BITS Pilani
Pilani Campus

Instruction Level Parallelism

K Hari Babu
Department of Computer Science & Information Systems

innovate achieve lead

BITS Pilani
Pilani Campus

Pipelining

What is Pipelining?

BITS Pilani

- A way of speeding up execution of instructions
- Key idea:
  - overlap execution of multiple instructions

The Laundry Analogy

BITS Pilani

- 3 loads of cloths to wash, dry, and fold
  - Washer takes 20 minutes
  - Dryer takes 20 minutes
  - “Folder” takes 20 minutes

One person in the laundry room

1PM	2PM	3PM
WDF		
	WDF	
		WDF

3 loads=3hrs

Pipelining:
Separate laundry room into stations each with a resource

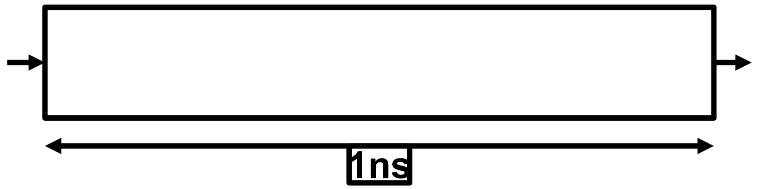
20 min	20 min	20 min
W		
W	D	
W	D	F
	D	F
		F

3 loads=60min; one load still takes 60 mins to finish, but throughput =1load/20min

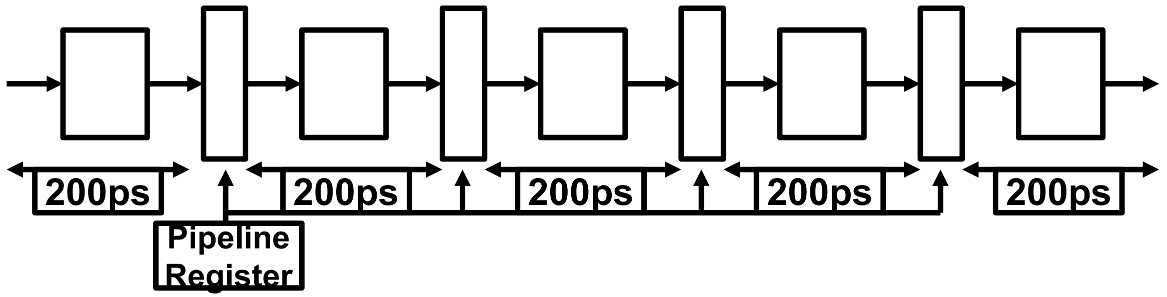
Pipelining

BITS Pilani

- Break big computation up into pieces



- Separate each piece with a pipeline register



- Pipelining increases throughput, but not latency
  - Answer available every 200ps, BUT A single computation still takes 1ns
- Limitations:
  - Computations must be divisible into stage size
  - Pipeline registers add overhead

Pipelining a Processor

BITS Pilani

- The 5 steps in instruction execution:
  1. Instruction Fetch (IF)
  2. Instruction Decode and Register Read (ID)
  3. Execution operation or calculate address (EX)
  4. Memory access (MEM)
  5. Write result into register (WB)

Example

BITS Pilani

- Consider the unpipelined processor introduced previously. Assume that it has a 1 ns clock cycle and it uses 4 cycles for ALU operations and branches, and 5 cycles for memory operations, assume that the relative frequencies of these operations are 40%, 20%, and 40%, respectively. Suppose that due to clock skew and setup, pipelining the processor adds 0.2ns of overhead to the clock. Ignoring any latency impact, how much speedup in the instruction execution rate will we gain from a pipeline?
  - Average instruction execution time
  - = 1 ns * ((40% + 20%)*4 + 40%*5)
  - = 4.4ns
  - Speedup from pipeline = Average instruction time unpiplined/Average instruction time pipelined

    = 4.4ns/1.2ns = 3.7

Pipelining

- Pros
  - Multiple instructions are being processed at same time
  - This works because stages are isolated by registers
  - Best case speedup of N (N stages)
- Cons
  - Instructions interfere with each other - hazards
  - Example: different instructions may need the same piece of hardware (e.g., memory) in same clock cycle
  - Example: instruction may require a result produced by an earlier instruction that is not yet complete

