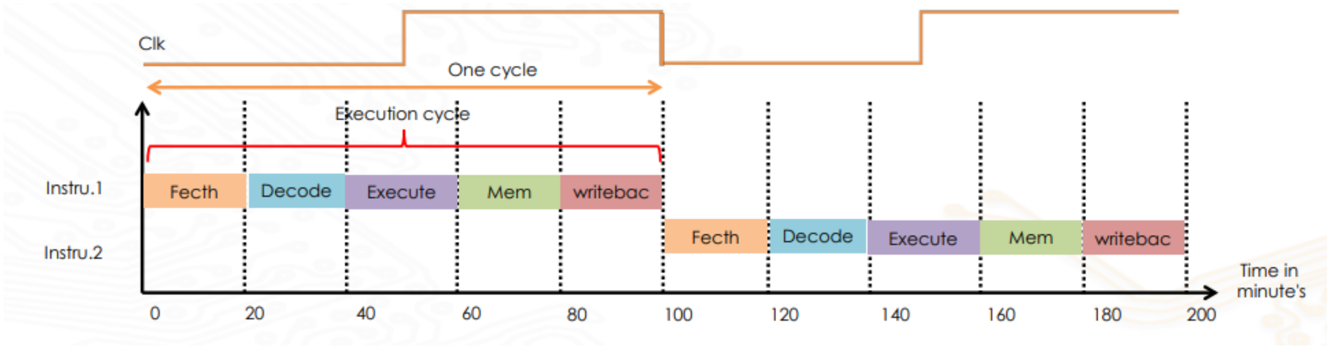
M4 Instruction Level Parallelism

1. Instruction Execution

1.1 Pipelining

Single cycle execution instructions

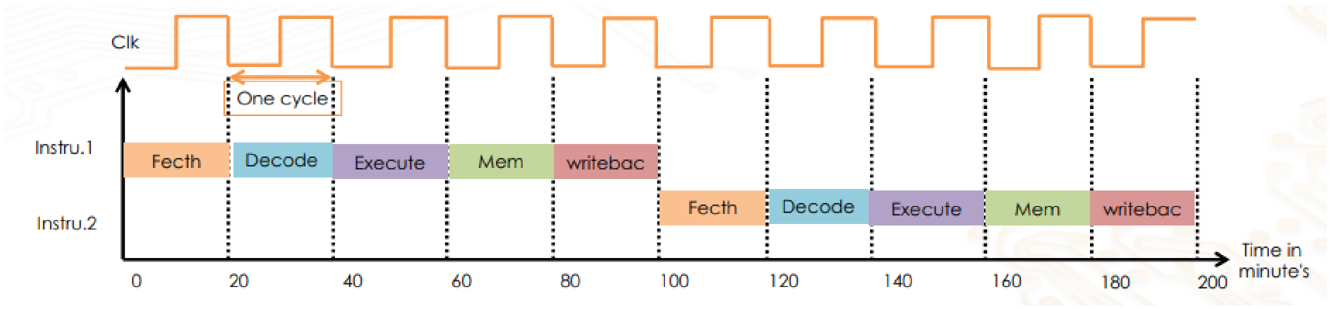
- Instructions are executed in a single cycle (low clk + high clk)



- Remember that Execution Time = IC x CPI x Time Period (dependent on the worst case instruction in data path)
  - Execution Cycle $\neq$ Execution Time
  - For Single cycle execution instructions, CPI = 1, reducing execution time, however there is a long clock period

Multi cycle execution instructions

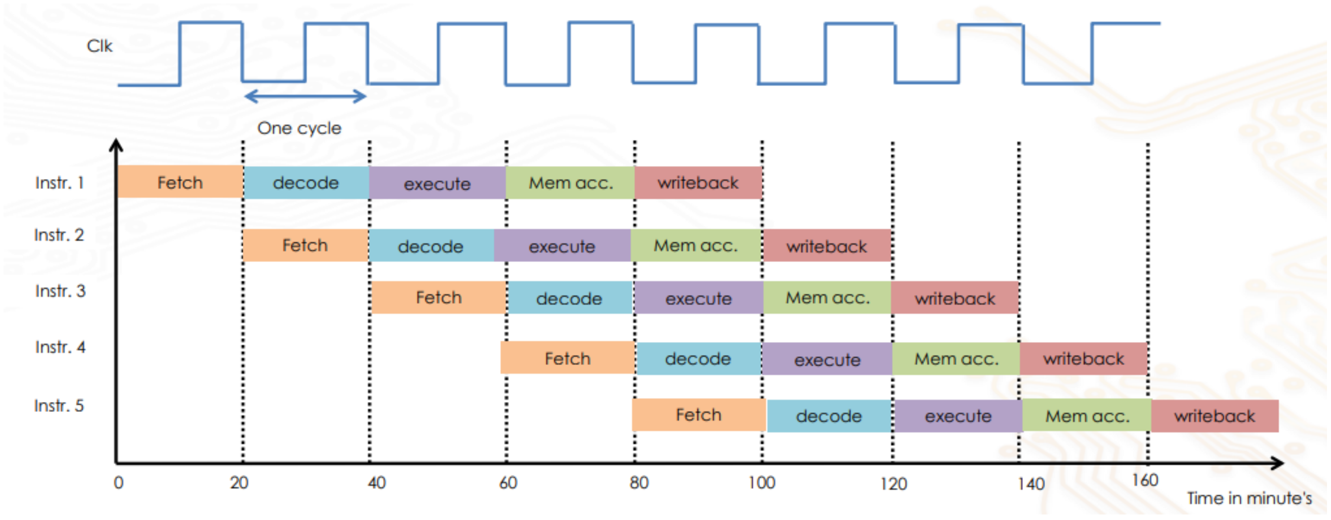
- Instructions are executed in multiple clock cycles (CPI $>$ 1)



- This reduces execution time since Time Period is reduced, but CPI is increased instead

