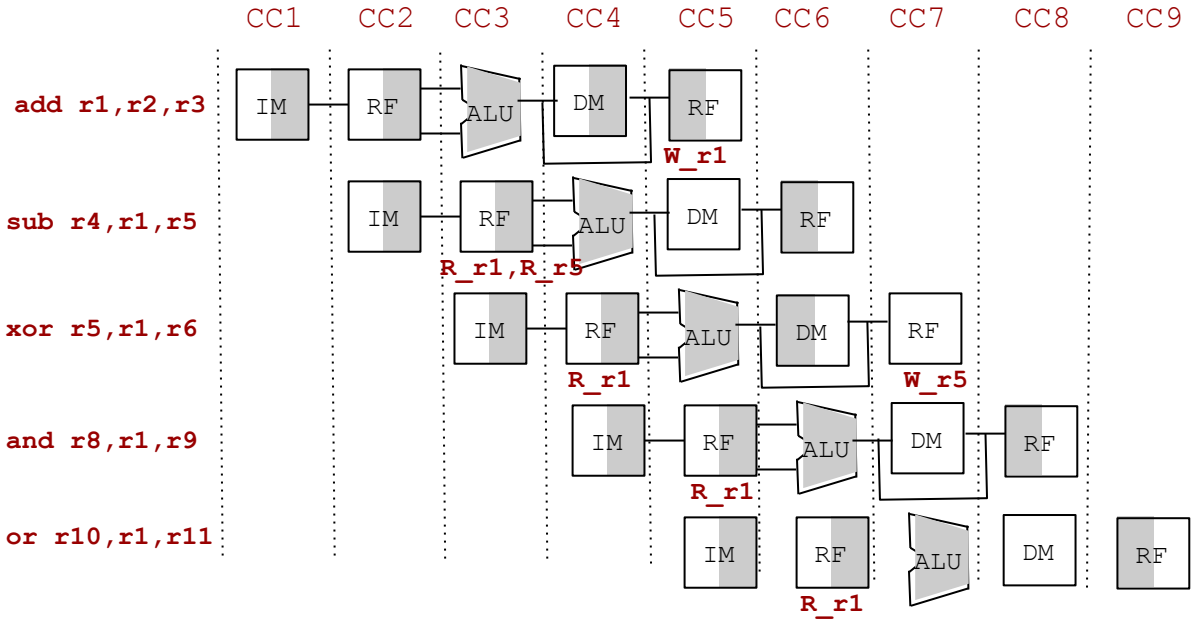
Computer Architecture

OoO Processing: Motivation

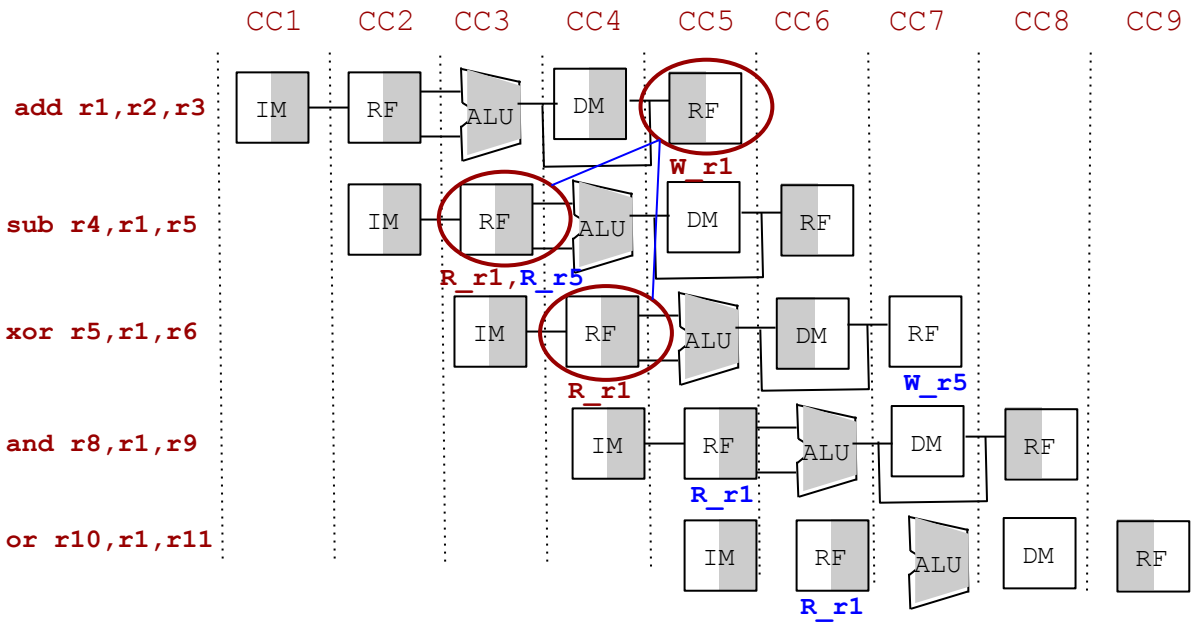
Debadatta Mishra, CSE, IITK

Data hazard (recap)



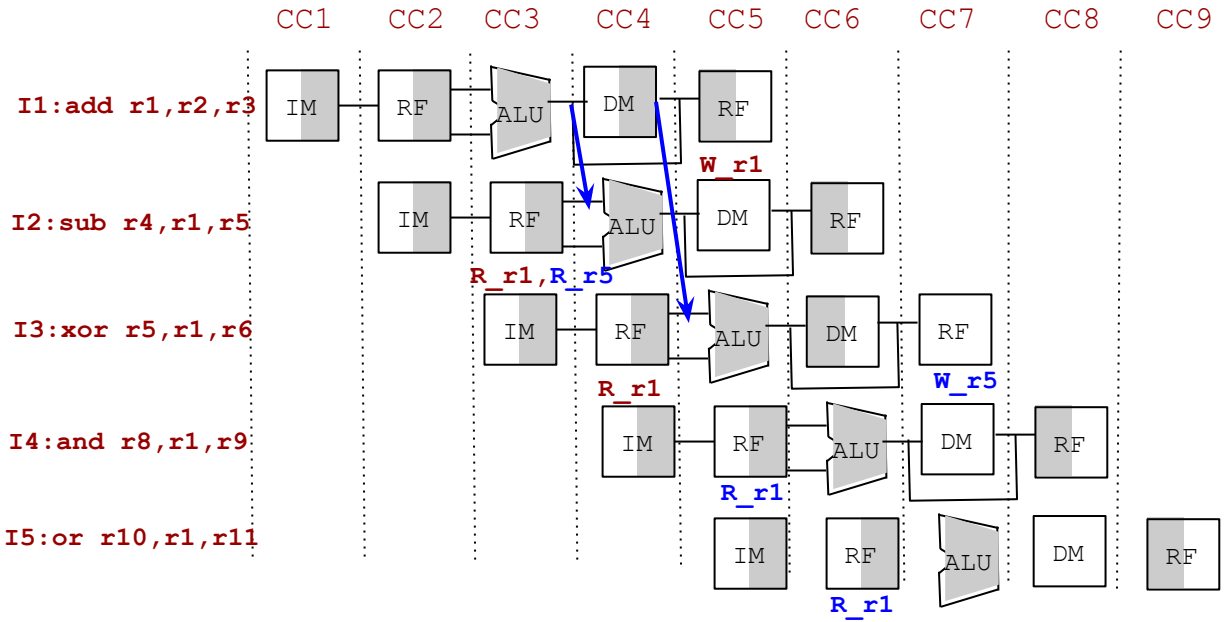
- Register **`r1`** is written in CC5 (first half of the clock) by **`add`**
- **`r1`** is read by **`sub`**,**`xor`**,**`and`**, **`or`** in the second half of CC3, CC4, CC5 and CC6, respectively
- Which instructions can not proceed?