Pipeline Hazards

BITS Pilani

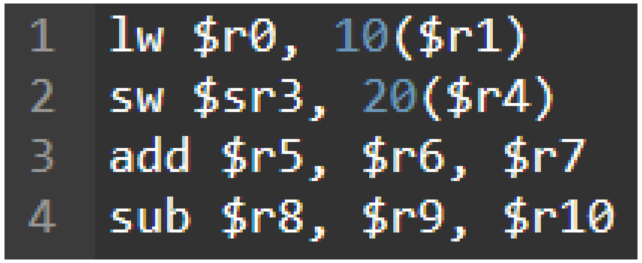
- Where one instruction cannot immediately follow another
- Types of hazards
  - Structural hazards - attempt to use the same resource by two or more instructions
  - Control hazards - attempt to make branching decisions before branch condition is evaluated
  - Data hazards - attempt to use data before it is ready
- Can always resolve hazards by waiting

Structural Hazards

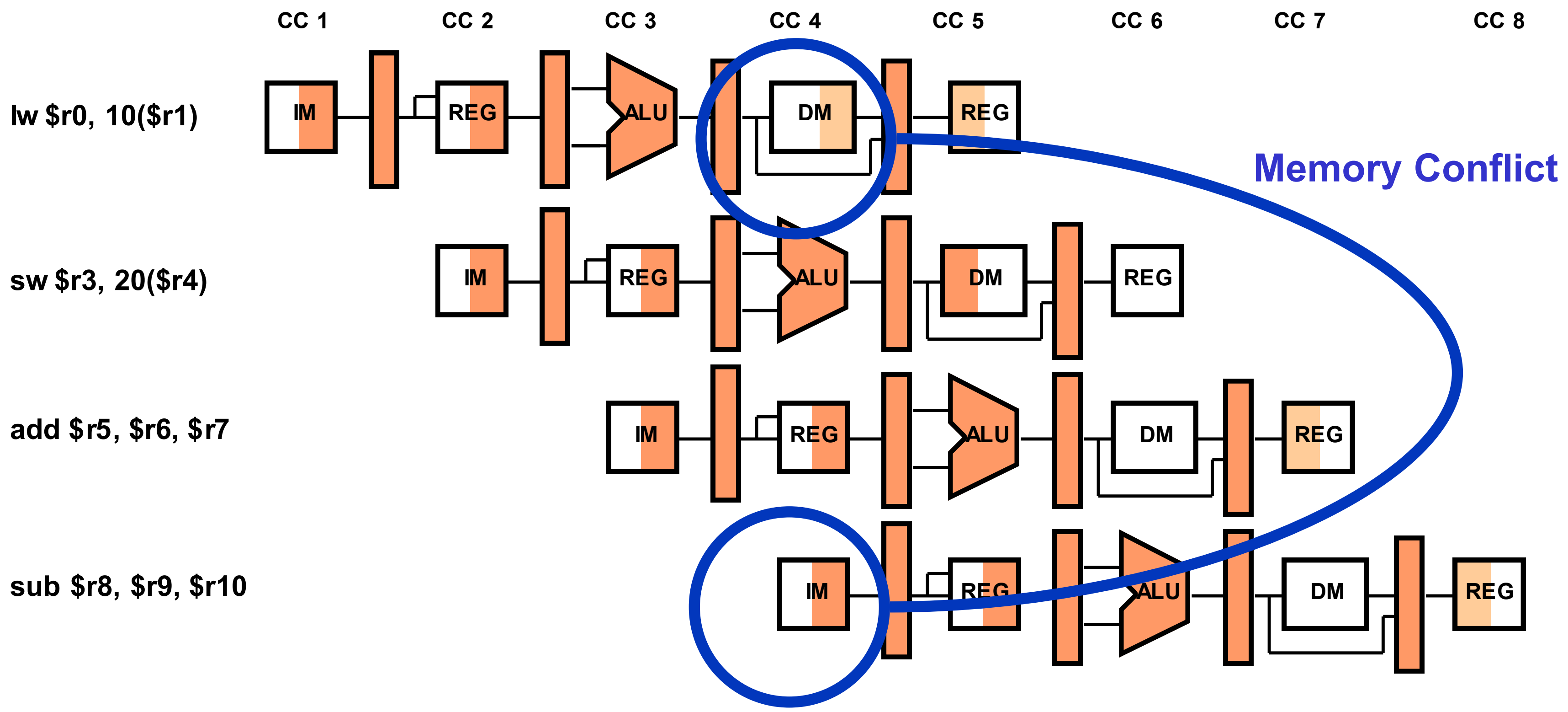
- Attempt to use the same resource by two or more instructions at the same time
- Example: Single Memory for instructions and data
  - Accessed by IF stage
  - Accessed at same time by MEM stage
- Solutions
  - Delay the second access by one clock cycle, OR
  - Provide separate memories for instructions & data
    - This is called a “Harvard Architecture”
    - Real pipelined processors have separate caches

