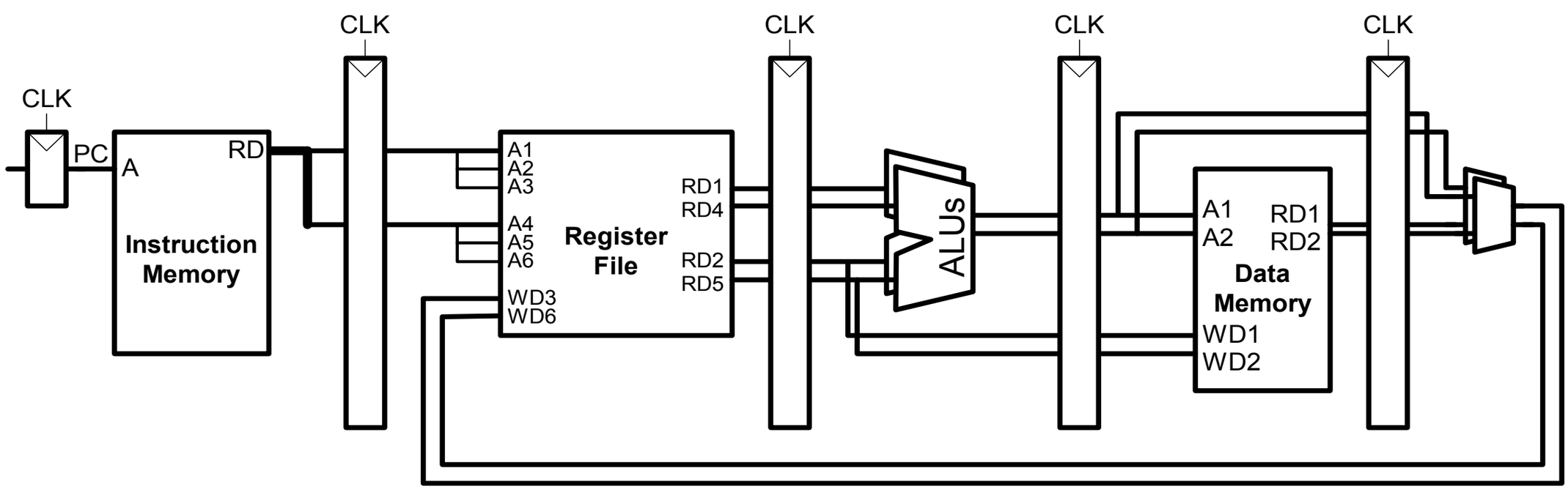
- Deep Pipelining
- Branch Prediction
- Superscalar Processors
- Out of Order Processors
- Register Renaming
- SIMD
- Multithreading
- Multiprocessors

Deep Pipelining

- 10-20 stages typical
- Number of stages limited by:
  - Pipeline hazards
  - Sequencing overhead
  - Power
  - Cost

Superscalar

- Multiple copies of datapath execute multiple instructions at once
- Dependencies make it tricky to issue multiple instructions at once

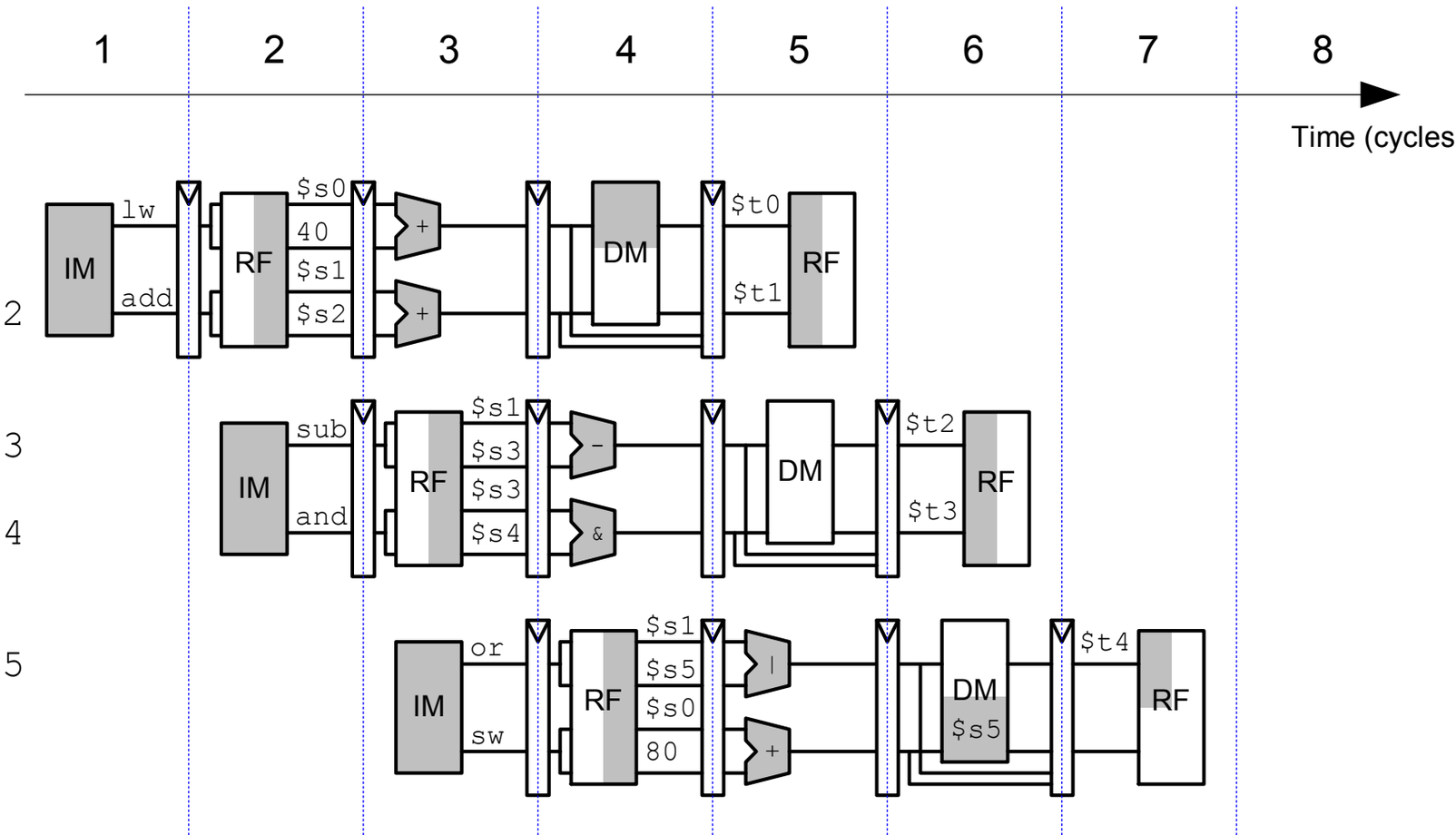


Superscalar Example

```
lw  $t0, 40($s0)
add $t1, $s1, $s2
sub $t2, $s1, $s3
and $t3, $s3, $s4
or  $t4, $s1, $s5
sw  $s5, 80($s0)
```

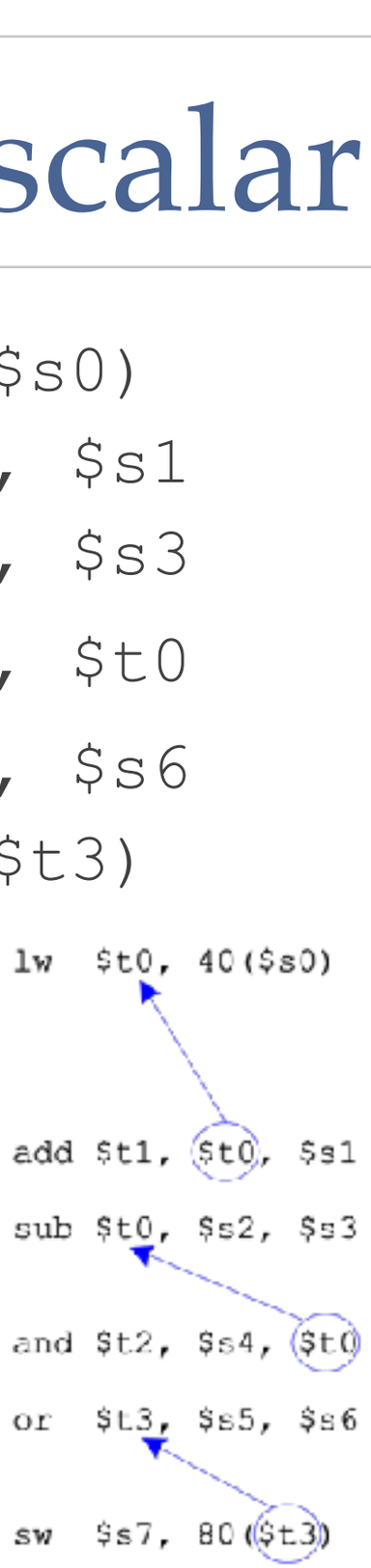
Ideal IPC (Instruction Per Cycle): 2

Practical IPC = ?



