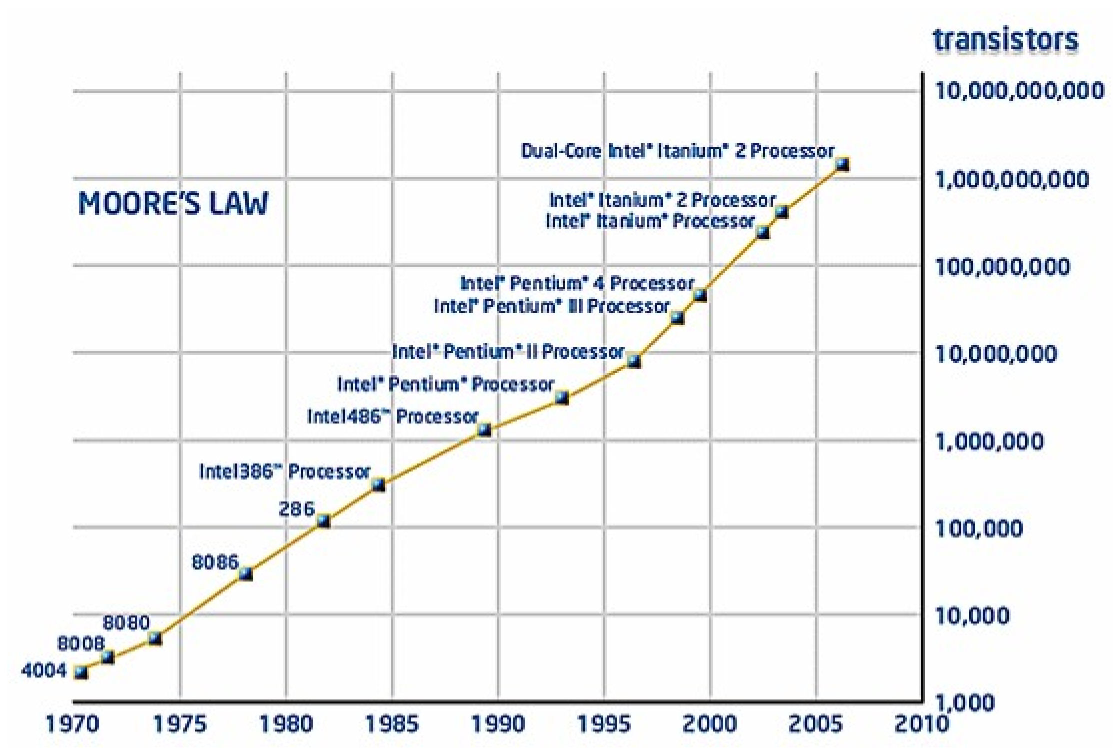
Computer Design and Organization

- Architecture = Design + Organization + Performance
- Topics in this class:
  - Central processing unit: deeply pipelined, multiple instr. per cycle, exploitation of instruction level parallelism (in-order and out-of-order), support for speculation (branch prediction, spec. loads).
  - Memory hierarchy: multi-level cache hierarchy, includes hardware and software assists for enhanced performance
  - Multiprocessors: SMP's and CMP's –cache coherence and synchronization
  - Multithreading: Fine, coarse and SMT
  - Some "advanced" topic: current research in dept.

Technological improvements

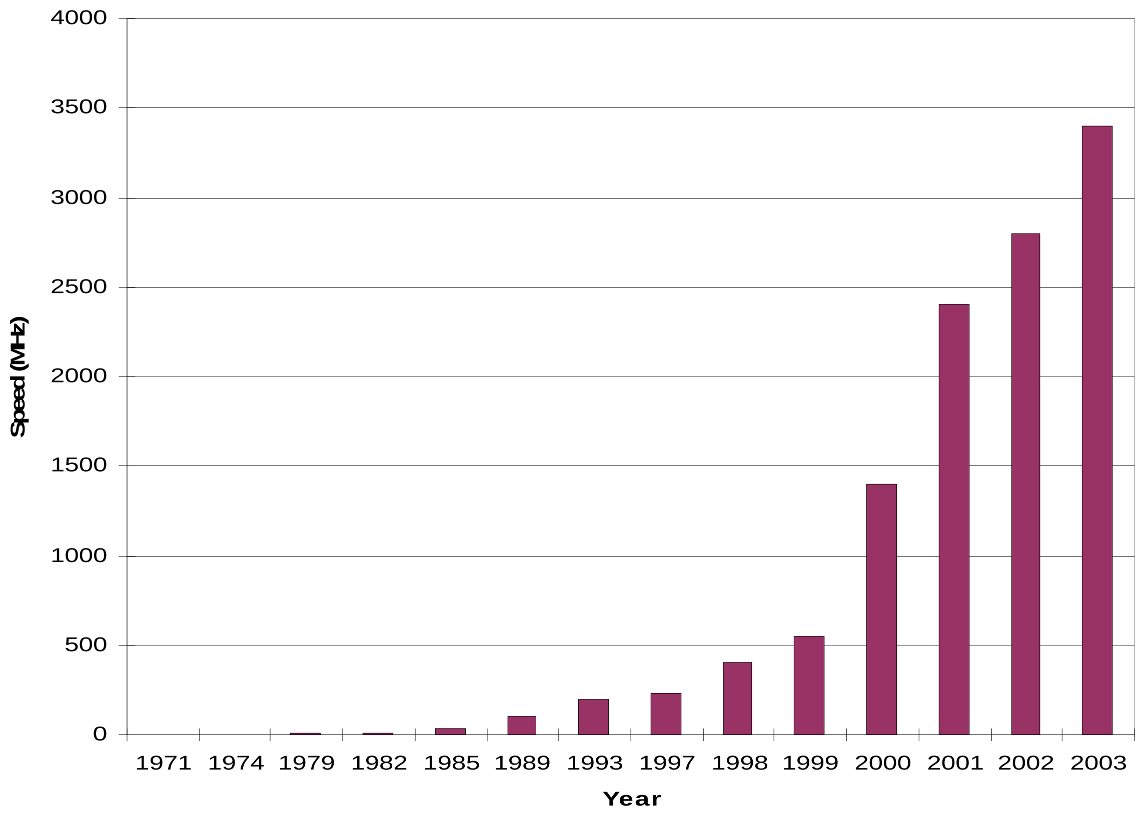
- CPU :
  - Annual rate of speed improvement is 35% before 1985 and 60% from 1985 until 2003
  - Slightly faster than increase in number of transistors on-chip (Moore's law)
- Memory:
  - Annual rate of speed improvement (decrease in latency) is < 10%
  - Density quadruples in 3 years.
- I/O :
  - Access time has improved by 30% in 10 years
  - Density improves by 50% every year

