Computer Architecture

计 算 机 体 系 结 构

第4讲：ISA & ILP（3）

张献伟

xianweiz.github.io

DCS3013, 9/28/2022

Quiz Questions

For remote attendees, plz email to zhangxw79@mail.sysu.edu.cn.

- Q1: list the execution stages of 'add R3, R1, R2'.

  IF, ID, EX, WB

- Q2: for pipelining, impsbl to reach ideal speedup. Why?

  Imbalanced stages, pipelining overhead

- Q3: list three differences between CISC vs. RISC

  Complex vs. reduced, differences on instructions, perf, code size, …

- Q4: explain structural hazard.

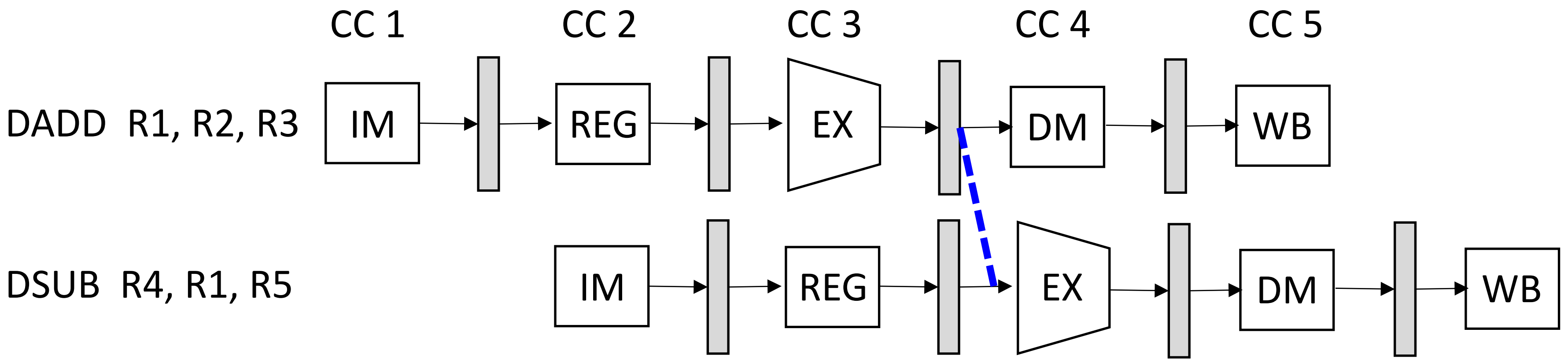
  HW cannot support some combination of instructions

- Q5: suppose a program has 90% portion that can be fully parallelized, and you have 10 CPU cores to run it. Is it possible to achieve 5x speedup? If yes, how many cores are needed?

