中山大學
SUN YAT-SEN UNIVERSITY

国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

Advanced Computer Architecture
高级计算机体系结构

第3讲： ISA and ILP (2)

张献伟

xianweiz.github.io

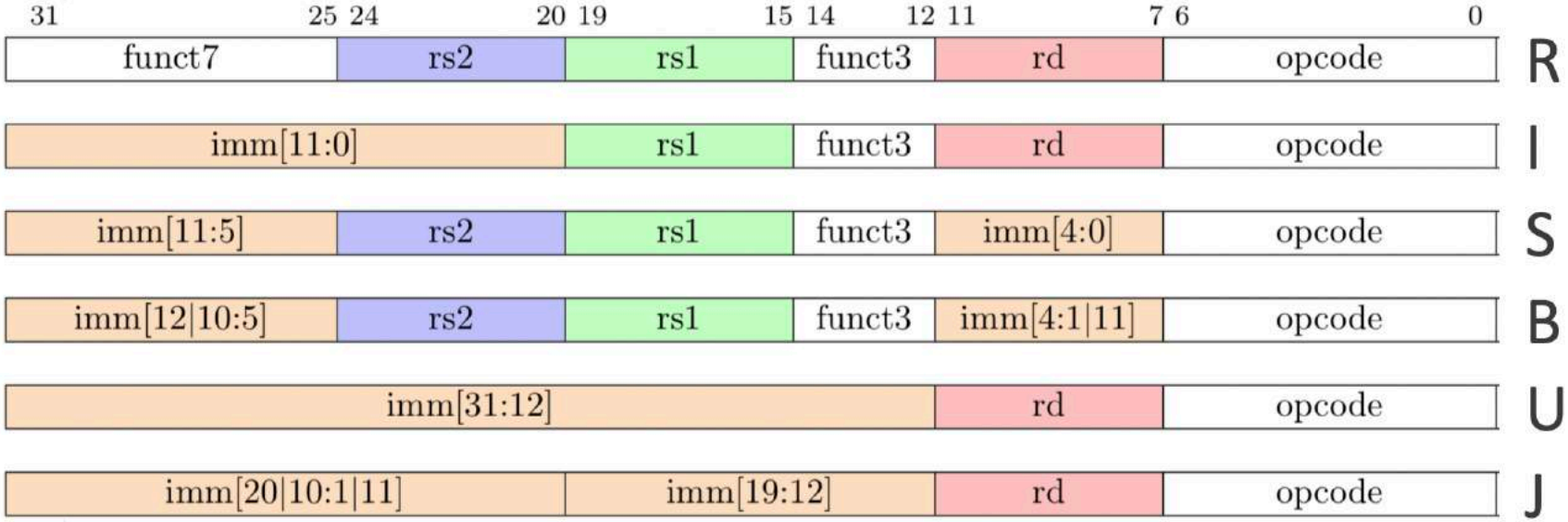
DCS5367, 9/28/2021

Review Questions

- List some goals of architecture designs?
  Functional, high performance, reliable, low cost, low power, …
- Amdahl's Law ?
  Speedup is limited by the fraction.
- If 80% of a program can be parallelized to run 4x faster, what's the overall speedup? The theoretical max?
  1 / (20% + 80%/4) = 2.5x max: 1 / 20% = 5
- CPI vs. IPC?
  Cycles per instruction instructions per cycle
- ISA vs. u-arch?
  u-arch is the specific implementation of ISA. Arch = ISA + u-arch
- CISC vs. RISC
  Complex vs. reduced, differences on instructions, perf, code size, …

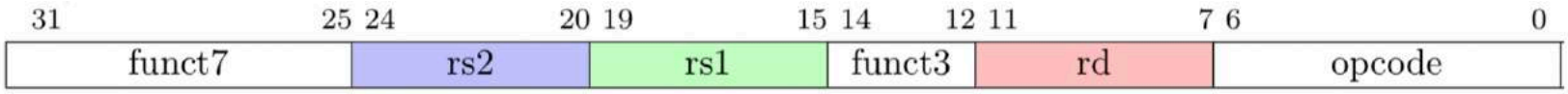
RISC-V Instructions[指令]

- All RISC-V instructions are 32 bits long, have 6 formats
  - R-type: instructions using 3 register inputs
  - I-type: instructions with immediates, loads
  - S-type: store instructions
  - B-type: branch instructions (beq, bge)
  - U-type: instructions with upper immediates
  - J-type: jump instructions (jal)

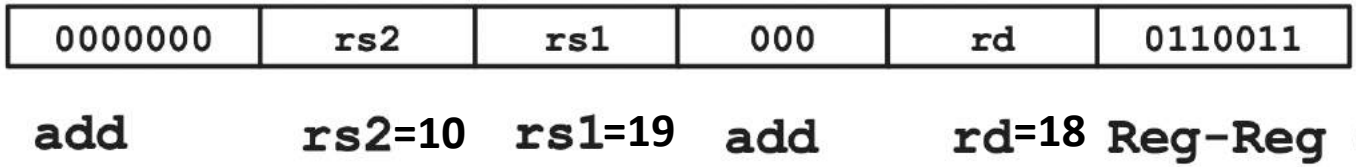


https://riscv.org/wp-content/uploads/2018/05/13.15-13-50-Talk-riscv-base-isa-20180507.pdf

Example

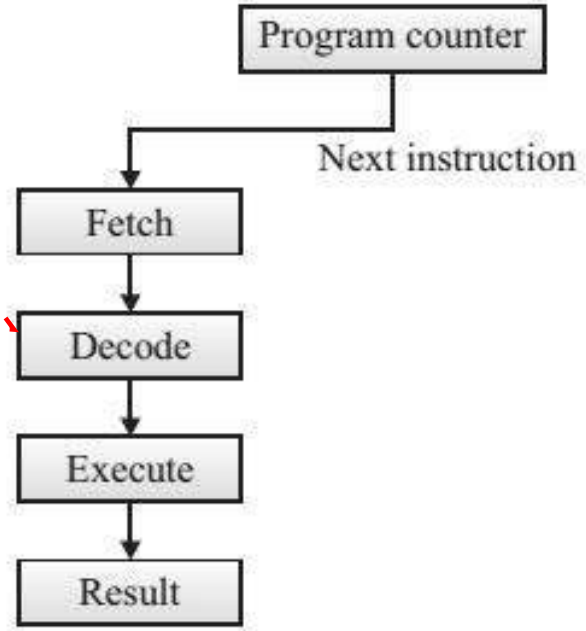


- Fields of R-type
  - **opcode**: partially specifies what instruction it is
  - **funct7**+**funct3**: combined with opcode, these two fields describe what operation to perform
  - **rs1** (source register #1): specifies register of first operand
  - **rs2**: specifies second register operand
  - **rd** (destination register): specifies register which will receive result of computation
    - Each register field holds a 5-bit unsigned integer (0-31) corresponding to a register number (x0-x31)
- add x18,x19,x10



