Module 13: CS31003: Compilers

Code Generation for Pipeline Architecture

Course Summary

Partha Pratim Das & Pralay Mitra

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in ; pralay@cse.iitkgp.ac.in

November 08, 09, 15 & 16 2021

Module Outline

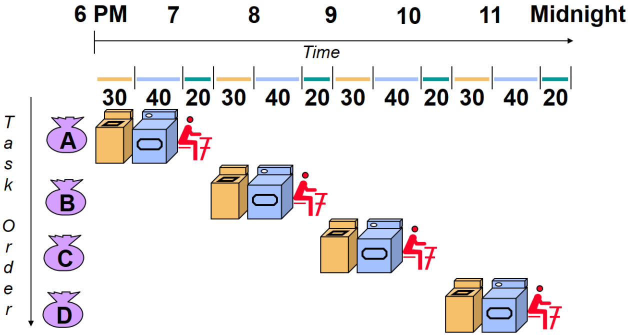
1. Pipelining
   - Laundry Analogy
   - Instruction Pipeline
   - Pipeline Hazards
     - Structural Hazard
     - Data Hazard
     - Control Hazard
2. Assignment 2
3. Control Hazards: Compiler Actions for Resolutions
4. Loop Unrolling
   - Parallel Loops
   - Unroll & Reorder
   - Software Pipeline
5. Branch Prediction
   - Static Branch Prediction
   - Dynamic Branch Prediction
6. Course Summary
7. Assignment 3

What is Pipelining?

- A way of speeding up execution of instructions
- **Key idea**: *Overlap* execution of *Multiple* instructions
- It also known as **data pipeline**
- It is a set of data processing elements *connected in series*, where the output of one element is the input of the next one
- The elements of a pipeline are *often executed in parallel* or *in time-sliced fashion*
- Some amount of *buffer storage* is often inserted between elements.
- Pipelining also refers to:
  - **Instruction pipelines**: RISC
  - *Graphics pipelines*: GPU
  - *Software pipelines*: Commands, program runs, tasks, threads, procedures, etc.
  - *HTTP pipelining*: Issuing multiple HTTP requests through the same TCP connection

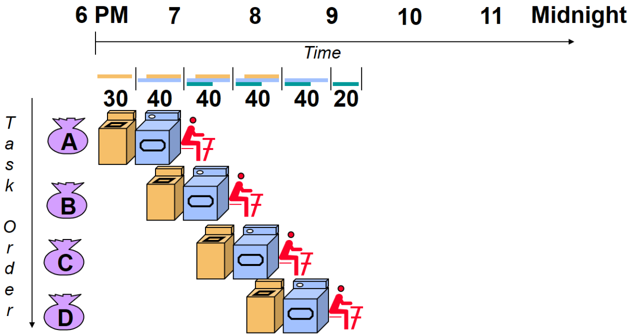
Laundry Analogy

- Ramesh, Kamala, Yusuf, Radha – each have one load of clothes to – wash, dry, and fold
  - ○ Washer takes 30 minutes
  - ○ Dryer takes 40 minutes
  - ○ Folder takes 20 minutes
- **Sequential laundry** takes 6 hours for 4 loads



Laundry Analogy

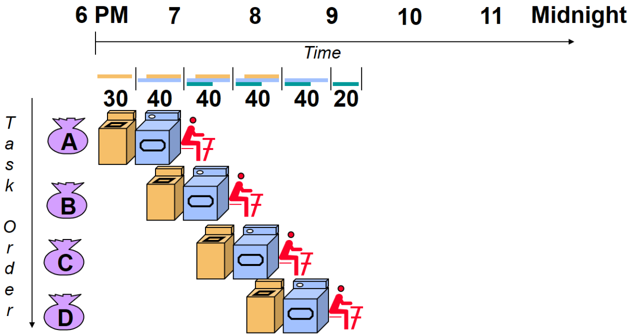
- **Pipelined laundry** takes 3.5 hours for 4 loads



- **Latency** vs. **Throughput**
  - What is the *latency* (*time delay between the cause and the effect*) in both cases? 90 min. & 90 min.
  - What is the *throughput* (*rate of production*) in both cases? 90 min. & $210/4 = 52.5$ min.
- Pipelining doesn't help **latency** *of single task*, it helps **throughput** *of entire workload*

Laundry Analogy

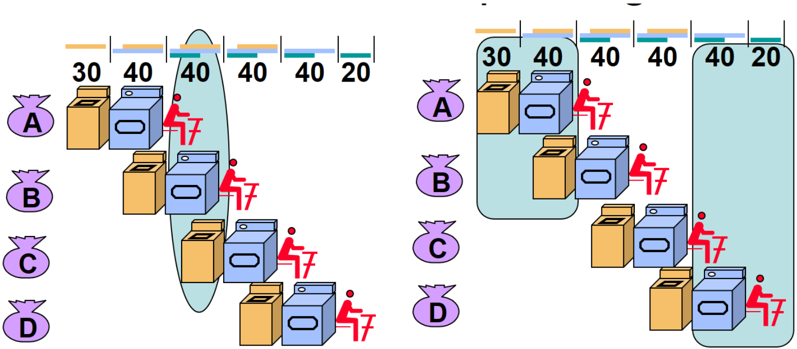
- Pipeline rate limited by **slowest** *pipeline stage*



- **Speed of Operations**
  - What is the fastest operation in the example? Folder: 20 min.
  - What is the slowest operation in the example? Dryer: 40 min.

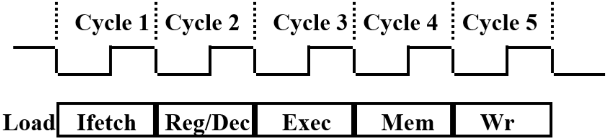
Laundry Analogy

- **Multiple tasks** *operating simultaneously using* **different resources**



- **How to increase speed-up?**
  - *Would the speedup increase if we had* ***more steps****?*: Potential Speedup = Number of pipe stages
  - **Unbalanced lengths** *of pipe stages reduces speedup*
  - *Time to* ***fill*** *pipeline and time to* ***drain*** *it reduces speedup*

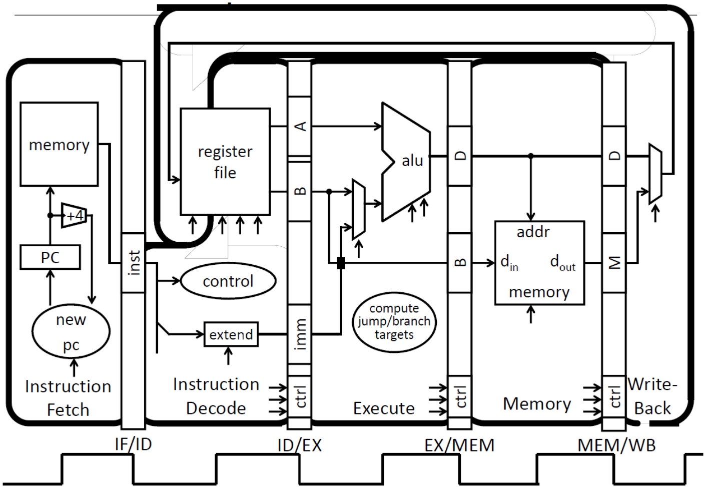
Five Stages of an Instruction



- **Ifetch (IF)**: Instruction Fetch: Read an instruction from memory
- **Reg/Dec (ID)**: Instruction Decode: Read source registers and generate control signals
- **Exec (EX)**: Compute an R-type result or a branch outcome
- **Mem (MEM)**: Read or write the data memory
- **Wr (WB)**: Write the data back to the register file: Store a result in the destination register

Instruction	Steps required				
beq	IF	ID	EX		
R-type	IF	ID	EX		WB
sw	IF	ID	EX	MEM	
lw	IF	ID	EX	MEM	WB

Pipeline Processor

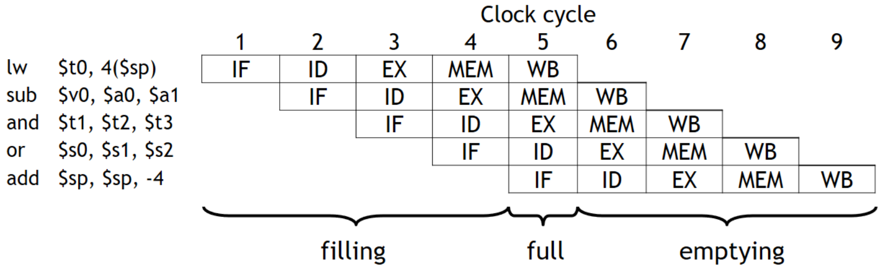


Pipelined Execution

		Clock cycle								
		1	2	3	4	5	6	7	8	9
lw	$t0, 4($sp)	IF	ID	EX	MEM	WB				
sub	$v0, $a0, $a1		IF	ID	EX	MEM	WB			
and	$t1, $t2, $t3			IF	ID	EX	MEM	WB		
or	$s0, $s1, $s2				IF	ID	EX	MEM	WB	
add	$sp, $sp, -4					IF	ID	EX	MEM	WB