Data hazard (recap)



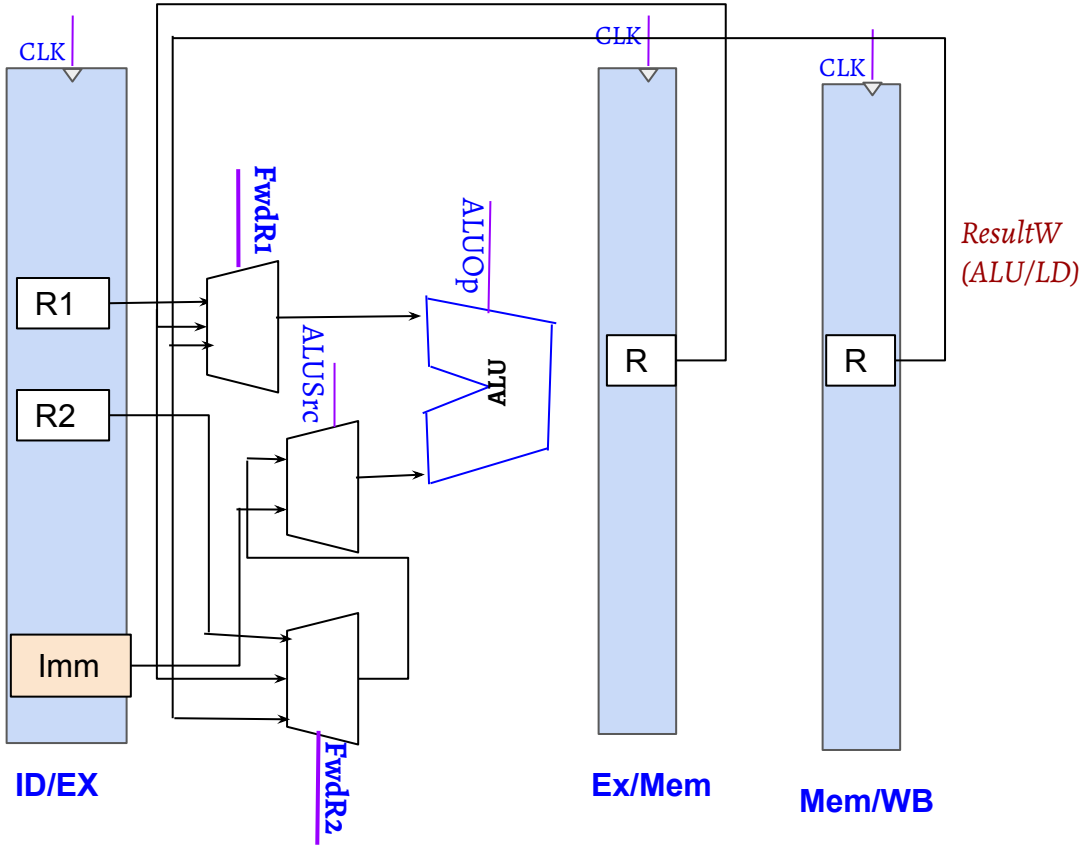
- Register **r1** is written in CC5 (first half of the clock) by **add**
- **r1** is read by **sub,xor,and, or** in the second half of CC3, CC4, CC5 and CC6, respectively
- Which instructions can not proceed? For how long?
- **sub** for 2 cycles and **xor** for 1 cycle