Moore’s Law

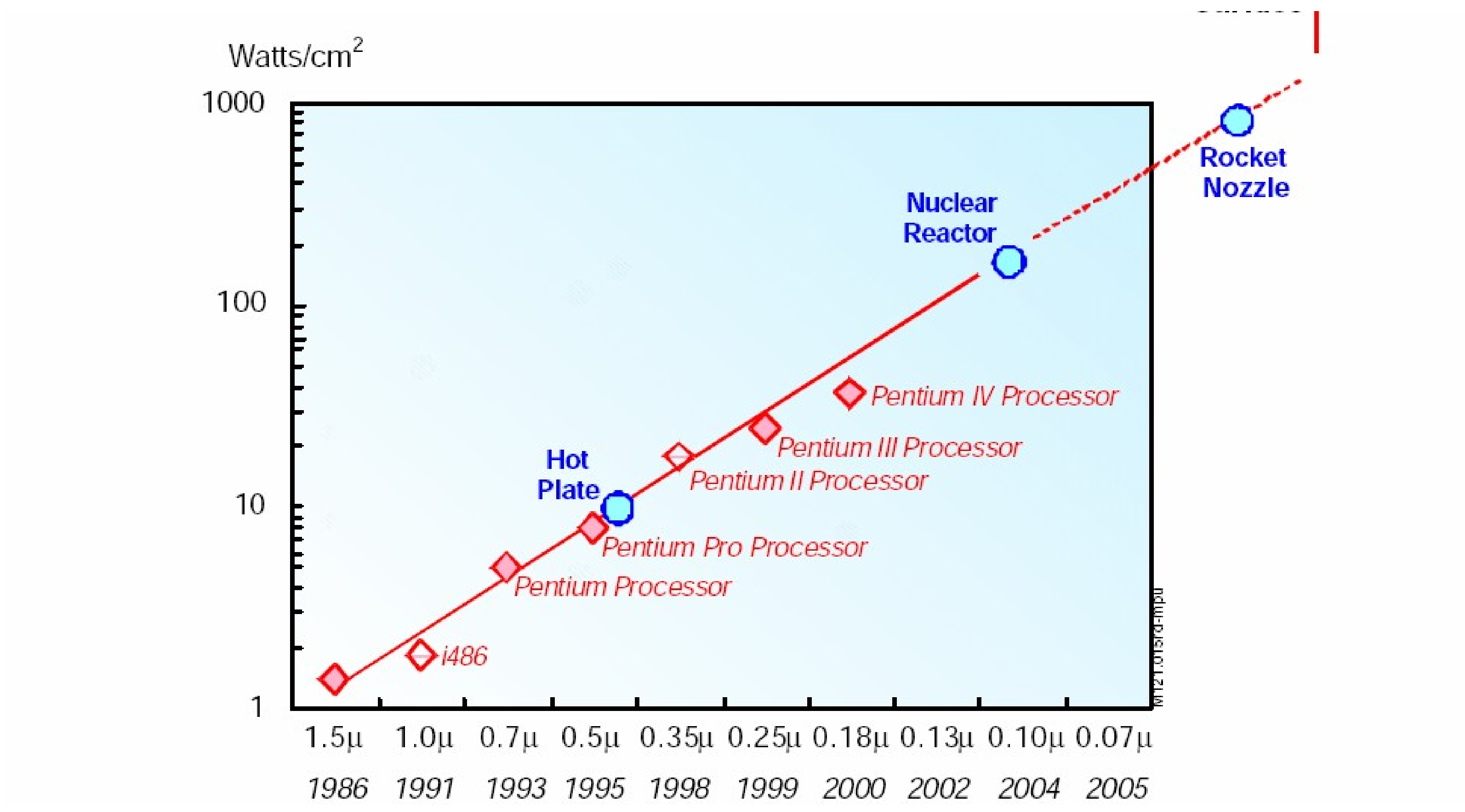


MOORE'S LAW
transistors
10,000,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000
1970
1975
1980
1985
1990
1995
2000
2005
2010
4004
8008
8080
8086
286
Intel386™ Processor
Intel486™ Processor
Intel® Pentium® Processor
Intel® Pentium® II Processor
Intel® Pentium® III Processor
Intel® Pentium® 4 Processor
Intel® Itanium® Processor
Intel® Itanium® 2 Processor
Dual-Core Intel® Itanium® 2 Processor

