Exceptions in a Pipeline

- Another form of control hazard
- Given the instruction sequence

40_{hex} sub $11, $2, $4

44_{hex} and $12, $2, $5

48_{hex} or $13, $2, $6

$4C_{hex}$ **add $1, $2, $1**

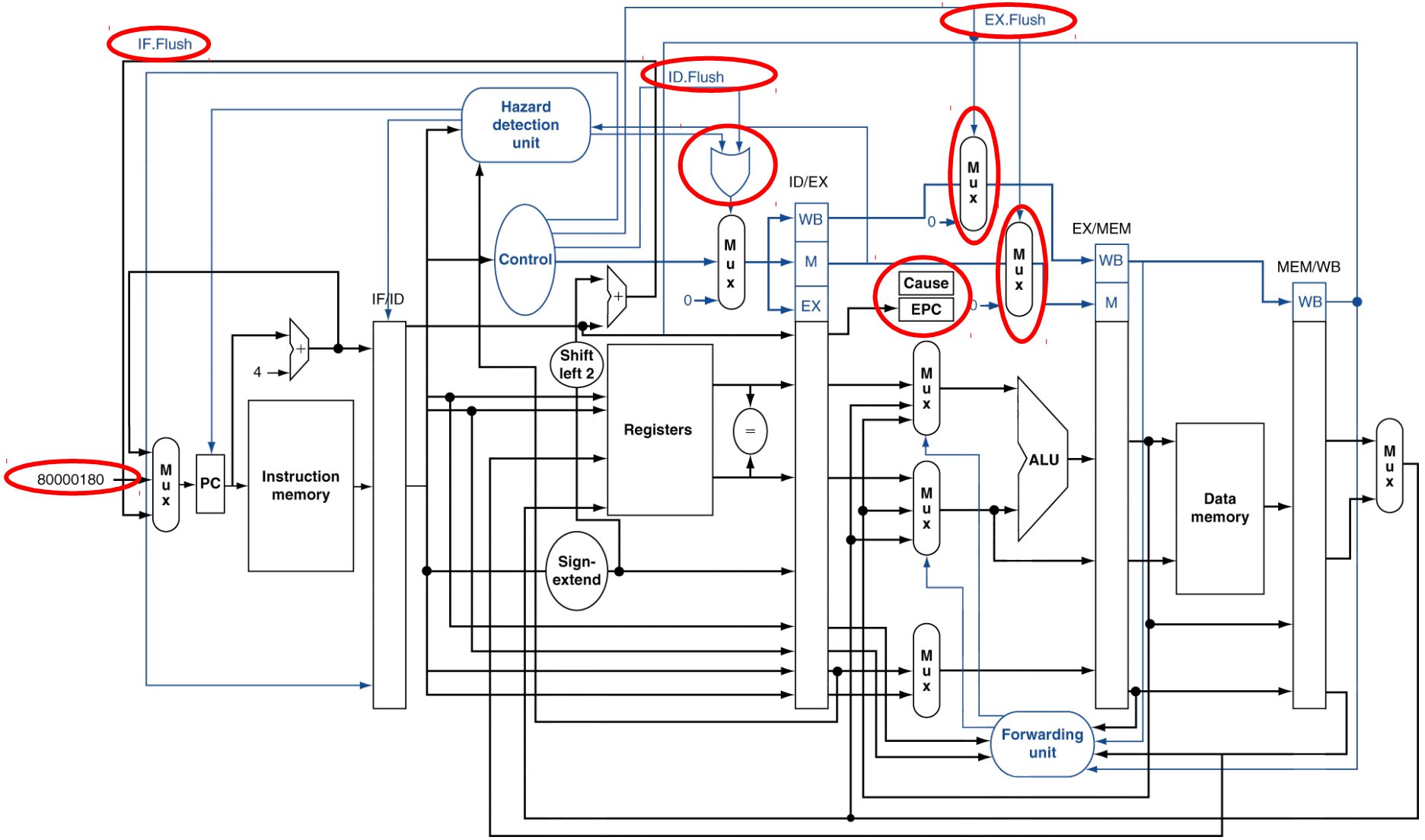
50_{hex} slt $15, $6, $7

54_{hex} lw $16, 50($7)

- Flush **EXEC**, ID, and IF stages
- Set Cause and EPC register values
- Transfer control to handler

Triggers an overflow

Pipeline with Exceptions



the ALU overflow signal is an input to the control unit.