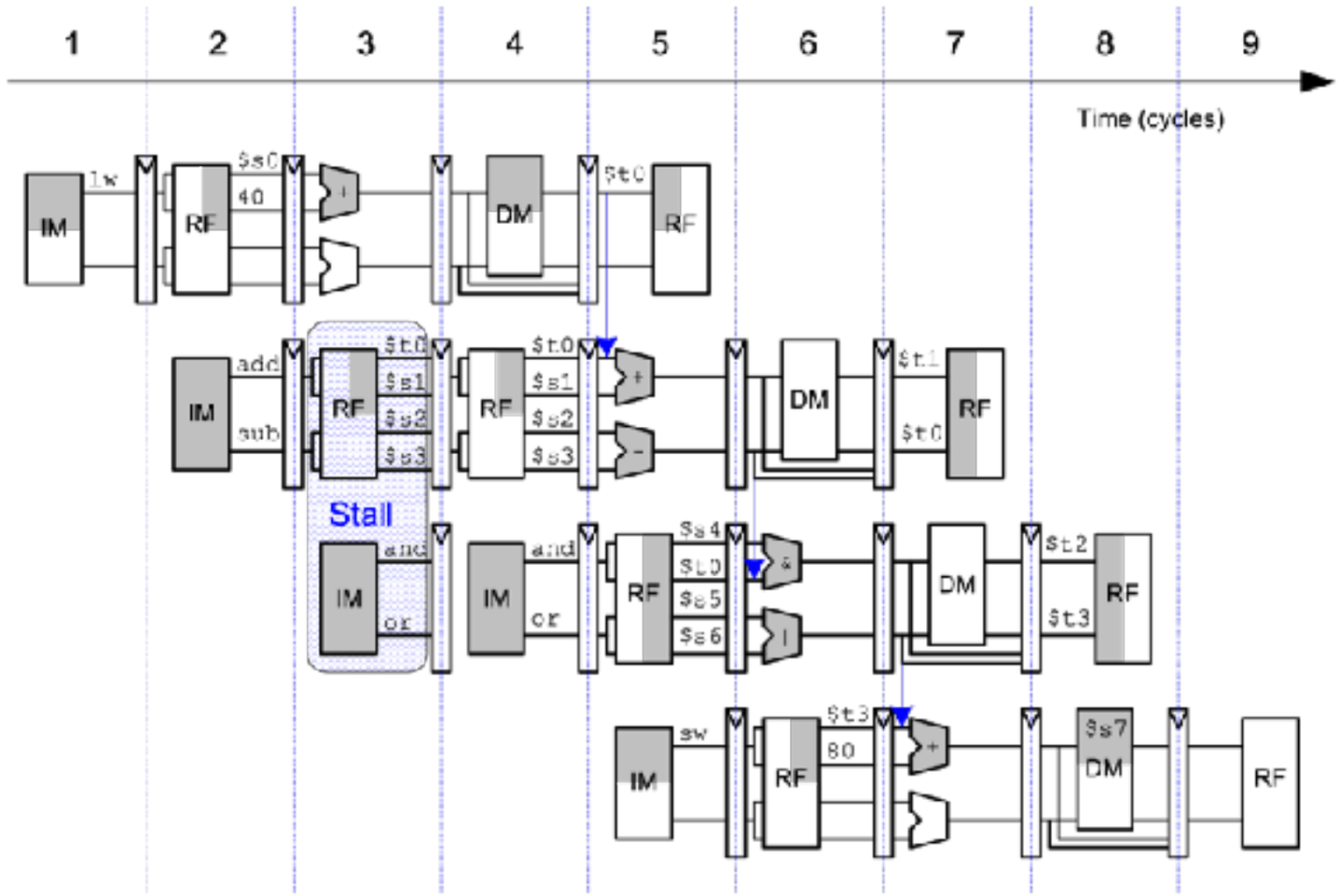
Superscalar with Dependencies

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```



Ideal IPC: 2

Actual IPC: 6/5 = 1.17



Out of Order Processor

- Looks ahead across multiple instructions
- Issue as many instructions as possible at once
- Issue instructions out of order (**as long as no dependencies**)
- **Dependencies:**
  - **RAW** (read after write): one instruction writes, later instruction reads a register, (e.g., lw instruction)
  - **WAR** (write after read): one instruction reads, later instruction writes a register
  - **WAW** (write after write): one instruction writes, later instruction writes a register

Out of Order Processor

- **Instruction level parallelism (ILP):** number of instruction that can be issued simultaneously (average < 3)
- **Scoreboard:** table that keeps track of:
  - Instructions waiting to issue
  - Available functional units
  - Dependencies

Out of Order Processor Example

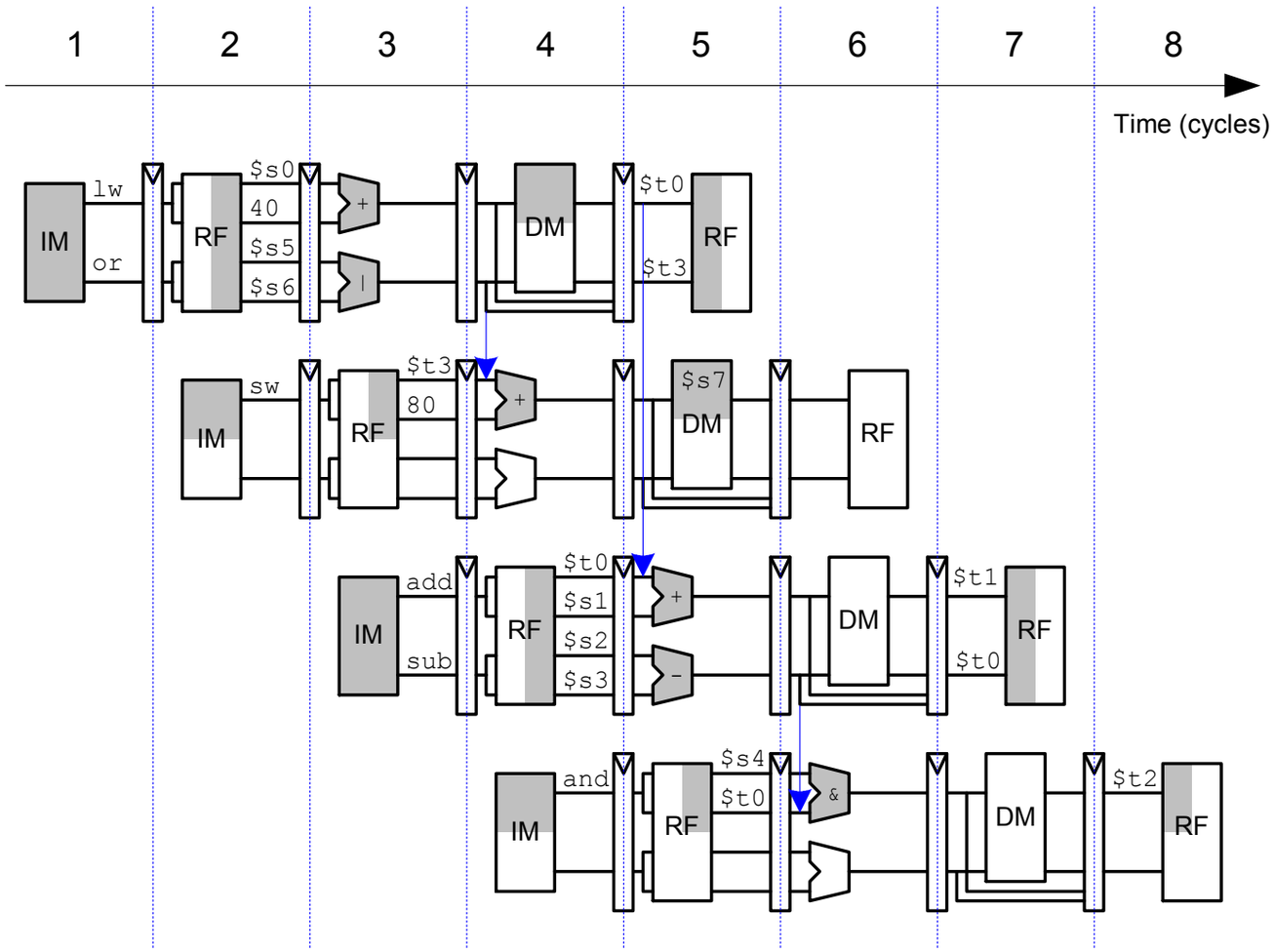
original code

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```

```
lw  $t0, 40($s0)
or  $t3, $s5, $s6
        RAW
sw  $s7, 80($t3)
two cycle latency
between load and   RAW
use of $t0
add $t1, $t0, $s1
        WAR
sub $t0, $s2, $s3
        RAW
and $t2, $s4, $t0
```

Ideal IPC: 2

Actual IPC: 6/4 = 1.5



Out of Order Processor Example

original code

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```

```
lw  $t0, 40($s0)
or  $t3, $s5, $s6
sw  $s7, 80($t3)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
```