Evolution of Intel Microprocessor Speeds



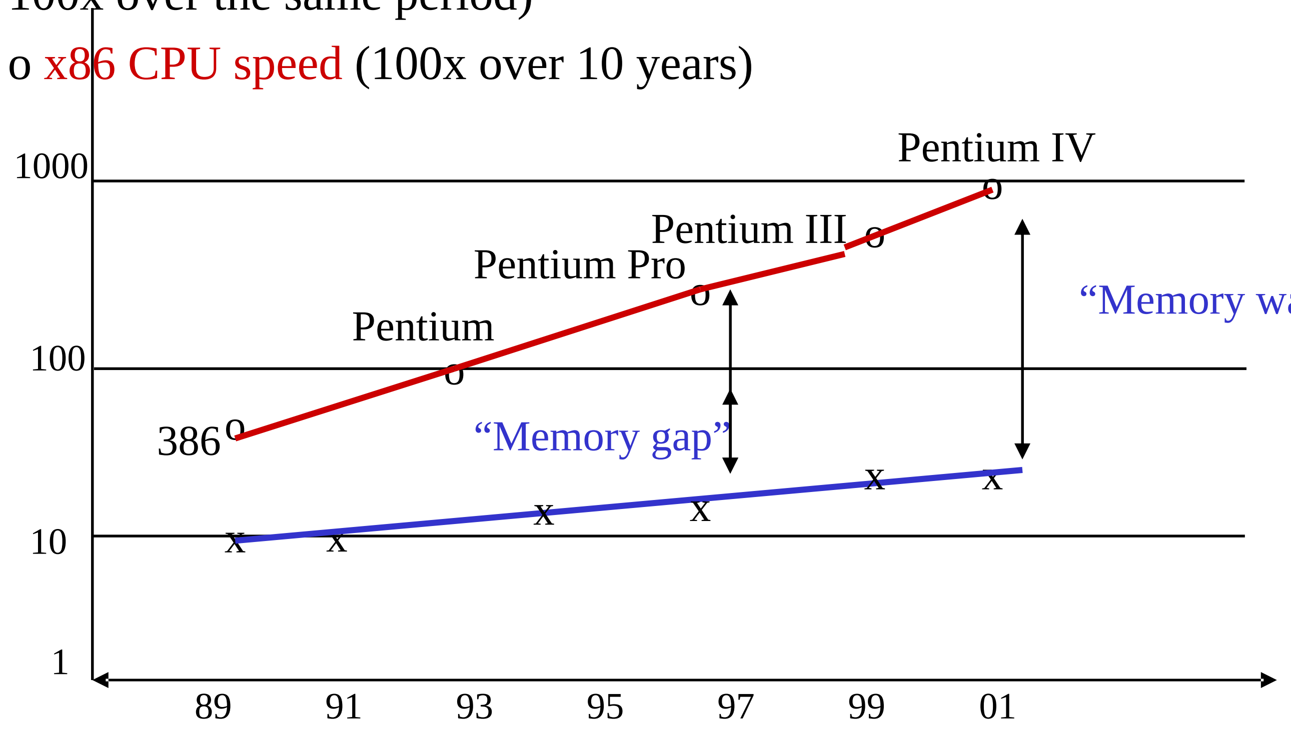
Power Dissipation



Processor-Memory Performance Gap

(there is a much nicer graph in H&P 4th Ed Figure 5.2 page 289 although it assumes that the processor speed is still improving)

- x Memory latency decrease (10x over 8 years but densities have increased 100x over the same period)
- o x86 CPU speed (100x over 10 years)



Performance evaluation basics

- Performance inversely proportional to execution time
- Elapsed time includes:

    user + system; I/O; memory accesses; CPU per se

- CPU execution time (for a given program): 3 factors
  - Number of instructions executed
  - Clock cycle time (or rate)
  - CPI: number of cycles per instruction (or its inverse IPC)

  CPU execution time = Instruction count * CPI * clock cycle time

Components of the CPI

- CPI for single instruction issue with ideal pipeline = 1
- Previous formula can be expanded to take into account classes of instructions
  - For example in RISC machines: branches, f.p., load-store.
  - For example in CISC machines: string instructions

  $CPI = \Sigma\, CPI_i * f_i$ where f_i is the frequency of instructions in class i
- We'll talk about "contributions to the CPI" from, e.g,:
  - memory hierarchy
  - branch (misprediction)
  - hazards etc.

Comparing and summarizing benchmark performance

- For execution times, use *(weighted) arithmetic mean:*

  Weighted Ex. Time = Σ $\text{Weight}_i * \text{Time}_i$

- For rates, use *(weighted) harmonic mean:*

  Weighted Rate = $1 / \Sigma (\text{Weight}_i / \text{Rate}_i)$

- As per Jim Smith (1988 – CACM)

  "Simply put, we consider one computer to be faster than another if it executes the same set of programs in less time"

- Common benchmark suites: SPEC for int and fp (SPEC92, SPEC95, SPEC2000, SPEC2006), SPECweb, SPECjava etc., Ogden benchmark (linked lists), multimedia etc.

Computer design: Make the common case fast

- Amdahl's law (speedup)

  Speedup = (performance with enhancement)/(performance base case)

  Or equivalently

  Speedup = (exec.time base case)/(exec.time with enhancement)