Structural Hazards

- Consider instruction sequence:



```
lw $r0, 10($r1)
sw $sr3, 20($r4)
add $r5, $r6, $r7
sub $r8, $r9, $r10
```



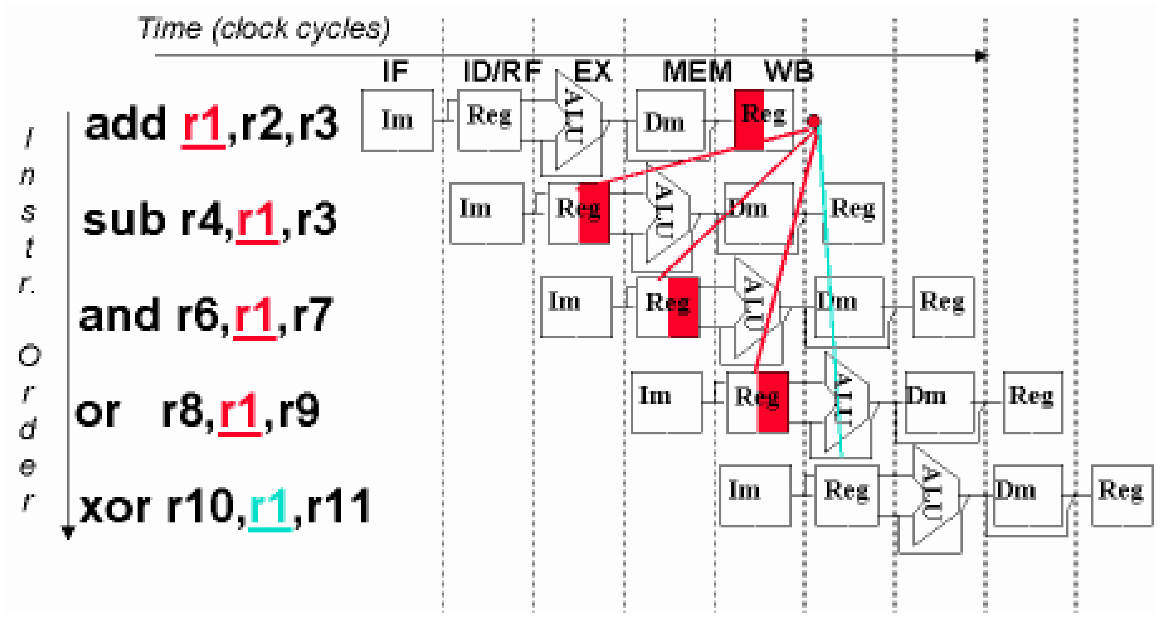
Dealing with Structural Hazards

- Stall
  - low cost, simple
  - Increases CPI
  - use for rare case since stalling has performance effect
- Pipeline hardware resource
  - useful for multi-cycle resources
  - good performance
  - sometimes complex e.g., RAM
- Replicate resource
  - good performance
  - increases cost (+ maybe interconnect delay)
  - useful for cheap resources

Data Hazards

BITS Pilani

- Data hazards occur when data is used before it is ready



Data Hazards

- Read After Write (RAW)
  - I: add r1,r2,r3
  - J: sub r4,r1,r3
  - InstrJ tries to read operand before InstrI writes it
- Caused by a "Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.
- Write After Read (WAR)
  - I: sub r4,r1,r3
  - J: add r1,r2,r3
  - K: mul r6,r1,r7
  - InstrJ tries to write operand before InstrI reads r1. Insti may get wrong operand
  - Called an "anti-dependence" by compiler writers. This results from reuse of the name "r1".