Exception Example

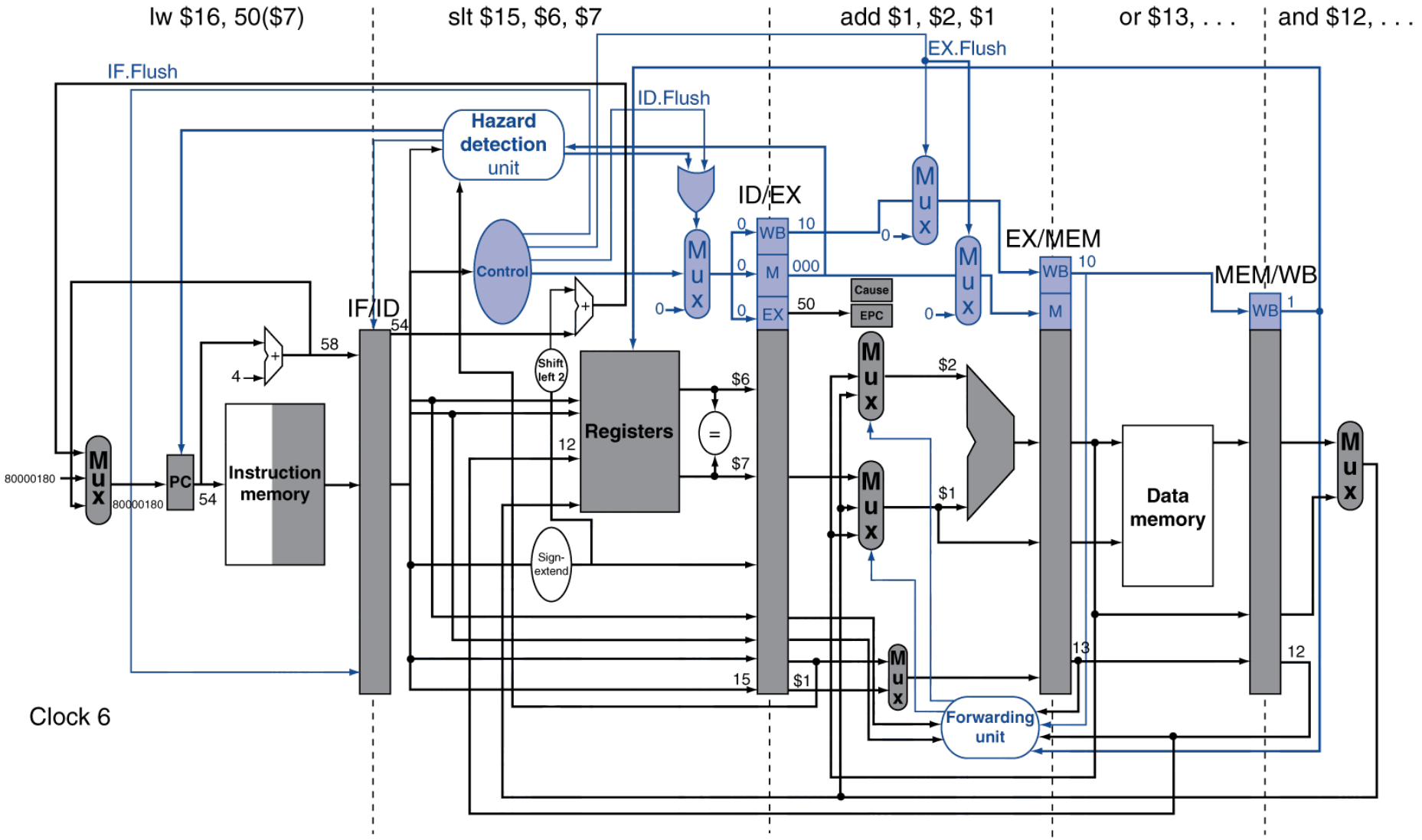
- Exception on add in

```
40    sub  $11, $2, $4
44    and  $12, $2, $5
48    or   $13, $2, $6
4C    add  $1,  $2, $1
50    slt  $15, $6, $7
54    lw   $16, 50($7)
...
```