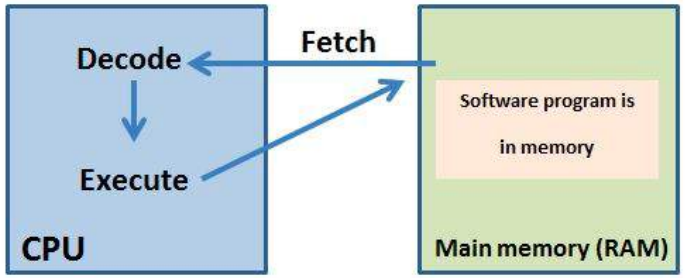
Executing an Instruction[执行指令]

- Very generally, what steps do you take to figure out the effect/result of the next RISC-V instruction?
  - Get the instruction[获取指令]
    - add x18,x19,x10
  - What instruction is it?[操作符?]
    - add
  - Gather data read[操作数?]
    - R[x19], R[x10]
  - Perform operation[操作]
    - calc R[x19]+R[x10]
  - Store result[结果]
    - save into x18



Five-Stage Execution(§C.1)[5阶段执行]

- **Instruction fetch** (IF)[取指令]/(IM: instruction memory)
  - Fetch the next instruction from memory (and update PC to the next sequential instruction)
- **Instruction decode** (ID)[解码]/(REG: register fetch)
  - Decode the inst and read the registers corresponding to register source specifiers
- **Execution**/effective address (EX)[执行]/(ALU)
  - Operate on the operands prepared in the prior cycle
- **Memory access** (MEM)[访存]/(DM: data memory)
  - Load: read using the effective address
  - Store: write to memory
- **Write-back** (WB)[回写]/(REG)
  - Writes the result into the register



Examples

- Arithmetic/logic instructions: R-type rd, rs1, rs2
  - IF: fetch instruction
  - ID: read registers rs1 and rs2
  - EX: compute result (use ALU)
  - WB: write to register rd
- Load instructions: lw rd, c(rs1)
  - IF: fetch instruction
  - ID: read register rs1 (and rs2 ??)
  - EX: use ALU to compute memory address = content of rs1 + c
  - MEM: read from memory
  - WB: write to register rd
- Store instructions: sw rs2, c(rs1)
  - IF: fetch instruction
  - ID: read registers rs1 and rs2
  - EX: use ALU to compute memory address = content of rs1 + c
  - WB: write value of rs2 to memory at address rs1+c

Why Five Stages?

- Could we have a different number of stages?
  - Yes, and other architectures do
- So why does RISC-V have five if instructions tend to idle for at least one stage?
  - The five stages are the union of all the operations needed by all the instructions
  - There is one instruction that uses all five stages: load (lw/lb)

R-type rd, rs1, rs2
IF: fetch instruction
ID: read registers rs1 and rs2
EX: compute result (use ALU)
WB: write to register rd

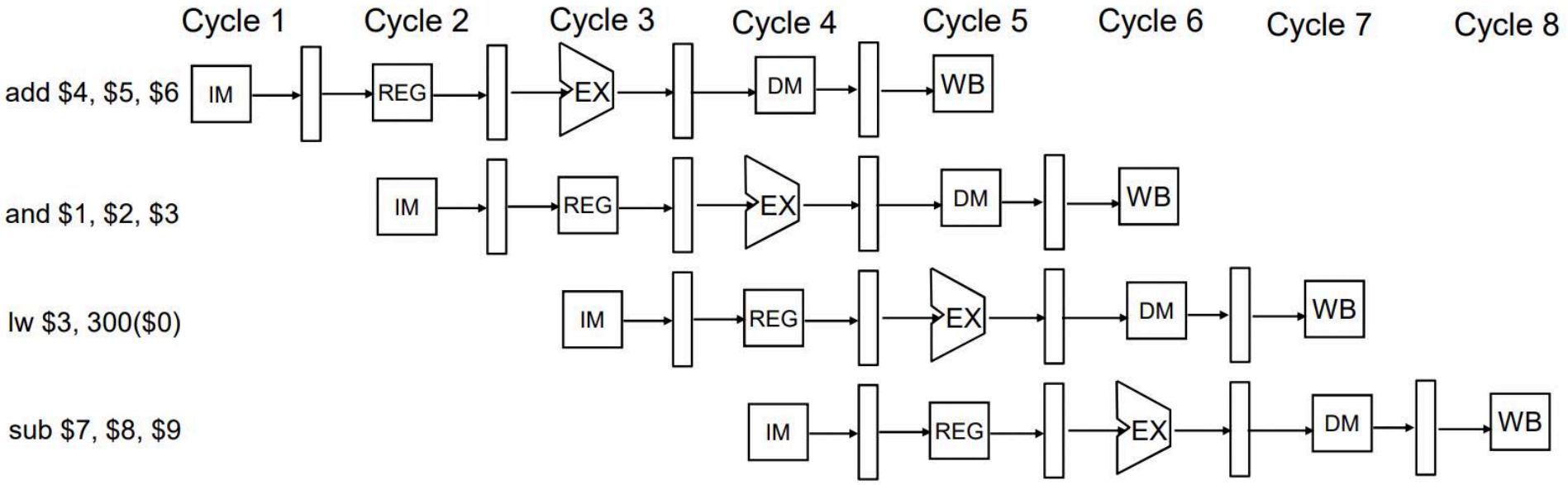
lw rd, c(rs1)
IF: fetch instruction
ID: read register rs1
EX: use ALU to compute address = rs1 + c
MEM: read from memory
WB: write to register rd

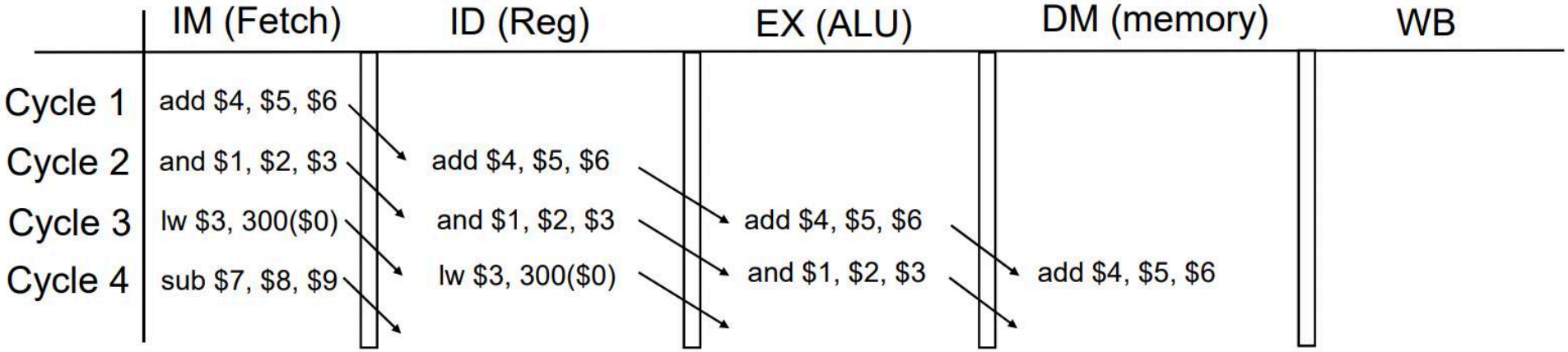
Pipelining[指令流水]