- A **pipeline diagram** shows the execution of a series of instructions.
  - The instruction sequence is shown vertically, from top to bottom
  - Clock cycles are shown horizontally, from left to right
  - Each instruction is divided into its component stages
- This clearly indicates the overlapping of instructions. For example, there are three instructions active in the third cycle above:
  - The "lw" instruction is in its Execute stage
  - Simultaneously, the "sub" is in its Instruction Decode stage
  - Also, the "and" instruction is just being fetched

Pipelined Execution



		1	2	3	4	5	6	7	8	9
lw	$t0, 4($sp)	IF	ID	EX	MEM	WB				
sub	$v0, $a0, $a1		IF	ID	EX	MEM	WB			
and	$t1, $t2, $t3			IF	ID	EX	MEM	WB		
or	$s0, $s1, $s2				IF	ID	EX	MEM	WB	
add	$sp, $sp, -4					IF	ID	EX	MEM	WB

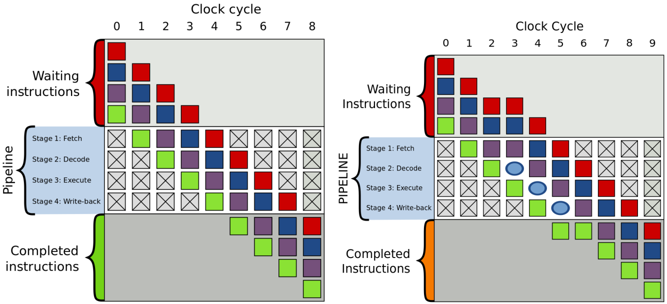
- The **pipeline depth** is the number of stages—in this case, five
- In the first four cycles here, the pipeline is **filling**, since there are unused functional units
- In cycle 5, the pipeline is **full**. Five instructions are being executed simultaneously, so all hardware units are in use
- In cycles 6-9, the pipeline is **emptying**
- **Latency**: 5 Cycles
- **Throughput**: 1 Instruction / Cycle
- **Concurrency**: 5
- **Cycles per Instruction, CPI**: 1

Pipeline Hazards and Stall

- Ideally we expect a **CPI** (*cycles per instruction*) value of 1 and a speedup equal to the number of stages in the pipeline
- But, there are a number of factors that limit this
- The problems that occur in the pipeline are called **hazards**
- Hazards that arise in the pipeline prevent the next instruction from executing during its designated clock cycle. There are three types of hazards:
  - **Structural hazards**: Hardware cannot support certain combinations of instructions (two instructions in the pipeline require the same resource)
  - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline
  - **Control hazards**: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)
- Resolution of a hazard may cause a delay in execution of an instruction. This is called a **Pipeline Stall**

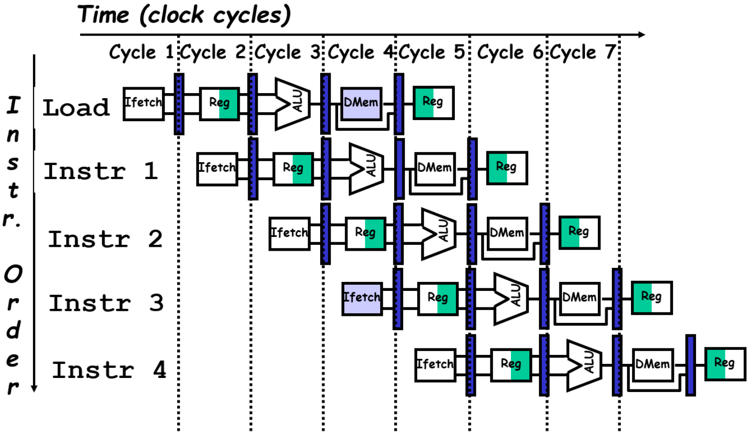
Pipeline Stall

- Resolution of a hazard may cause a delay in execution of an instruction called **Pipeline Stall**
- When a pipeline stalls (usually due to a delayed fetch of the next instruction, a bubble event has occurred where the designated stage cannot do anything (does a NOP if it must)
- The following is two executions of the same four instructions through a 4-stage pipeline but, for whatever reason, a delay in fetching of the purple instruction in cycle #2 leads to a bubble being created delaying all instructions after it as well



Structural Hazard

- When two or more different instructions want to use same hardware resource in same cycle
- For example, MEM uses the same memory port as IF as shown below



Structural Hazard: Resolution

- **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, for example, RAM
- **Replicate resource**
  - good performance
  - increases cost (+ maybe interconnect delay)
  - useful for cheap or divisible resources

Data Hazard

- These occur when at any time, there are instructions active that need to access the same data (memory or register) locations
- Where there's real trouble is when we have:
  instruction A
  instruction B
  and B manipulates (reads or writes) data before A does
- This violates the order of the instructions, since the architecture implies that A completes entirely before B is executed
- Ignoring potential data hazards can result in **race conditions** (or **race hazards**). There are three situations in which a data hazard can occur:
  - **Read after Write (RAW)**, a *true dependency*
  - **Write after Read (WAR)**, an *anti-dependency*
  - **Write after Write (WAW)**, an *output dependency*

  Read after read (RAR) is not a hazard case

Data Hazard: RAW

- Consider two instructions $\texttt{i}_1$ and $\texttt{i}_2$, with $\texttt{i}_1$ occurring before $\texttt{i}_2$ in program order
- **Read after Write (RAW)**: $\texttt{i}_2$ tries to read a source before $\texttt{i}_1$ writes to it – a situation where an instruction refers to a result that has not yet been calculated or retrieved
- This can occur because even though an instruction is executed after a prior instruction, the prior instruction has been processed only partly through the pipeline. For example:

```
i1. R2 <- R5 + R3
i2. R4 <- R2 + R3
```

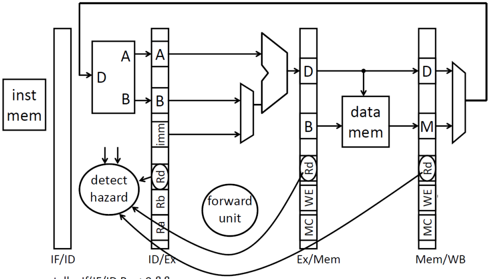
- The 1st instruction is calculating a value to be saved in register **R2**, and the 2nd is going to use this value to compute a result for register R4
- However, in a pipeline, when operands are fetched for the 2nd operation, the results from the 1st have not yet been saved, and hence a data dependency occurs
- In compiler nomenclature, a *data dependency occurs* with instruction $\texttt{i}_2$, as it is dependent on the completion of instruction $\texttt{i}_1$
- This hazard results from an actual need for communication (*data transfer*)

Data Hazard: RAW: Resolution

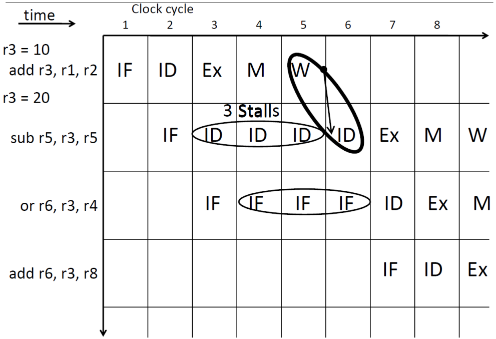
- **Simple Solution to RAW**
  - **Hardware detects RAW and stalls**
  - Assumes register written then read each cycle
    - ▷ + low cost to implement, simple
    - ▷ – reduces IPC
  - Try to minimize stalls
- **Minimizing RAW stalls**
  - **Forward / Bypass**
  - Use data before it is in the register
    - ▷ + reduces/avoids stalls
    - ▷ – complex
  - Crucial for common RAW hazards
  - Three types:
    - ▷ Forwarding from Ex/Mem registers to Ex stage (M → Ex)
    - ▷ Forwarding from Mem/WB register to Ex stage (W → Ex)
    - ▷ RegisterFile Bypass

Data Hazard: RAW: Resolution

- **Hardware detects RAW and stalls**



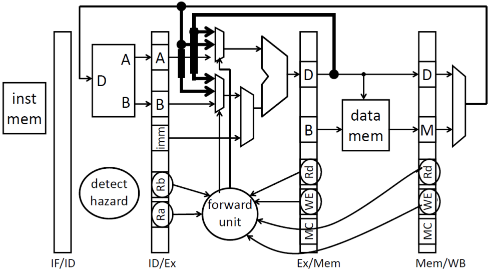
stall = If(IF/ID.Ra ≠ 0 &&
(IF/ID.Ra == ID/Ex.Rd
IF/ID.Ra == Ex/M.Rd
IF/ID.Ra == M/W.Rd))



	1	2	3	4	5	6	7	8	
r3 = 10									
add r3, r1, r2	IF	ID	Ex	M	W				
r3 = 20									
sub r5, r3, r5		IF	ID	ID	ID	ID	Ex	M	W
or r6, r3, r4			IF	IF	IF	IF	ID	Ex	M
add r6, r3, r8							IF	ID	Ex