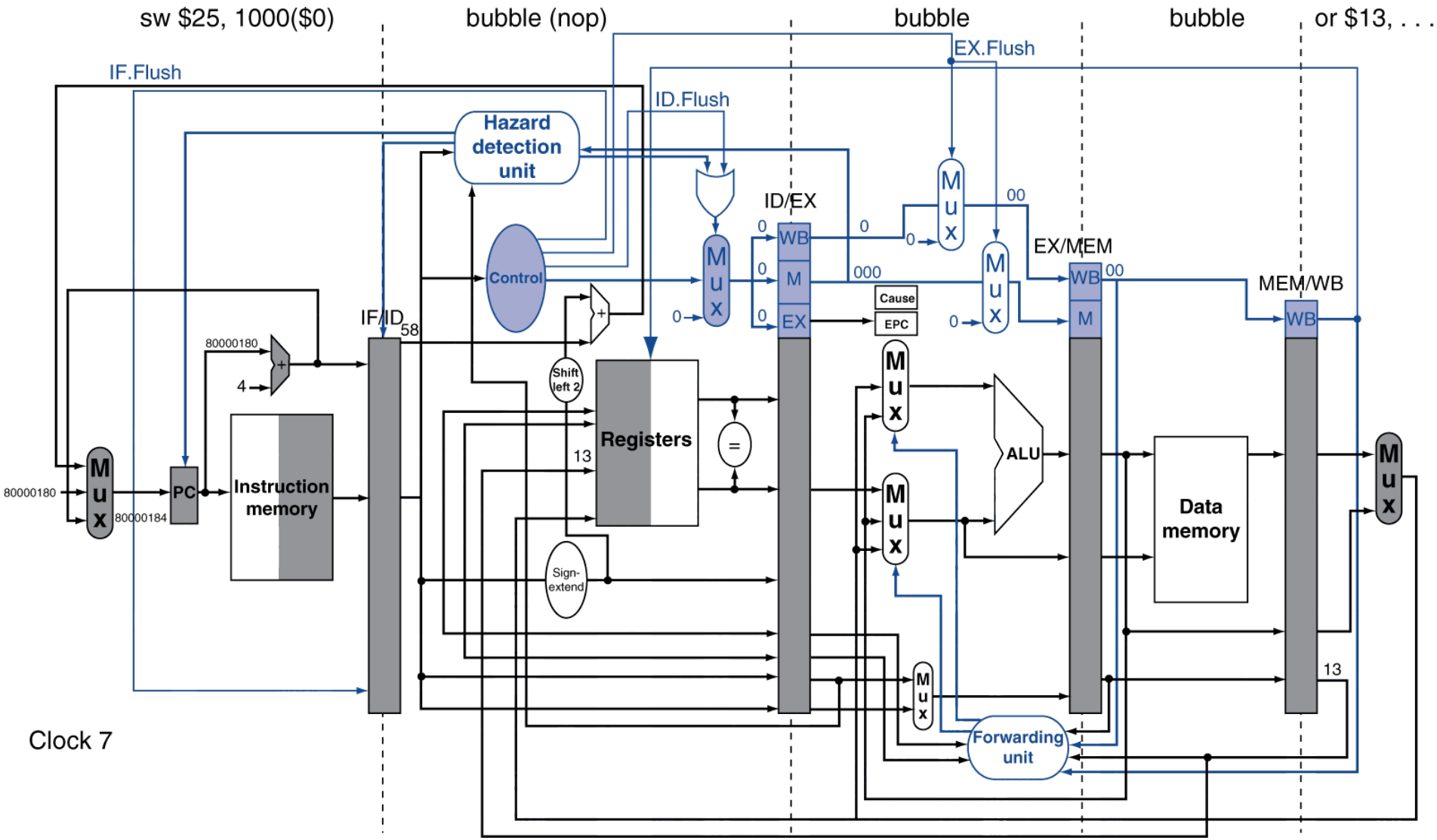
- Handler

```
80000180    sw   $25, 1000($0)
80000184    sw   $26, 1004($0)
...
```