- Pipelining: an implementation technique whereby multiple instructions are overlapped in execution
  - Just like an assembly line
  - Takes advantage of parallelism that exists among the actions needed to execute an instruction
  - Pipelining is the key technique to make fast processors

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Visualize Pipelining[表示?]





Revisiting Pipeline

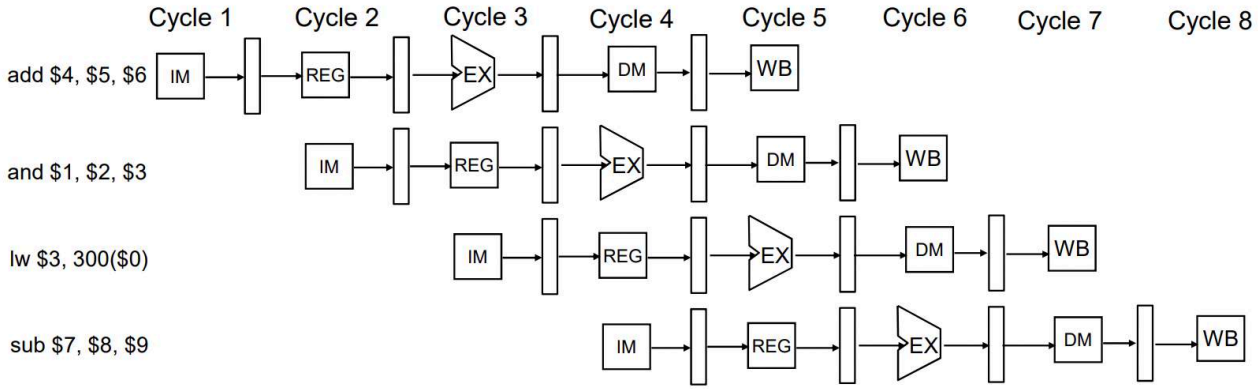
- Automobile assembly line[汽车流水线]
  - How often a completed car exits the assembly line (i.e., #cars/hour)[产能]
  - The longest step determines the time between advancing the line
- Instruction pipelining[指令流水线]
  - **Throughput**: how often an instruction exits the pipeline[吞吐]
  - **Processor cycle**: the time required between moving an inst one step down the pipeline
    - The length of a processor cycle is determined by the time required for the slowest pipe stage
    - Usually 1 clock cycle
- Pipeline designer should balance the length of each stage

Pipelining Effects[效果]

- If stages are perfectly balanced, then the time per inst on the pipelined processor (assuming ideal conditions)
  - $\frac{Time\ per\ instruction\ on\ unpipelined\ machine}{Number\ of\ pipe\ stages}$
- Speedup from pipelining equals the number of stages
  - An assembly pipeline with *n* stages can ideally produce cars *n* times fast
  - Instruction exit: every *n* cycles vs. every single cycle
- Pipelining reduces the avg execution time per inst
  - Baseline of multi clock cycles/inst: pipelining reduces CPI
  - Baseline of single clock cycle/inst: pipelining decreases the clock cycle time

Pipelining Effects (cont.)

- Pipelining exploits parallelism among the insts[并行]
  - Not visible to the programmer
- Pipelining improves instruction throughput rather instruction latency[提高吞吐]
  - Goal is to make programs, not individual insts, go faster
  - Single instruction latency
    - Doesn't really matter, billions of insts in a program
    - Difficult to reduce anyway
  - In fact, pipelining usually slightly increases the execution time of each inst



Performance Issues in Pipelining[问题]

- Impossible to reach the ideal speedup (= *n* stages)
  - Usually, the stages will not be perfectly balanced[不平衡]
    - The clock can run no faster than the time needed for the slowest pipeline stage
  - Furthermore, pipelining does involve some overhead[开销]
    - Pipeline register delay + clock skew[时钟漂移]

Example: one unpipelined processor has 1ns clock cycle and instructions are ALUs (4 cycles, 40%), branches (4 cycles, 20%), memory (5 cycles, 40%). Suppose that pipelining the processor adds 0.2ns overhead to the clock. How much pipelining speedup?

- Unpipelined processor, avg inst exe time = clock cycle x avg CPI = 1 ns x (40%x4 + 20%x4 + 40%x5) = 4.4ns
- Pipelined processor, avg inst exe time = 1 + 0.2 ns = 1.2 ns

Dependences and Hazards[依赖和冒险]

- **Dependence**[依赖]: relationship between two insts
  - Data: two insts use same storage location
  - Control: one inst affects whether another executes at all
  - Not a bad thing, programs would be boring without them
  - Enforced by making older inst go before younger one
    - Happens naturally in single-/multi-cycle designs
    - But not in a pipeline
- **Hazard**[冒险]: dependence & possibility of wrong inst order
  - Effects of wrong inst order cannot be externally visible
    - Stall: for order by keeping younger inst in same stage
  - Hazards are a bad thing: stalls reduce performance

Pipeline Hazards (§C.2)[流水冒险]