RAW

two cycle latency between load and use of $t0

RAW

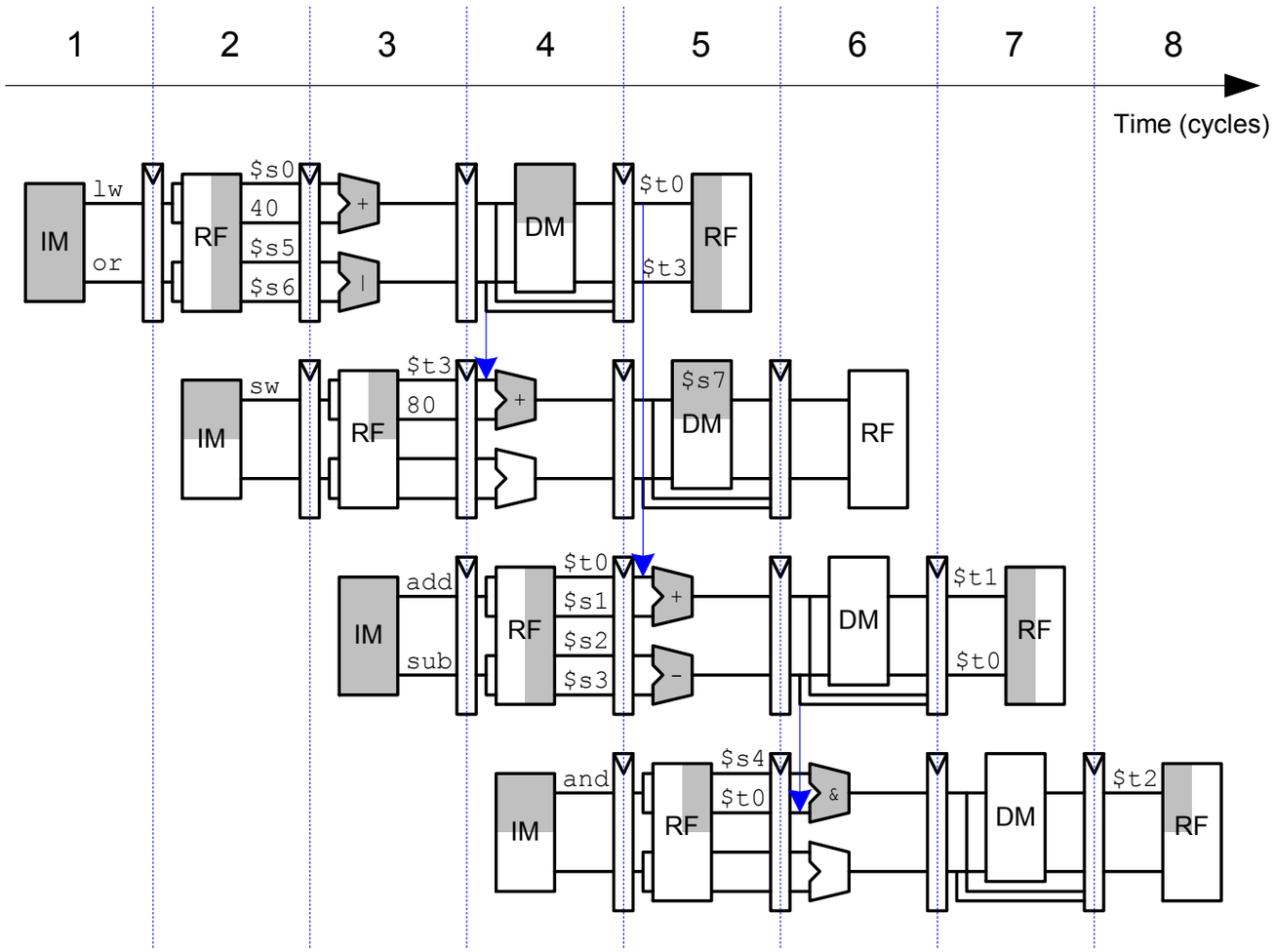
WAR

RAW

easier to solve

Ideal IPC: 2

Actual IPC: 6/4 = 1.5

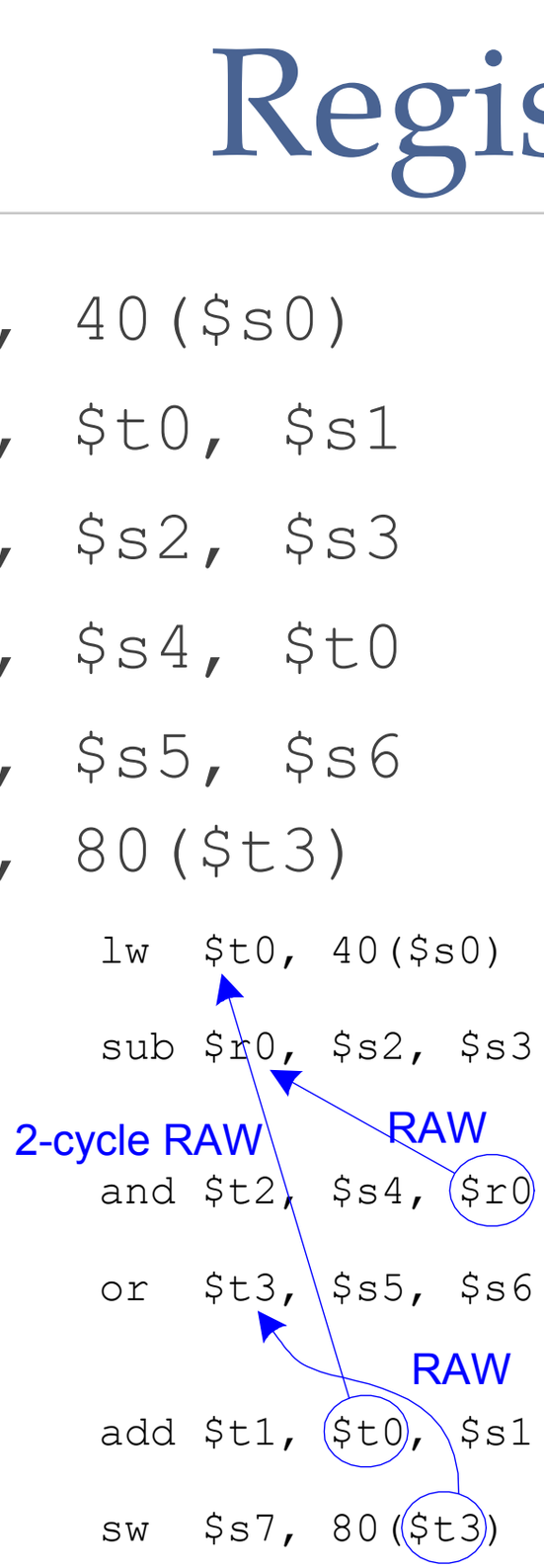


Register Renaming

- MIPS processor might add 20 renaming registers, called $r0-$r19
- The programmer cannot use these registers **directly**, because they are not part of the architecture
- However, the processor is free to use them to eliminate hazards

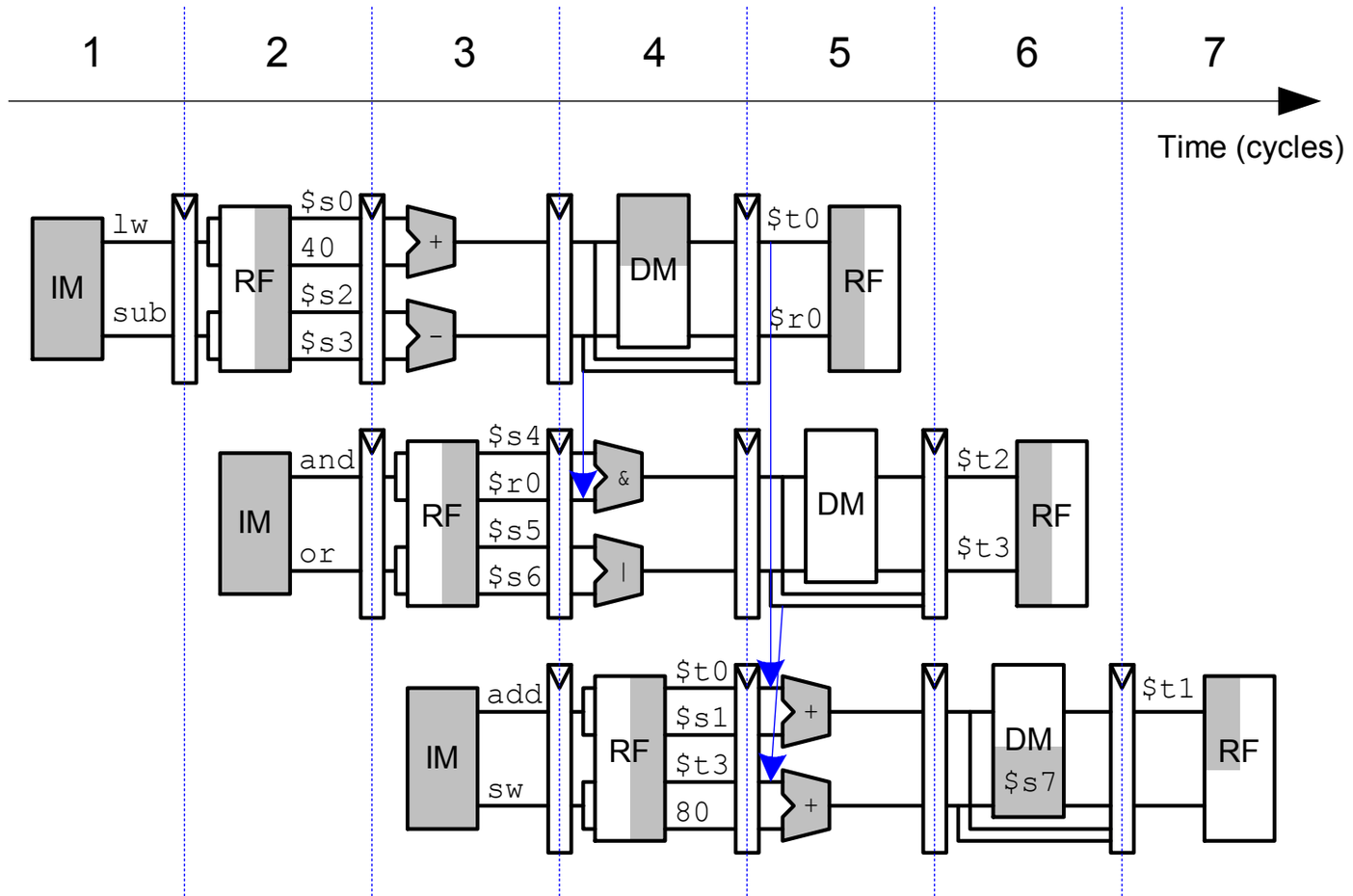
Register Renaming

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```



Ideal IPC: 2

Actual IPC: 6/3 = 2



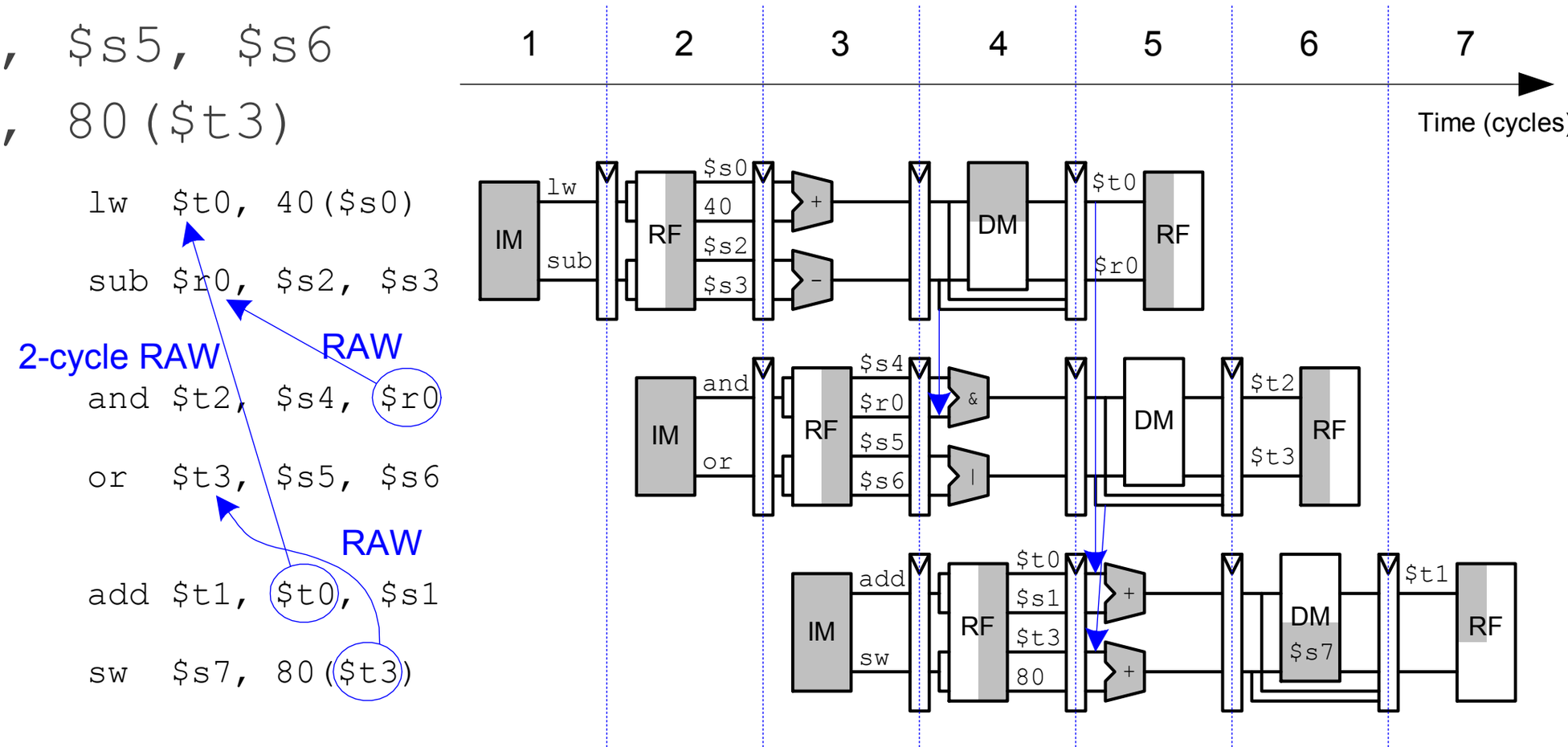
Register Renaming

```
lw   $t0, 40($s0)
add  $t1, $t0, $s1
sub  $t0, $s2, $s3
and  $t2, $s4, $t0
or   $t3, $s5, $s6
sw   $s7, 80($t3)
```

r: renaming registers

Ideal IPC: 2

Actual IPC: 6/3 = 2



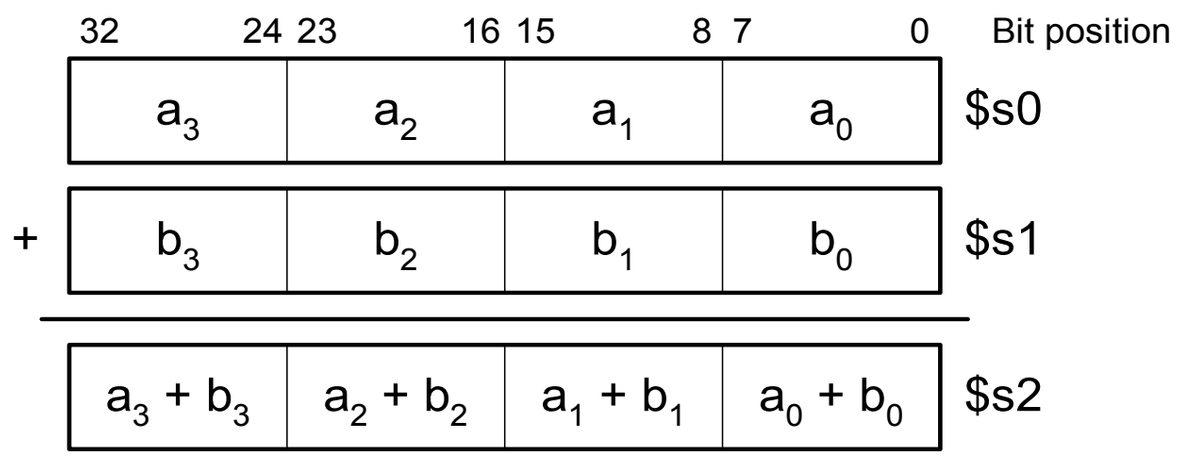
Some Advanced Concepts

SIMD

- Single Instruction Multiple Data (SIMD)
  - Single instruction acts on multiple pieces of data at once
  - Common application: graphics (8-bit RGB)
  - Perform short arithmetic operations (also called *packed arithmetic*)