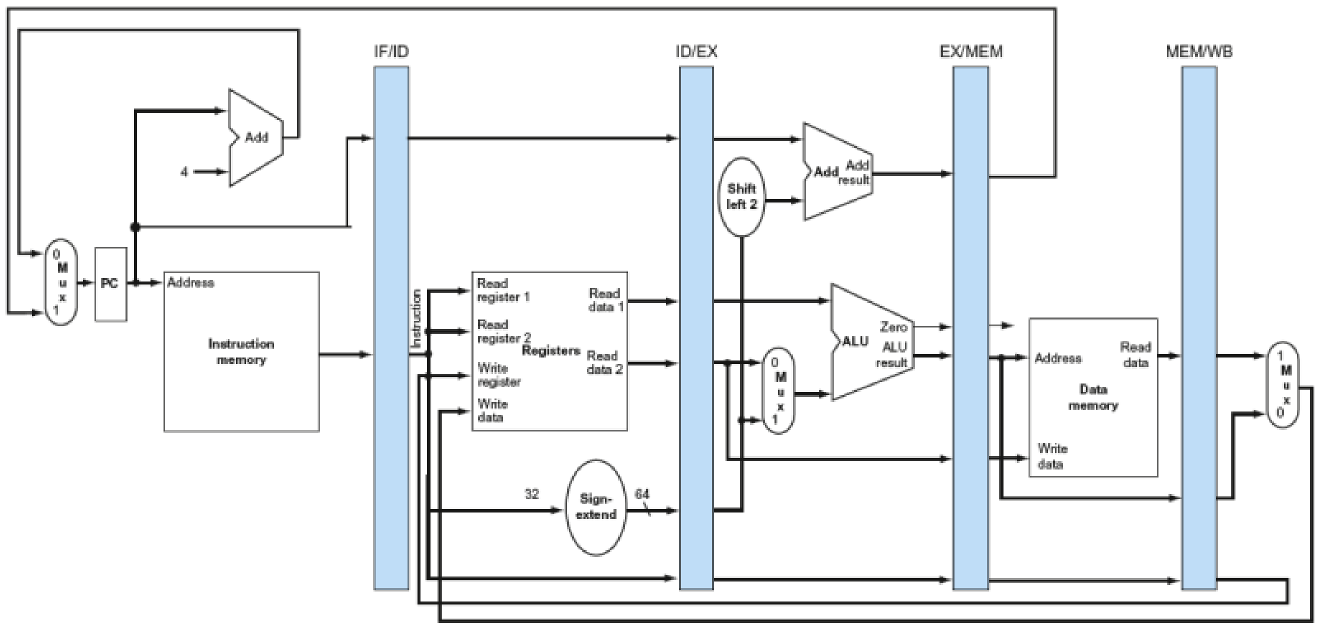
Solution

- Pipelining is used to allow multiple sub-tasks to be carried out simultaneously using independent resources



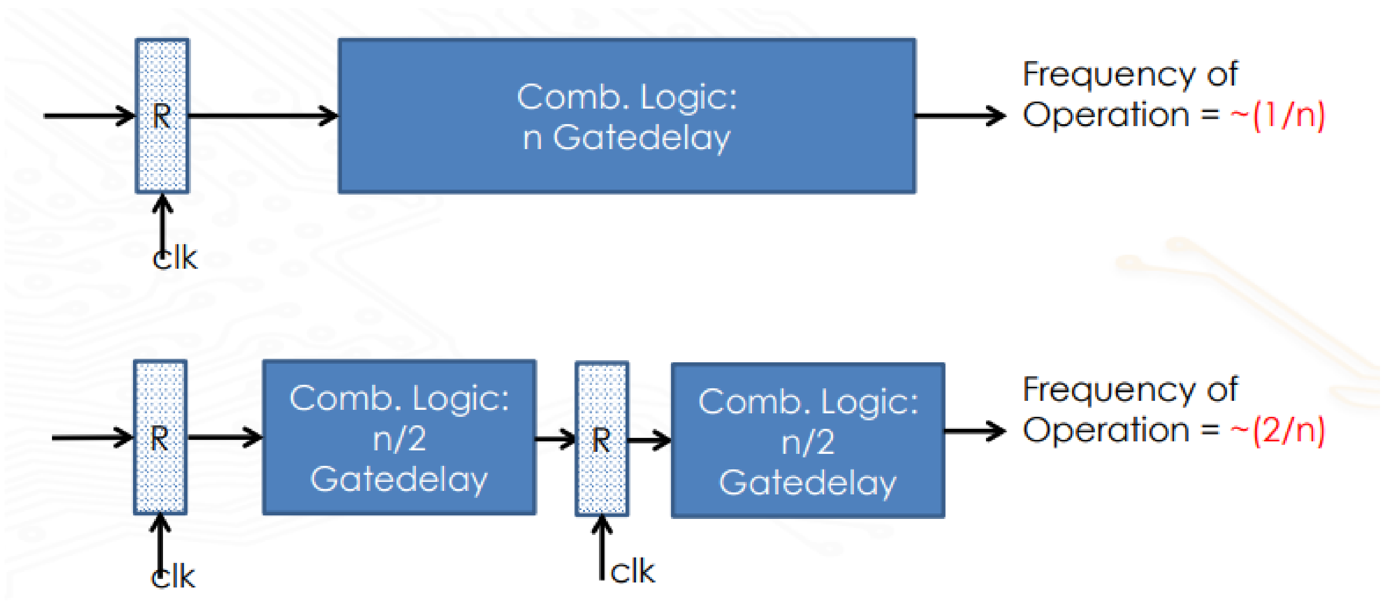
- Each component uses different resources, this allows up to 5 instructions to process at the same time (80-100) since there are no conflicting resources