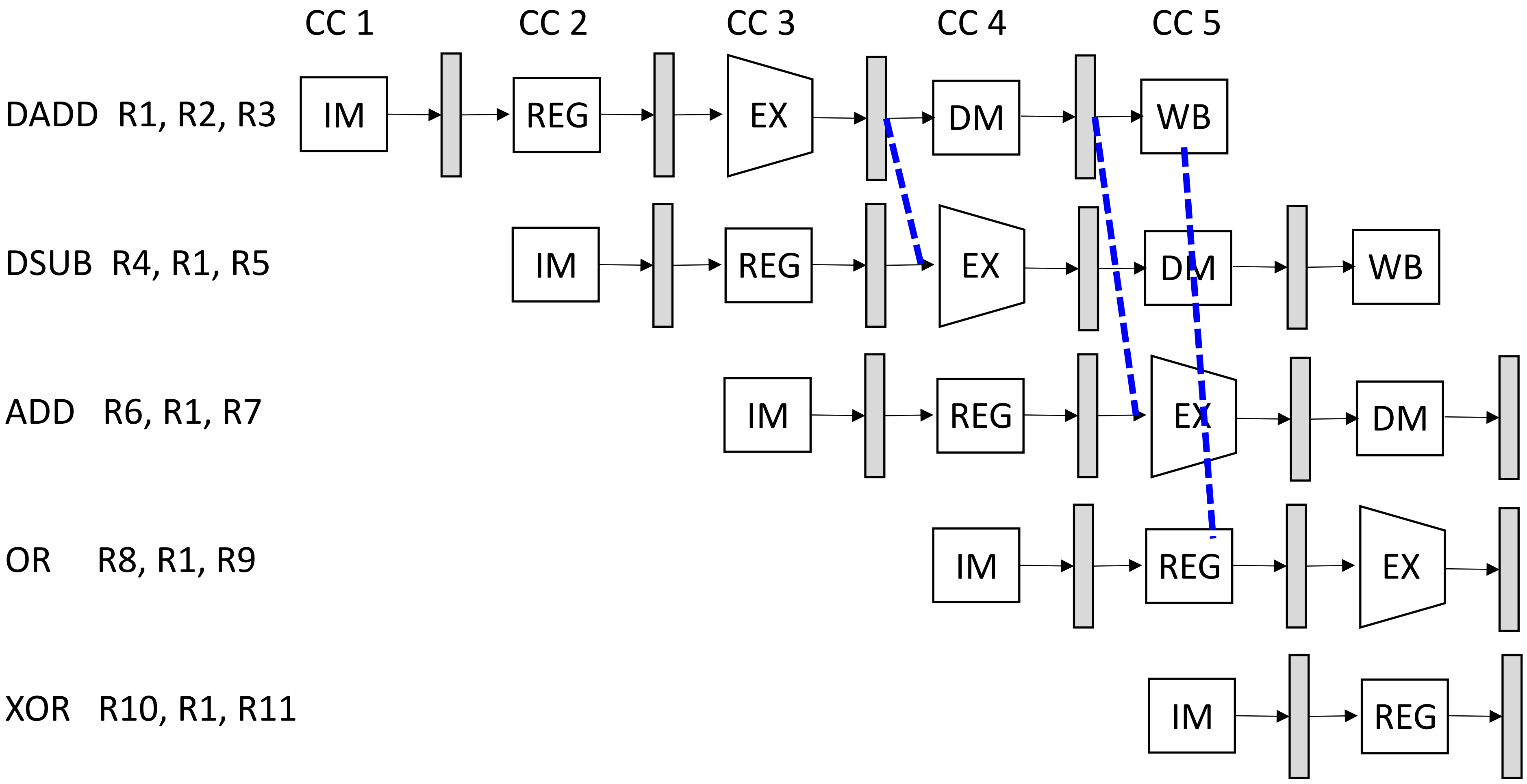
  Yes. $1/(90\%/10+10\%) = 5.26 > 5$, $N = 90\%/(1/5-10\%) = 9$