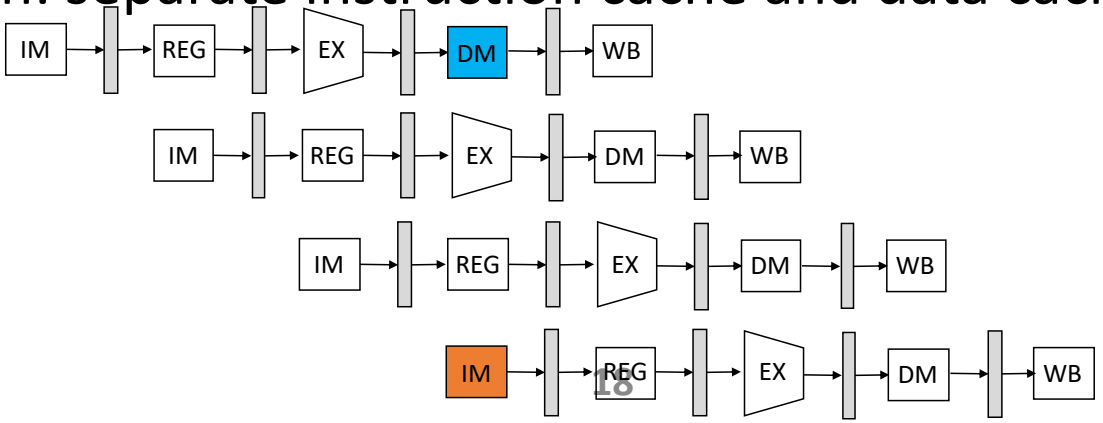
- Hazards prevent next instruction from executing during its designated clock cycle[妨碍执行]
  - Hazards reduce the performance from the ideal speedup gained by pipelining
- Three classes of hazards
  - **Structural hazards**[结构]: HW cannot support some combination of instructions
  - **Data hazards**[数据]: An instruction depends on result of prior instruction still in the pipeline
  - **Control hazards**[控制]: Pipelining of branches & other instructions stall the pipeline until the hazard bubbles in the pipeline

Structural Hazards[结构冒险]

- Different instructions are using the same resource at the same time[资源冲突]
  - Some functional unit is not fully pipelined
    - Then a sequence of insts using that unpipelined unit cannot proceed at the rate of one per clock cycle
  - Some resource has not been duplicated enough
- To fix structural hazards: proper ISA/pipeline design[解决]
  - Each inst uses every structure exactly one, for at most one cycle
  - Always at the same stage relative to Fetch
- Tolerate structural hazards[容忍]
  - Add stall logic to stall pipeline when hazards occur

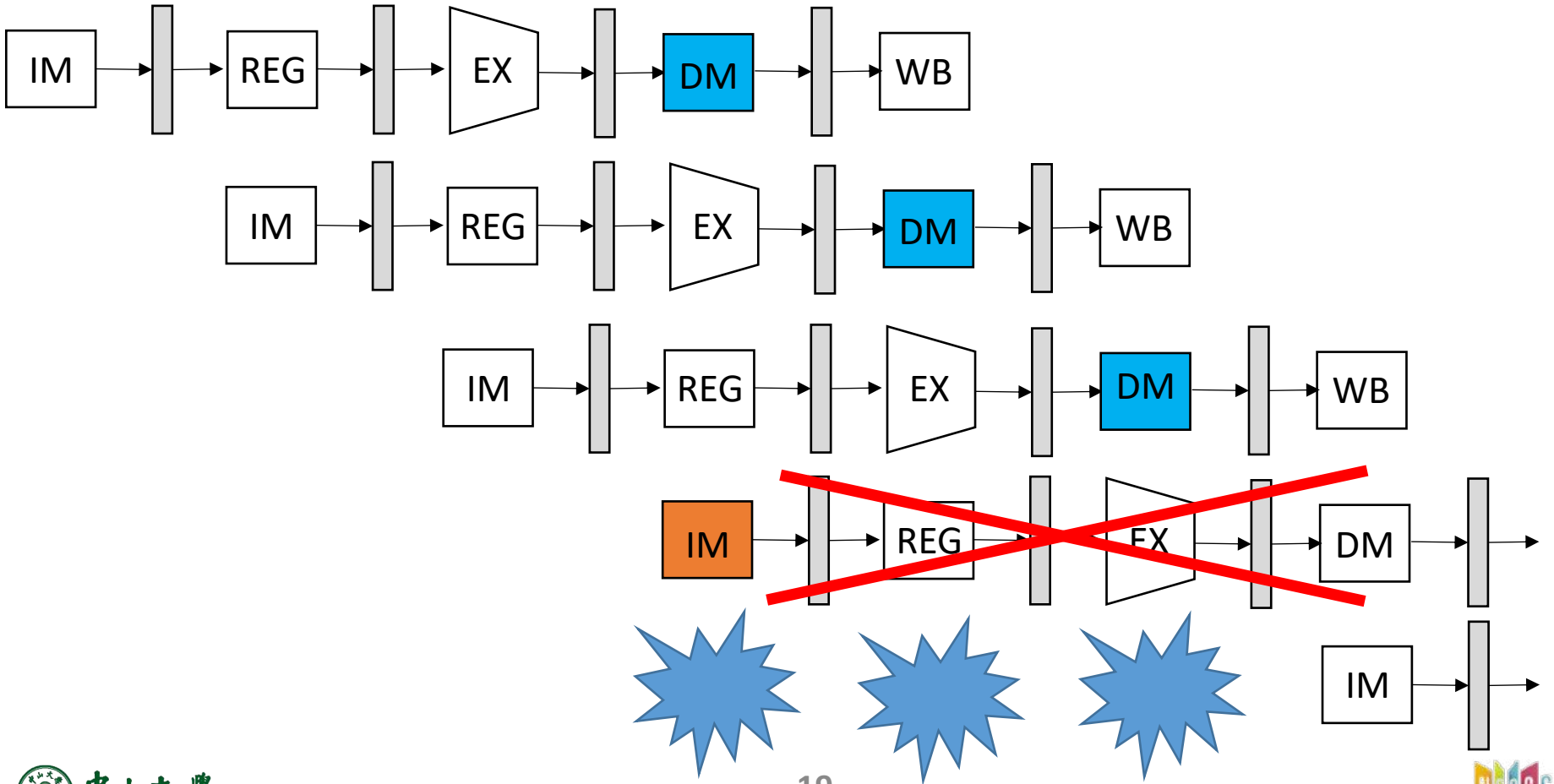
Structural Hazards (cont.)

- Register file[寄存器]
  - Accessed in two stages
    - Read during stage 2 (ID)
    - Write during stage 5 (WB)
  - Solution: one read port, one write port
- Memory[内存]
  - Accessed in two stages
    - Instruction fetch during stage 1 (IF)
    - Data read/write during stage 4 (MEM)
  - Solution: separate instruction cache and data cache



Structural Hazards (cont.)