1.2 Pipeline datapath

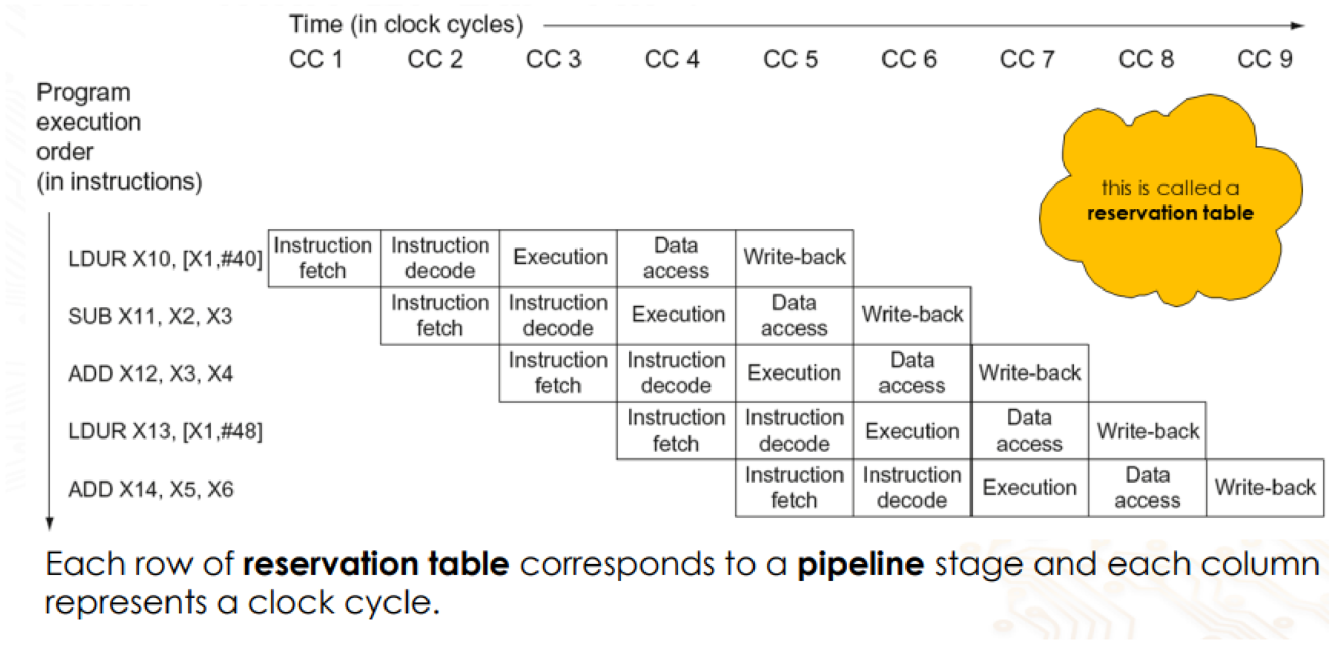


- Every pipeline starts and end with a register
  - IF: Starts with PC, ends with IF/ID pipeline register
  - ID: Starts with IF/ID pipeline register, ends with ID/EX pipeline register
  - EX: Starts with ID/EX pipeline register, ends with EX/MEM pipeline register
  - MEM: Starts with EX/MEM pipeline register, ends with MEM/WB pipeline register
  - WB: Starts with MEM/WB pipeline register, ends with Register File

Ideal pipelining


R
clk
Comb. Logic: n Gatedelay
Frequency of Operation = ~(1/n)
R
clk
Comb. Logic: n/2 Gatedelay
R
clk
Comb. Logic: n/2 Gatedelay
Frequency of Operation = ~(2/n)

- **Comb. Logic** exists between two registers in pipeline
- Frequency = 1 / time, when Comb. Logic is divided (increasing the number of stages), we end up increasing the frequency of operation
  - A pipeline with n-stages would mean that CPU can operate n time faster (ideally)
  - As we increase the number of stages, number of registers also increases, increasing overall latency due to register delay. Combinational Logic no longer bottleneck, but registers.
- **Example of ideal pipelining**



Time (in clock cycles) →

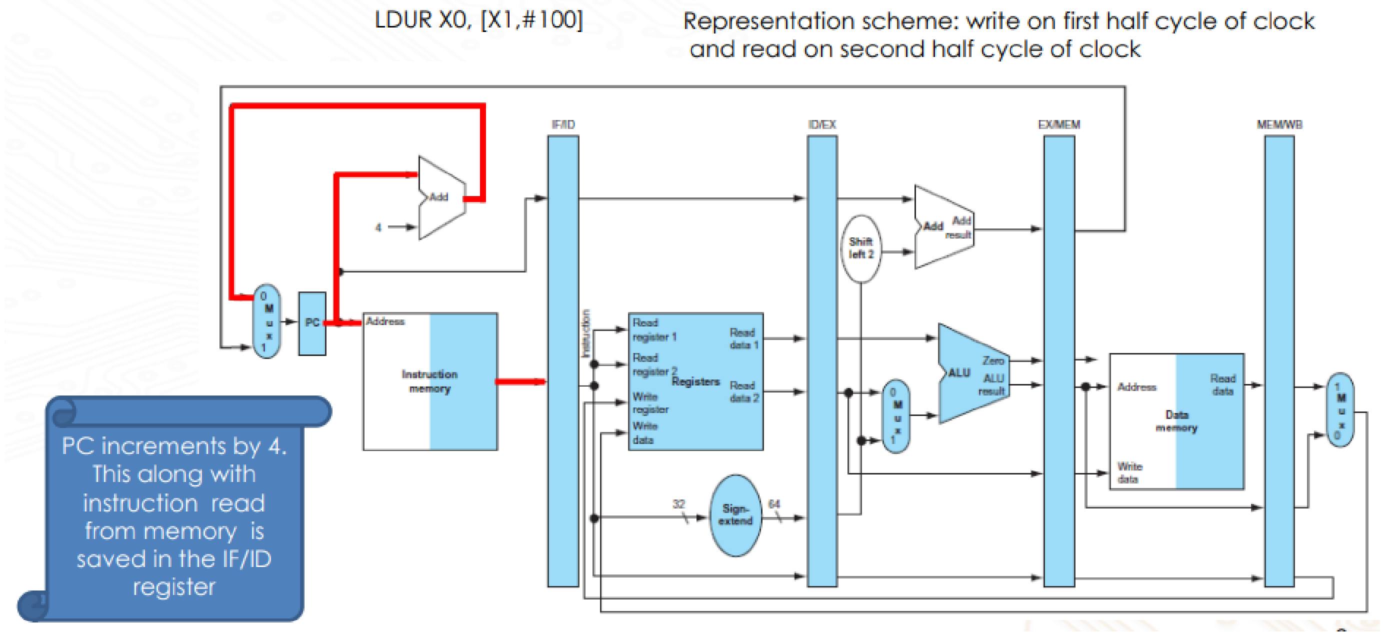
Program execution order (in instructions)	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
LDUR X10, [X1,#40]	Instruction fetch	Instruction decode	Execution	Data access	Write-back				
SUB X11, X2, X3		Instruction fetch	Instruction decode	Execution	Data access	Write-back			
ADD X12, X3, X4			Instruction fetch	Instruction decode	Execution	Data access	Write-back		
LDUR X13, [X1,#48]				Instruction fetch	Instruction decode	Execution	Data access	Write-back	
ADD X14, X5, X6					Instruction fetch	Instruction decode	Execution	Data access	Write-back

this is called a **reservation table**

Each row of **reservation table** corresponds to a **pipeline** stage and each column represents a clock cycle.