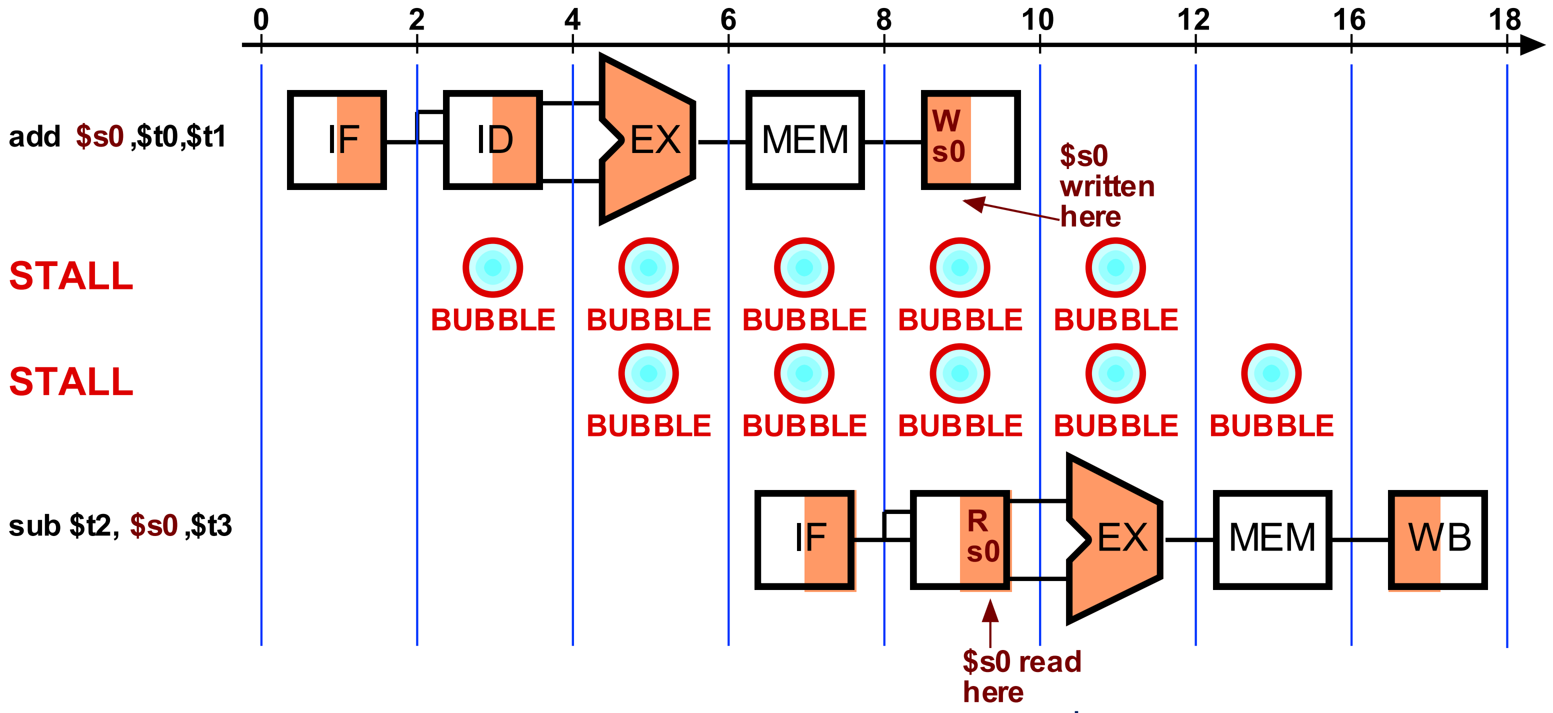
Data Hazards

- Write After Write (WAW)
  - I: sub r1,r4,r3
  - J: mul r6,r1,r7
  - K: add r1,r2,r3
  - InstrK tries to write operand before InstrI writes it
  - InstrJ may read wrong result (of InstrK not InstrI)
  - Called an “output dependence” by compiler writers
    This also results from the reuse of name “r1”.