Exception Example



Exception Example



Computer Memory

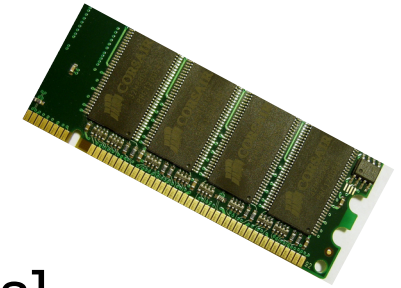
Dr. Ahmed Zahran

WGB 182

a.zahran@cs.ucc.ie

Memory Technology (1/2)

- **SRAM** uses more transistors per bit and keeps the data as long as it is powered
- **DRAM** keeps the charge for few milli-sec → needs periodic refresh (Dynamic!)
  - Organised in banks that can be accessed simultaneously
  - DDR (Double Data Rate) [Transfer on rising and falling clock edges]
- **Flash memory** [EEPROM]
  - Avoid wears by distributing dispersing the written blocks
- **Magnetic disc**
  - Platters having tracks split into sectors
  - Large access time (seek delay + rotational delay + transfer time)



Memory Technology

- Four primary memory technologies

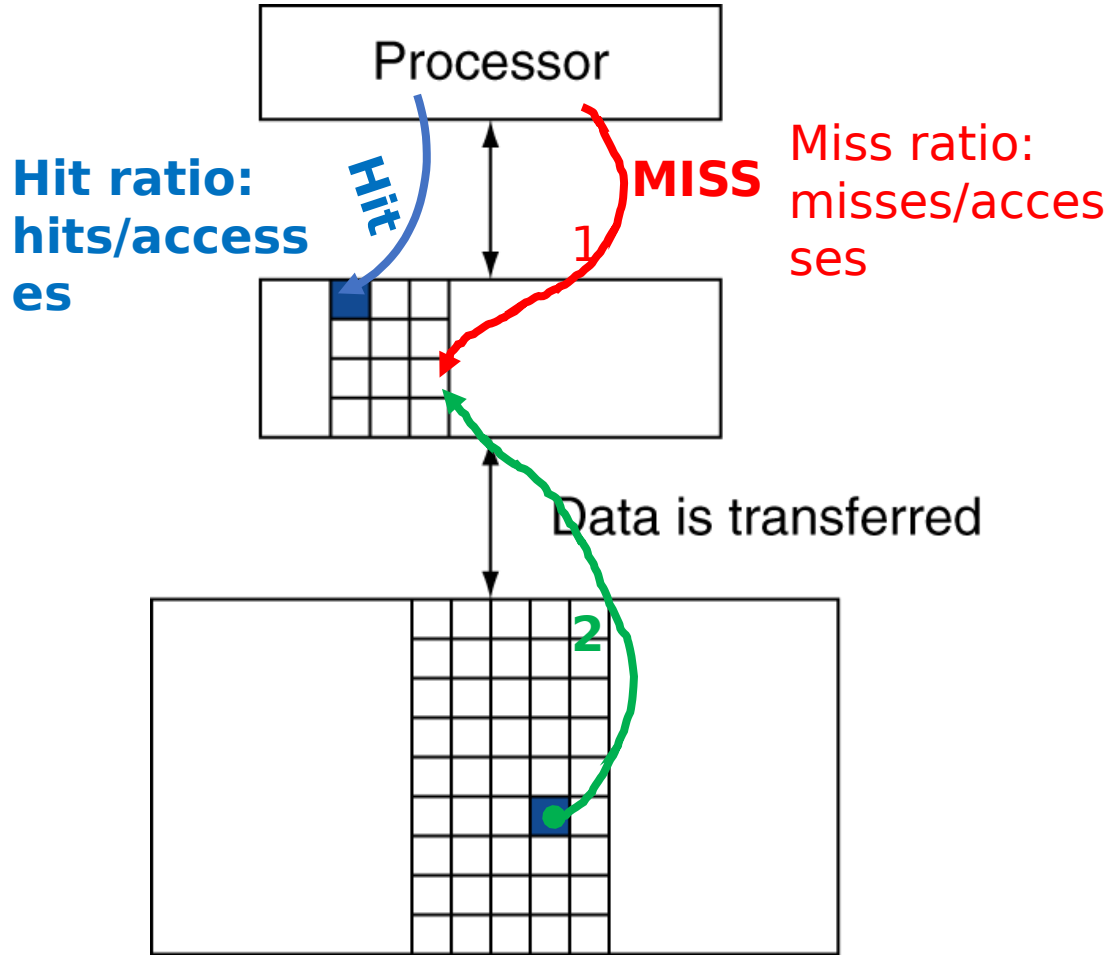
IDEAL

Fast as SRAM