- Bubbles are inserted
  - Wasted cycles → performance loss

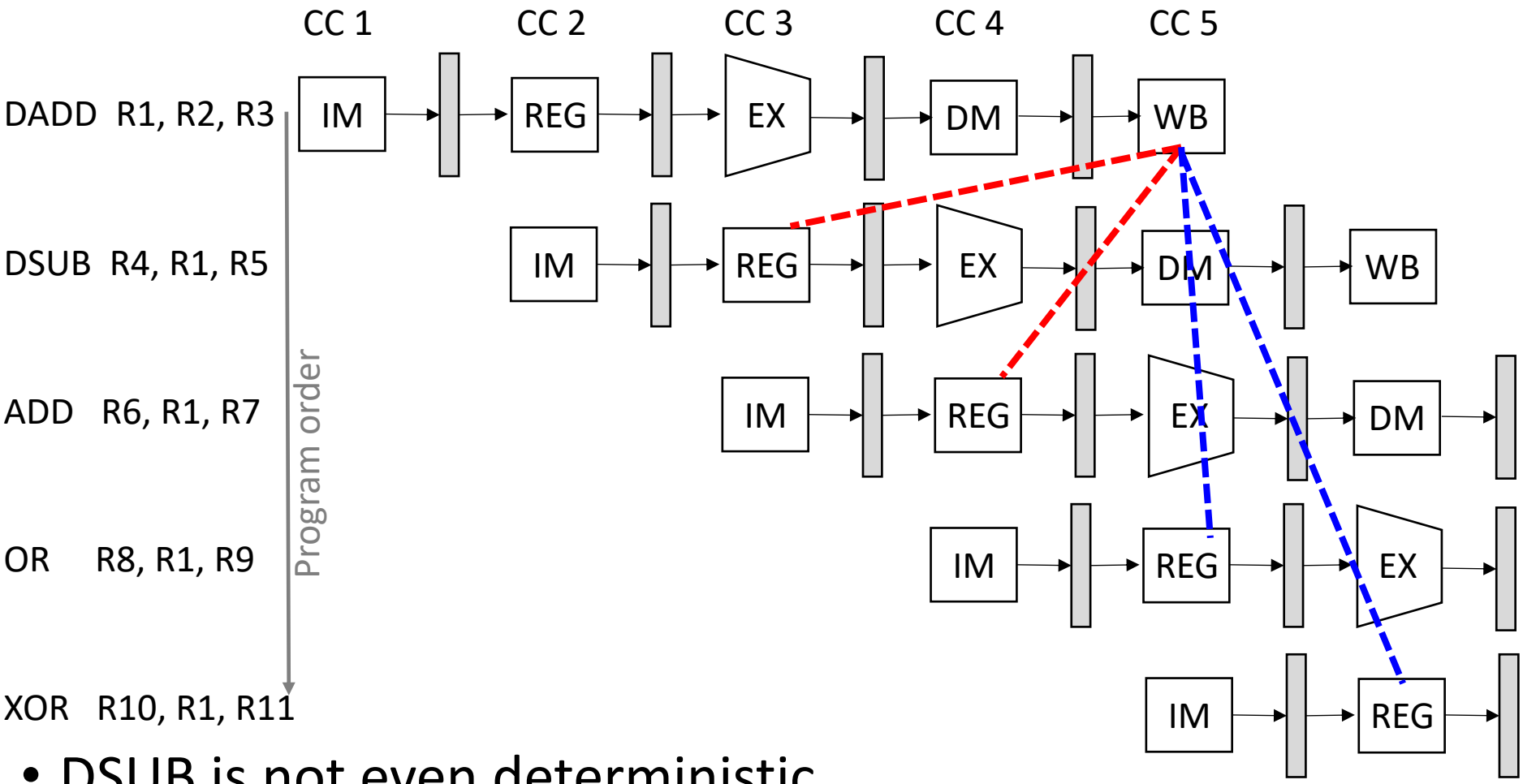


Data Hazards[数据冒险]

- Pipeline changes the order of read/write accesses to operands
  - The order may differ from the order seen by sequentially executing insts on an unpipelined processor
- Example
  - All the instructions after the DADD use the result of the DADD instruction
    - What if the old result is being accessed?
      - DADD writes into R1, happening in stage 5 (WB)
      - DSUB reads from R1, happening in stage 2 (ID)

```
DADD    R1, R2, R3
DSUB    R4, R1, R5
AND     R6, R1, R7
OR      R8, R1, R9
XOR     R10, R1, R11
```

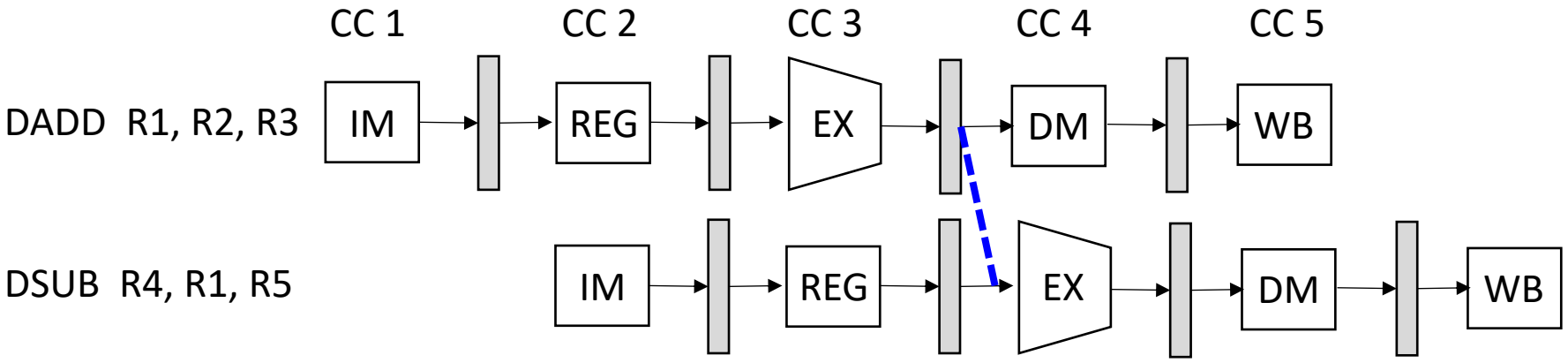
Data Hazards (cont.)