Data Hazards

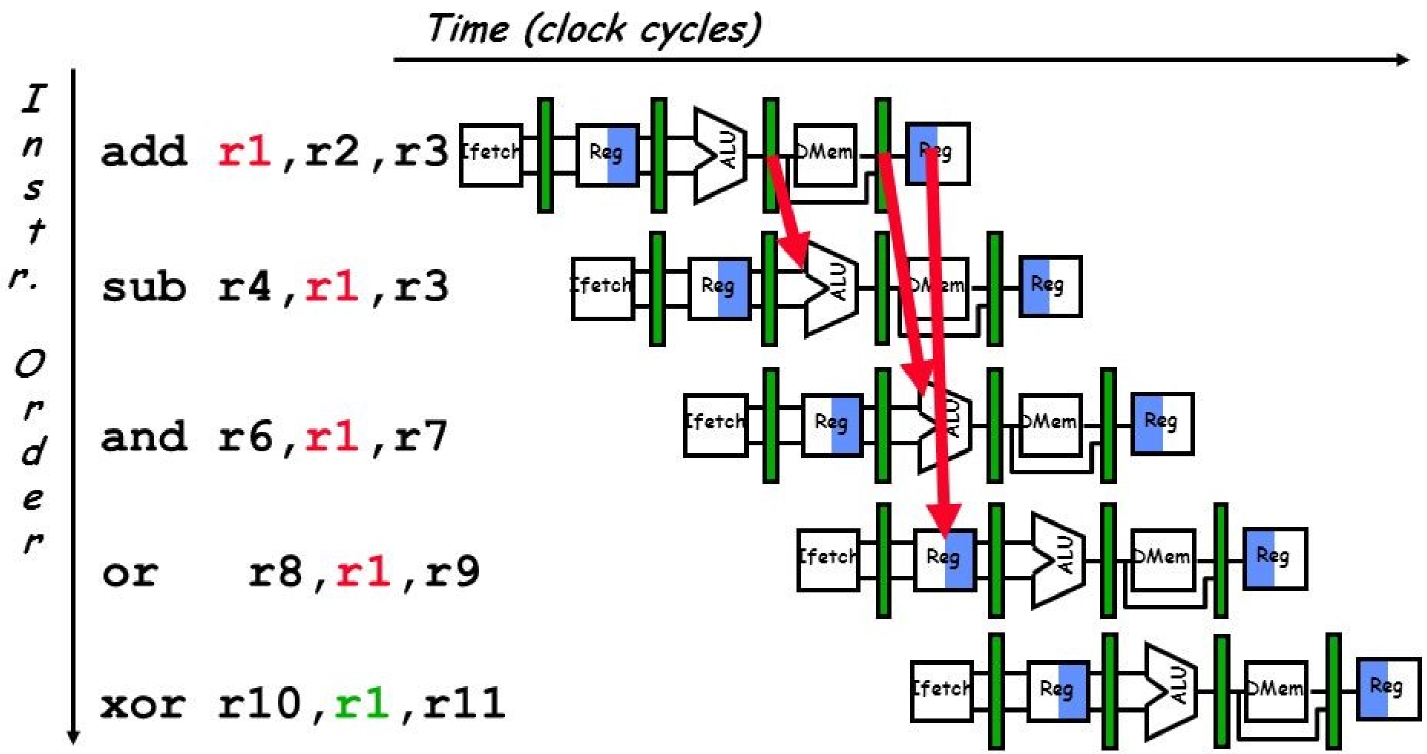
- Three types of data hazards
  - RAW (MIPS)
  - WAW (not in MIPS)
  - WAR (not in MIPS)
- Solution to RAW in MIPS
  - Stalling
  - Forwarding:
    - connect new value directly to next stage
  - Reordering

Stalling



- SUB is made to wait till ADD writes to $S0

Forwarding



- Connect data internally before it's stored
  - Assumption: The register file forwards values that are read and written during the same cycle

Data Dependences and Hazards

- Identify the true data hazard in this sequence:
  - LW $s0, 100($t0) ;$s0 = memory value
  - ADD $t2, $s0, $t3 ;$t2 = $s0 + $t3
  - SUB $S5, $s3, $s4

	1	2	3	4	5	6
LW	IF	ID	EX	MEM	WB	
ADD		IF	ID	EX	MEM	WB
SUB			IF	ID	EX	MEM

Data Dependences and Hazards

- Identify the true data hazard in this sequence:
  - LW $s0, 100($t0) ;$s0 = memory value
  - ADD $t2, $s0, $t3 ;$t2 = $s0 + $t3
  - SUB $S5, $s3, $s4

	1	2	3	4	5	6
LW	IF	ID	EX	MEM	WB	
ADD		IF	ID	EX	MEM	WB
SUB			IF	ID	EX	MEM

  - LW doesn’t write $s0 to Reg File until the end of CC5, but ADD reads $s0 from Reg File in CC3

Data Dependences and Hazards

- Identify the true data hazard in this sequence:
  - LW $s0, 100($t0) ;$s0 = memory value
  - ADD $t2, $s0, $t3 ;$t2 = $s0 + $t3
  - SUB $S5, $s3, $s4