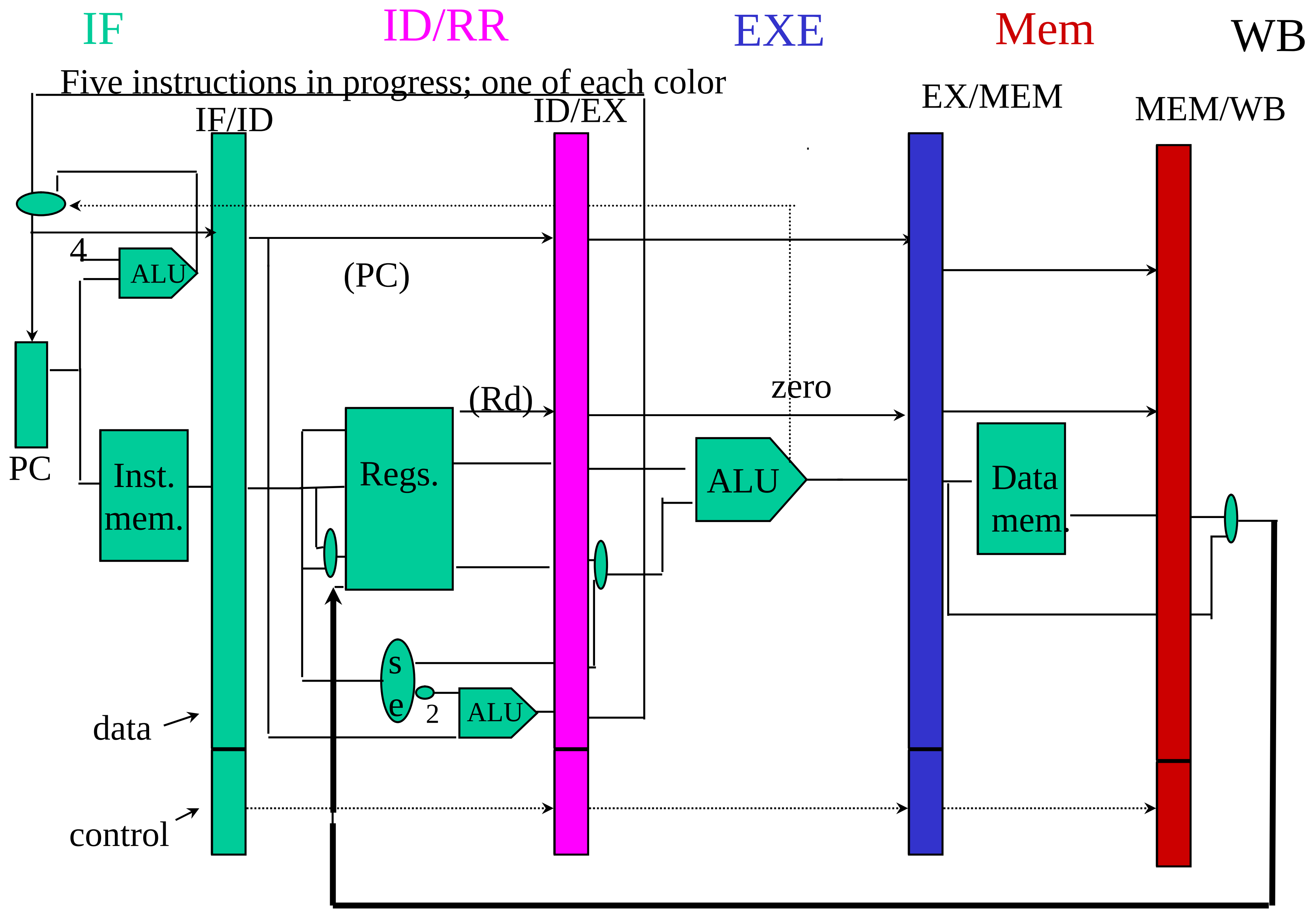
- Application to parallel processing
  - *s* fraction of program that is sequential
  - Speedup S is at most 1/*s*
  - That is if 20% of your program is sequential the maximum speedup with an infinite number of processors is at most 5

Pipelining

- One instruction/result every cycle (ideal)
  - Not in practice because of *hazards*
- Increase throughput (wrt non-pipelined implementation)
  - Throughput = number of results/second
- Speed-up (over non-pipelined implementation)
  - In the ideal case, if *n* stages , the speed-up will be close to *n*. Can't make *n* too large: physical limitations and load balancing between stages & hazards
- Might slightly increase the latency of individual instructions (pipeline overhead)

Basic pipeline implementation

- Five stages: IF, ID, EXE, MEM, WB
- What are the resources needed and where
  - ALU’s, Registers, Multiplexers etc.
- What info. is to be passed between stages
  - Requires pipeline registers between stages: IF/ID, ID/EXE, EXE/MEM and MEM/WB
  - What is stored in these pipeline registers?
- Design of the control unit.


IF
ID/RR
EXE
Mem
WB
Five instructions in progress; one of each color
IF/ID
ID/EX
EX/MEM
MEM/WB
4
ALU
(PC)
PC
Inst.
mem.
Regs.
(Rd)
zero
ALU
Data
mem.
s
e
2
ALU
data
control

Hazards

- Structural hazards
  - Resource conflict (mostly in multiple instruction issue machines; also for resources which are used for more than one cycle)
- Data dependencies
  - Most common RAW but also WAR and WAW in OOO execution
- Control hazards
  - Branches and other flow of control disruptions
- Consequence: stalls in the pipeline
  - Equivalently: insertion of *bubbles* or of no-ops

Pipeline speed-up

$$\text{Speedup\_ideal} = \frac{\text{pipeline depth}}{1}$$

$$\text{Speedup\_hazards} = \frac{\text{pipeline depth}}{1 + \text{CPI contributed by hazards}}$$