- For example, add four 8-bit elements

```
padd8 $s2, $s0, $s1
```



Advanced Architecture Techniques

- **Multithreading**
  - Wordprocessor: thread for typing, spell checking, printing
  - More than one copy of the architectural state
- **Multiprocessors**
  - Multiple processors (cores) on a single chip

Threading: Definitions

- **Process:** program running on a computer
  - Multiple processes can run at once: e.g., surfing Web, playing music, writing a paper
- **Thread:** part of a program
  - Each process has multiple threads: e.g., a word processor may have threads for typing, spell checking, printing

Threads in Conventional Processor

- One thread runs at once
- When one thread stalls (for example, waiting for memory):
  - Architectural state of that thread stored
  - Architectural state of waiting thread loaded into processor and it runs
  - Called **context switching**
- Appears to user like all threads running simultaneously

Multithreading

- Multiple copies of architectural state
- Multiple threads **active** at once:
  - When one thread stalls, another runs immediately
  - If one thread can't keep all execution units busy, another thread can use them
- Does not increase instruction-level parallelism (ILP) of single thread, but increases throughput

Intel calls this "hyperthreading"

Multiprocessors

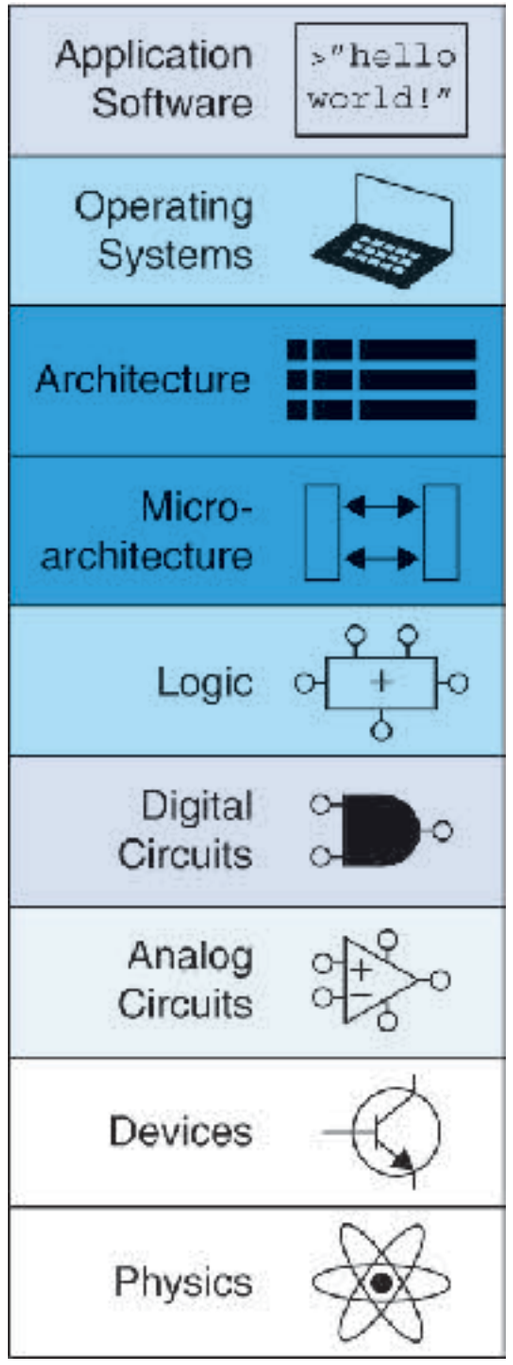
- Multiple processors (cores) with a method of communication between them
- Types:
  - **Homogeneous:** multiple cores with shared memory
  - **Heterogeneous:** separate cores for different tasks (for example, DSP and CPU in cell phone)
  - **Clusters:** each core has own memory system

Other Resources

- Patterson & Hennessy's: *Computer Architecture: A Quantitative Approach*
- Conferences:
  - www.cs.wisc.edu/~arch/www/
  - ISCA (International Symposium on Computer Architecture)
  - HPCA (International Symposium on High Performance Computer Architecture)

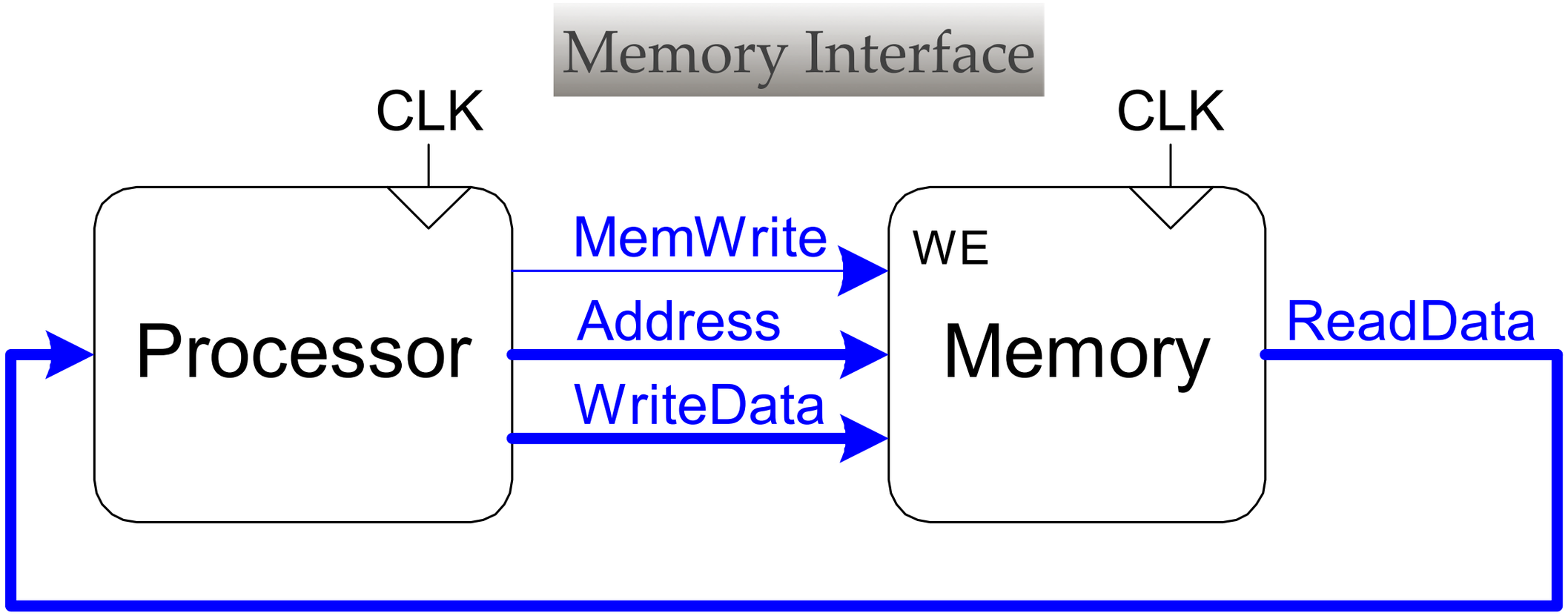
Other Related Topics

- Memory System Performance Analysis
- Caches
- Virtual Memory
- Memory-Mapped I/O



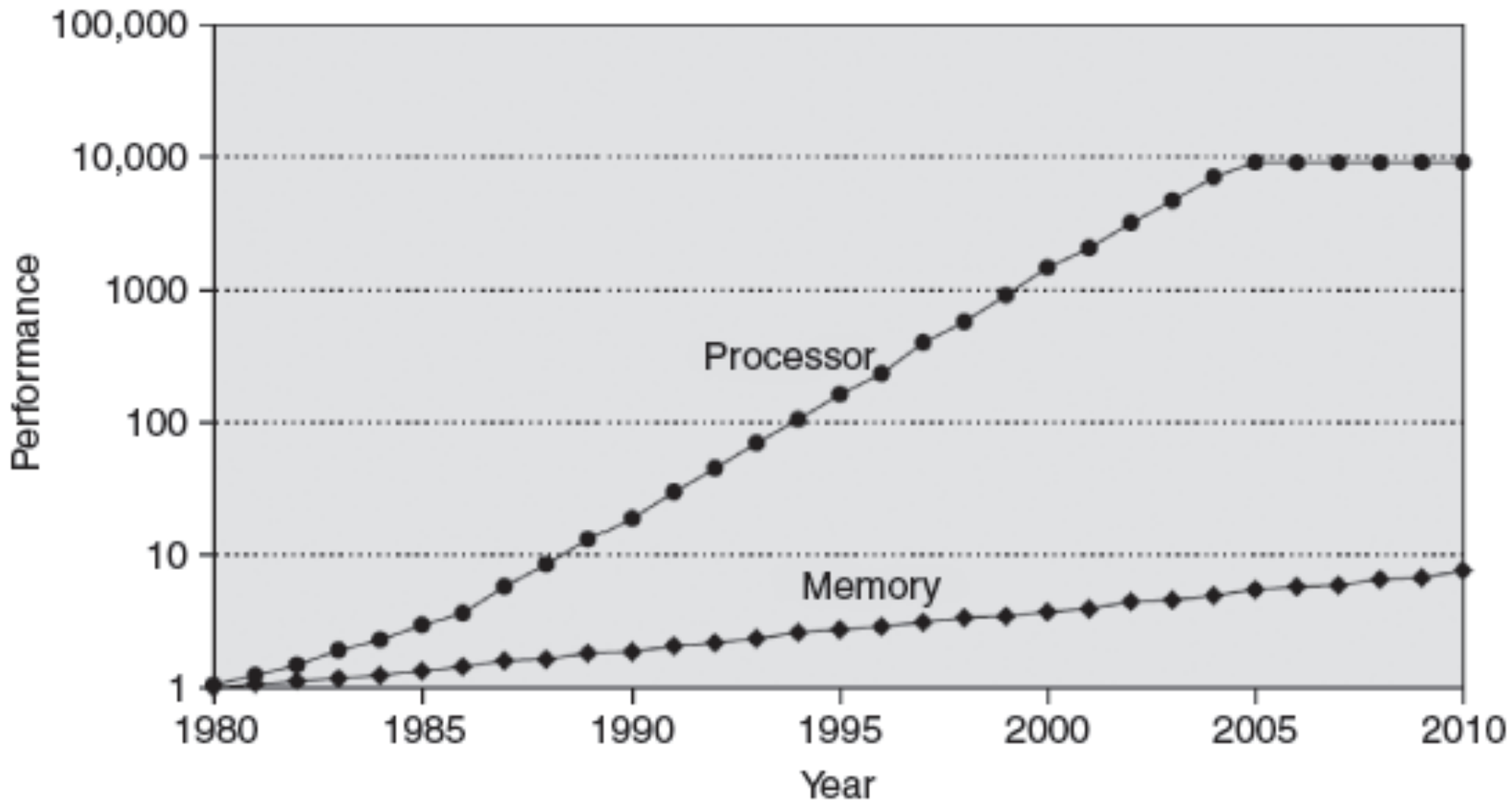
Memory as an Important Component

- Computer performance depends on:
  - Processor performance
  - Memory system performance