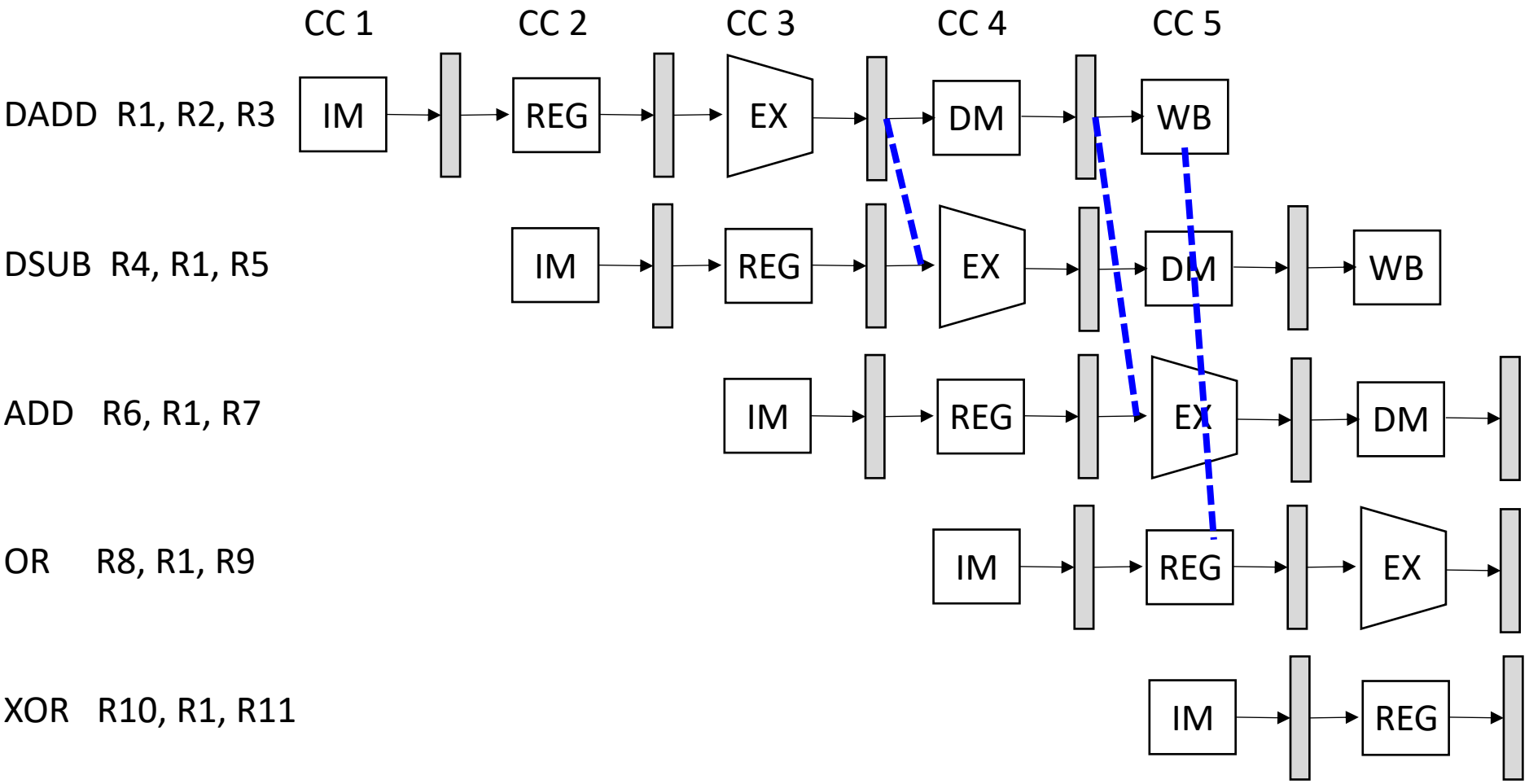
- DSUB is not even deterministic
  - Right, if an interrupt occurs between DADD and DSUB
  - Wrong, otherwise

Forwarding[转发]

- Minimizing data hazards stalls by forwarding
  - a.k.a., bypassing, short-circuiting
  - The result is not really needed by the *DSUB* until after the *DADD* actually produces it
  - If the result can be moved from the pipeline register where the *DADD* stores it to where the *DSUB* needs it, then the need for a stall can be avoided

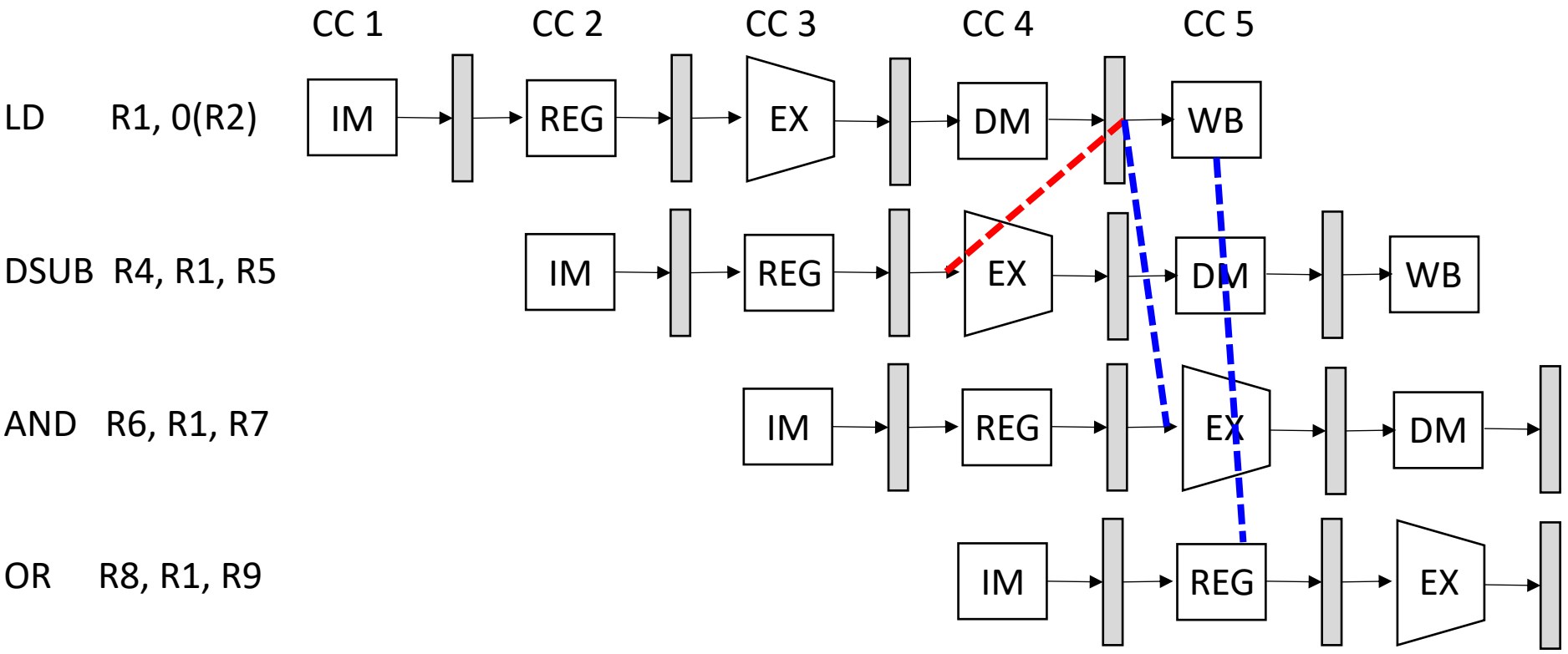


Forwarding (cont.)



- ALU inputs could use forwarded inputs from either the same pipeline register or from different pipeline registers

Forwarding is Insufficient[仅转发不够]



- LD can bypass its results to AND and OR instructions
- But not to the DSUB
  - Forwarding the result in “negative time” !