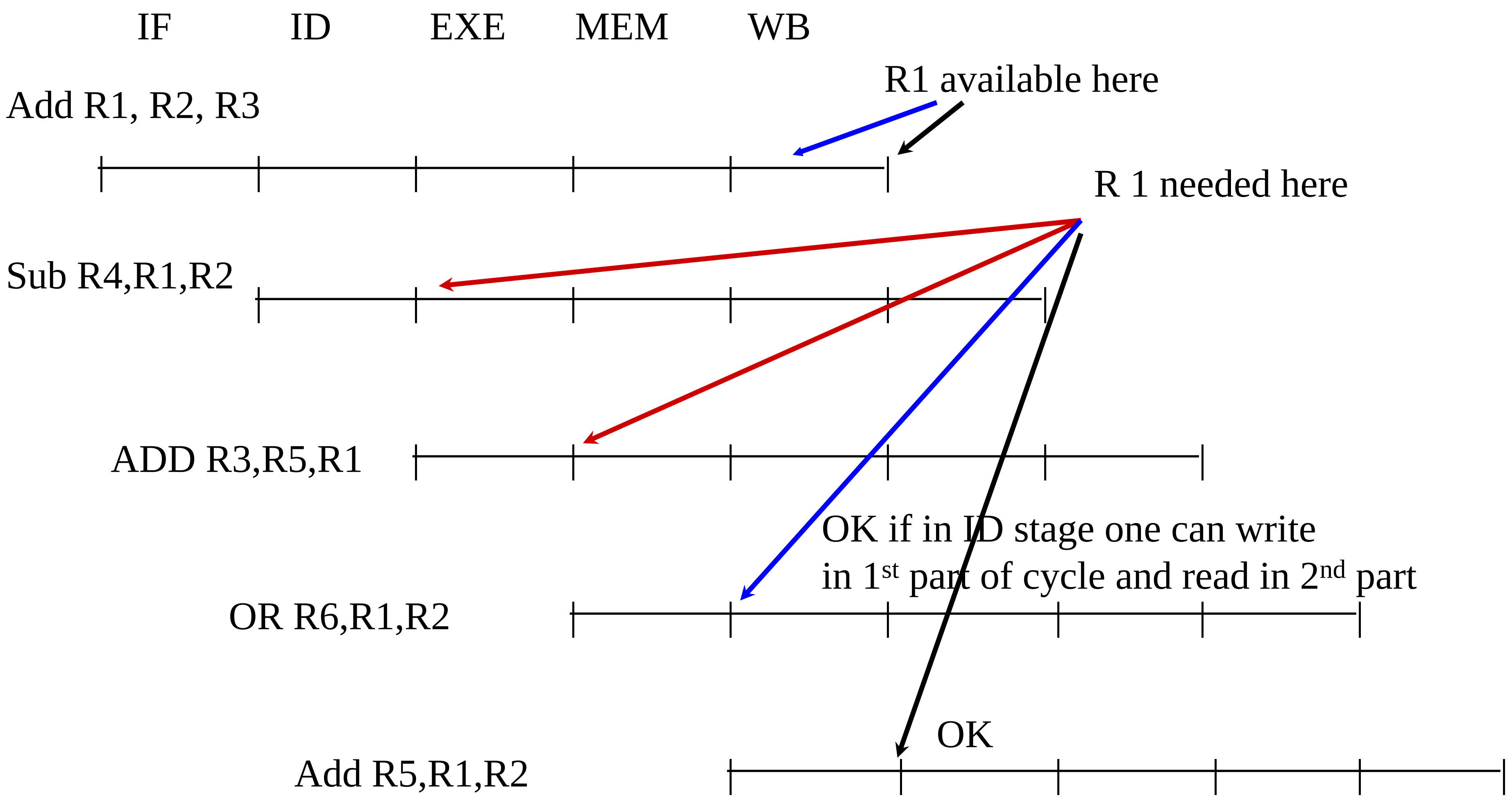
Example of structural hazard

- For single issue machine: common data and instruction memory (unified cache)
  - Pipeline stall every load-store instruction (control easy to implement)
- Better solutions
  - Separate I-cache and D-cache
  - Instruction buffers
  - Both + sophisticated instruction fetch unit!
- Will see more cases in multiple issue machines

Data hazards

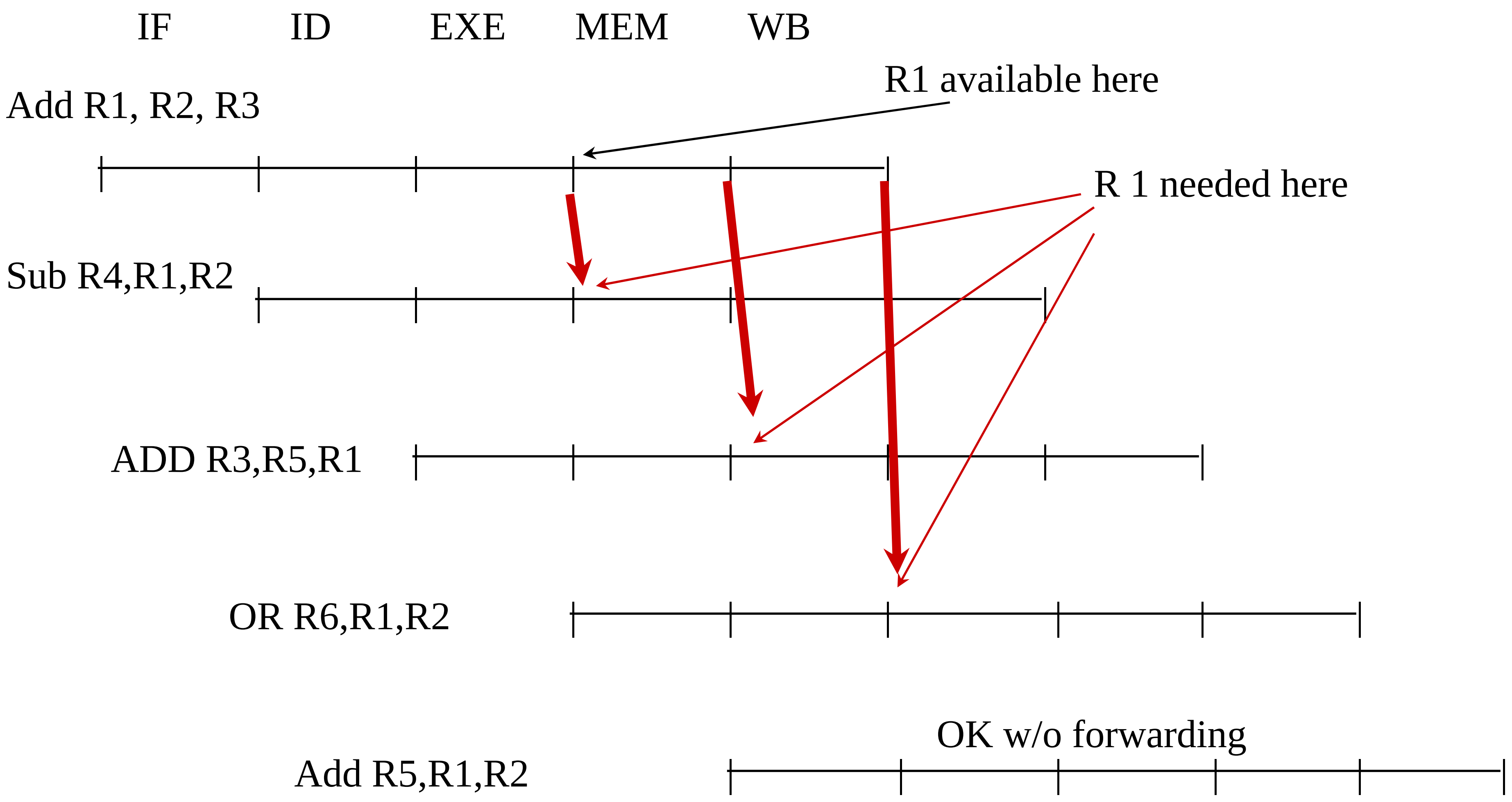
- Data dependencies between instructions that are in the pipe at the same time.
- For single pipeline in order issue: Read After Write hazard (RAW)

```
Add     R1, R2, R3          #R1 is result register
Sub     R4, R1,R2           #conflict with R1
Add     R3, R5, R1          #conflict with R1
Or      R6,R1,R2            #conflict with R1
Add      R5, R2, R1         #R1 OK now (5 stage pipe)
```