Data hazard: Bypass network



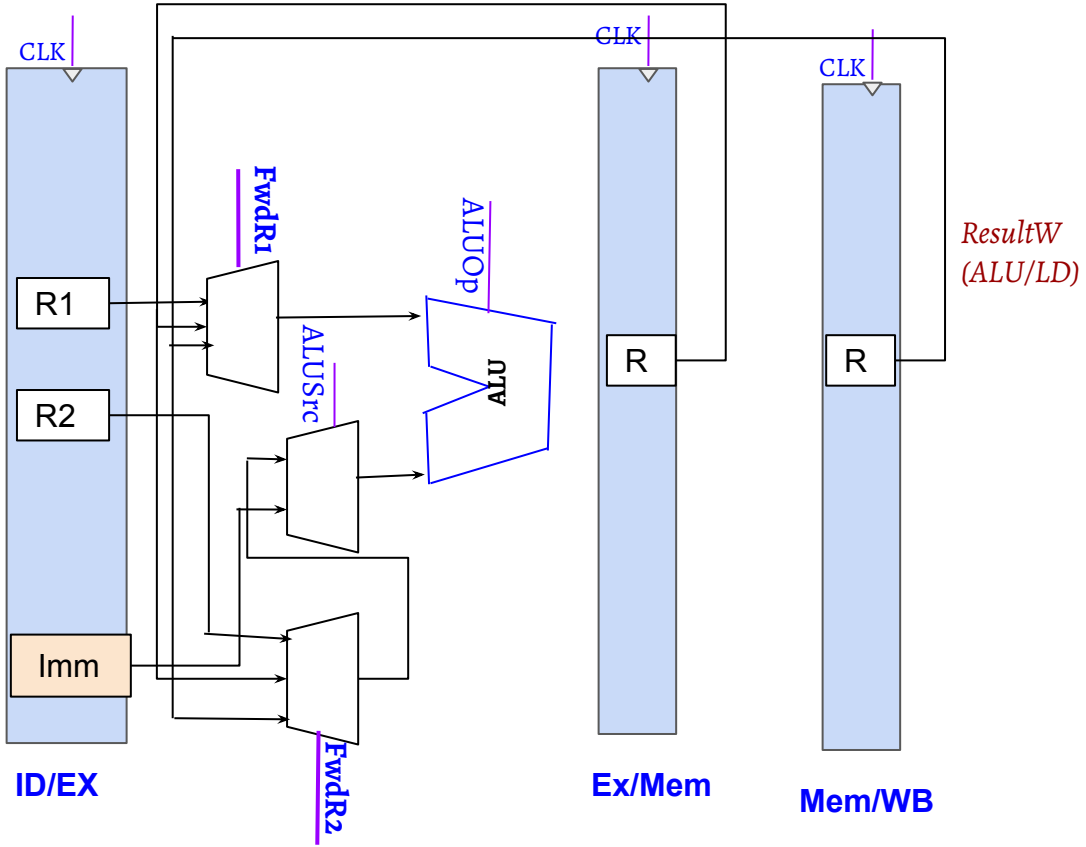
- Register **r1** value is already generated in CC3 (as ALU result)
- Can forward the value of R1 from the memory (CC4) and writeback stages (CC5) as inputs to the ALU
- How to implement?
- How to detect dependency?

Bypass Network/Forwarding



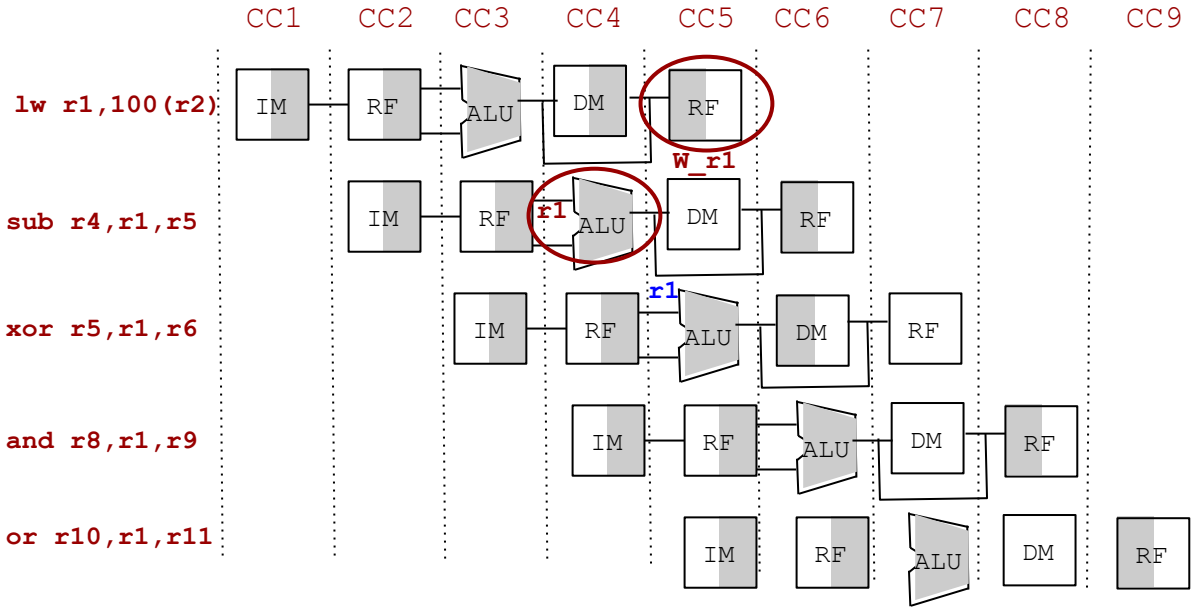
- How to implement?
- Forward the result of ALU output or write back result to the Ex stage, generate additional control signals to select them
- How to detect dependency?