Example Pipeline datapath for LDUR X0, [X1,#100]

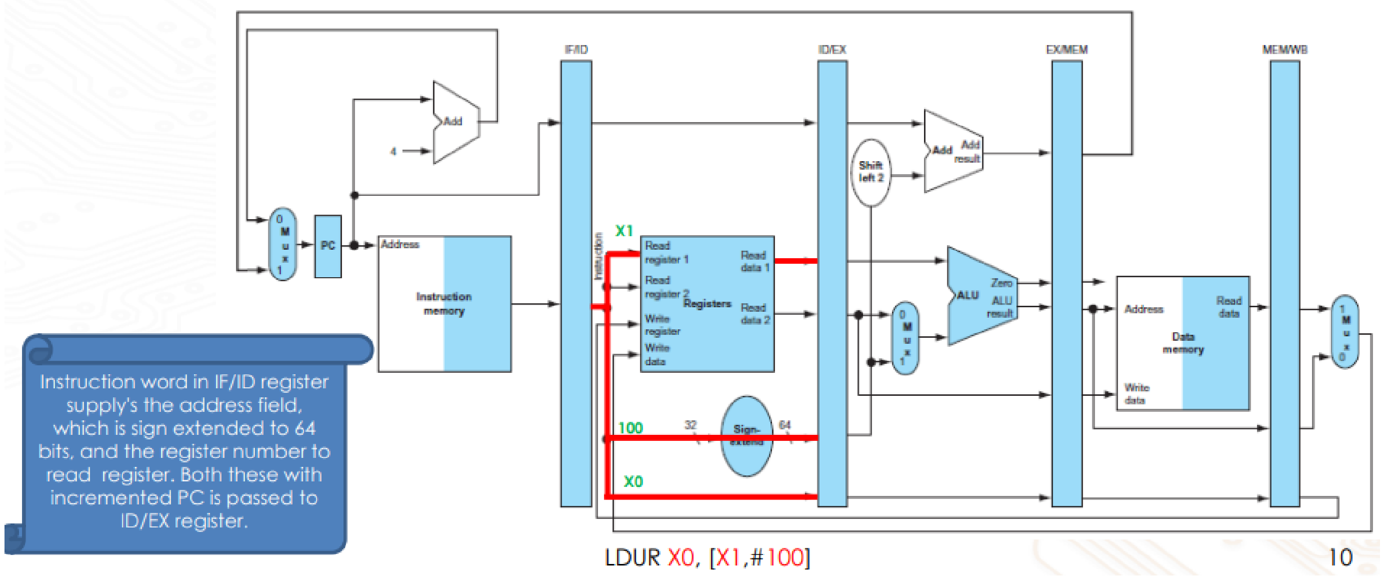
1. IF


LDUR X0, [X1,#100]

Representation scheme: write on first half cycle of clock and read on second half cycle of clock

PC increments by 4. This along with instruction read from memory is saved in the IF/ID register

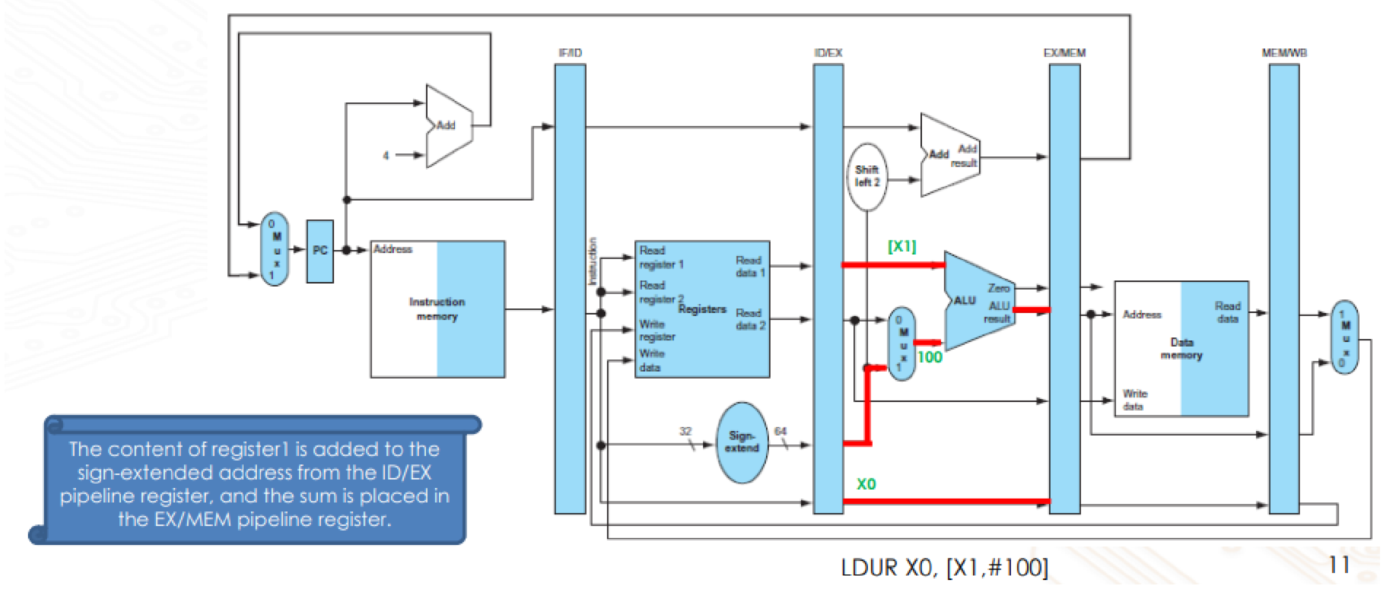
2. ID


Instruction word in IF/ID register supply's the address field, which is sign extended to 64 bits, and the register number to read register. Both these with incremented PC is passed to ID/EX register.