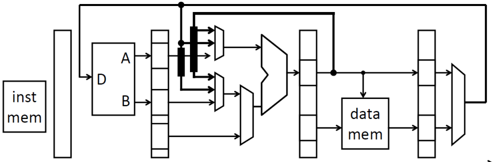
Data Hazard: RAW: Resolution

- **Forward / Bypass: Forwarding from Ex/Mem registers to Ex stage (M → Ex)**



Three types of forwarding/bypass

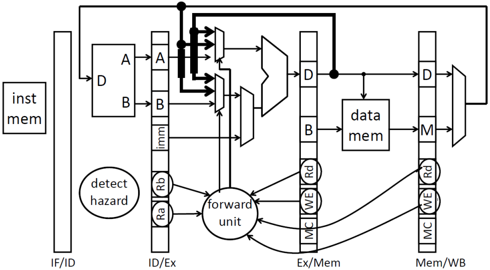
- Forwarding from Ex/Mem registers to Ex stage (M→Ex)
- Forwarding from Mem/WB register to Ex stage (W → Ex)
- RegisterFile Bypass



add r3, r1, r2	IF	ID	Ex	M	W			
sub r5, r3, r1		IF	ID	Ex	M	W		
or r6, r3, r4			IF	ID	Ex	M	W	
add r6, r3, r8				IF	ID	Ex	M	W

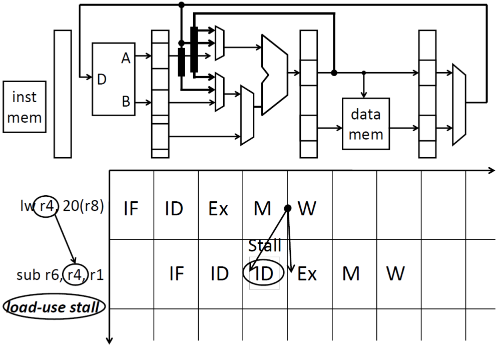
Data Hazard: RAW: Resolution

- **Forward / Bypass: Forwarding from Mem/WB register to Ex stage (W → Ex)**



Three types of forwarding/bypass

- Forwarding from Ex/Mem registers to Ex stage (M→Ex)
- Forwarding from Mem/WB register to Ex stage (W → Ex)
- RegisterFile Bypass



Data Hazard: RAW: Resolution

- RAW can be fully or partially avoided by instruction scheduling (movement) by the compiler

Before Scheduling

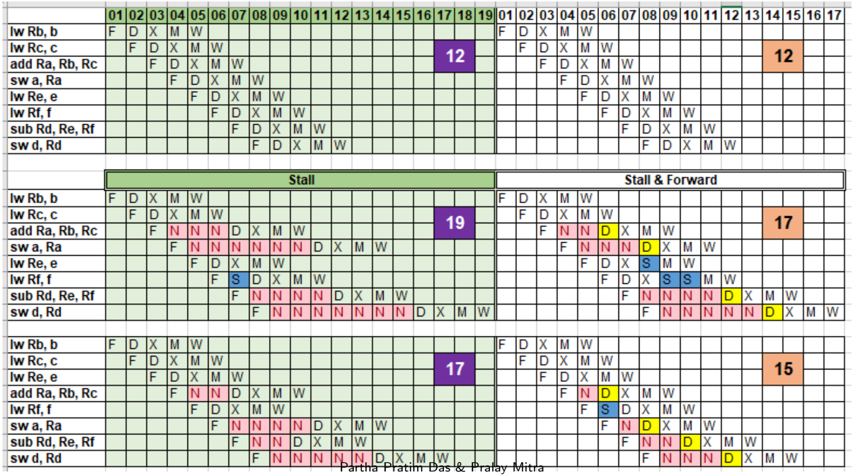
```
lw Rb, b        // code sequence for a = b+c before scheduling
lw Rc, c
Add Ra, Rb, Rc  // stall
sw a, Ra
lw Re, e        // code sequence for d = e-f before scheduling
lw Rf, f
sub Rd, Re, Rf  // stall
sw d, Rd
```

After Scheduling

```
lw Rb, b
lw Rc, c
lw Re, e
Add Ra, Rb, Rc
lw Rf, f
sw a, Ra
sub Rd, Re, Rf
sw d, Rd
```

Data Hazard: RAW: Resolution

- NOP's for data hazard are shown as "N" and NOP's for structural hazard are shown as "S"
- "D" in yellow marks forwarded data. Here we have assumed forwarding to ID only. It may be forwarded to EXE too in some cases shortening the schedules further



	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
lw Rb, b	F	D	X	M	W														
lw Rc, c		F	D	X	M	W											12		
add Ra, Rb, Rc			F	D	X	M	W												
sw a, Ra				F	D	X	M	W											
lw Re, e					F	D	X	M	W										
lw Rf, f						F	D	X	M	W									
sub Rd, Re, Rf							F	D	X	M	W								
sw d, Rd								F	D	X	M	W							

	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17
lw Rb, b	F	D	X	M	W												
lw Rc, c		F	D	X	M	W								12			
add Ra, Rb, Rc			F	D	X	M	W										
sw a, Ra				F	D	X	M	W									
lw Re, e					F	D	X	M	W								
lw Rf, f						F	D	X	M	W							
sub Rd, Re, Rf							F	D	X	M	W						
sw d, Rd								F	D	X	M	W					

Stall

	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
lw Rb, b	F	D	X	M	W														
lw Rc, c		F	D	X	M	W											19		
add Ra, Rb, Rc			F	N	N	N	D	X	M	W									
sw a, Ra				F	N	N	N	N	N	N	D	X	M	W					
lw Re, e					F	D	X	M	W										
lw Rf, f						F	S	D	X	M	W								
sub Rd, Re, Rf							F	N	N	N	N	D	X	M	W				
sw d, Rd								F	N	N	N	N	N	N	N	D	X	M	W

Stall & Forward

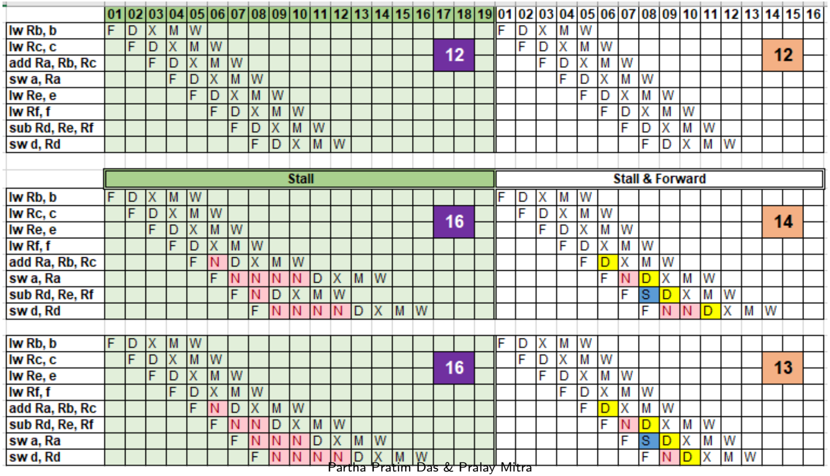
	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17
lw Rb, b	F	D	X	M	W												
lw Rc, c		F	D	X	M	W								17			
add Ra, Rb, Rc			F	N	N	D	X	M	W								
sw a, Ra				F	N	N	N	D	X	M	W						
lw Re, e					F	D	X	S	M	W							
lw Rf, f						F	D	X	S	S	M	W					
sub Rd, Re, Rf							F	N	N	N	N	D	X	M	W		
sw d, Rd								F	N	N	N	N	N	D	X	M	W

	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
lw Rb, b	F	D	X	M	W														
lw Rc, c		F	D	X	M	W											17		
lw Re, e			F	D	X	M	W												
add Ra, Rb, Rc				F	N	N	D	X	M	W									
lw Rf, f					F	D	X	M	W										
sw a, Ra						F	N	N	N	N	D	X	M	W					
sub Rd, Re, Rf							F	N	N	D	X	M	W						
sw d, Rd								F	N	N	N	N	N	D	X	M	W		

	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17
lw Rb, b	F	D	X	M	W												
lw Rc, c		F	D	X	M	W								15			
lw Re, e			F	D	X	M	W										
add Ra, Rb, Rc				F	N	D	X	M	W								
lw Rf, f					F	S	D	X	M	W							
sw a, Ra						F	N	D	X	M	W						
sub Rd, Re, Rf							F	N	N	D	X	M	W				
sw d, Rd								F	N	N	N	D	X	M	W		

Data Hazard: RAW: Resolution

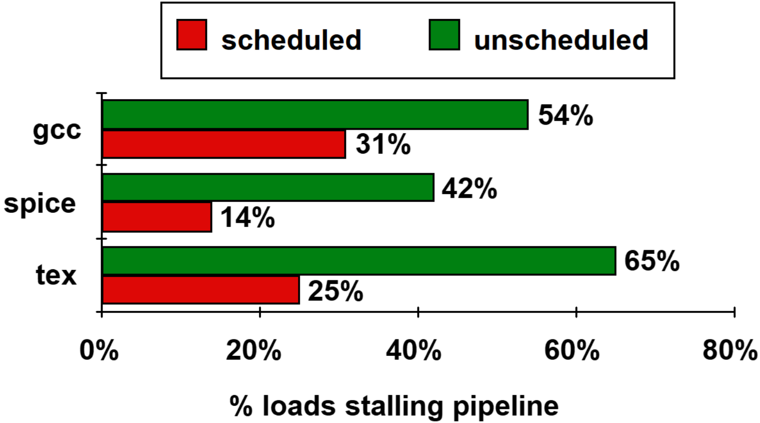
- NOP's for data hazard are shown as "N" and NOP's for structural hazard are shown as "S"
- "D" in yellow marks forwarded data. Here we have assumed forwarding to ID only. It may be forwarded to EXE too in some cases shortening the schedules further