Bypass Network/Forwarding



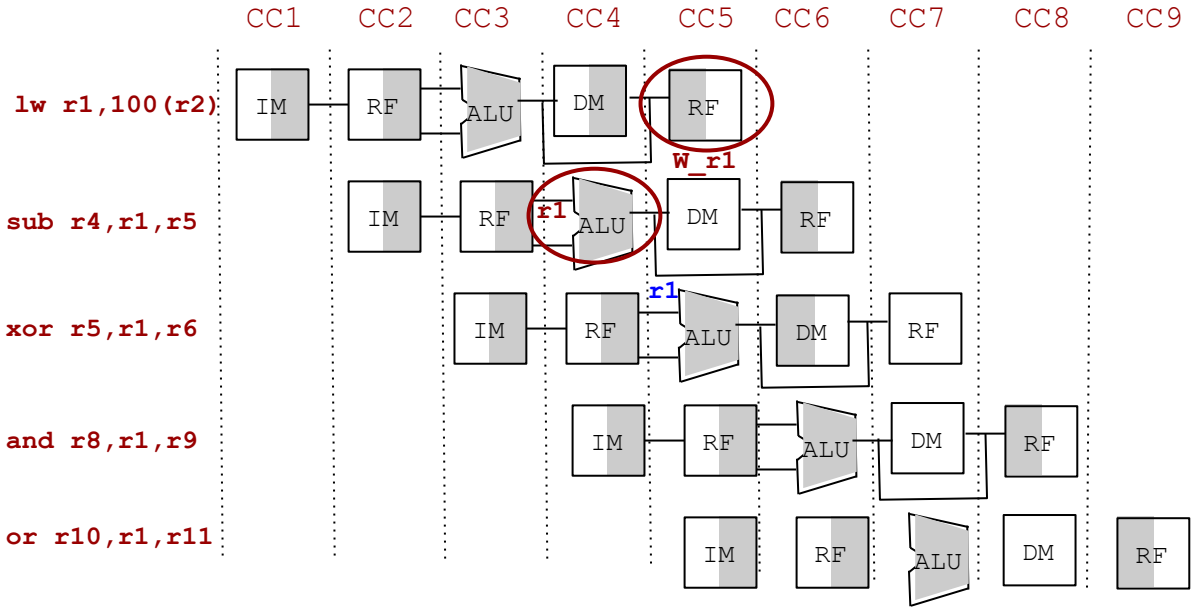
- How to implement?
- Forward the result of ALU output or write back result to the Ex stage, generate additional control signals to select them
- How to detect dependency?
- Additional control logic (hazard detection unit) generates control signals by comparing the source operand registers (in Ex stage) with the destination register (in Mem or WB stage)

Does forwarding work always?