LDUR X0, [X1,#100]

10

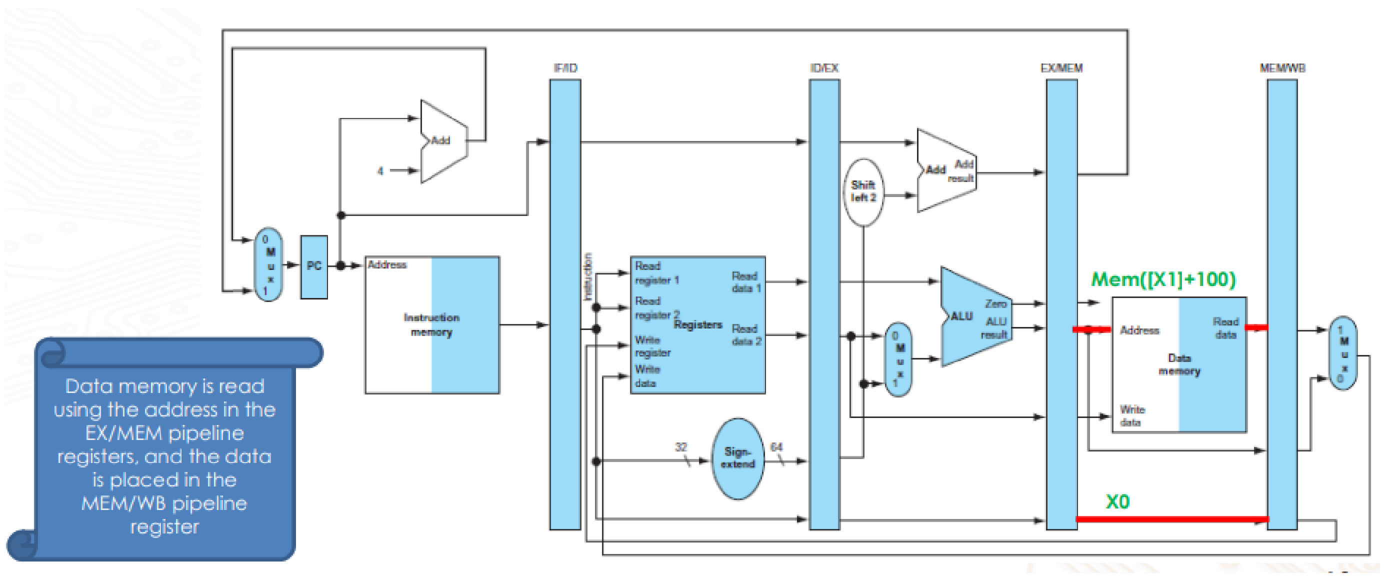
3. EX


The content of register1 is added to the sign-extended address from the ID/EX pipeline register, and the sum is placed in the EX/MEM pipeline register.

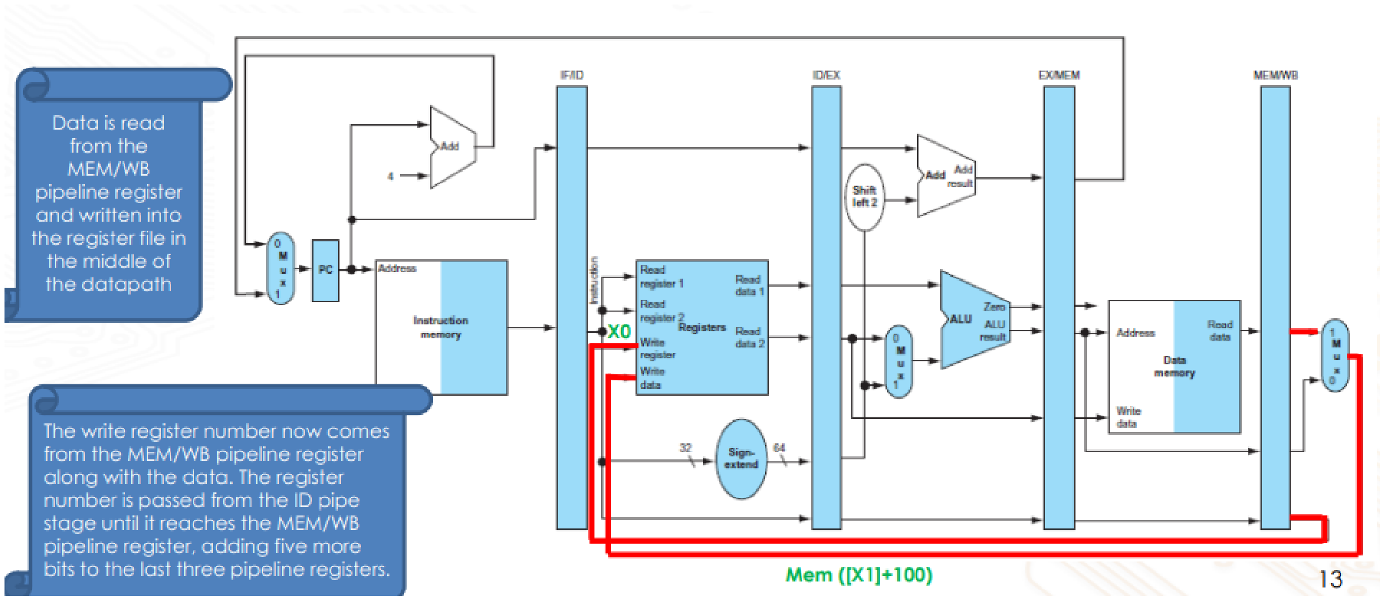
LDUR X0, [X1,#100]

11

4. MEM


Data memory is read using the address in the EX/MEM pipeline registers, and the data is placed in the MEM/WB pipeline register

5. WB


Data is read from the MEM/WB pipeline register and written into the register file in the middle of the datapath

The write register number now comes from the MEM/WB pipeline register along with the data. The register number is passed from the ID pipe stage until it reaches the MEM/WB pipeline register, adding five more bits to the last three pipeline registers.