	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16
lw Rb, b	F	D	X	M	W															F	D	X	M	W											
lw Rc, c		F	D	X	M	W											12				F	D	X	M	W								12		
add Ra, Rb, Rc			F	D	X	M	W															F	D	X	M	W									
sw a, Ra				F	D	X	M	W															F	D	X	M	W								
lw Re, e					F	D	X	M	W															F	D	X	M	W							
lw Rf, f						F	D	X	M	W															F	D	X	M	W						
sub Rd, Re, Rf							F	D	X	M	W															F	D	X	M	W					
sw d, Rd								F	D	X	M	W															F	D	X	M	W				
	Stall																			Stall & Forward															
lw Rb, b	F	D	X	M	W															F	D	X	M	W											
lw Rc, c		F	D	X	M	W											16				F	D	X	M	W								14		
lw Re, e			F	D	X	M	W															F	D	X	M	W									
lw Rf, f				F	D	X	M	W															F	D	X	M	W								
add Ra, Rb, Rc					F	N	D	X	M	W														F	D	X	M	W							
sw a, Ra						F	N	N	N	N	D	X	M	W											F	N	D	X	M	W					
sub Rd, Re, Rf							F	N	D	X	M	W														F	S	D	X	M	W				
sw d, Rd								F	N	N	N	N	D	X	M	W											F	N	N	D	X	M	W		
lw Rb, b	F	D	X	M	W															F	D	X	M	W											
lw Rc, c		F	D	X	M	W											16				F	D	X	M	W								13		
lw Re, e			F	D	X	M	W															F	D	X	M	W									
lw Rf, f				F	D	X	M	W															F	D	X	M	W								
add Ra, Rb, Rc					F	N	D	X	M	W														F	D	X	M	W							
sub Rd, Re, Rf						F	N	N	D	X	M	W													F	N	D	X	M	W					
sw a, Ra							F	N	N	N	D	X	M	W												F	S	D	X	M	W				
sw d, Rd								F	N	N	N	N	D	X	M	W											F	N	D	X	M	W			

Data Hazard: RAW: Stats on Scheduling

- Here are some example stats for benefits of scheduling to avoid RAW



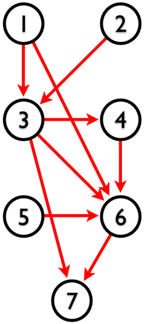
Data Hazard: RAW: Resolution: Instruction Scheduling

- We would like to reorder the instructions within each basic block in a way which
  - preserves the dependencies between those instructions (and hence the correctness of the program), and
  - achieves the minimum possible number of pipeline stalls
- Firstly, we can construct a directed acyclic graph (DAG) to represent the dependencies between instructions:
  - For each instruction in the basic block, create a corresponding vertex in the graph
  - For each dependency between two instructions, create a corresponding edge in the graph
    - This edge is directed: it goes from the earlier instruction to the later one.

Data Hazard: RAW: Resolution: Preserving Dependencies

- Construct a directed acyclic graph (DAG) to represent the dependencies between instructions:
  - For each instruction in the basic block, create a corresponding vertex in the graph.
  - For each dependency between two instructions, create a corresponding edge in the graph.
    - This edge is directed: it goes from the earlier instruction to the later one.

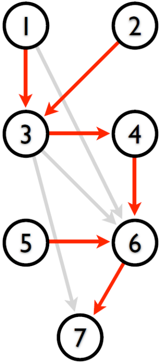
```
1 lw $1,0($0)
2 lw $2,4($0)
3 add $3,$1,$2
4 sw $3,12($0)
5 lw $4,8($0)
6 add $3,$1,$4
7 sw $3,16($0)
```



- Any topological sort of this DAG (i.e. any linear ordering of the vertices which keeps all the edges "pointing forwards") will maintain the dependencies and hence preserve the correctness of the program.

Data Hazard: RAW: Resolution: Preserving Dependencies

- Now, we want to choose an instruction order which causes the fewest possible pipeline stalls
- Unfortunately, this problem is (as usual) NP-complete and hence difficult to solve in a reasonable amount of time for realistic quantities of instructions
- However, we can devise some static scheduling heuristics to help guide us; we will hence choose a sensible and reasonably optimal instruction order, if not necessarily the absolute best one possible



1, 2, 3, 4, 5, 6, 7
2, 1, 3, 4, 5, 6, 7

1, 2, 3, 5, 4, 6, 7
1, 2, 5, 3, 4, 6, 7
1, 5, 2, 3, 4, 6, 7
5, 1, 2, 3, 4, 6, 7

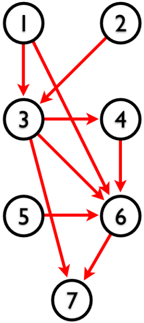
2, 1, 3, 5, 4, 6, 7
2, 1, 5, 3, 4, 6, 7
2, 5, 1, 3, 4, 6, 7
5, 2, 1, 3, 4, 6, 7

Data Hazard: RAW: Resolution: Preserving Dependencies

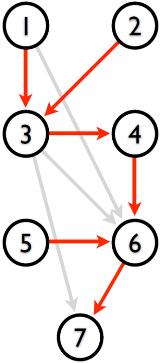
- **Heuristics**
  - Each time we emit the next instruction, we should try to choose one which:
    - does not conflict with the previous emitted instruction
    - is most likely to conflict if first of a pair (e.g. prefer lw to add)
    - is as far away as possible (along paths in the DAG) from an instruction which can validly be scheduled last
- **Algorithm**
  - Construct the scheduling DAG.
    - We can do this in $O(n^2)$ by scanning backwards through the basic block and adding edges as dependencies arise.
  - Initialise the candidate list to contain the minimal elements of the DAG.
  - While the candidate list is non-empty:
    - If possible, emit a candidate instruction satisfying all three of the static scheduling heuristics;
    - if no instruction satisfies all the heuristics, either emit NOP (on MIPS) or an instruction satisfying only the last two heuristics (on SPARC).
    - Remove the instruction from the DAG and insert the newly minimal elements into the candidate list.

Data Hazard: RAW: Resolution: Preserving Dependencies

```
1 lw $1,0($0)
2 lw $2,4($0)
3 add $3,$1,$2
4 sw $3,12($0)
5 lw $4,8($0)
6 add $3,$1,$4
7 sw $3,16($0)
```



Data Hazard: RAW: Resolution: Preserving Dependencies

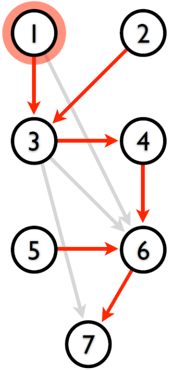


1, 2, 3, 4, 5, 6, 7
2, 1, 3, 4, 5, 6, 7

1, 2, 3, 5, 4, 6, 7
1, 2, 5, 3, 4, 6, 7
1, 5, 2, 3, 4, 6, 7
5, 1, 2, 3, 4, 6, 7

2, 1, 3, 5, 4, 6, 7
2, 1, 5, 3, 4, 6, 7
2, 5, 1, 3, 4, 6, 7
5, 2, 1, 3, 4, 6, 7

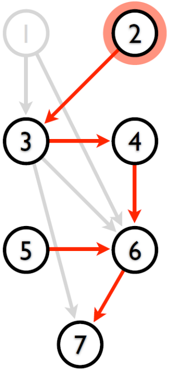
Data Hazard: RAW: Resolution: Preserving Dependencies



Candidates:
{ 1, 2, 5 }

1 `lw $1,0($0)`

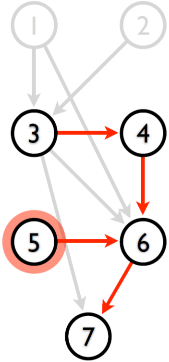
Data Hazard: RAW: Resolution: Preserving Dependencies



Candidates:
$\{ 2, 5 \}$

```
1 lw $1,0($0)
2 lw $2,4($0)
```

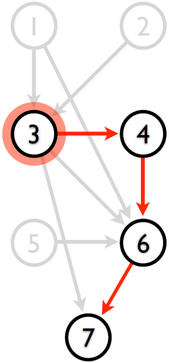
Data Hazard: RAW: Resolution: Preserving Dependencies



Candidates:
{ 3, 5 }

```
1 lw $1,0($0)
2 lw $2,4($0)
5 lw $4,8($0)
```

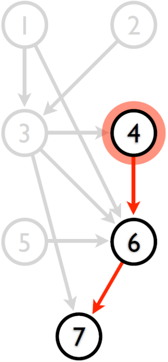
Data Hazard: RAW: Resolution: Preserving Dependencies



Candidates:
{ 3 }

```
1 lw $1,0($0)
2 lw $2,4($0)
5 lw $4,8($0)
3 add $3,$1,$2
```

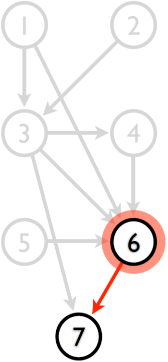
Data Hazard: RAW: Resolution: Preserving Dependencies



Candidates:
{ 4 }

```
1 lw $1,0($0)
2 lw $2,4($0)
5 lw $4,8($0)
3 add $3,$1,$2
4 sw $3,12($0)
```