Processor-Memory Gap

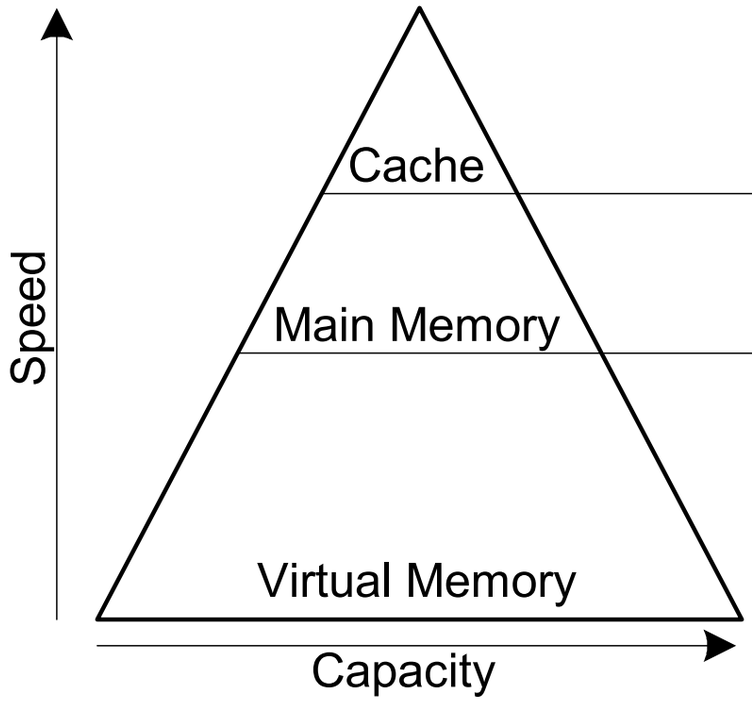
In prior chapters, assumed access memory in 1 clock cycle – but hasn't been true since the 1980's



Memory System Challenge

- Make memory system appear as fast as processor
  - Use hierarchy of memories
- Ideal memory:
  - Fast
  - Cheap (inexpensive)
  - Large (capacity)
- **But one can only has two!**

Memory Hierarchy



Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	$10,000	1	25+
DRAM	$10	10 - 50	10
SSD	$1	100,000	0.5
HDD	$0.1	10,000,000	0.1

Locality

Exploit locality to make memory accesses fast

- **Temporal Locality:**
  - Locality in time
  - If data used recently, likely to use it again soon
  - **How to exploit:** keep recently accessed data in higher levels of memory hierarchy
- **Spatial Locality:**
  - Locality in space
  - If data used recently, likely to use nearby data soon
  - **How to exploit:** when access data, bring nearby data into higher levels of memory hierarchy too

Memory Performance

Hit: data found in that level of memory hierarchy

Miss: data not found (must go to next level)

Hit Rate = # hits / # memory accesses
= 1 – Miss Rate

Miss Rate = # misses / # memory accesses
= 1 – Hit Rate

Average memory access time (AMAT): average time for processor to access data

AMAT = $t_{\text{cache}} + MR_{\text{cache}}[t_{MM} + MR_{MM}(t_{VM})]$

Memory Performance Example 1

- A program has 2,000 loads and stores 1,250 of these data values in cache
- Rest supplied by other levels of memory hierarchy
- What are the hit and miss rates for the cache?
- **Hit Rate = 1250/2000 = 0.625**
- **Miss Rate = 750/2000 = 0.375 = 1 – Hit Rate**

Memory Performance Example 2

Suppose processor has 2 levels of hierarchy: cache and main memory

t_{cache} = 1 cycle, t_{MM} = 100 cycles

What is the AMAT of the program from Example 1?

Memory Performance Example 2

Suppose processor has 2 levels of hierarchy: cache and main memory

t_{cache} = 1 cycle, t_{MM} = 100 cycles

What is the AMAT of the program from Example 1?

$$\begin{aligned} \text{AMAT} &= t_{\text{cache}} + MR_{\text{cache}}(t_{MM}) \\ &= [1 + 0.375(100)] \text{ cycles} \\ &= \mathbf{38.5 \text{ cycles}} \end{aligned}$$

Cache

- Highest level in memory hierarchy
- Fast (typically ~ 1 cycle access time)
- Ideally supplies most data to processor
- Usually holds most recently accessed data

Cache Design Questions

- What data is held in the cache?
- How is data found?
- What data is replaced?

Focus on data loads, but stores follow same principles

What data is held in the cache?

- Ideally, cache anticipates needed data and puts it in cache
- But **impossible to predict future**
- Use past to predict future – temporal and spatial locality:
  - **Temporal locality:** copy newly accessed data into cache
  - **Spatial locality:** copy neighboring data into cache too

Cache Terminology

- **Capacity (*C*):**
  - number of data bytes in cache
- **Block size (*b*):**
  - bytes of data brought into cache at once
- **Number of blocks (*B = C/b*):**
  - number of blocks in cache: $B = C / b$
- **Degree of associativity (*N*):**
  - number of blocks in a set
- **Number of sets (*S = B/N*):**
  - each memory address maps to exactly one cache set

How is data found?

- Cache organized into *S* sets
- Each memory address maps to exactly one set
- Caches categorized by # of blocks in a set:
  - **Direct mapped:** 1 block per set
  - ***N*-way set associative:** *N* blocks per set
  - **Fully associative:** all cache blocks in 1 set
- Examine each organization for a cache with:
  - Capacity ($C = 8$ words)
  - Block size ($b = 1$ word)
  - So, number of blocks ($B = 8$)

Example Cache Parameters

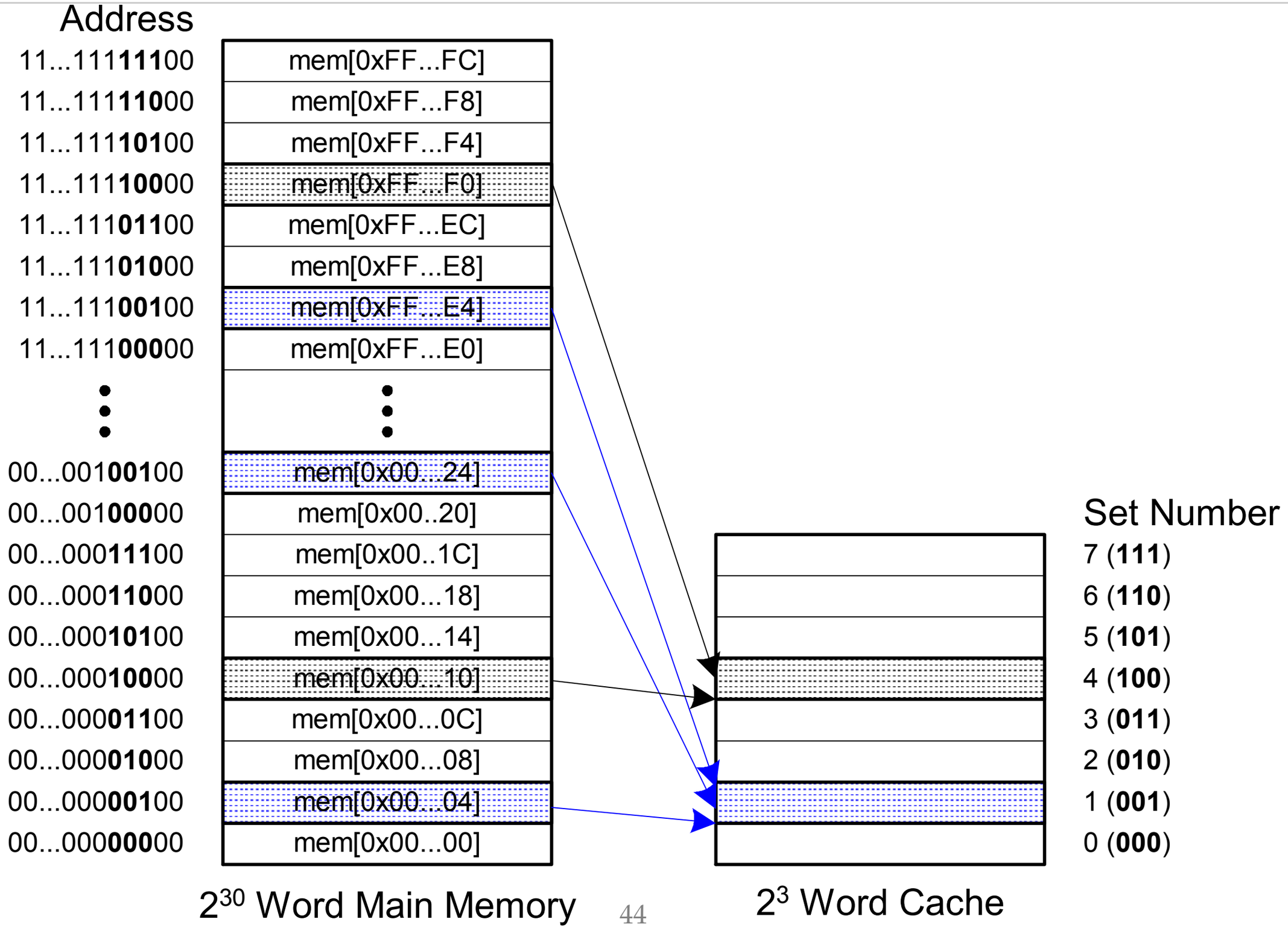
- ***C* = 8** words (capacity)
- ***b* = 1** word (block size)
- So, ***B* = 8** (# of blocks)