1.3 Challenges in Instruction Pipelining Realisation

Ideal

- Identical task of all instructions
- Uniform decomposition of task
- Independent computations

Real

- Only LDUR instructions need all pipelining stages
- Not all pipeline stages involve the same time to complete their subtasks
- Execution of one instruction may depend on a previous instruction

Complications

- Datapath: Many data in flight
- Control: Must correspond to multiple instructions

- Instructions may have data and control flow dependencies / One may have to stall and wait for another

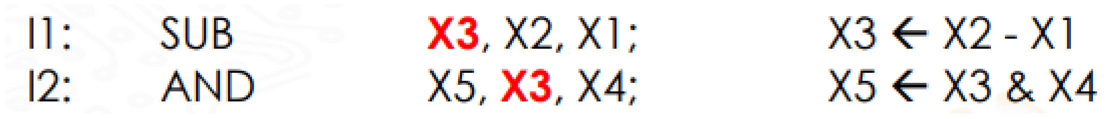
2. Pipelining Hazards and methods to solve

- Pipeline hazards are caused by stalling of pipeline (decrease in efficiency)
- **Structural** - Happens when two instructions require to use a given hardware resource at the same time (one instruction has to wait for at least a clock cycle)
  - Solved by additional hardware elements
- **Data** - Either source / destination register of instruction not available at expected time in pipeline
  - Need to know the dependencies between data in the program and eliminate them
- **Control** - Jumps / subroutine calls can stall a pipeline because of a delay in the availability of an instruction
  - Need to predict location of un/conditional branches to avoid stalling

2.1 Data Dependencies

True / flow dependencies (RAW - read after write)

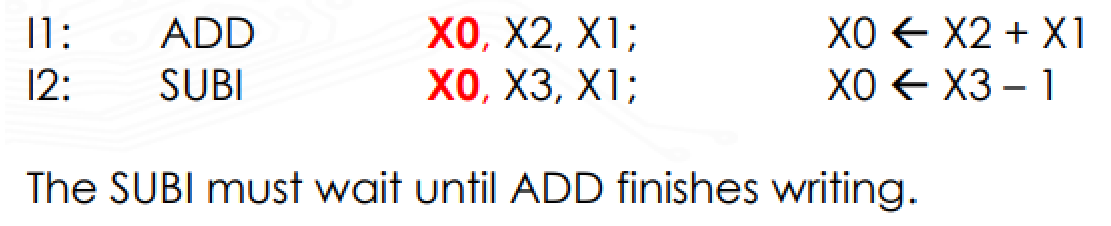
- Instruction j cannot execute until instruction i processes result



I1:	SUB	**X3**, X2, X1;	X3 ← X2 - X1
I2:	AND	X5, **X3**, X4;	X5 ← X3 & X4

Output dependencies (WAW - write after write)

- Instruction j cannot write its result until instruction i has written its result

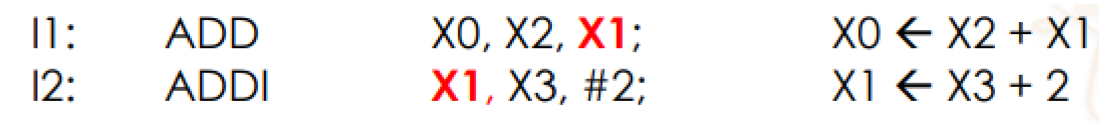


I1:	ADD	**X0**, X2, X1;	X0 ← X2 + X1
I2:	SUBI	**X0**, X3, X1;	X0 ← X3 – 1

The SUBI must wait until ADD finishes writing.

Anti dependence (WAR - write after read)

- Instruction j cannot write its result until instruction i has read its sources



I1:	ADD	X0, X2, **X1**;	X0 ← X2 + X1
I2:	ADDI	**X1**, X3, #2;	X1 ← X3 + 2

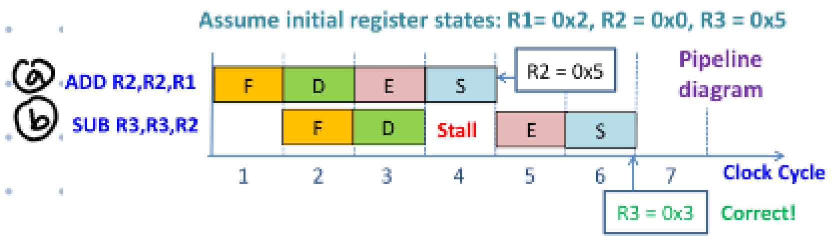
2.2 Data Dependency Handling (focus on true dependencies)

Steady State CPI = (no. of instructions + no. of stalls) / no. of instructions

Detect and wait until value is available

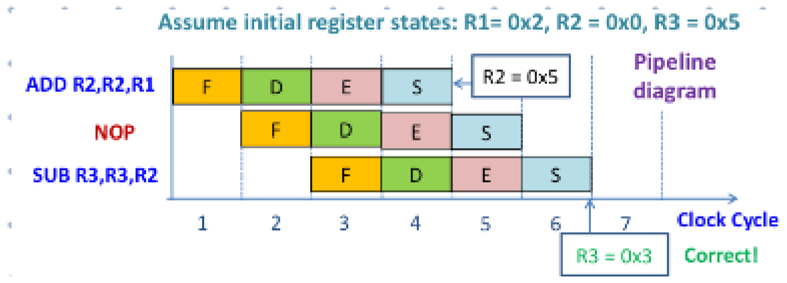
- Recall 1106 pipeline datadependency section

- Hardware - Stall the program



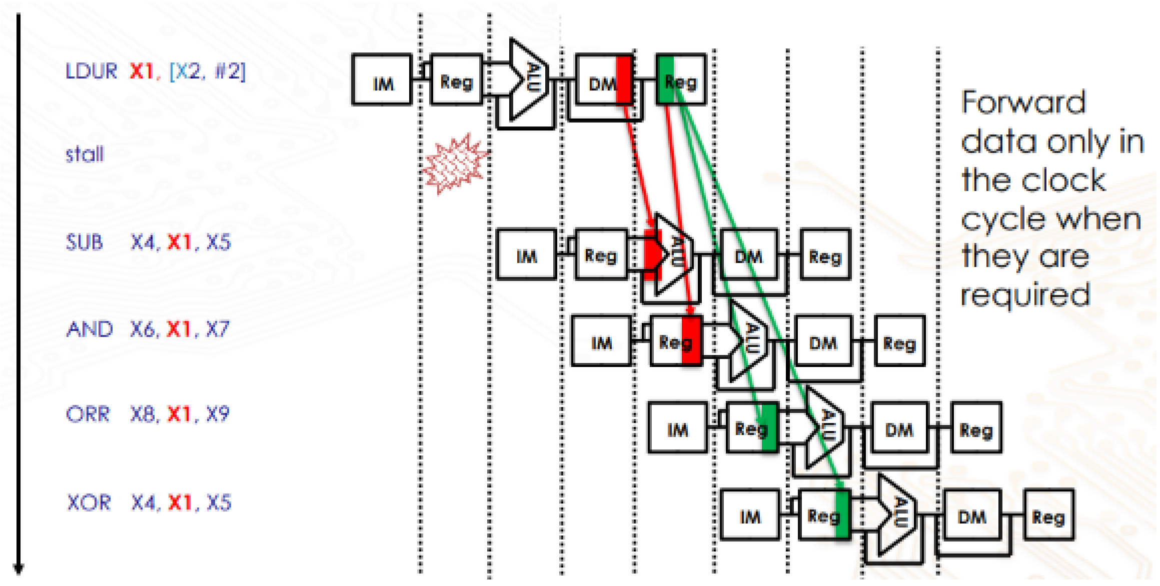
  - When data dependency is detected, allow `a` to continue execution but stall `b`