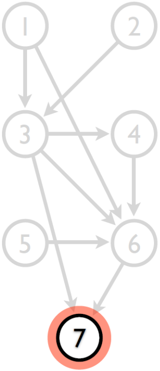
Data Hazard: RAW: Resolution: Preserving Dependencies



Candidates:
{ 6 }

```
1 lw $1,0($0)
2 lw $2,4($0)
5 lw $4,8($0)
3 add $3,$1,$2
4 sw $3,12($0)
6 add $3,$1,$4
```

Data Hazard: RAW: Resolution: Preserving Dependencies



Candidates:
{ 7 }

```
1 lw $1,0($0)
2 lw $2,4($0)
5 lw $4,8($0)
3 add $3,$1,$2
4 sw $3,12($0)
6 add $3,$1,$4
7 sw $3,16($0)
```

Data Hazard: RAW: Resolution: Preserving Dependencies

Original code:

```
1 lw $1,0($0)
2 lw $2,4($0)
3 add $3,$1,$2
4 sw $3,12($0)
5 lw $4,8($0)
6 add $3,$1,$4
7 sw $3,16($0)
```

2 stalls
13 cycles

Scheduled code:

```
1 lw $1,0($0)
2 lw $2,4($0)
5 lw $4,8($0)
3 add $3,$1,$2
4 sw $3,12($0)
6 add $3,$1,$4
7 sw $3,16($0)
```

no stalls
11 cycles

Data Hazard: WAR

- Consider two instructions $\texttt{i}_1$ and $\texttt{i}_2$, with $\texttt{i}_1$ occurring before $\texttt{i}_2$ in program order
- **Write after Read (WAR)**: $\texttt{i}_2$ tries to write a destination before it is read by $\texttt{i}_1$
- A WAR data hazard represents a problem with concurrent execution. For example:

```
i1. R4 <- R1 + R5
i2. R5 <- R1 + R2
```

- In any situation with a chance that $\texttt{i}_2$ may finish before $\texttt{i}_1$ (that is, with concurrent execution), it must be ensured that the result of register R5 is not stored before $\texttt{i}_1$ has had a chance to fetch the operands
- Called an *anti-dependence* by compiler writers, this results from reuse of the name **R5**
- This is caused by a *name dependence*. There is *no actual data transfer*. It is the same name that causes the problem. **Resolve by renaming**
- It cannot happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5

Data Hazard: WAW

- Consider two instructions i_1 and i_2, with i_1 occurring before i_2 in program order
- **Write after Write (WAW)**: i_2 tries to write an operand before it is written by i_1
- A WAW data hazard may occur in a concurrent execution environment. For example:

```
i1. R2 <- R4 + R7
i2. R2 <- R1 + R3
```

- The write back (WB) of i_2 must be delayed until i_1 finishes executing
- Called an *output dependence* by compiler writers, this also results from the reuse of name **R2**
- This is caused by a *name dependence*. There is *no actual data transfer*. It is the same name that causes the problem. **Resolve by renaming**
- It cannot happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5
  - WAR and WAW happen in more complicated pipes

Control Hazard

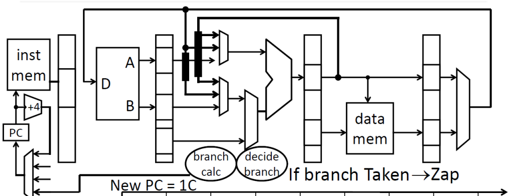
- A **Control Hazard** occurs if there is a control instruction (for example, BEQ) because the program counter (PC) following the control instruction is not known until the control instruction computes *if the branch should be taken or not*

```
0x10: beq r1, r2, L
0x14: add r3, r0, r3
0x18: sub r5, r4, r6
0x1C: L: or r3, r2, r4
```

- Note:
  - Instructions are fetched in stage 1 (IF)
  - Branch and jump decisions occur in stage 3 (EX)
  - That is, next PC is not known until *2 cycles* after branch/jump
- What happens to instr following a branch, if branch *not taken*?
  - Continue on the pipeline
- What happens to instr following a branch, if branch *taken*?
  - *Pipeline needs to be zapped or flushed*

Control Hazard: Pipeline Flush

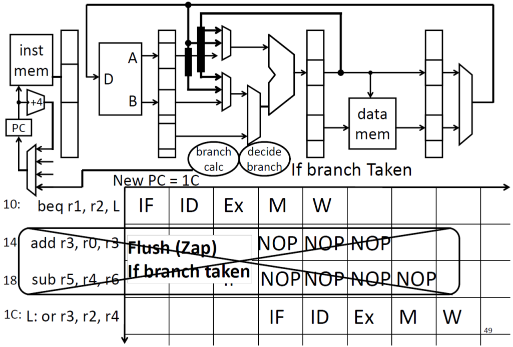
- If a branch is taken, pipeline stalls and needs to be flushed:
  - prevent PC update
  - clear IF/ID pipeline register
    - ▷ instruction just fetched might be wrong one, so convert to NOP
  - allow branch to continue into EX stage
- **Stall with NOP**



10: beq r1, r2, L	IF	ID	Ex	M	W				
14: add r3, r0, r3		IF	ID	NOP	NOP	NOP			
18: sub r5, r4, r6			IF	NOP	NOP	NOP	NOP		
1C: L: or r3, r2, r4				IF	ID	Ex	M	W	

Control Hazard: Pipeline Flush

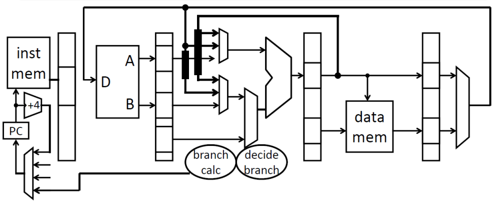
- If a branch is taken, pipeline stalls and needs to be flushed:
  - prevent PC update
  - clear IF/ID pipeline register
    - ▷ instruction just fetched might be wrong one, so convert to NOP
  - allow branch to continue into EX stage
- **Zap / Flush**

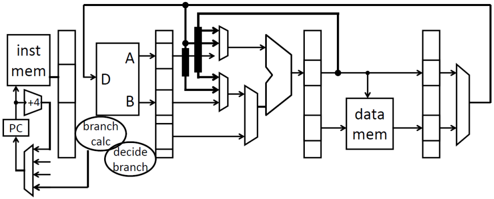


10: beq r1, r2, L	IF	ID	Ex	M	W			
14 add r3, r0, r3				NOP	NOP	NOP		
18 sub r5, r4, r6				NOP	NOP	NOP	NOP	
1C: L: or r3, r2, r4				IF	ID	Ex	M	W

Control Hazard: Resolution

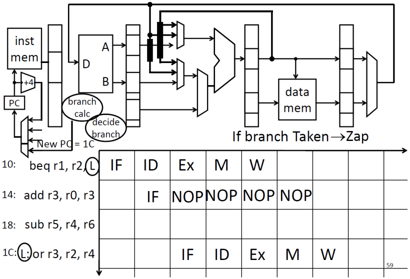
- **Forward/bypass values for branches**
  - We can move branch calculation from EX to ID
  - will require new bypasses into ID stage; or can just zap the second instruction
  - Still need to zap/flush instructions if the branch is taken



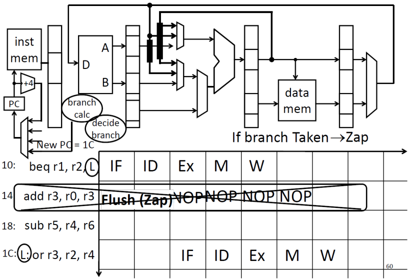


Control Hazard: Resolution

- **Forward/bypass values for branches**
  - We can move branch calculation from EX to ID
  - will require new bypasses into ID stage; or can just zap the second instruction
  - Still need to zap/flush instructions if the branch is taken



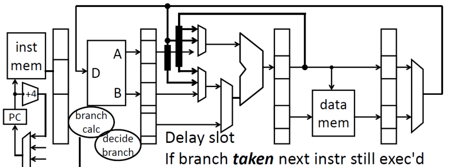
10: beq r1, r2, L	IF	ID	Ex	M	W			
14: add r3, r0, r3		IF	NOP	NOP	NOP	NOP		
18: sub r5, r4, r6								
1C: L: or r3, r2, r4			IF	ID	Ex	M	W	



10: beq r1, r2, L	IF	ID	Ex	M	W			
14 add r3, r0, r3	Flush (Zap)		NOP	NOP	NOP	NOP		
18: sub r5, r4, r6								
1C: L: or r3, r2, r4			IF	ID	Ex	M	W	

Control Hazard: Resolution

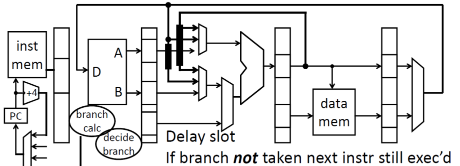
- **Delay Slot**
  - ISA says N instructions after branch/jump always executed
    - MIPS has 1 branch delay slot
    - That is, Whether branch taken or not, instruction following branch is always executed