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

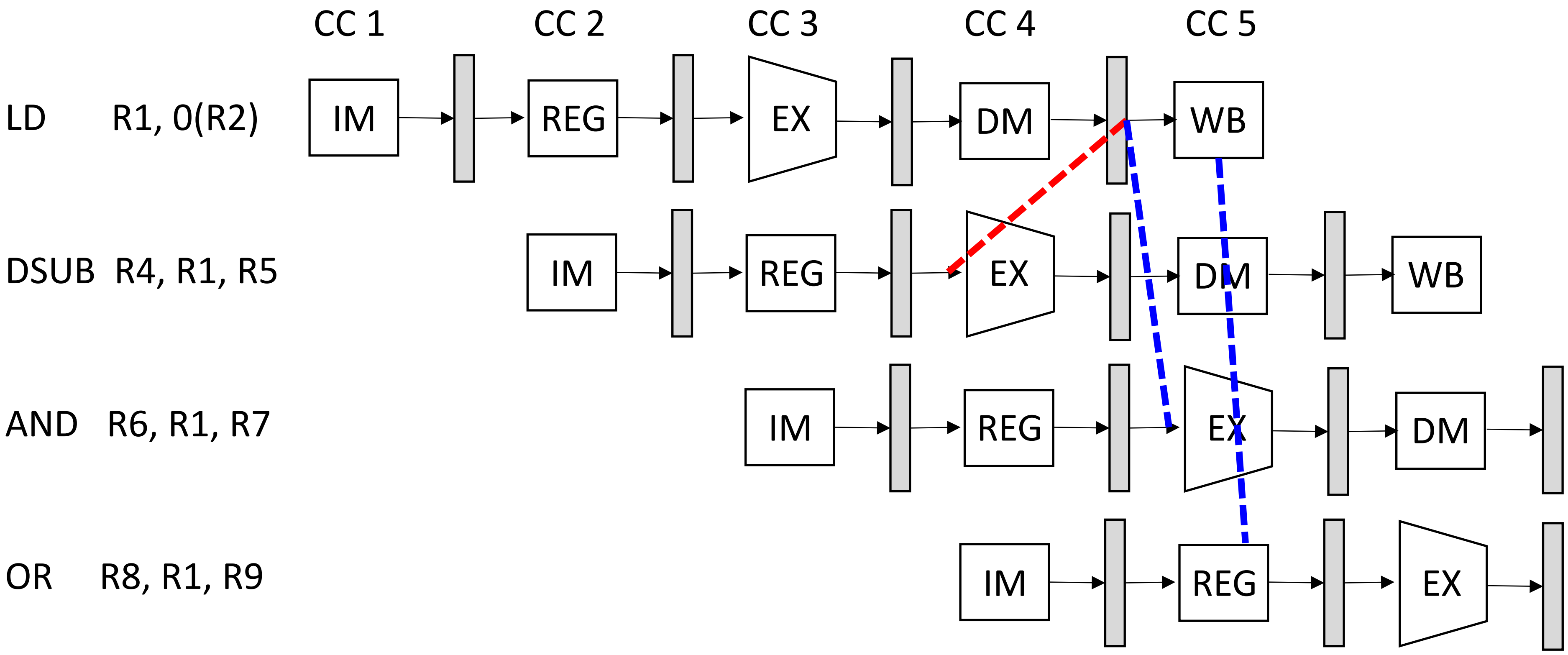


Forwarding (cont.)



- 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 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

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

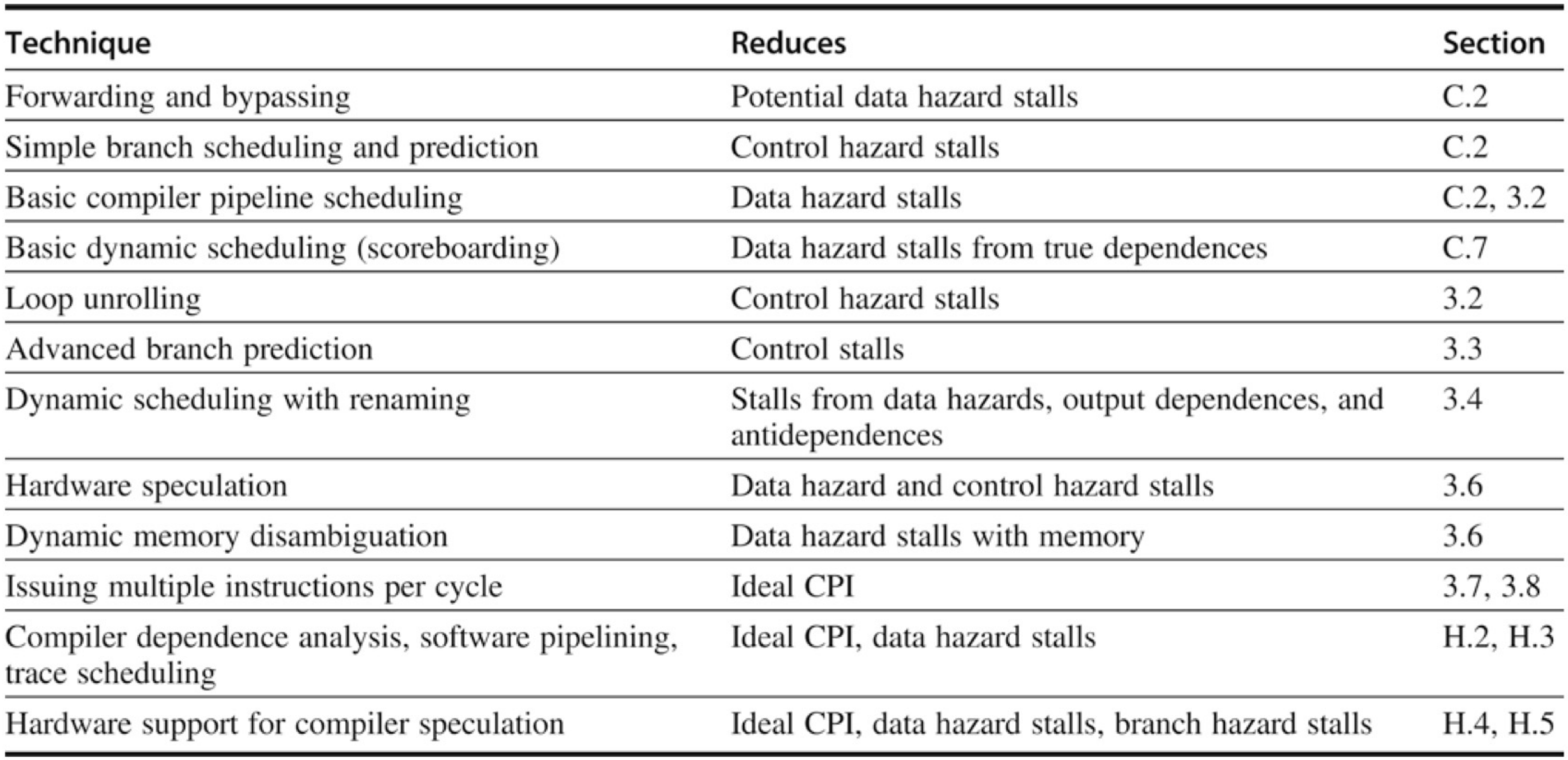
https://courses.cs.washington.edu/courses/cse471/09sp/lectures/pipeliningBasics.pdf