Large as magnetic disc

Large and Fast: Exploiting Memory Hierarchy

Memory Hierarchy

- Store everything on disk
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory
  - Main memory [RAM]
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
  - Cache memory attached to CPU



Miss: block copied from lower level
Takes time (processor stall)
Processor then access data

Why memory hierarchy is good?

- **Principle of Locality**
- Programs access a small proportion of their address space at any time

Temporal locality

Items accessed recently are likely to be accessed again soon
e.g., instructions in a loop, induction variables

Spatial locality

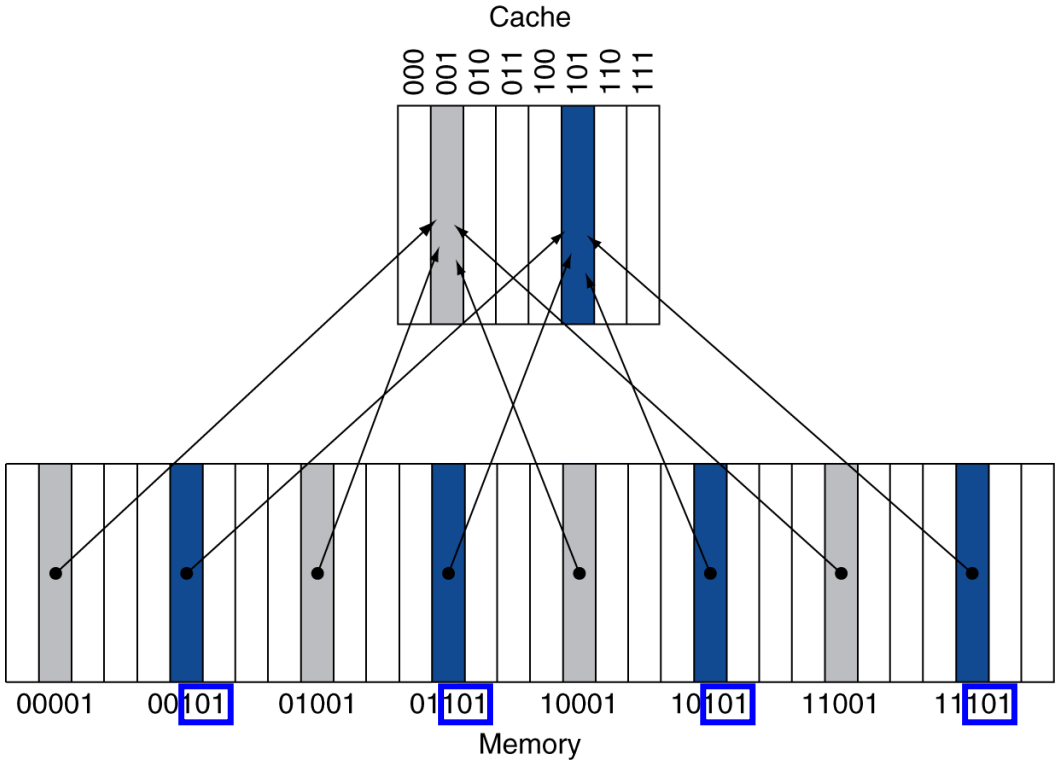
Items near those accessed recently are likely to be accessed soon
E.g., sequential instruction access, array data