IF
ID
EXE
MEM
WB
Add R1, R2, R3
R1 available here
R 1 needed here
Sub R4,R1,R2
ADD R3,R5,R1
OK if in ID stage one can write in 1st part of cycle and read in 2nd part
OR R6,R1,R2
OK
Add R5,R1,R2

Forwarding

- Result of ALU operation is known at end of EXE stage
- Forwarding between:
  - EXE/MEM pipeline register to ALUinput for instructions *i* and *i+1*
  - MEM/WB pipeline register to ALUinput for instructions *i* and *i+2*
    - Note that if the same register has to be forwarded, forward the last one to be written
  - Forwarding through register file (write 1st half of cycle, read 2nd half of cycle)
- Need of a "forwarding box" in the Control Unit to check on conditions for forwarding
- Forwarding between load and store (memory copy)


IF
ID
EXE
MEM
WB
R1 available here
Add R1, R2, R3
R 1 needed here
Sub R4,R1,R2
ADD R3,R5,R1
OR R6,R1,R2
OK w/o forwarding
Add R5,R1,R2

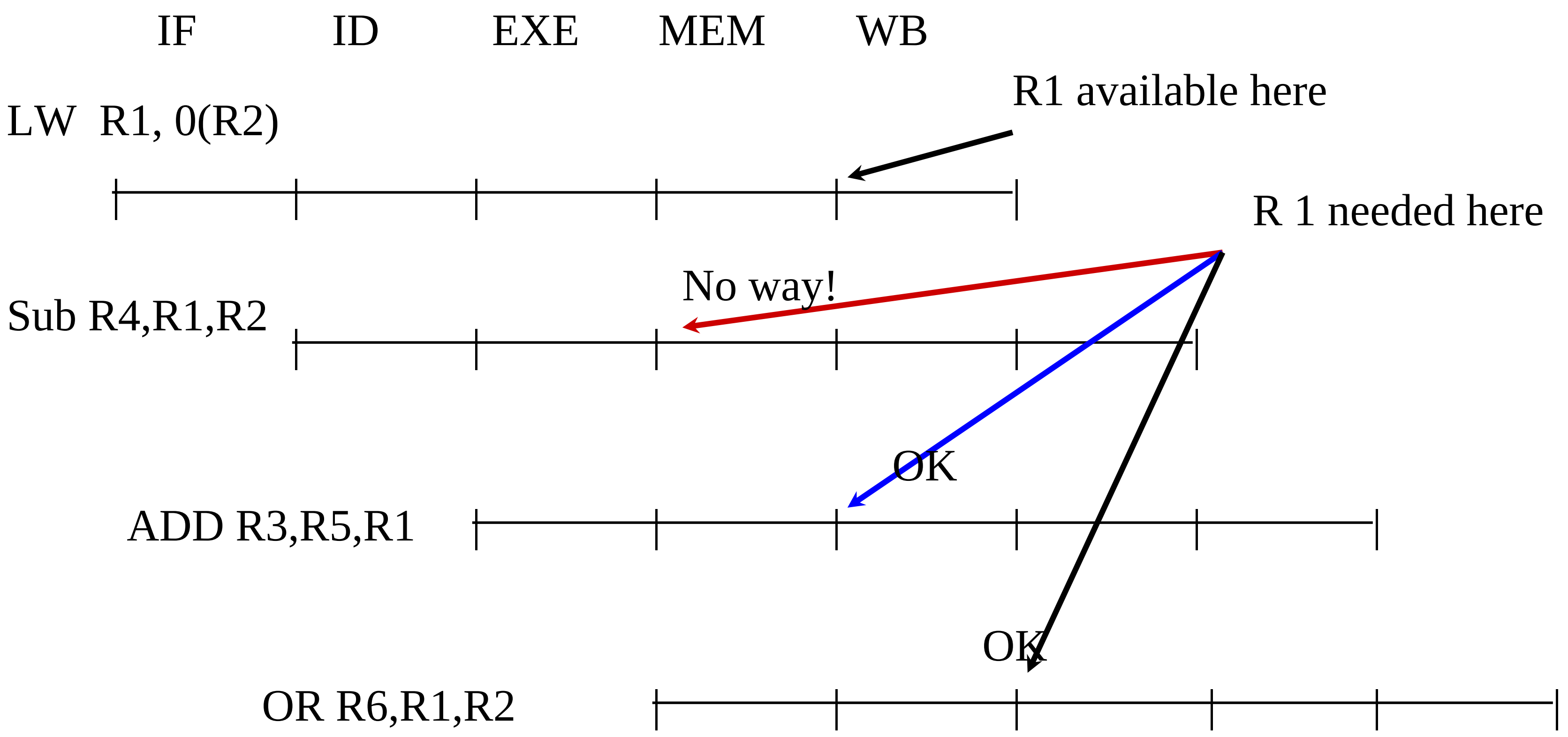
Other data hazards

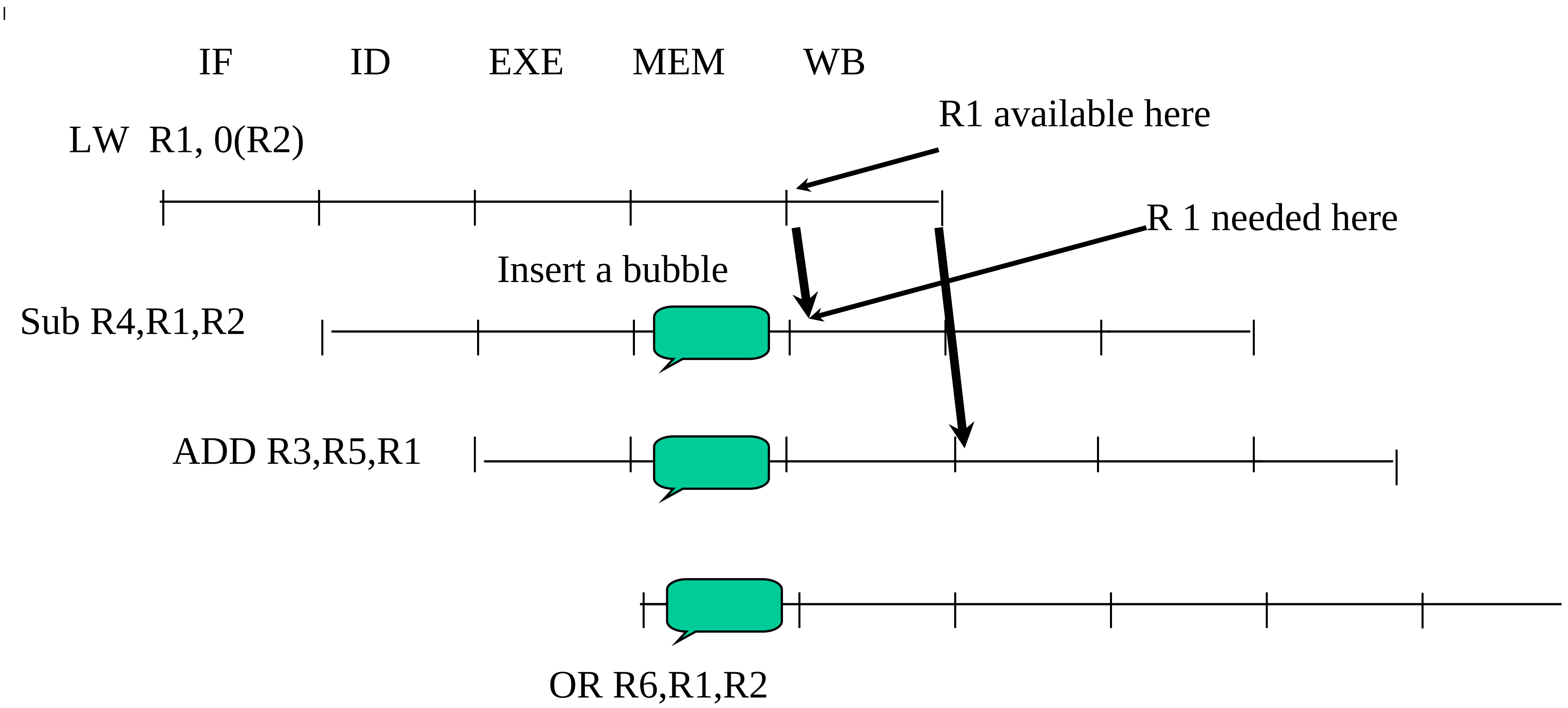
- Write After Write (WAW). Can happen in
  - Pipelines with more than one write stage
  - More than one functional unit with different latencies (see later)
- Write After Read (WAR). Very rare
  - With VAX-like autoincrement addressing modes