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

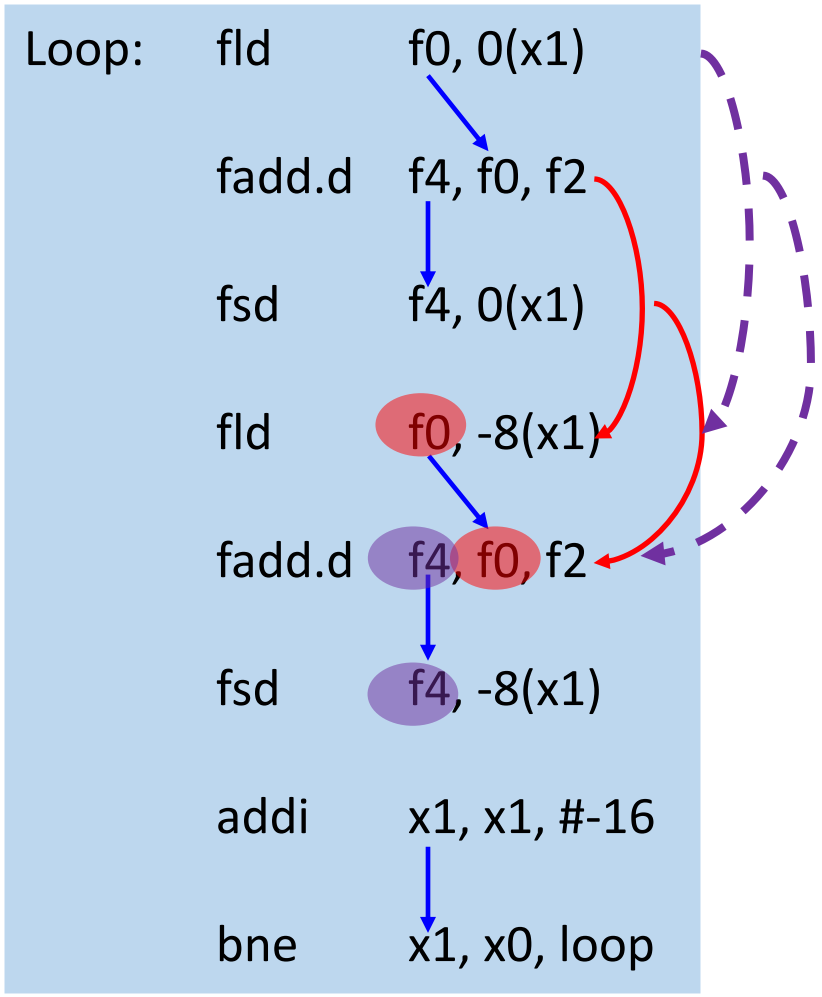


Technique	Reduces	Section
Forwarding and bypassing	Potential data hazard stalls	C.2
Simple branch scheduling and prediction	Control hazard stalls	C.2
Basic compiler pipeline scheduling	Data hazard stalls	C.2, 3.2
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences	C.7
Loop 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]

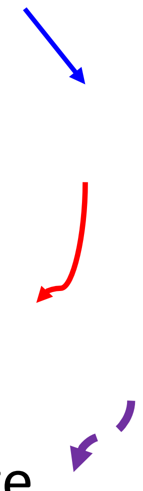
- True **data dependences**: may cause RAW hazards[数据]
  - Instruction *Q* uses data produced by instruction *P* or by an instruction which is data dependent on *P*
  - Easy to determine for registers but hard to determine for memory locations since addresses are computed dynamically
    - Example: is 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