Four Fundamental Design Questions

1. Where can a block be placed? **(Q1)**
2. How is a block found? **(Q2)**
3. What block is replaced on a miss? **(Q3)**
4. How are writes handled? **(Q4)**

Direct Mapped Cache

- Data location in cache is determined by its memory address **(Q1)**
- ***Direct mapped :*** **(Q3)**
  Memory address → one possible cache location
  (Block address) modulo (#Blocks in cache) ≈ *mapping function*



- #Blocks is a power of 2
- Use low-order address bits
- E.g., 8-block cache uses three lower bits $8 = 2^3$

Is my block in the cache? (Q2)

- ***Tags***
  - Store block identifier [TAG] as well as the data
  - Actually, TAG is the high-order bits of the memory address

- How do we know the cache block has valid data?
  - ***Valid bit:*** 1 = present, 0 = not present
  - Initially 0

Cache Example

- 8-block cache, 1 word/block, direct mapped
- Initial state

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 **110**	Miss	

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	**Y**	**10**	**Mem[10110]**
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
26	11 010	Miss	010

Index	V	Tag	Data
000	N		
001	N		
010	**Y**	**11**	**Mem[11010]**
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Hit	110
26	11 010	Hit	010

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

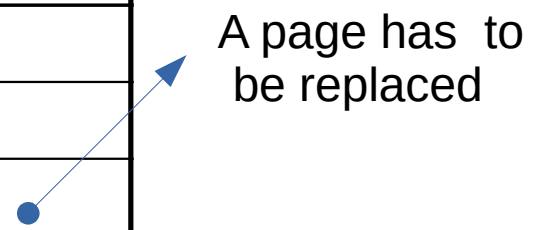
Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Miss	000

Index	V	Tag	Data
000	**Y**	**10**	**Mem[10000]**
001	N		
010	Y	11	Mem[11010]
011	**Y**	**00**	**Mem[00011]**
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