- Even with forwarding, **`sub`** stalls for one cycle
- What else can we do?

Does forwarding work always?



- Even with forwarding, **`sub`** stalls for one cycle
- What else can we do?
- What if we execute an independent instruction after **`lw`**?

OoO Scheduling

Program Order

```
0:lw r1,100(r2)
4:sub r4,r1,r5
8:xor r5,r1,r6
12:and r8,r1,r9
16:or r10,r1,r11
20:add r3,r3,r2
```

Scheduling Order

```
0:lw r1,100(r2)
20:add r3,r3,r2
4:sub r4,r1,r5
8:xor r5,r1,r6
12:and r8,r1,r9
16:or r10,r1,r11
```

- Even with forwarding, `sub` stalls for one cycle
- What else can we do?
- What if we execute an independent instruction after `lw`?

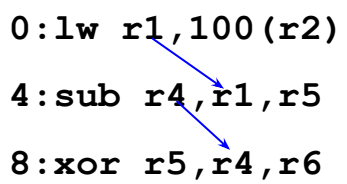
The strategy looks promising. However, some other things to worry about – dependency because of reordering, violation of program order, finding independent instructions …

Data dependance

- Data dependance: Instruction ***j*** is dependent on Instruction ***i*** if
  - Instruction ***i*** produces a results which is used by instruction ***j***
  - Instruction ***j*** is data dependent on ***k***, and instruction ***k*** is data dependent on instruction ***i***

Data dependance

- Data dependance: Instruction ***j*** is dependent on Instruction ***i*** if
  - Instruction ***i*** produces a results which is used by instruction ***j***
  - Instruction ***j*** is data dependent on ***k***, and instruction ***k*** is data dependent on instruction ***i***

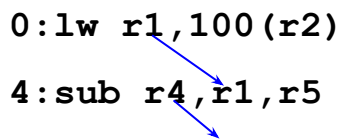


Instruction `sub` and `xor` are dependent on `lw`

How easy it is to determine data dependence?

Data dependance

- Data dependance: Instruction ***j*** is dependent on Instruction ***i*** if
  - Instruction ***i*** produces a results which is used by instruction ***j***
  - Instruction ***j*** is data dependent on ***k***, and instruction ***k*** is data dependent on instruction ***i***



Instruction `sub` and `xor` are dependent on `lw`

- Detecting register data dependence is comparatively easier than detecting memory dependence
- Example: `r1=1000 r2=900`
  `sw r5, 110(r2)` and `lw r7, 10(r1)` are data dependent

Name dependance

- Name dependance occurs when two instructions use same memory location or register but there is no data flow between the two instructions. Assuming instruction ***i*** precedes instruction ***j*** in program order,
  - Instruction ***i*** is name dependent on ***j*,** if ***j*** writes to a register/memory location which is read by instruction ***i*** (Anti-dependence)
  - Instruction ***j*** and ***i*** write to the same register/memory location (Output dependence)

```
0:lw r1,100(r2)
4:sub r4,r1,r5
8:xor r1,r1,r6
```

- Why worry about name dependence if they are not true data dependencies?
- How can we address name dependence?

Name dependance

- Name dependance occurs when two instructions use same memory location or register but there is no data flow between the two instructions. Assuming instruction ***i*** precedes instruction ***j*** in program order,
  - Instruction ***i*** is name dependent on ***j*,** if ***j*** writes to a register/memory location which is read by instruction ***i*** (Anti-dependence)
  - Instruction ***j*** and ***i*** write to the same register/memory location (Output dependence)

```
0:lw r1,100(r2)
4:sub r4,r1,r5
8:xor r1,r1,r6
```

- Why worry about name dependence if they are not true data dependencies? It matters when instructions are not executed in program order
- How can we address name dependence? For registers, register renaming can be used

Program order and hazards

- Objective of any software/hardware technique is to preserve the program order
- Dependent instructions should be executed such that the output matches the sequential execution order (exploiting maximum parallelism possible)

```
0:lw r1,100(r2)
4:sub r4,r1,r5
```