Ridiculously small, but will illustrate organizations

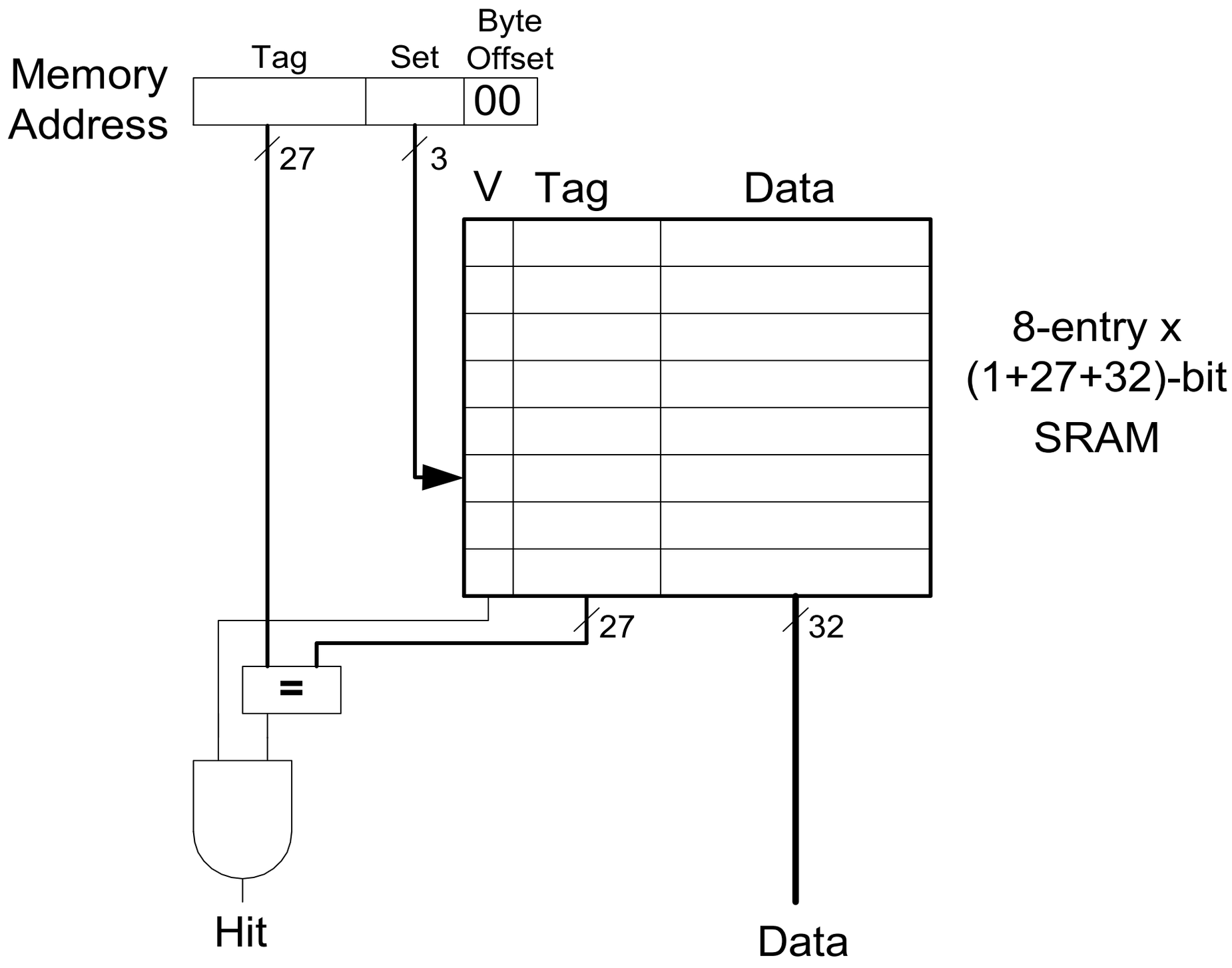
Mapping

- Mapping: the relationship **between the address of data in main memory and the location of that data in the cache**
- Each memory address maps to exactly one set in the cache
  - Some of the address bits are used to determine which cache set contains the data

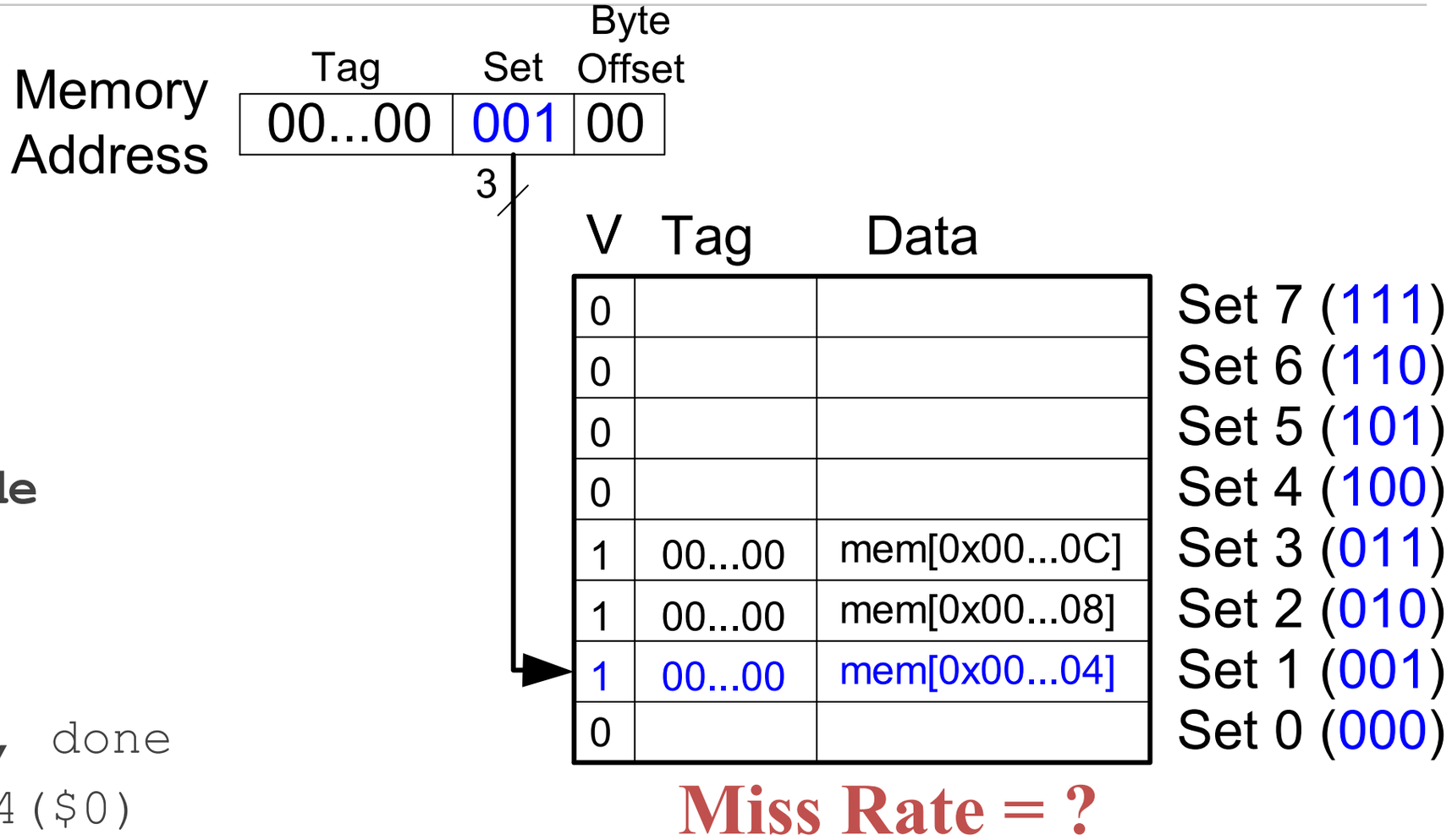
Direct Mapped Cache



Direct Mapped Cache Hardware



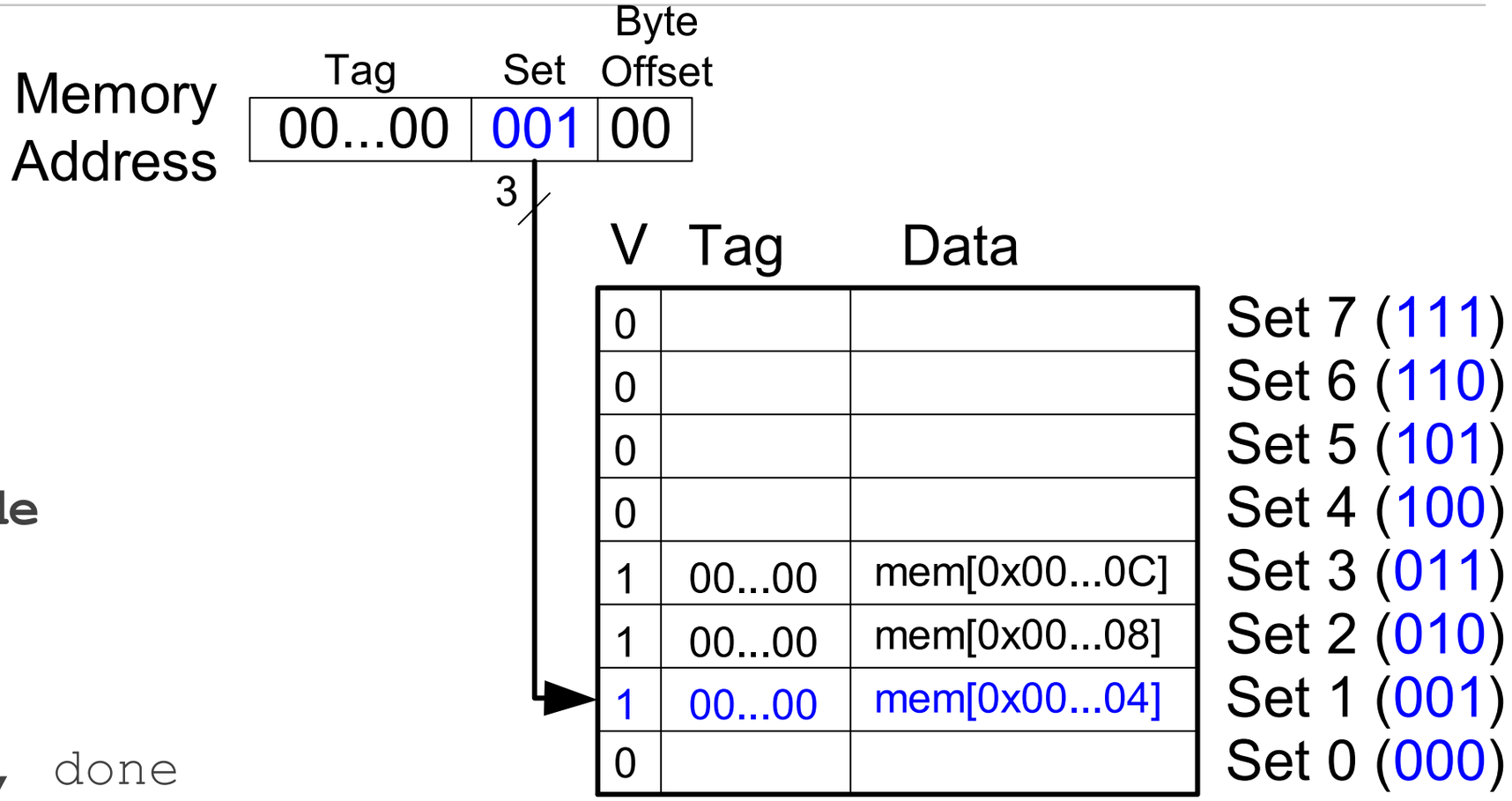
Direct Mapped Cache Performance



Miss Rate = ?

```
# MIPS assembly code


    addi $t0, $0, 5
loop: beq  $t0, $0, done
      lw   $t1, 0x4($0)
      lw   $t2, 0xC($0)
      lw   $t3, 0x8($0)
      addi $t0, $t0, -1
      j    loop
done:
```