- Software - Compiler plug in NOP instructions in between



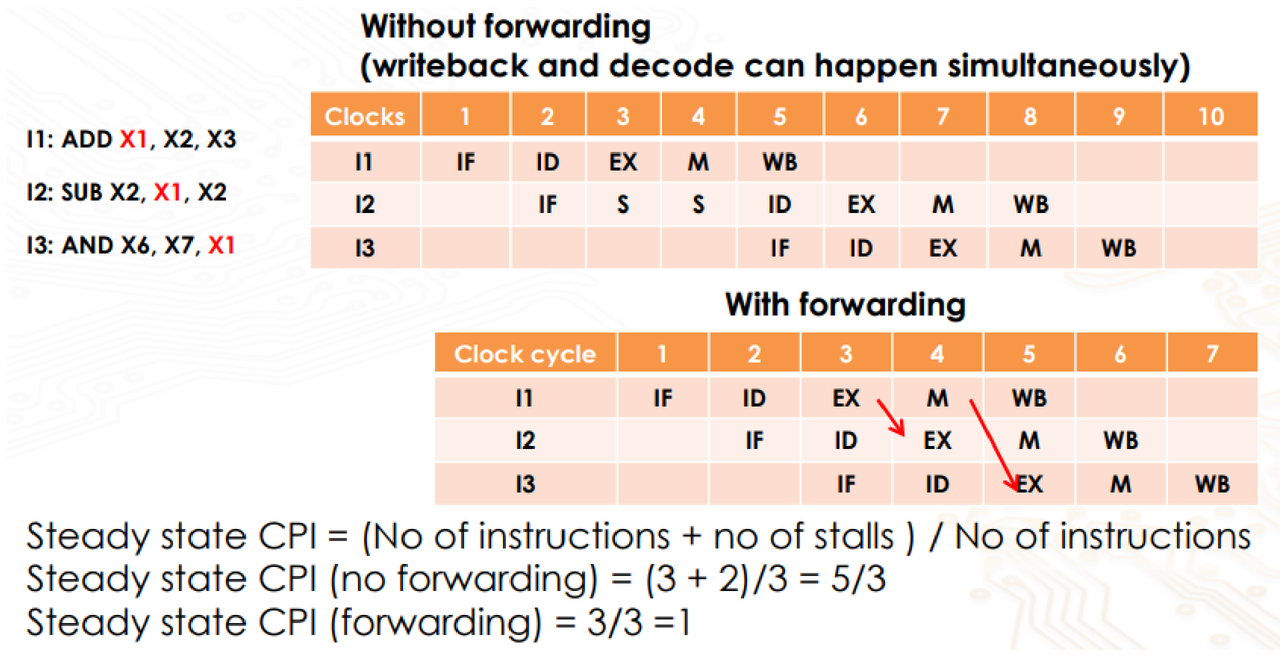
Data Forwarding

- **Through register**
  - Writing and reading of a register can be done in the same cycle
- **Bypass (Red)**



  - We can obtain value to be loaded into X1 at the end of the MEM stage of I1 and pass it into the ALU of I2 (stall is not counted as an instruction)
  - In I3, we can read the value in X1 immediately after it is written in I1
  - **Red arrows indicate data forwarding**

- The green arrows can be carried out normally



Without forwarding
(writeback and decode can happen simultaneously)

I1: ADD X1, X2, X3

I2: SUB X2, X1, X2

I3: AND X6, X7, X1

Clocks	1	2	3	4	5	6	7	8	9	10
I1	IF	ID	EX	M	WB					
I2		IF	S	S	ID	EX	M	WB		
I3					IF	ID	EX	M	WB	

With forwarding

Clock cycle	1	2	3	4	5	6	7
I1	IF	ID	EX	M	WB		
I2		IF	ID	EX	M	WB	
I3			IF	ID	EX	M	WB

Steady state CPI = (No of instructions + no of stalls) / No of instructions
Steady state CPI (no forwarding) = (3 + 2)/3 = 5/3
Steady state CPI (forwarding) = 3/3 =1

Out-of-Order instruction execution (Rearrangement of instructions)

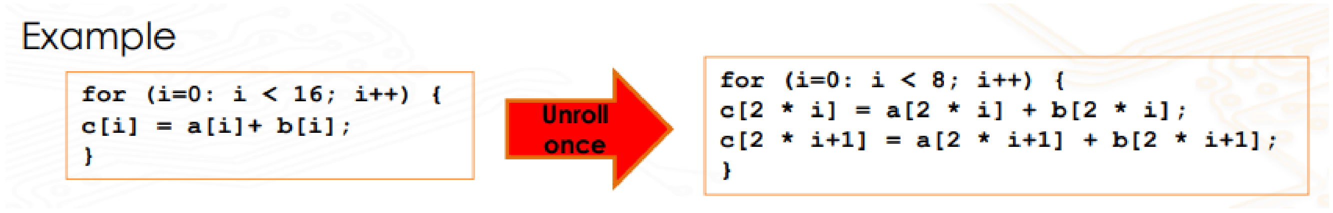
- Instructions fetched in compiler-generated order but may be executed in some other order due to dynamic scheduling during run time
- Independent instructions behind a stalled instruction may pass it
- After ID
  - check for structural hazard (available functional units)
  - check for data hazard (data dependencies)
  - Independent ready instructions should execute without stalling if they do not depend on previous instructions

Register Renaming

- Renames register to minimise structural hazards but may require more register resources

Loop unrolling and reordering

- Creates longer code sequence and leads to multiple replications of the loop body, more efficient to execute iterations in parallel



Example

```
for (i=0: i < 16; i++) {
c[i] = a[i]+ b[i];
}
```

Unroll
once

```
for (i=0: i < 8; i++) {
c[2 * i] = a[2 * i] + b[2 * i];
c[2 * i+1] = a[2 * i+1] + b[2 * i+1];
}
```