Delay slot

If branch *taken* next instr still exec'd

10: beq r1, r2, L	IF	ID	Ex	M	W			
14: add r3, r0, r3		IF	ID	Ex	M	W		
18: sub r5, r4, r6								
1C: L: or r3, r2, r4			IF	ID	Ex	M	W	



Delay slot

If branch **not** taken next instr still exec'd

10: beq r1, r2, L	IF	ID	Ex	M	W			
14: add r3, r0, r3		IF	ID	Ex	M	W		
18: sub r5, r4, r6			IF	ID	Ex	M	W	
1C: L: or r3, r2, r4				IF	ID	Ex	M	W

Control Hazard: Resolution

- **Speculative Execution**
  - “Guess” direction of the branch
    - ▷ Allow instructions to move through pipeline
    - ▷ Zap them later if wrong guess always executed
  - Useful for long pipelines
- Predict Branching
  - Make prediction based on last branch:
  - Predict “take branch” if last branch “taken”
  - Or Predict “do not take branch” if last branch “not taken”
  - Need one bit to keep track of last branch
- Need methods for good prediction

In-Class Assignment 2: 09-Nov-2021

Marks 10

- Consider the following instructions:

```
r1 := a
r2 := b
r3 := r1 + r2
r3 := r3 + 1
r2 := c
r3 := r3 + r2
a := r3
```

- For a MIPS 5 stage pipeline, schedule with stall to avoid hazards
- Construct the dependency preservation DAG and schedule to minimize stalls
- You may write your solution on notepad or on paper
- Submit by email to ppd@cse.iitkgp.ac.in within class hours (9:55am)
- Mention your name and roll number

Pipeline Hazards and Resolutions

- **Structural hazards**: Resource Conflict
  - Resolutions
    - ▷ Stall
    - ▷ Pipeline hardware resource
    - ▷ Replicate resource
- **Data hazards**: Data Conflict that can result in *race conditions* (or *race hazards*). It may be of three types: *Read after Write (RAW)*, a *true dependency*, *Write after Read (WAR)*, an *anti-dependency*, and *Write after Write (WAW)*, an *output dependency*
  - Resolutions
    - ▷ Hardware detects RAW and stalls
    - ▷ Forward / Bypass
    - ▷ Renaming (WAR & WAW)
    - ▷ Instruction Scheduling
- **Control hazards**: Control Conflict
  - Resolutions
    - ▷ Stall & Zap / Flush
    - ▷ Forward / Bypass
    - ▷ Delay Slot
    - ▷ Instruction Scheduling & Speculative Execution

Control Hazard

- A **Control Hazard** occurs if there is a control instruction (for example, BEQ) because the program counter (PC) following the control instruction is not known until the control instruction computes *if the branch should be taken or not*

```
0x10: beq r1, r2, L
0x14: add r3, r0, r3
0x18: sub r5, r4, r6
0x1C: L: or r3, r2, r4
```

- Note:
  - Instructions are fetched in stage 1 (IF)
  - Branch and jump decisions occur in stage 3 (EX)
  - That is, next PC is not known until *2 cycles* after branch/jump
- What happens to instr following a branch, if branch *not taken*?
  - Continue on the pipeline
- What happens to instr following a branch, if branch *taken*?
  - *Pipeline needs to be zapped or flushed*

Control Hazards: Compiler Actions for Resolutions

- Instruction Scheduling
  - Exploit Delay Slots
  - Unroll loops to minimize branches
- Speculative Execution
  - Predict branching to reduce probability of control hazards

Loop Unrolling

- Determine loop unrolling useful by finding that *loop iterations were independent*
  - Determine address offsets for different loads/stores
  - Increases program size
- *Use different registers* to avoid unnecessary constraints forced by using same registers for different computations
  - Stress on registers
- Eliminate the *extra test and branch* instructions and adjust the *loop termination and iteration* code
- If a *loop only has dependencies within an iteration*, the loop is considered **parallel** - multiple iterations can be executed together so long as order within an iteration is preserved
- If a *loop has dependencies across iterations*, it is not parallel and these dependencies are referred to as **loop-carried**

Loop Unrolling: Example Set 1

```
for (i=1000; i>0; i=i-1)
     x[i] = x[i] + s;        // No dependences

for (i=1; i<=100; i=i+1) {
     A[i+1] = A[i] + C[i];   // S1: S1 depends on S1 from previous iteration
     B[i+1] = B[i] + A[i+1]; // S2: S2 depends on S1 in the same iteration
}                            // S2 depends on S2 from previous iteration

for (i=1; i<=100; i=i+1) {
     A[i] = A[i] + B[i];      // S1: S1 depends on S2 from previous iteration
     B[i+1] = C[i] + D[i];    // S2
}

for (i=1000; i>0; i=i-1)
     x[i] = x[i-3] + s;       // S1: S1 depends on S1 from 3 prev iterations
                              // Referred to as a recursion
                              // Dependence distance 3; limited parallelism
```

Source: Loop Unrolling

Loop Unrolling: Constructing Parallel Loops

- If loop-carried dependencies are not cyclic (S1 depending on S1), loops can be restructured to be parallel

```
for (i=1; i<=100; i=i+1) {
     A[i] = A[i] + B[i];   // S1: S1 depends on S2 from previous iteration
     B[i+1] = C[i] + D[i]; // S2
}

A[1] = A[1] + B[1];        // Iter 1
B[2] = C[1] + D[1];        // Iter 1
A[2] = A[2] + B[2];        // Iter 2
B[3] = C[2] + D[2];        // Iter 2
A[3] = A[3] + B[3];        // Iter 3
B[4] = C[3] + D[3];        // Iter 3
...
A[99] = A[99] + B[99];     // Iter 99
B[100] = C[99] + D[99];    // Iter 99
A[100] = A[100] + B[100]; // Iter 100
B[101] = C[100] + D[100]; // Iter 100

A[1] = A[1] + B[1];
for (i=1; i<=99; i=i+1) {
     B[i+1] = C[i] + D[i];     // S3
     A[i+1] = A[i+1] + B[i+1]; // S4: S4 depends on S3 of same iteration
}
B[101] = C[100] + D[100];
```

Source: Loop Unrolling

Loop Unrolling: Example Set 2

- Load: 2-cycles (1 cycle stall for consumer)
- FP ALU: 4-cycles (3 cycle stall for consumer; 2 cycle stall if the consumer is a store)
- One branch delay slot
- Int ALU: 1-cycle (no stall for consumer, 1 cycle stall if the consumer is a branch)

```
for (i=1000; i>0; i--) x[i] = x[i] + s;

Loop:   L.D     F0, 0(R1)      ; F0 = array element. R1 = &x[1000] in memory
        ADD.D   F4, F0, F2     ; add scalar. F2 = s, a loop invariant
        S.D     F4, 0(R1)      ; store result
        DADDUI  R1, R1,# -8    ; decrement address pointer
        BNE     R1, R2, Loop   ; branch if R1 != R2. R2 = &x[0]
        NOP
// 10 Cycle Schedule
Loop:   L.D     F0, 0(R1)     ; F0 = array element
        stall
        ADD.D   F4, F0, F2    ; add scalar
        stall
        stall
        S.D     F4, 0(R1)     ; store result
        DADDUI  R1, R1,# -8   ; decrement address pointer
        stall
        BNE     R1, R2, Loop ; branch if R1 != R2
        stall
```

Loop Unrolling: Example Set 2

- By re-ordering instructions, it takes 6 cycles per iteration instead of 10
- We were able to violate an anti-dependence easily because an immediate was involved
- Loop overhead (instrs that do book-keeping for the loop): 2
  Actual work (the ld, add.d, and s.d): 3 instrs
  Can we somehow get execution time to be 3 cycles per iteration?

```
for (i=1000; i>0; i--) x[i] = x[i] + s;

// 10 Cycle Schedule
Loop:   L.D     F0, 0(R1)
        stall
        ADD.D   F4, F0, F2
        stall
        stall
        S.D     F4, 0(R1)
        DADDUI  R1, R1,# -8
        stall
        BNE     R1, R2, Loop
        stall
```

```
for (i=1000; i>0; i--) x[i] = x[i] + s;

// 6 Cycle Schedule
Loop:   L.D     F0, 0(R1)
        DADDUI  R1, R1,# -8  // Cycle before ADD.D
        ADD.D   F4, F0, F2
        stall
        BNE     R1, R2, Loop // Cycle 2 before S.D
        S.D     F4, 8(R1)    // Delay slot
```

Source: Lecture: Static ILP

Loop Unrolling: Example Set 2

- Loop overhead: 2 instrs; Work: 12 instrs
- Separate registers used to avoid WAR / WAW
- How long will the below schedule take to complete?
- Stalls will need to be considered

```
        for (i=1000; i>0; i--) x[i] = x[i] + s;

Loop:   L.D     F0, 0(R1)     // Iter 1
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)

        L.D     F6, -8(R1)    // Iter 2
        ADD.D   F8, F6, F2
        S.D     F8, -8(R1)

        L.D     F10,-16(R1)   // Iter 3
        ADD.D   F12, F10, F2
        S.D     F12, -16(R1)

        L.D     F14, -24(R1) // Iter 4
        ADD.D   F16, F14, F2
        S.D     F16, -24(R1)

        DADDUI  R1, R1, #-32
        BNE     R1,R2, Loop
```