Direct Mapped Cache Performance



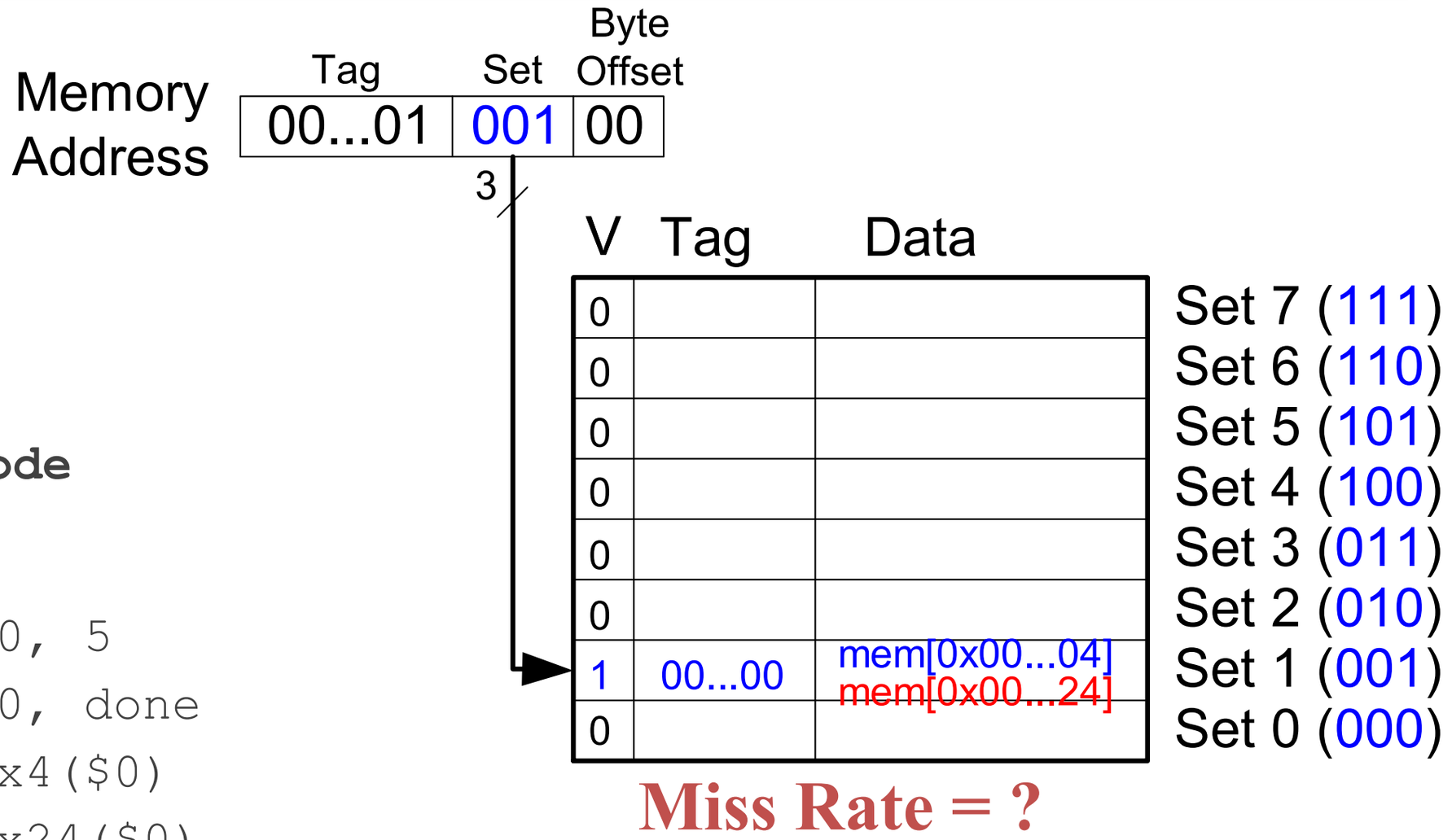

```
# MIPS assembly code


    addi $t0, $0, 5
loop: beq  $t0, $0, done
      lw   $t1, 0x4($0)
      lw   $t2, 0xC($0)
      lw   $t3, 0x8($0)
      addi $t0, $t0, -1
      j    loop
done:
```

Miss Rate = 3/15
= 20%

Temporal Locality
Compulsory Misses

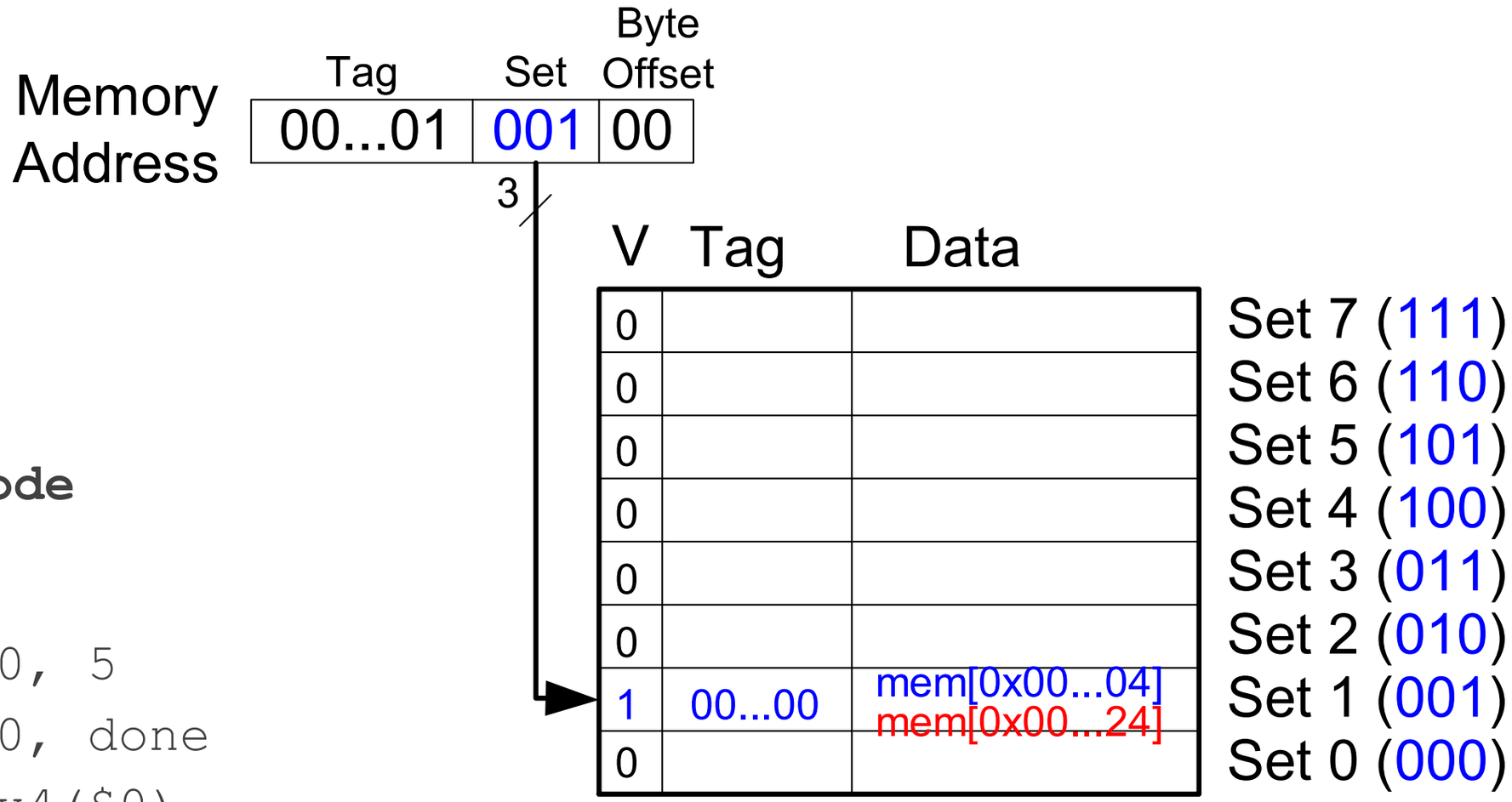
Direct Mapped Cache: Conflict

```
# MIPS assembly code


        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```



Miss Rate = ?

Direct Mapped Cache: Conflict



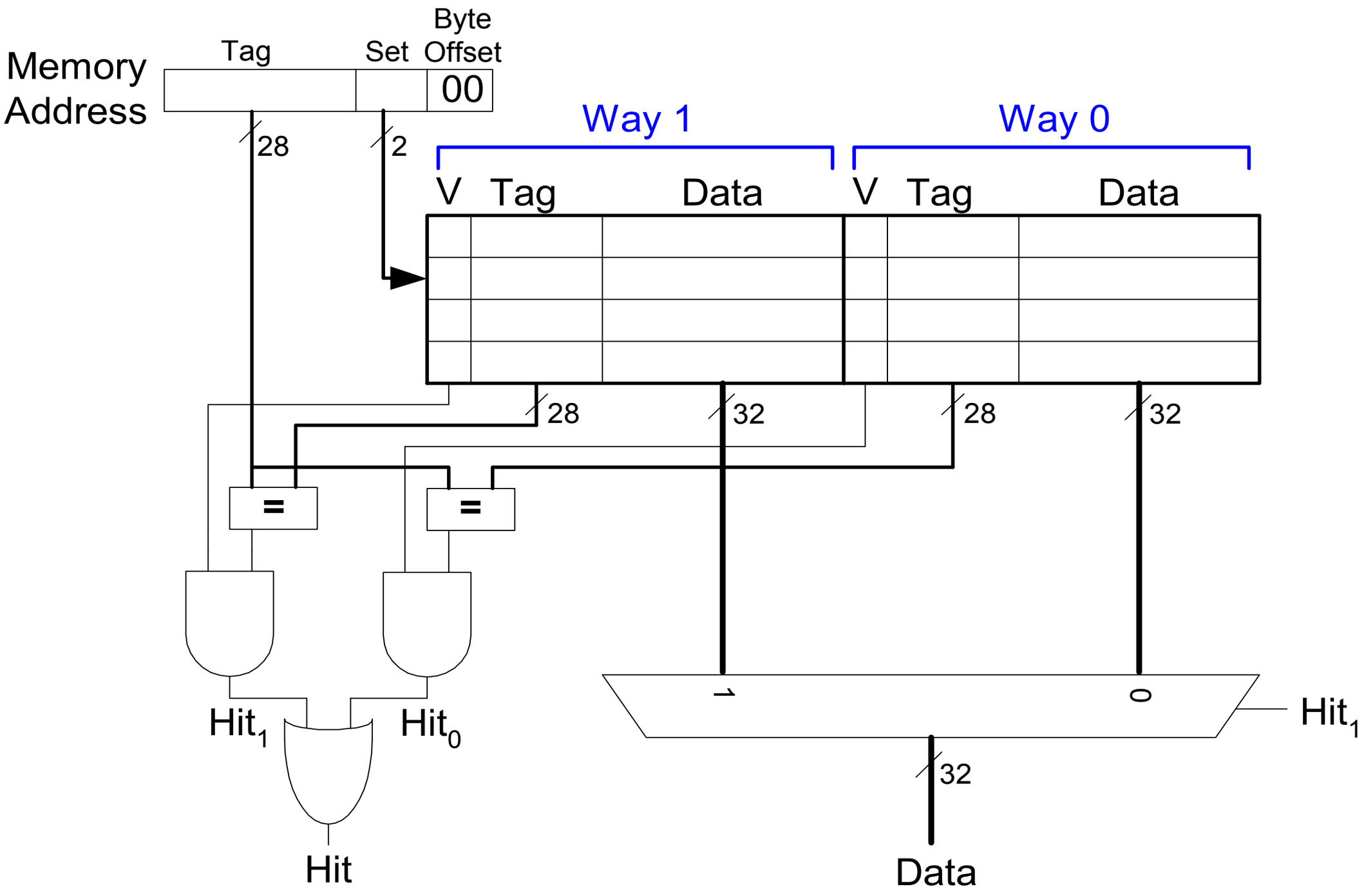