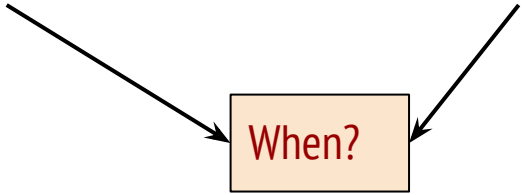
RAW
(True data dependence)

```
0:lw r1,100(r2)
4:sub r1,r2,r5
```

WAW
(Output dependence)

```
0:lw r1,100(r2)
4:sub r2,r4,r5
```

WAR
(Name dependence)



Program order and hazards

- Objective of any software/hardware technique is to preserve the program order
- Dependent instructions should be executed such that the output matches the sequential execution order (exploiting maximum parallelism possible)

```
0:lw r1,100(r2)
4:sub r4,r1,r5
```

RAW
(True data dependence)

```
0:lw r1,100(r2)
4:sub r1,r2,r5
```

WAW
(Output dependence)

```
0:lw r1,100(r2)
4:sub r2,r4,r5
```

WAR
(Name dependence)

- Instruction reordering
- Write/read is pipelined or their order is not preserved

Other reasons to reorder

- Consider a processor with a FPU which has the following stall behavior
  - Three cycles of stall for two dependent consecutive FP operations
  - One cycle stall for load of floating point followed by a FP operation
  - No stall required for two independent consecutive FP operations

Program Order

```
ldD f0,0(r1)
addD f2, f0, f1
subD f3, f2, f0
ld  r5,10(r1)
and r8,r5,r9
addD f5,f4,f7
add r3,r3,r2
```

How many stalls?

Other reasons to reorder

- Consider a processor with a FPU which has the following stall behavior
  - Three cycles of stall for two dependent consecutive FP operations
  - One cycle stall for load of floating point followed by a FP operation
  - No stall required for two independent consecutive FP operations

Program Order

```
ldD f0,0(r1)
addD f2, f0, f1
subD f3, f2, f0
ld  r5,10(r1)
and r8,r5,r9
addD f5,f4,f7
add r3,r3,r2
```

One cycle

Three cycles

One cycle

How many stalls?

Pipeline stalls for five cycles. Can we avoid that?

Other reasons to reorder

- Consider a processor with a FPU which has the following stall behavior
  - Three cycles of stall for two dependent consecutive FP operations
  - One cycle stall for load of floating point followed by a FP operation
  - No stall required for two independent consecutive FP operations

Program Order

```
ldD f0,0(r1)
addD f2, f0, f1
subD f3, f2, f0
ld  r5,10(r1)
and r8,r5,r9
addD f5,f4,f7
add r3,r3,r2
```

Scheduling Order

```
ldD f0,0(r1)
ld  r5,10(r1)
addD f2, f0, f1
addD f5,f4,f7
and r8,r5,r9
add r3,r3,r2
subD f3, f2, f0
```

How many stalls?

Pipeline stalls for five cycles. Can we avoid that? Yes, by carefully scheduling the instructions based on their data dependencies

OoO to handle non-deterministic delays

- Consider a processor with three levels of cache with following round trip latencies
  - L1 hit: 1 cycle, L2 hit: 10 cycles, L3 hit: 30 cycles, Memory: 100 cycles

Program Order

```
a = a + 2000;

addi r4,r0,#2000
lw r1,0(r2)
add r3,r1,r4
sw r3, 0(r2)
```

Assuming `r2` contains the address of variable `a`, how many cycles of stall are expected?

OoO to handle non-deterministic delays

- Consider a processor with three levels of cache with following round trip latencies
  - L1 hit: 1 cycle, L2 hit: 10 cycles, L3 hit: 30 cycles, Memory: 100 cycles

Program Order

```
a = a + 2000;

addi r4,r0,#2000
lw r1,0(r2)
add r3,r1,r4
sw r3, 0(r2)
```

Assuming `r2` contains the address of variable `a`, how many cycles of stall are expected? Depends on where **a** resides at the time of instructions execution. Most of the times non-deterministic. Any OoO technique should be designed to tackle the non-determinism.