Loop Unrolling: Example Set 2

```
Loop:   L.D     F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        L.D     F6, -8(R1)
        ADD.D   F8, F6, F2
        S.D     F8, -8(R1)
        L.D     F10,-16(R1)
        ADD.D   F12, F10, F2
        S.D     F12, -16(R1)
        L.D     F14, -24(R1)
        ADD.D   F16, F14, F2
        S.D     F16, -24(R1)
        DADDUI  R1, R1, #-32
        BNE     R1,R2, Loop
```

```
for (i=1000; i>0; i--)
    x[i] = x[i] + s;
```

```
Loop:   L.D     F0, 0(R1)
        stall
        ADD.D   F4, F0, F2
        stall (2)
        S.D     F4, 0(R1)
        L.D     F6, -8(R1)
        stall
        ADD.D   F8, F6, F2
        stall (2)
        S.D     F8, -8(R1)
        L.D     F10,-16(R1)
        stall
        ADD.D   F12, F10, F2
        stall (2)
        S.D     F12, -16(R1)
        L.D     F14, -24(R1)
        stall
        ADD.D   F16, F14, F2
        stall (2)
        S.D     F16, -24(R1)
        DADDUI  R1, R1, #-32
        stall
        BNE     R1,R2, Loop
        stall
```

```
Loop:   L.D     F0, 0(R1)
        L.D     F6, -8(R1)
        L.D     F10,-16(R1)
        L.D     F14, -24(R1)
        ADD.D   F4, F0, F2
        ADD.D   F8, F6, F2
        ADD.D   F12, F10, F2
        ADD.D   F16, F14, F2
        S.D     F4, 0(R1)
        S.D     F8, -8(R1)
        DADDUI  R1, R1, # -32
        // In stall. Offset adjusted
        S.D     F12, 16(R1)
        BNE     R1, R2, Loop
        // Delay slot. Offset adjusted
        S.D     F16, 8(R1)
```

14 cycles or 3.5 cycles per original iteration

Some Registers may be optimized

Source: Lecture: Static ILP

Loop Unrolling

- Increases program size
- Requires more registers
- To unroll an n-iteration loop by degree k, we will need (n/k) iterations of the larger loop, followed by $(n \mod k)$ iterations of the original loop

Source: Lecture: Static ILP

Automating Loop Unrolling

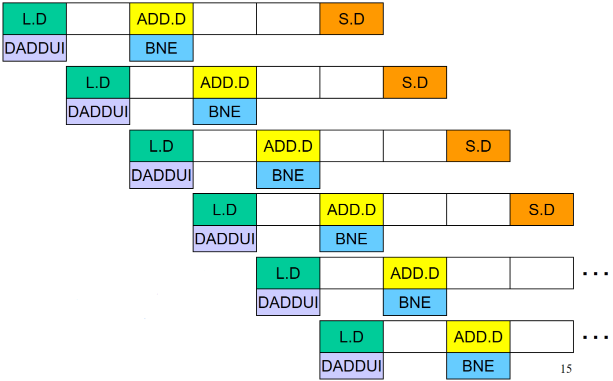
- Determine the dependencies across iterations: in the example, we knew that loads and stores in different iterations did not conflict and could be re-ordered
- Determine if unrolling will help – possible only if iterations are independent
- Determine address offsets for different loads/stores
- Dependency analysis to schedule code without introducing hazards; eliminate name dependencies by using additional registers

Source: Lecture: Static ILP

Software Pipeline

- Software pipelining is a technique used to optimize loops, in a manner that parallels hardware pipelining
- Software pipelining is a type of out-of-order execution, except that the reordering is done by a compiler (or in the case of hand written assembly code, by the programmer) instead of the processor
- Some computer architectures like Intel's IA-64 have explicit support for software pipelining

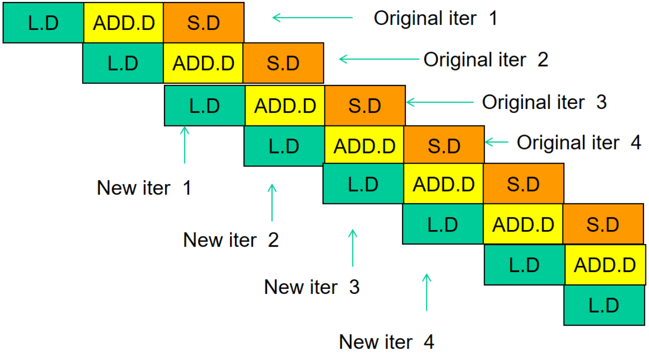
```
for (i=1000; i>0; i--)
    x[i] = x[i] + s;

// 10 Cycle Schedule
Loop:   L.D     F0, 0(R1)
        stall
        ADD.D   F4, F0, F2
        stall
        stall
        S.D     F4, 0(R1)
        DADDUI  R1, R1,# -8
        stall
        BNE     R1, R2, Loop
        stall
```



Source: Lecture: Static ILP and Software Approaches to Exploiting Instruction Level Parallelism

Software Pipeline

- Software pipelining eliminates NOP's by inserting instructions from different iterations of the same loop body



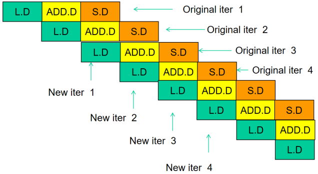
Source: Lecture: Static ILP and Software Approaches to Exploiting Instruction Level Parallelism

Software Pipeline

```
for (i=1000; i>0; i--) x[i] = x[i] + s;
```

```
// Original Loop
Loop:   L.D     F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        DADDUI  R1, R1,# -8
        BNE     R1, R2, Loop
```

```
// After software pipeline
Loop:   S.D     F4, 16(R1)
        ADD.D   F4, F0, F2
        L.D     F0, 0(R1)
        DADDUI  R1, R1,# -8
        BNE     R1, R2, Loop
```



- **Advantages**:
  - achieves nearly the same effect as loop unrolling, but without the code expansion
  - an unrolled loop may have inefficiencies at the start and end of each iteration, while a sw-pipelined loop is almost always in steady state
  - a sw-pipelined loop can also be unrolled to reduce loop overhead
- **Disadvantages**:
  - does not reduce loop overhead
  - may require more registers

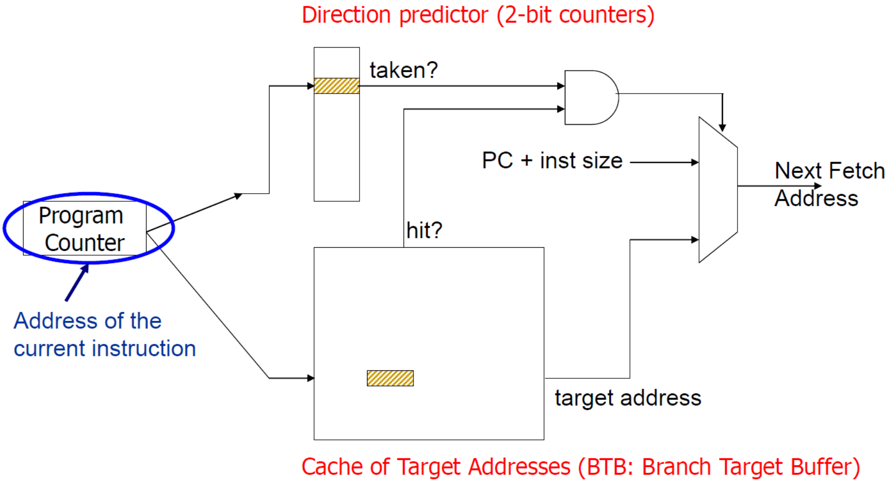
Source: Lecture: Static ILP and Software Approaches to Exploiting Instruction Level Parallelism

Branch Prediction

- **Idea**: Predict the next fetch address (to be used in the next cycle)
- Requires three things to be predicted at fetch stage:
  - Whether the fetched instruction is a branch
  - (Conditional) branch direction
  - Branch target address (if taken)
- **Observation**: Target address remains the same for a conditional direct branch across dynamic instances
  - Idea: Store the target address from previous instance and access it with the PC
  - Called Branch Target Buffer (BTB) or Branch Target Address Cache

Source: 18-447: Computer Architecture Lecture 11: Branch Prediction

Branch Prediction: Fetch Stage with BTB


Direction predictor (2-bit counters)
taken?
PC + inst size
Next Fetch Address
hit?
Program Counter
Address of the current instruction
target address
Cache of Target Addresses (BTB: Branch Target Buffer)

Source: 18-447: Computer Architecture Lecture 11: Branch Prediction

Branch Direction Prediction Schemes

- **Compile time (static)**
  - Always not taken
  - Always taken
  - Backward taken, forward not taken (BTFN)
  - Profile based (likely direction)
  - Program analysis based (likely direction)
  - Programmer based
- **Run time (dynamic)**
  - Last time prediction (1-bit bimodal)
  - Two-bit counter based prediction (2-bit bimodal)
  - Two-level prediction (global vs. local)
  - Hybrid

Source: 18-447: Computer Architecture Lecture 11: Branch Prediction

Static Branch Prediction

- **Always not taken**
  - Simple to implement: no need for BTB, no direction prediction
  - Low accuracy: ˜30-40%
  - Compiler can layout code such that the likely path is the *not-taken* path
- **Always taken**
  - No direction prediction
  - Better accuracy: ˜60-70%
    - Backward branches (that is, loop branches) are usually taken
    - Backward branch: target address lower than branch PC
- **Backward taken, forward not taken (BTFN)**
  - Predict backward (loop) branches as taken, others not-taken