Pipeline Interlock[互锁]

- Bypassing alone isn't sufficient
  - Hardware solution: detect this situation and inject a stall cycle
  - Software solution: ensure compiler doesn't generate such code
- Pipeline **interlock** should be added to detect a hazard and stall the pipeline until the hazard is cleared
  - The interlock stalls the pipeline, beginning with the inst that wants to use the data until the source inst produces it
  - The interlock introduces a stall or bubble

LD R1, 0(R2)	IF	ID	EX	MEM	WB				
DSUB R4, R1, R5		IF	ID	stall	EX	MEM	WB		
AND R6, R1, R7			IF	stall	ID	EX	MEM	WB	
OR R8, R1, R9				stall	IF	ID	EX	MEM	WB

Control[控制]

- Question: what should the fetch PC be in the next cycle?
- Answer: the address of the next instruction
  - If the fetched inst is a non-control-flow inst:
    - Next Fetch PC is the address of the next-sequential inst
  - If the inst that is fetched is a control-flow inst:
    - How do we determine the next Fetch PC
- Branch (beq, bne) determines flow of control
  - Fetching next inst depends on branch outcome
  - Pipeline cannot always fetch correct inst

```
    beq   R9, R10, L
    add   R1, R2, R3
    sw    R6, 200(R8)
    sub   R4, R5, R6
    mult  R1, R2, R3
… …
L:  lw R7, 100(R8)
```

Branch Hazards[分支冒险]

- Control hazard: branch has a delay in determining the proper inst to fetch
- Basic implementation
  - Branch decision is unknown until MEM stage
  - 3 clock cycles are wasted

```
    beq   R9, R10, L
    add   R1, R2, R3
    sw    R6, 200(R8)
    sub   R4, R5, R6
    mult  R1, R2, R3
… …
… …
L:  lw R7, 100(R8)
```

beq	R9, R10, L	IF	ID	EX	MEM	WB				
add	R1, R2, R3		IF	ID	EX	MEM	WB			
sw	R6, 200(R8)			IF	ID	EX	MEM	WB		
sub	R4, R5, R6				IF	ID	EX	MEM	WB	
mult OR lw						IF	ID	EX	MEM	WB

Depending on the beq condition

Branch Stall Impact[停顿]

- If CPI = 1, 10% branch, stall 3 cycles → new CPI = 1.3
- Two-part solution
  - Determine branch taken (or not) sooner, and[分支是否执行]
  - Compute taken branch address earlier[目标地址计算]
- RISC-V solution
  - Move Zero test to ID/EX stage
  - Adder to calculate new PC in ID/EX stage
  - 1 clock cycle penalty for branch vs. 3
- One stall cycle for every branch will yield a performance loss of 10% - 30% depending on the branch frequency
  - Need to deal with this loss

Pipeline Stall Reductions[减少停顿]

- #1: Stall until branch direction is clear[保持停顿]
  - Freeze or flush the pipeline, holding or deleting any insts after the branch until the branch destination is known
- #2: Predict branch not taken[预测分支不执行]
  - Treat every branch as taken as not taken, simply allowing the HW to continue as if the branch were not taken
  - If branch actually taken, turn fetched insts into no-op and restart the fetch at the target address
- #3: Predict branch taken[预测分支执行]
  - As soon as the branch is decoded and the target address is computed, begin fetching and executing at the target
  - One cycle improvement when the branch is actually taken

Pipeline Stall Reductions (cont.)

- #4: Delayed branch[延后分支]
  - Change semantics such that branching takes place AFTER the n insts following the branch execute
  - Branch delay slot: the sequential successor
    - This inst is executed whether or not the branch is taken
    - Typically one inst delay in practice
    - Compiler should make the successor insts valid and useful
  - One slot delay in the 5-stage pipeline if branch condition and target are resolved in the ID stage

branch instruction

sequential successor$_1$

branch target if taken

Performance of Branch Schemes[性能]

- When accounting for branch hazards
  - CPI = 1 + hazard_frequency * hazard_penalty
- Example: assume a pipeline in which the target address is known in stage 3 and the branch condition is evaluated in stage 4. Find the effective addition to the CPI arising from branches for this pipeline.
  - Assume that you know the percentages of branch instructions, and the probabilities for branch taken/not taken.
  - See textbook C-21
- Compilers technology can be used to increase efficiency of code if it knows branch probabilities (static branch prediction - profiling)

Performance of Branch Schemes

- When accounting for branch hazards
  - CPI = 1 + hazard_frequency * hazard_penalty

Example: assume a pipeline in which the target address is known in stage 3 and the branch condition is evaluated in stage 4. Find the effective addition to the CPI arising from branches for this pipeline.

- Assume that you know the percentages of branch instructions, and the probabilities for branch taken/not taken.
- See textbook C-21

- Compilers technology can be used to increase efficiency of code if it knows branch probabilities (static branch prediction - profiling)

Summary

- Pipelining overlaps multiple instructions in execution
  - Speed up programs
- Hazards reduce effective of pipelining
  - Structural hazards: conflict in use of a datapath component
  - Data hazards: need to wait for result of a previous instruction
  - Control hazards: address of next instruction uncertain/unknown
- To increase processor performance
  - Clock rate
    - Limited by technology and power dissipation
  - Pipelining
    - Deeper pipeline is challenging
  - Multi-issue processor
    - Several instructions executed simultaneously

Instruction-Level Parallelism(§3.1)

- ILP: overlap execution of instructions[指令级并行]
  - Overlap among instructions[重叠]
    - Pipelining or multiple instruction execution
  - Fine-grained parallelism[细粒度]
    - In contrast to process-/task/thread-level parallelism (coarse-grained)
- Pipelining: exploits ILP by executing several instructions “in parallel”
  - Overlaps execution of different instructions
  - Execute all steps in the execution cycle simultaneously, but on different instructions
- Pipeline CPI = Ideal pipeline CPI + stalls due to hazards
  - Structural stalls + Data hazard stalls + Control stalls

Instruction-Level Parallelism(cont.)