Forwarding cannot solve all conflicts

- For example, in a simple MIPS-like pipeline

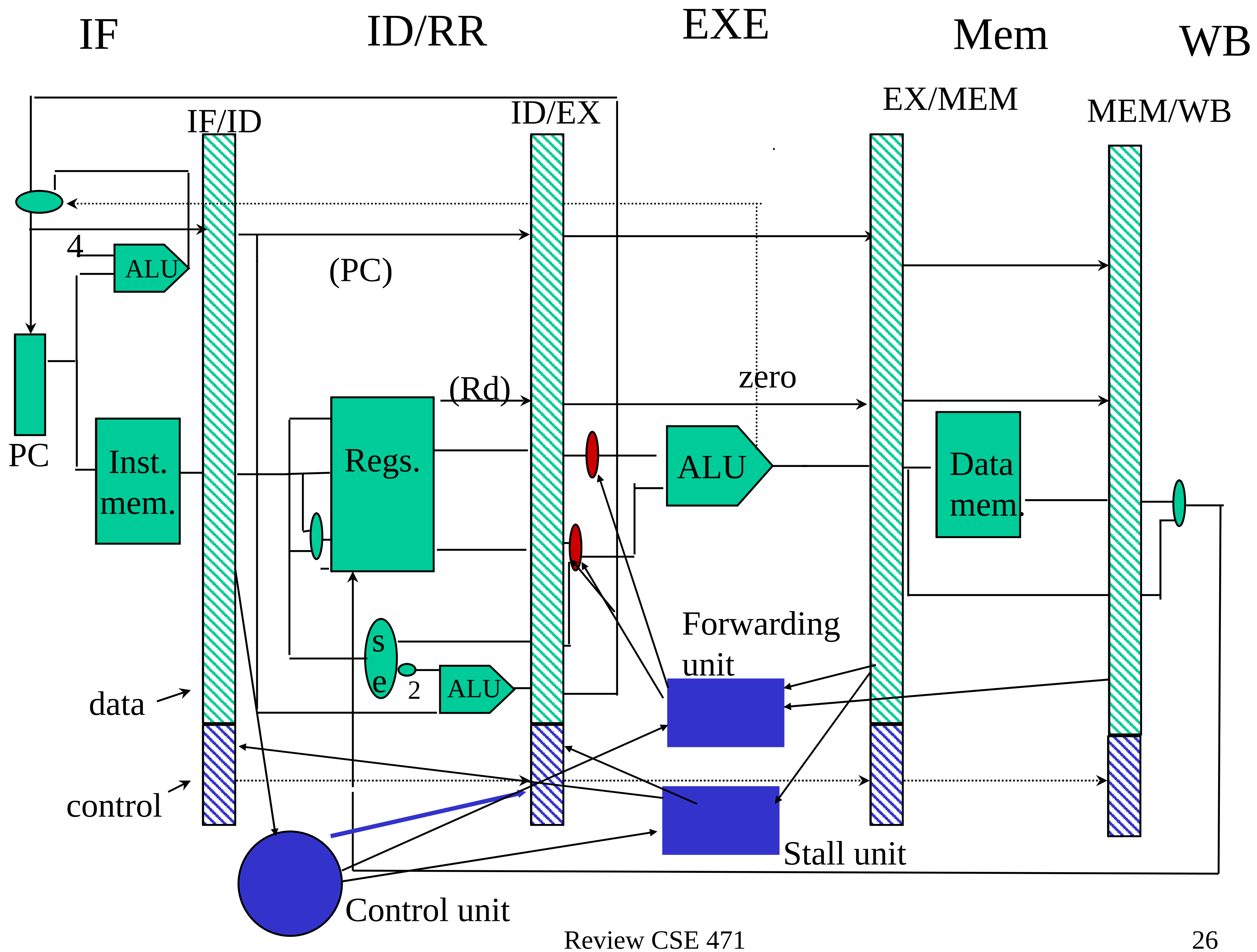
```
Lw      R1, 0(R2)          #Result at end of MEM stage
Sub     R4, R1,R2          #conflict with R1
Add     R3, R5, R1         #OK with forwarding
Or      R6,R1,R2           # OK with forwarding
```


IF
ID
EXE
MEM
WB
R1 available here
LW R1, 0(R2)
R 1 needed here
No way!
Sub R4,R1,R2
OK
ADD R3,R5,R1
OK
OR R6,R1,R2


IF
ID
EXE
MEM
WB
R1 available here
LW R1, 0(R2)
R 1 needed here
Insert a bubble
Sub R4,R1,R2
ADD R3,R5,R1
OR R6,R1,R2

Hazard detection unit

- If a Load (instruction *i-1*) is followed by instruction *i* that needs the result of the load, we need to stall the pipeline for one cycle , that is
  - instruction *i-1* should progress normally
  - instruction *i* should not progress
  - no new instruction should be fetched
- Controlled by a “hazard detection box” within the Control unit; it should operate during the ID stage


IF
ID/RR
EXE
Mem
WB
IF/ID
ID/EX
EX/MEM
MEM/WB
4
ALU
(PC)
PC
Inst. mem.
Regs.
(Rd)
zero
ALU
Data mem.
s e
2
ALU
Forwarding unit
data
control
Stall unit
Control unit

Control Hazards

- Branches (conditional, unconditional, call-return)
- Interrupts: asynchronous event (e.g., I/O)
  - Occurrence of an interrupt checked at every cycle
  - If an interrupt has been raised, don’t fetch next instruction, flush the pipe, handle the interrupt
- Exceptions (e.g., arithmetic overflow, page fault etc.)
  - Program and data dependent (repeatable), hence “synchronous”

Exceptions

- Occur “within” an instruction, for example:
  - During IF: page fault
  - During ID: illegal opcode
  - During EX: division by 0
  - During MEM: page fault; protection violation
- Handling exceptions
  - A pipeline is *restartable* if the exception can be handled and the program restarted w/o affecting execution

Precise exceptions

- If exception at instruction *i* then
  - Instructions *i-1, i-2 etc* complete normally (flush the pipe)
  - Instructions *i+1, i+2 etc*. already in the pipeline will be “no-oped” and will be restarted from scratch after the exception has been handled
- Handling precise exceptions: Basic idea
  - Force a trap instruction on the next IF
  - Turn off writes for all instructions *i* and following
  - When the target of the trap instruction receives control, it saves the PC of the instruction having the exception
  - After the exception has been handled, an instruction “return from trap” will restore the PC.

Precise exceptions (cont'd)

- Relatively simple for integer pipeline
  - All current machines have precise exceptions for integer and load-store operations (page fault)
- More complex for f-p
  - Might be more lenient in treating exceptions
  - Special f-p formats for overflow and underflow etc.

Integer pipeline (RISC) precise exceptions

- Recall that exceptions can occur in all stages but WB
- Exceptions must be treated in *instruction order*
  - Instruction *i* starts at time *t*
  - Exception in MEM stage at time *t + 3* (treat it first)
  - Instruction *i* + 1 starts at time *t +1*
  - Exception in IF stage at time *t + 1* (occurs earlier but treat in 2nd)

Treating exceptions in order

- Use pipeline registers
  - Status vector of possible exceptions carried on with the instruction.
  - Once an exception is posted, no writing (no change of state; easy in integer pipeline -- just prevent store in memory)
  - When an instruction leaves MEM stage, check for exception.