Source: 18-447: Computer Architecture Lecture 11: Branch Prediction

Static Branch Prediction

- **Profile based**
  - **Idea**: *Compiler determines likely direction for each branch using profile run*
    - Encodes that direction as a hint bit in the branch instruction format
  - **Advantages**
    - Per branch prediction (more accurate than earlier schemes) → accurate if profile is representative!
  - **Disadvantages**
    - Requires hint bits in the branch instruction format
    - Accuracy depends on dynamic branch behavior:
      TTTTTTTTTTNNNNNNNNNN → 50% accuracy
      TNTNTNTNTNTNTNTNTNTN → 50% accuracy
    - Accuracy depends on the representativeness of profile input set

Source: 18-447: Computer Architecture Lecture 11: Branch Prediction

Static Branch Prediction

- **Program based / Program analysis based**
  - **Idea**: *Use heuristics based on program analysis to determine statically-predicted direction*
    - ▷ Opcode heuristic: Predict BLEZ as NT (negative integers used as error values in many programs)
    - ▷ Loop heuristic: Predict a branch guarding a loop execution as taken (that is, execute the loop)
    - ▷ Pointer and FP comparisons: Predict not equal
  - **Advantages**
    - ▷ Does not require profiling
  - **Disadvantages**
    - ▷ Heuristics might be not representative or good
    - ▷ Requires compiler analysis and ISA support

Source: 18-447: Computer Architecture Lecture 11: Branch Prediction

Static Branch Prediction

- **Programmer-based**
  - **Idea**: *Programmer provides the statically-predicted direction*
    - ▷ Via pragmas in the programming language that qualify a branch as *likely-taken* versus *likely-not-taken*

```
if (likely(x))  ...         // likely-taken

if (unlikely(error))  ...   // likely-not-taken
```

  - **Advantages**
    - ▷ Does not require profiling or program analysis
    - ▷ Programmer may know some branches and their program better than other analysis techniques
  - **Disadvantages**
    - ▷ Requires programming language, compiler, ISA support
    - ▷ Burdens the programmer

Source: 18-447: Computer Architecture Lecture 11: Branch Prediction

Static Branch Prediction

- All previous techniques can be combined
  - Profile based
  - Program based
  - Programmer-based
- What are common **disadvantages** of all three techniques?
  - *Cannot adapt to dynamic changes in branch behavior*
    - ▷ This can be mitigated by a dynamic compiler, but not at a fine granularity (and a dynamic compiler has its overheads)

Source: 18-447: Computer Architecture Lecture 11: Branch Prediction

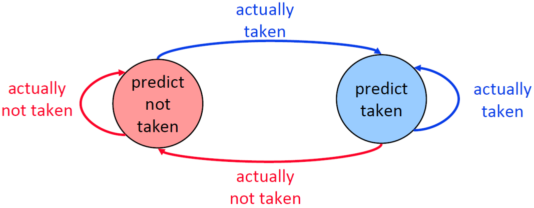
Dynamic Branch Prediction

- **Idea**: *Predict branches based on dynamic information* (collected at run-time)
- **Advantages**
  - Prediction based on history of the execution of branches
  - It can adapt to dynamic changes in branch behavior
  - No need for static profiling: input set representativeness problem goes away
- **Disadvantages**
  - More complex (requires additional hardware)

Source: 18-447: Computer Architecture Lecture 11: Branch Prediction

Dynamic Branch Prediction: Last time predictor: 1-bit Bimodal

- **Last time predictor**
  - Single bit per branch (stored in BTB)
  - Indicates which direction branch went last time it executed
    TTTTTTTTTTNNNNNNNNNN → 90% accuracy
- Always mispredicts the last iteration and the first iteration of a loop branch
  - Accuracy for a loop with N iterations = (N-2)/N
- **Advantages**
  - Loop branches for loops with large number of iterations
- **Disadvantages**
  - Loop branches for loops will small number of iterations
    TNTNTNTNTNTNTNTNTNTN → 0% accuracy



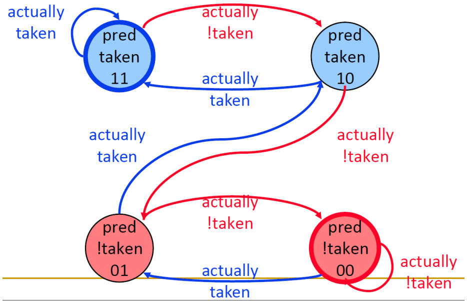
Dynamic Branch Prediction: 2-bit Bimodal

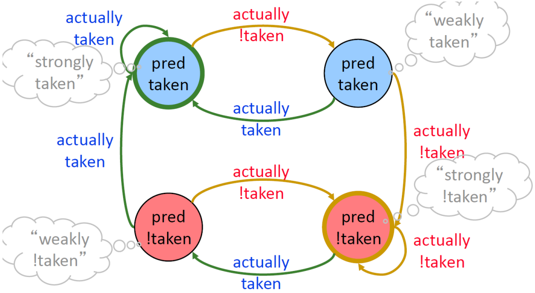
- **Problem**: *A last-time predictor changes its prediction from T → NT or NT → T too quickly*
  - even though the branch may be mostly taken or mostly not taken
- **Solution Idea**: **2-bit Bimodal**: *Add hysteresis to the predictor so that prediction does not change on a single different outcome*
  - Use two bits to track the history of predictions for a branch instead of a single bit
  - Can have 2 states for T or NT instead of 1 state for each
  - Each branch associated with a two-bit counter. One more bit provides *hysteresis*
- A strong prediction does not change with one single different outcome
- Accuracy for a loop with N iterations = (N-1)/N
  TNTNTNTNTNTNTNTNTNTN → 0% accuracy (assuming init to weakly taken)
- **Advantages**
  - Better prediction accuracy
- **Disadvantages**
  - More hardware cost (but counter can be part of a BTB entry)

Source: 18-447: Computer Architecture Lecture 11: Branch Prediction

Dynamic Branch Prediction: 2-bit Bimodal

- For each branch, maintain a **2-bit saturating counter**:
  - if the branch is taken: `counter = min(3,counter+1)`
  - if the branch is not taken: `counter = max(0,counter-1)`
- If (counter $\geq$2), predict taken, else predict not taken
- **Hysteresis**: Change prediction after 2 consecutive mistakes





Source: 18-447: Computer Architecture Lecture 11: Branch Prediction

Dynamic Branch Prediction: Correlation Predictor

- 1-bit (Last-time) and 2-bit Bimodal predictors exploit *last-time* predictability
- **Observation 1**: *A branch's outcome can be correlated with other branches' outcomes*
  - **Global branch correlation**
- **Observation 2**: *A branch's outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch "last-time" it was executed)*
  - **Local branch correlation**
- **Hybrid branch correlation**

Source: 18-447: Computer Architecture Lecture 11: Branch Prediction

Course Summary: Topics Covered

- Compiler Flow
- Front-End
  - Lexical Analysis, Flex
  - Syntax Analysis, Bison
  - Semantic Analysis and Machine Independent Code Generation
  - Machine Independent Optimization
  - Data Flow Analysis
- Intermediate Representation
- Run-Time Environment
- Symbol Table
- Back-End
  - Register Allocation
  - Target Code Generation and Optimization
  - Loop Optimization
  - Code Generation for Pipeline Architecture

Course Summary: Topics for the Future ...

- Front-End
  - C Preprocessor
  - `inline` function
  - Recursion Optimization (Tail-call)
  - `const` for optimization
  - Reference and Overloading
  - Classes, Inheritance, Polymorphism (Object Oriented Programming)
  - Exception handling
  - Templates (Meta Programming)
  - Lambda's (Functional Programming)
- Back-End
  - Code Generation for:
    - Super-scalar, VLIW, Vector, Multi-core architecture
    - Debugging
    - Exception handling
  - Virtual Machine
  - Garbage Collection
- Code Retargeting
  - LLVM
  - Application Binary Interface (ABI)

In-Class Assignment 3: 16-Nov-2021

Marks 10

- Consider the following loop:

```
int a[100]; // sizeof(int) = 4
a[0] = 0;
for(i = 1; i < 100; ++i)
    a[i] = a[i-1] + 1;
```

- Identify the dependency in the loop
- For a MIPS 5 stage pipeline, schedule with stall to avoid hazards. Assume:

```
R0 = &a[0] in memory
R1 = &a[100] in memory
Load <reg>, <mem> needs 1 cycle stall. <reg> <- <mem>
Inc <reg> does not need a stall. <reg> = <reg> + 1
Add <reg1>, <reg2>, <const> needs 1 cycle stall. <reg1> = <reg2> + <const>
Store <mem>, <reg>, does not need a stall. <mem> <- <reg>
Jne <reg1>, <reg2>, <label> needs 1 cycle stall. if (<reg1> != <reg2>) go to <label>
There is one cycle delay slot
There are 10 registers (R0 to R9) available
```

- Unroll the loop to optimize stalls in the generated code
- You may write your solution on notepad or on paper
- Submit to Moodle by 10:10am. DO NOT MAIL
- Mention your name and roll number in the submission file