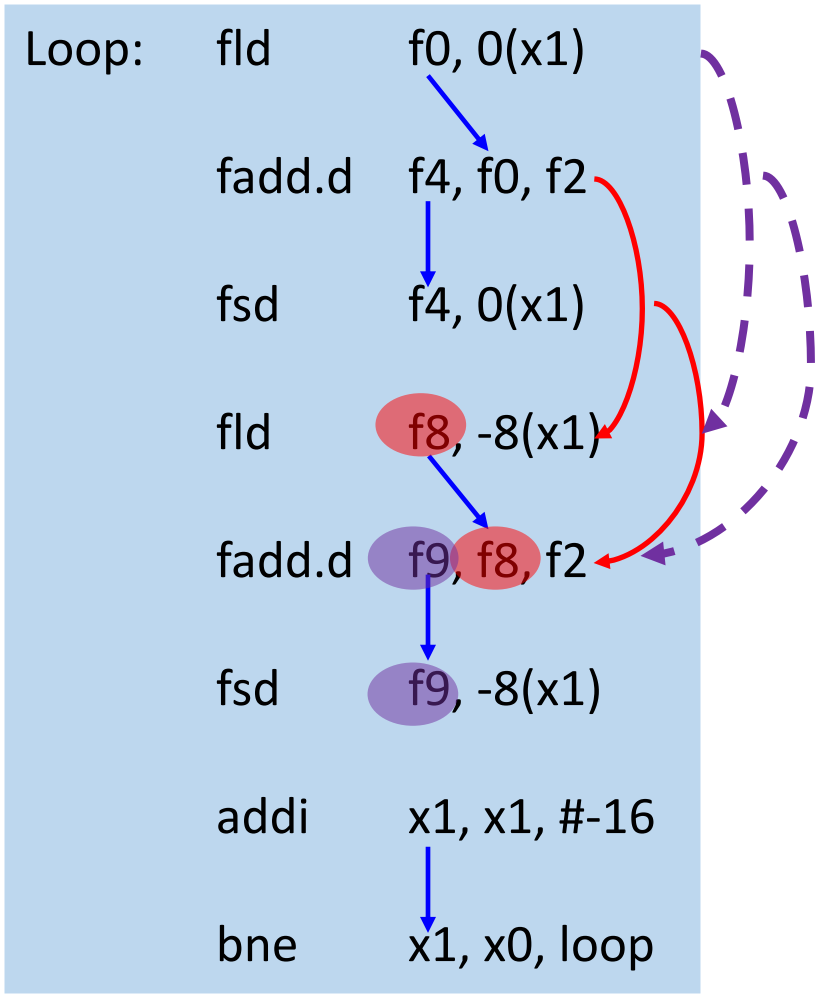


```
Loop:   fld     f0, 0(x1)
        fadd.d  f4, f0, f2
        fsd     f4, 0(x1)
        fld     f0, -8(x1)
        fadd.d  f4, f0, f2
        fsd     f4, -8(x1)
        addi    x1, x1, #-16
        bne     x1, x0, loop
```

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



Register Renaming[重命名]



```
Loop:   fld       f0, 0(x1)
        fadd.d    f4, f0, f2
        fsd       f4, 0(x1)
        fld       f8, -8(x1)
        fadd.d    f9, f8, f2
        fsd       f9, -8(x1)
        addi      x1, x1, #-16
        bne       x1, x0, loop
```

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

Control Dependences[控制依赖]

- Determine the order of instructions with respect to branches[相对分支的指令顺序]

if *P1* then *S1* ;	*S1* is control dependent on *P1* and
if *P2* then *S2* ;	*S2* is control dependent on *P2* (and P1 ??)

- An instruction that is control dependent on *P* cannot be moved to a place where it is no longer control dependent on *P*, and visa-versa[不可移动]

Example 1:

```
    add     x1, x2, x3
    beq     x4, x0, L
    sub     x1, x5, x6
L: …
     or     x7, x1, x8
```

"or" depends on the execution flow

Example 2:

```
    add     x1, x2, x3
    beq     x12, x0, skip
    sub     x4, x5, x6
    add     x5, x4, x9
skip:
     or     x7, x8, x9
```

possible to move "sub" before "beq" (if x4 is not used after skip)