- Approaches to exploit ILP[利用方法]
  - Rely on hardware to help discover and exploit the parallelism dynamically
  - Rely on software technology to find parallelism, statically at compile-time
- What determines the degree of ILP?[并行度]
  - Dependences: property of the program
  - Hazards: property of the pipeline (or the architecture)
- ILP challenge: overcoming data and control dependencies

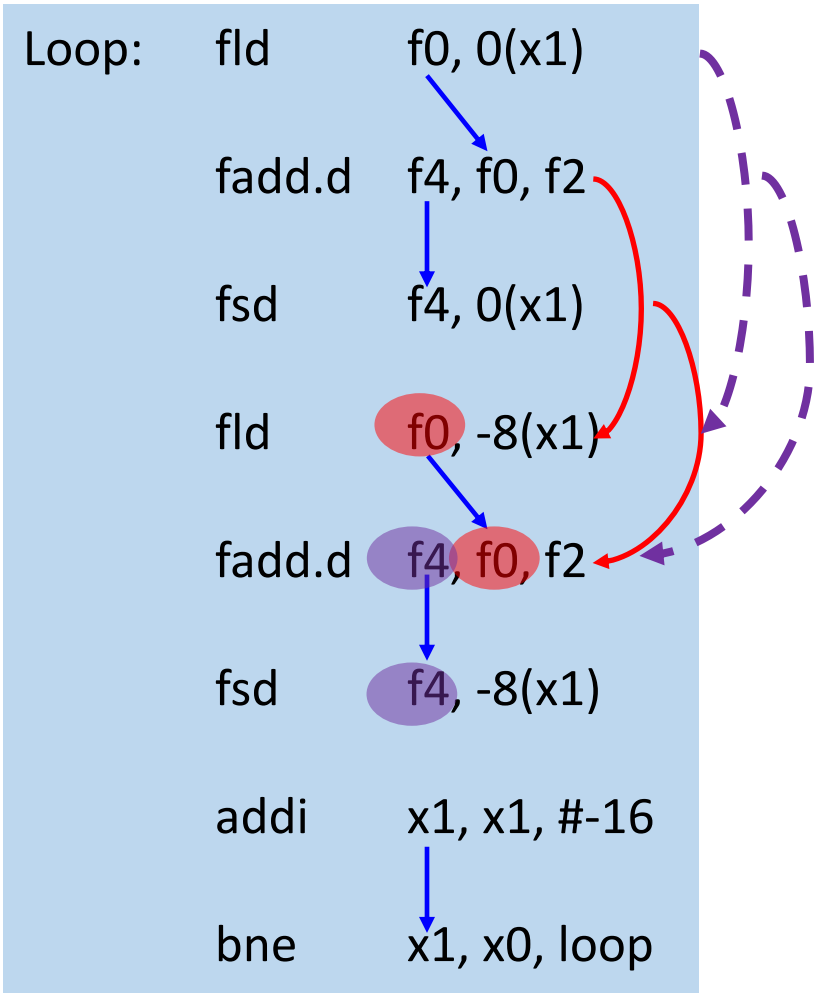
Techniques to Improve ILP

Technique	Reduces	Section
Forwarding and bypassing	Potential data hazard stalls	C.2
Simple branch scheduling and prediction	Control hazard stalls	C.2
Basic compiler pipeline scheduling	Data hazard stalls	C.2, 3.2
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences	C.7
Loop unrolling	Control hazard stalls	3.2
Advanced branch prediction	Control stalls	3.3
Dynamic scheduling with renaming	Stalls from data hazards, output dependences, and antidependences	3.4
Hardware speculation	Data hazard and control hazard stalls	3.6
Dynamic memory disambiguation	Data hazard stalls with memory	3.6
Issuing multiple instructions per cycle	Ideal CPI	3.7, 3.8
Compiler dependence analysis, software pipelining, trace scheduling	Ideal CPI, data hazard stalls	H.2, H.3
Hardware support for compiler speculation	Ideal CPI, data hazard stalls, branch hazard stalls	H.4, H.5

Types of Dependences[依赖类型]

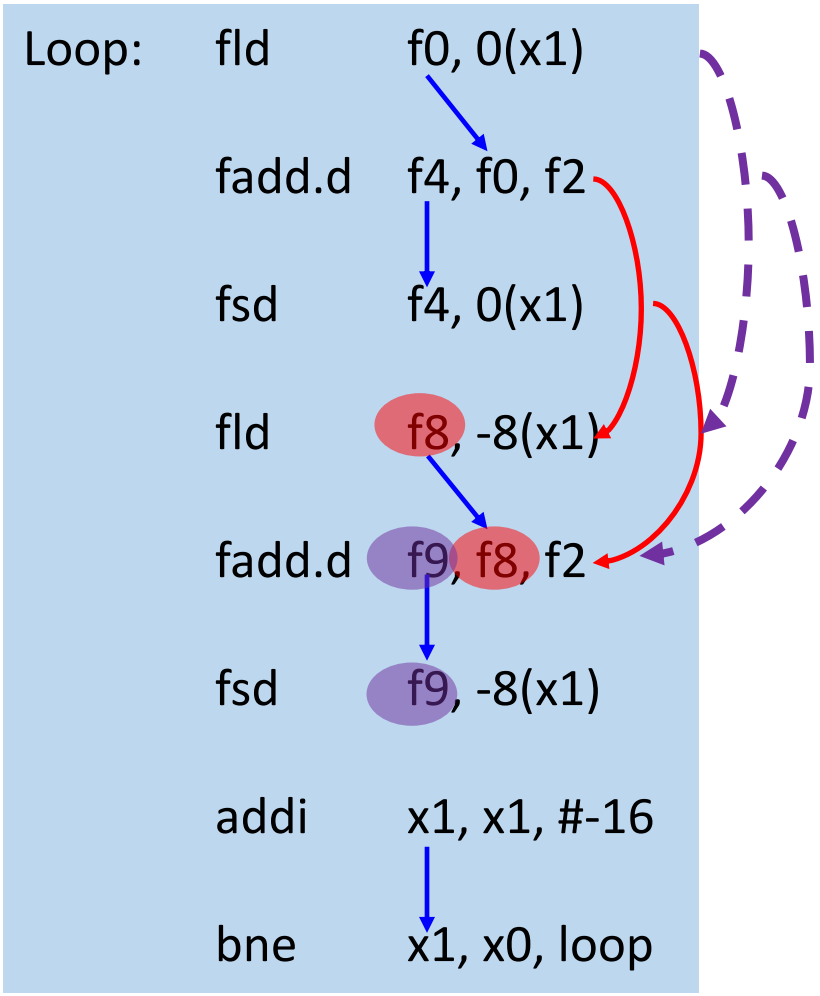
- True **data dependences**: may cause RAW hazards[数据]
  - Instruction *Q* uses data produced by instruction *P* or by an instruction which is data dependent on *P*
  - Easy to determine for registers but hard to determine for memory locations since addresses are computed dynamically
    - Example: is 100(R1) the same location as 200(R2)?
- **Name dependences**: two instructions use the same name but do not exchange data (no data dependency)[名字]
  - **Anti-dependence**[反依赖]: instruction *P* reads from a register (or memory) followed by instruction *Q* writing to that register (or memory). May cause WAR hazards
  - **Output dependence**[输出依赖]: instructions *P* and *Q* write to the same location. May cause WAW hazards.

Example



- Data dependence
  - RAW: read after write
- Anti-dependence
  - WAR: write after read
- Output dependence
  - WAW: write after write

Register Renaming[重命名]



- How to remove name dependences?
  - Rename the dependent uses of f0 and f4