	1	2	3	4	5	6	
LW	IF	ID	EX	MEM	WB		
ADD		IF	ID	bubbl e	EX	MEM	WB
SUB			IF		ID	EX	MEM

  - One way: handle this hazard by "stalling" the pipeline for 1 Clock Cycle (bubble). Now data can be forwarded internally to ALU

Data Dependences and Hazards

- Identify the true data hazard in this sequence:
  - LW $s0, 100($t0) ;$s0 = memory value
  - ADD $t2, $s0, $t3 ;$t2 = $s0 + $t3
  - SUB $s5, $s3, $s4

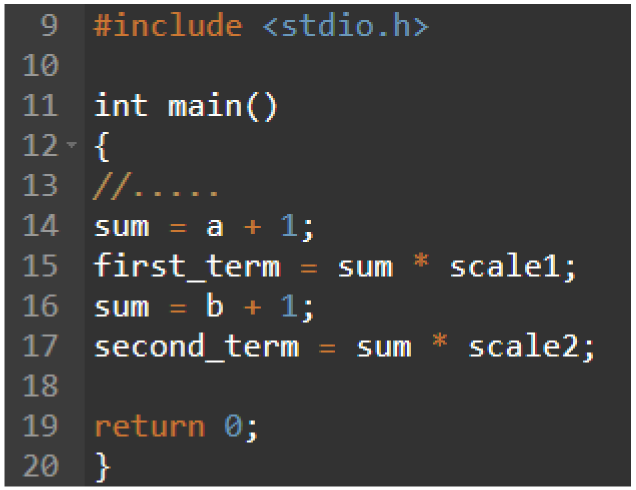
	1	2	3	4	5	6	
LW	**IF**	**ID**	**EX**	**MEM**	**WB**		
SUB		**IF**	**ID**	**EX**	**MEM**	**WB**	
ADD			**IF**	**ID**	**EX**	**MEM**	**WB**

  - Another way: handle this hazard by bringing independent instruction SUB before ADD
  - **Out of order execution**

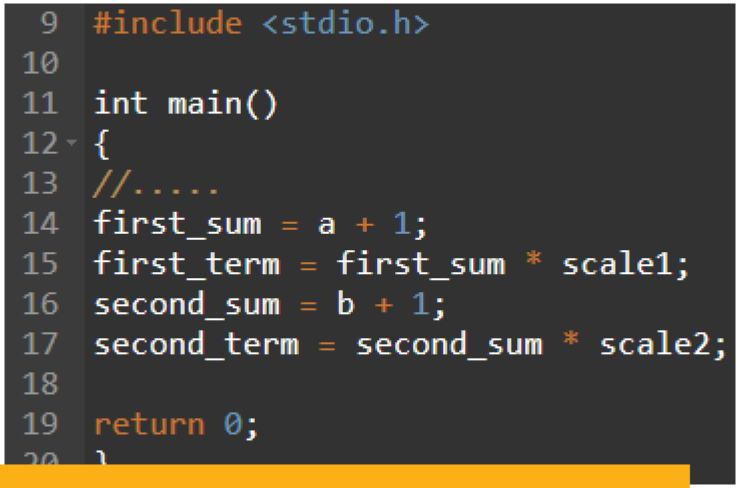
Example

BITS Pilani

- Can line 14-17 be run concurrently?
- What are data flow dependencies?
  - Read after Write (RAW)
  - Line 14 and 15
  - Line 16 and 17
- Is there any anti-dependency?
  - Write after read (WAR)
  - “sum” can’t be over-written (line no 16) without line 15 executing first
  - This dependency can be removed by using a different variable in line 16
  - Line 14,15 and 16,17 can be executed concurrently



```
 9  #include <stdio.h>
10
11  int main()
12  {
13  //.....
14  sum = a + 1;
15  first_term = sum * scale1;
16  sum = b + 1;
17  second_term = sum * scale2;
18
19  return 0;
20  }
```



```
 9  #include <stdio.h>
10
11  int main()
12  {
13  //.....
14  first_sum = a + 1;
15  first_term = first_sum * scale1;
16  second_sum = b + 1;
17  second_term = second_sum * scale2;
18
19  return 0;
```

By renaming, anti-dependency and output dependency can be removed.
At the cost of increased memory