Static scheduling

Program Order

```
for(i=999; i>=0; i-=1)
   X[i] = x[i] + s
```

```
loop: lD f0, 0(r1) ; f0 = array element
      addD f4, f0, f2 ;  f2 = s
      sD f4, 0(r1)    ; store back
      addi r1, r1, #-8 ; decrement ptr
      bne r1, r2, loop ; branch r1!=r2
```

How many stalls?

Static scheduling

Program Order

```
for(i=999; i>=0; i-=1)
   X[i] = x[i] + s
```

```
loop: lD f0, 0(r1) ; f0 = array element
      addD f4, f0, f2 ;  f2 = s
      sD f4, 0(r1)    ; store back
      addi r1, r1, #-8 ; decrement ptr
      bne r1, r2, loop ; branch r1!=r2
```

One cycle

Two cycles

one cycle

How many stalls? Four stalls in every iteration.
How much is the impact on CPI? (Homework)
Can the compiler schedule instructions better?

Static scheduling

Program Order

```
for(i=999; i>=0; i-=1)
   X[i] = x[i] + s
```

```
loop: lD f0, 0(r1) ; f0 = array element
      addi r1, r1, #-8 ; decrement ptr
      addD f4, f0, f2 ;  f2 = s
      sD f4, 8(r1)    ; store back
      addi r1, r1, #-8 ; decrement ptr
      bne r1, r2, loop ; branch r1!=r2
```

two cycles

How many stalls? Four stalls in every iteration.
How much is the impact on CPI? (Homework)
Can the compiler schedule instructions better? Yes, the loop counter can be decremented before

Static scheduling (loop unrolling)

Unrolled program

```
for(i=1024; i>=0; i-=4){
    X[i] = x[i] + s
    X[i-1] = x[i-1] + s
    X[i-2] = x[i-2] + s
    X[i-3] = x[i-3] + s
}
```

Assembly

```
loop: lD f0, 0(r1)
      ld f1, -8(r1)
      ld f3, -16(r1)
      ld f4, -24(r1)
      addD f5, f0, f2
      addD f6, f1, f2
      addD f7, f3, f2
      addD f8, f4, f2
      sD f5, 0(r1)
      sd f6, -8(r1)
      addi r1, r1, #-32
      sd f7, -16(r1)
      sd f8, -24(r1)
      bne r1, r2, loop ; branch r1!=r2
```

- Stalls can be avoided by ordering instructions across four iterations
- How to handle dynamic loops?

Static scheduling (loop unrolling)

Unrolled program

```
for(i=1024; i>=0; i-=4){
    X[i] = x[i] + s
    X[i-1] = x[i-1] + s
    X[i-2] = x[i-2] + s
    X[i-3] = x[i-3] + s
}
```

Assembly

```
loop: lD f0, 0(r1)
      ld f1, -8(r1)
      ld f3, -16(r1)
      ld f4, -24(r1)
      addD f5, f0, f2
      addD f6, f1, f2
      addD f7, f3, f2
      addD f8, f4, f2
      sD f5, 0(r1)
      sd f6, -8(r1)
      addi r1, r1, #-32
      sd f7, -16(r1)
      sd f8, -24(r1)
      bne r1, r2, loop ; branch r1!=r2
```

- Stalls can be avoided by ordering instructions across four iterations
- How to handle dynamic loops? By creating two consecutive loops– one normal loop and another unrolled loop

Why static scheduling may not be enough?

- Can not efficiently predict and handle unpredictable delays
  - A memory load may be served in different cycles at different times of execution
- Tight coupling with the underlying micro-architecture
  - Code generated for a particular machine may not be suitable for another machine
- Not all dependencies can be known at the compiler time
  - Memory dependencies, dependencies after dynamic linking

Dynamic scheduling may avoid stalls and handle unpredictable delays