```
# MIPS assembly code


        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = 10/10

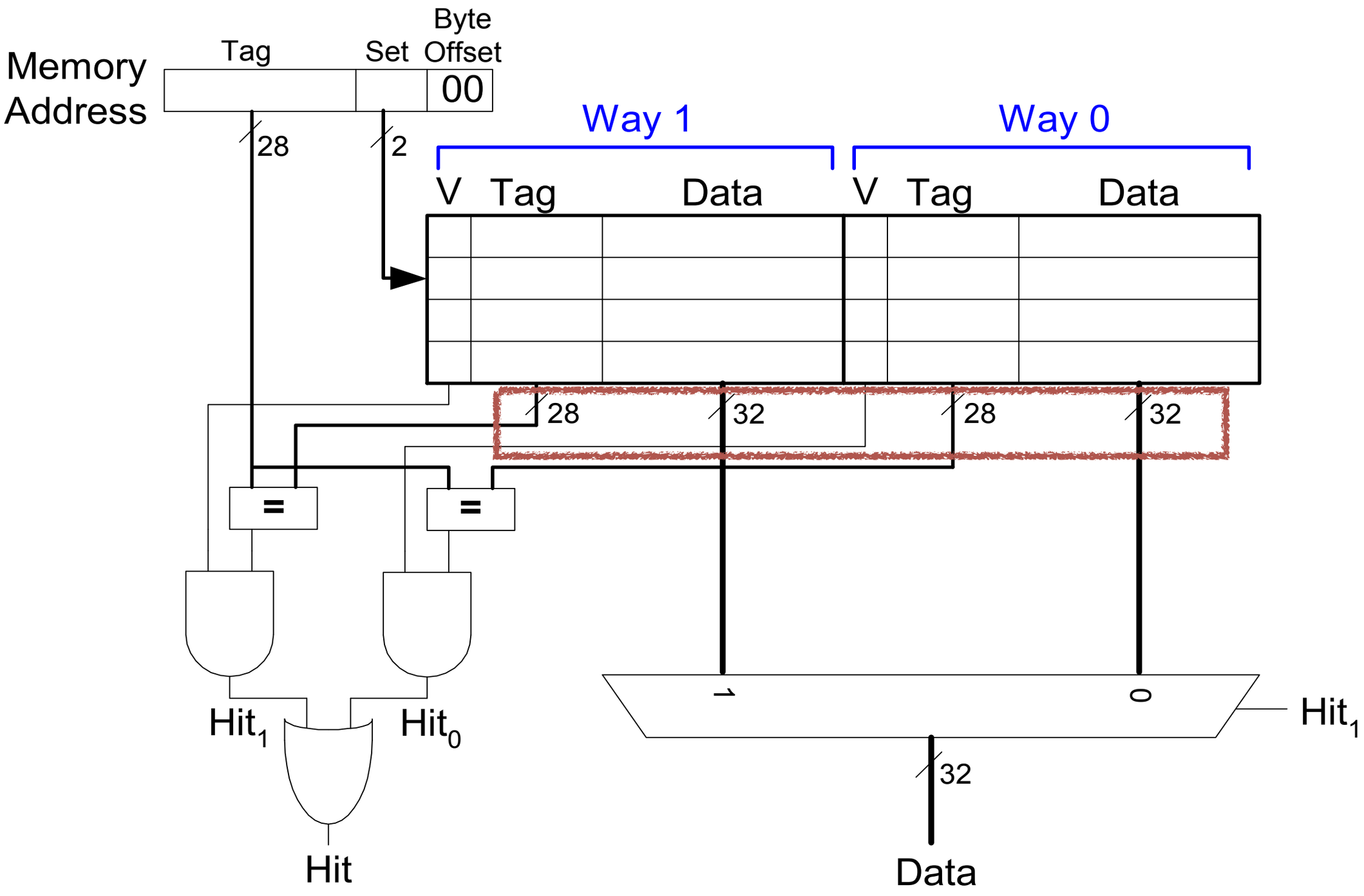
= 100%

Conflict Misses

N-Way Set Associative Cache



N-Way Set Associative Cache

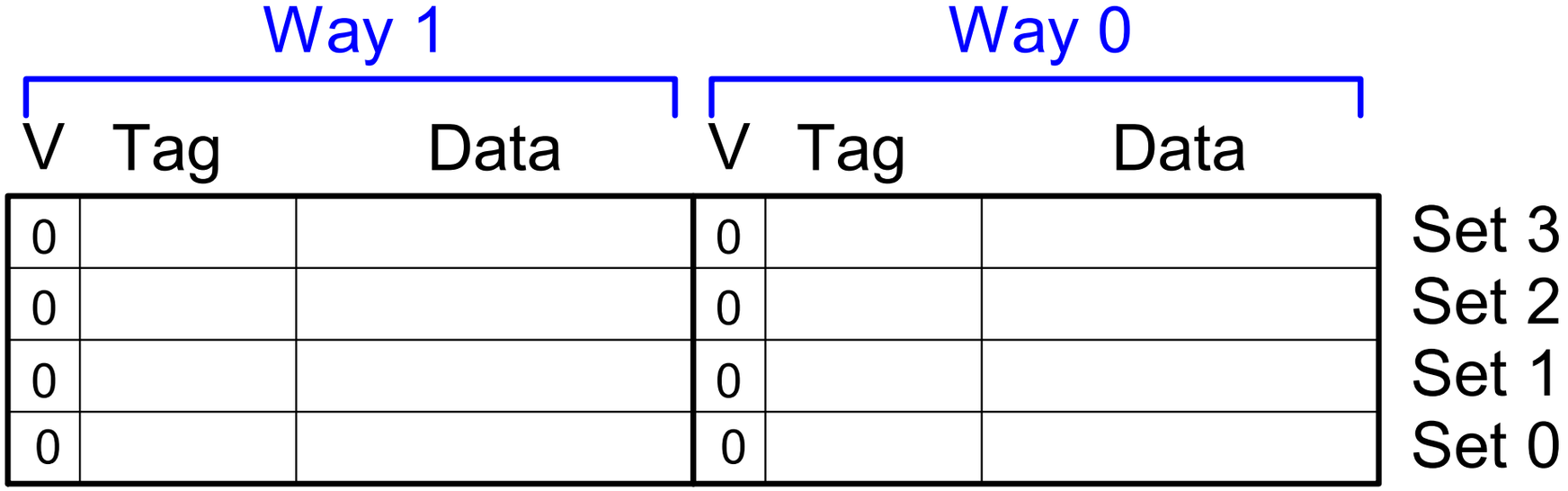


N-Way Set Associative Performance

```
# MIPS assembly code
```

Miss Rate = ?

```
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```



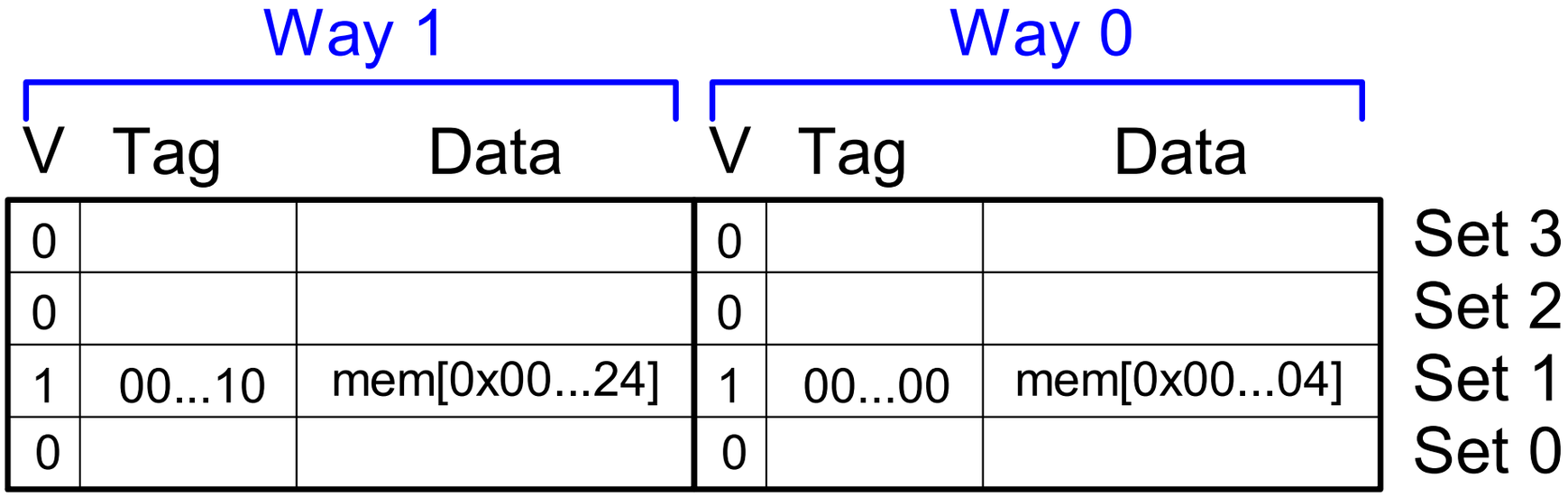
N-Way Set Associative Performance

```
# MIPS assembly code

        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

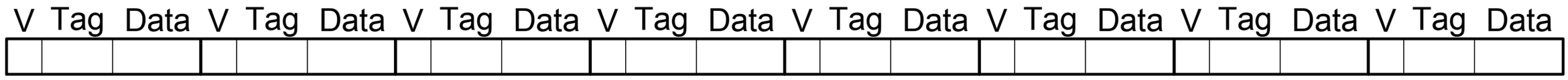
Miss Rate = 2/10

= 20%

Associativity reduces conflict misses



Fully Associative Cache



Reduces conflict misses

Expensive to build