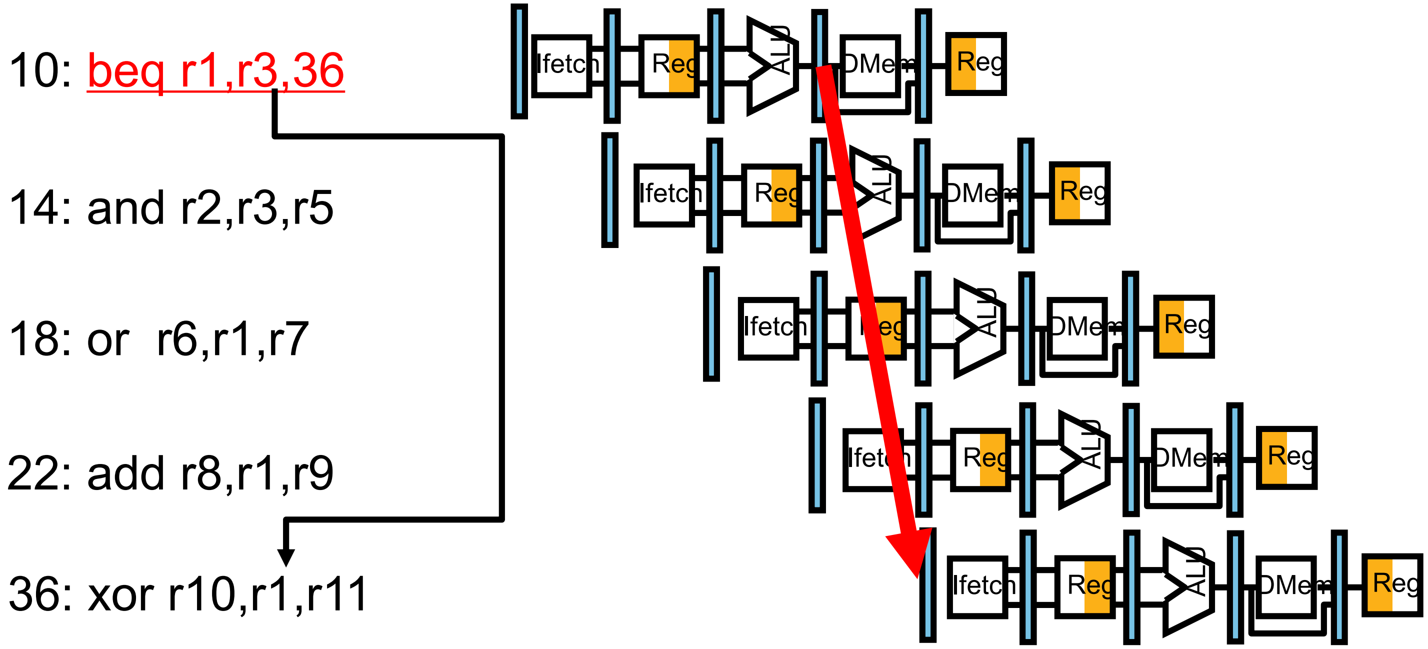
Control Hazards

- A control hazard is when we need to find the destination of a branch, and can't fetch any new instructions until we know that destination.

- A branch is either
  - Taken: PC <= PC + 4 + Immediate
  - Not Taken: PC <= PC + 4
- Just stalling for each branch is not practical
  - Common assumption: branch not taken
  - When assumption fails: flush three instructions

Control Hazards

- A branch is either
  - Taken: PC <= PC + 4 + Immediate
  - Not Taken: PC <= PC + 4
- Just stalling for each branch is not practical
  - Common assumption: branch not taken
  - When assumption fails: flush three instructions



Control Hazard Solutions

BITS Pilani

- Stall
  - stop loading instructions until result is available
- Predict
  - assume an outcome and continue fetching (undo if prediction is wrong)
  - lose cycles only on mis-prediction
- Delayed branch
  - specify in architecture that the instruction immediately following branch is always executed
- Delayed branches – code rearranged by compiler to place independent instruction after every branch (in delay slot).

```
add  $R4,$R5,$R6        beq  $R1,$R2,20
beq  $R1,$R2,20    ➡    add  $R4,$R5,$R6
lw $R3,400($R0)         lw $R3,400($R0)
```

  - Insert "NOP" (no-op) operations when can't do this (~50%)

Q&A

BITS Pilani
Pilani Campus

Thank You