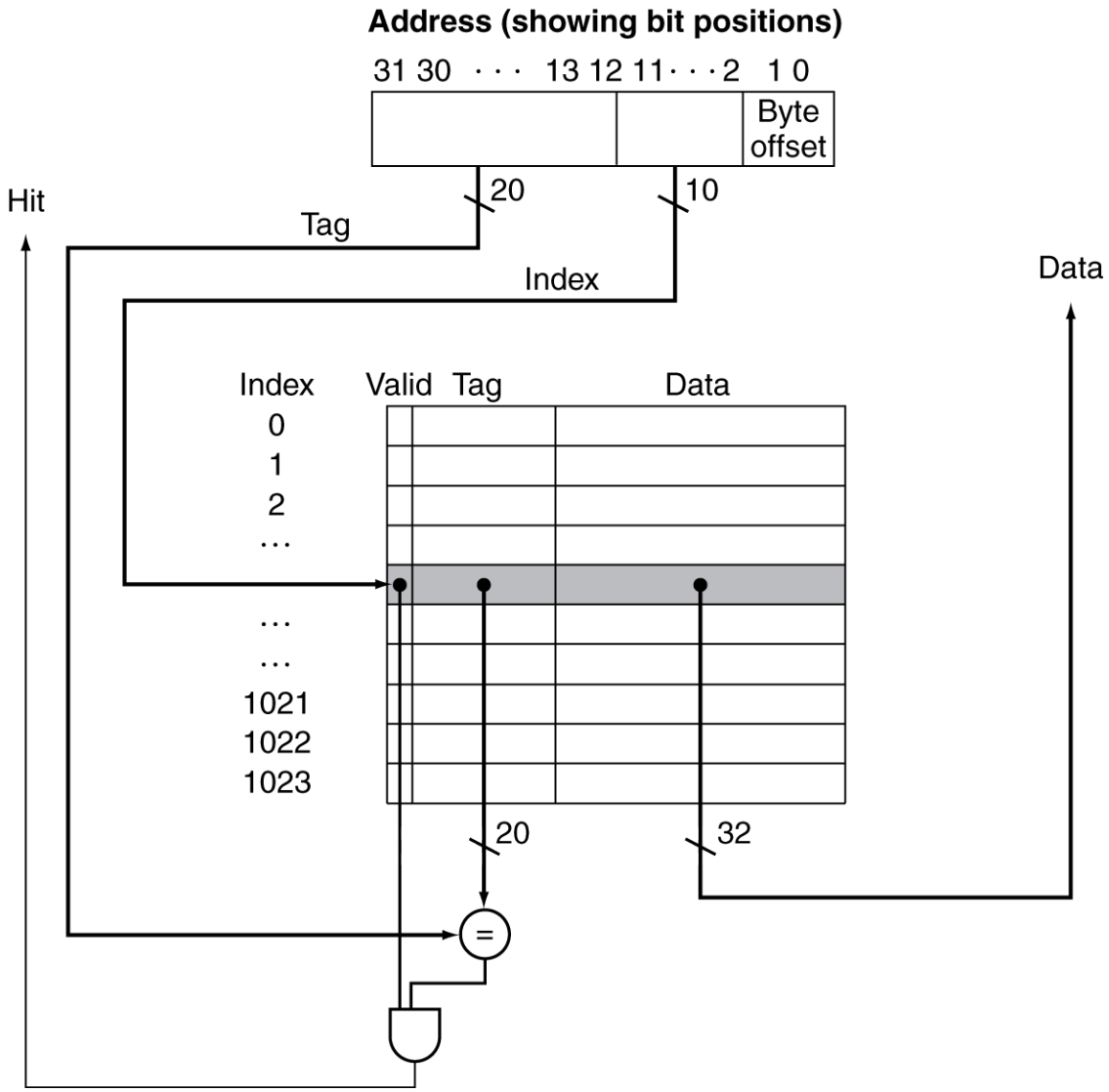
Cache Example

Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		



Address Subdivision



If tag and upper 20 are equal and valid bit set to 1 then we have a hit

Cache Size

- Cache stores **[data + Tag + valid bit]** for every block
- Cache size = number of blocks *
  (block size + tag size + valid field size)
- Tag size = Address size - Index size ← General case
  = Address size – (n + m)

Where cache has 2^n blocks and each block has 2^m **words**

- MIPS tag size = 32 – (n+m+2)
  aligned memory [two LSBs are insignificant]
- *Reported cache size represents the data part only*

Example: Larger Block Size

- 64 blocks cache, 16 bytes/block
  - To which memory block does address 1200 map?
- ***Memory Block*** =
  Byte address/ Byte per block = $\lfloor 1200/16 \rfloor = 75$
- ***Cache Block*** =
  75 modulo 64 = **11**
- 64 blocks = 2^6 blocks → **n = 6**
  16 byte/block = 2^2 words/block → **m = 2**
  MIPS TAG size = 32 – (6+2+2) = 20 bits

1200 =

0..01	**001011**	**0000**
TAG	Block Index	Block Offset

Reading

- Sections 5.1, 5.2, and 5.3