  - Instead of looping 16 times and doing the addition once each loop, we loop 8 times and do the addition twice each loop
- Look at slide 38-40 for example

3. Control Hazards

- During ID of B and CB type instructions, the decoding unit comes to know that there is an alteration in the program control flow (i.e. a branch is executed)
  - Incorrectly fetched instructions by IF needs to be discarded and new instruction at branch location must be fetched, casing a stall in the pipeline (time lost is known as branch penalty)

3.1 Counter

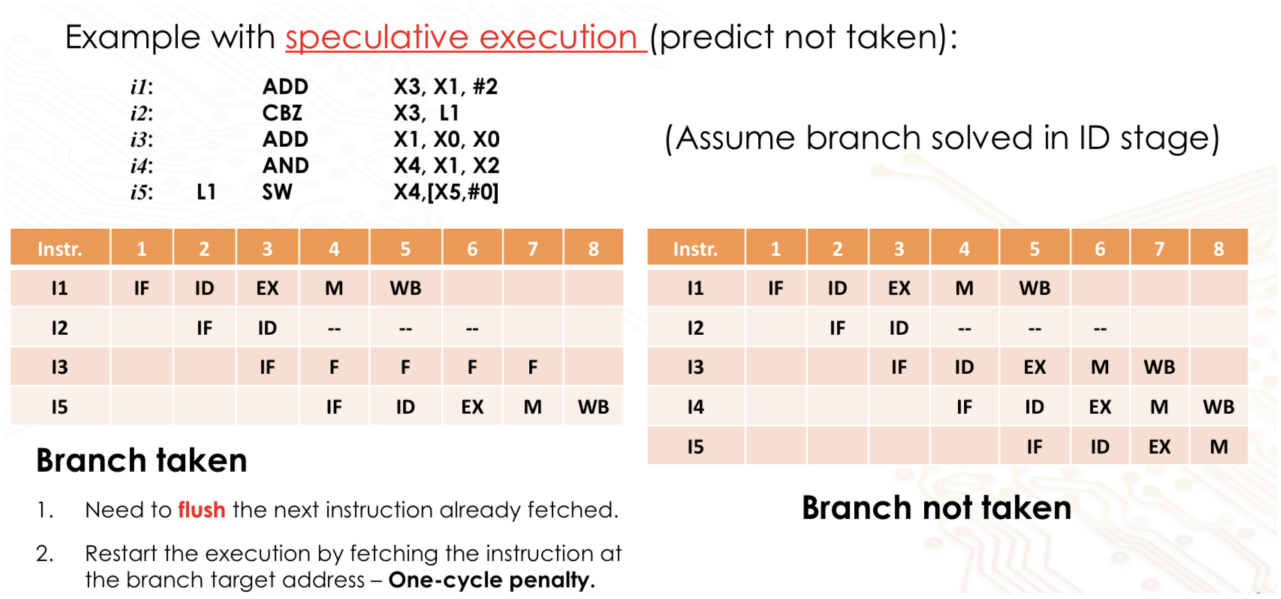
Conservative Way

- Immediately stall pipeline as a branch instruction is fetched, only fetch next instruction after waiting for pipeline to determine outcome of branch
  - 3 stalls needed is Branch Target Address (BTA) is updated in WB stage of Branching instruction execution
  - If PCSrc (PC source) and BTA is updated in EXE stage using hardware changes, Branch can update PC in EXE stage instead of WB, decreasing needed stalls to 2
  - Having 1 stall in pipeline is possible by conducting early evaluation of PC
    - Moving the adder in charge of adding PC value and 2 bit shifted offset before ID allows instruction at BTA to be fetched after the ID stage of Branch instruction execution
- Having only 1 pipeline stall still yields a performance loss depending on branch frequency
  - Pipeline stall cycles per instruction due to branches = branch frequency x branch penalty

Branch Prediction

- Deals with branch penalty by continuing executing the instructions following the branch speculatively
  - If branching does not happen, no stall occurs
  - Otherwise speculated instructions need to be flushed
- **Static Prediction** (for highly predictable behaviour)
  - Actions for branch fixed for each branch during entire execution

- **Predict-Not-Taken**



Example with speculative execution (predict not taken):

```
i1:        ADD    X3, X1, #2
i2:        CBZ    X3,  L1
i3:        ADD    X1, X0, X0
i4:        AND    X4, X1, X2
i5:   L1   SW     X4,[X5,#0]
```

(Assume branch solved in ID stage)

Instr.	1	2	3	4	5	6	7	8
I1	IF	ID	EX	M	WB			
I2		IF	ID	--	--	--		
I3			IF	F	F	F	F	
I5				IF	ID	EX	M	WB

Branch taken

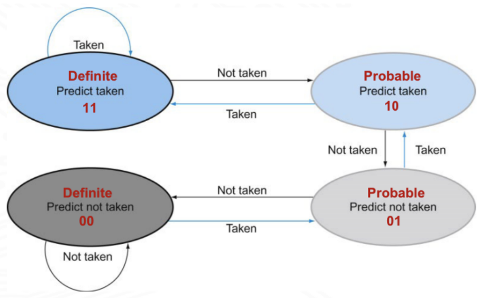
1. Need to **flush** the next instruction already fetched.
2. Restart the execution by fetching the instruction at the branch target address – **One-cycle penalty.**

Instr.	1	2	3	4	5	6	7	8
I1	IF	ID	EX	M	WB			
I2		IF	ID	--	--	--		
I3			IF	ID	EX	M	WB	
I4				IF	ID	EX	M	WB
I5					IF	ID	EX	M

Branch not taken

- **Delayed Branch**
  - Compiler schedules an independent instruction in the branch delay slot so that it gets executed no matter the decision of the branch instruction (assumes only 1 stall with hardware architecture specified in conservative way)

- **Dynamic Prediction** (for less predictable behaviour)
  - Prediction decision changes based on execution history; when CPU meets a branch, CPU uses the hardware predictor to make a decision on which path to speculate on and updates the path afterwards with the actual path from BTA
  - **1-bit predictor**
    - T is set to 1 when a branch is confirmed taken, 0 otherwise. CPU uses. this value to speculate which path to take
    - Problem: changes too quickly
    - Solution: Add hysteresis to predictor to make it not change on every single different outcome
      - use 2 bits to track history instead

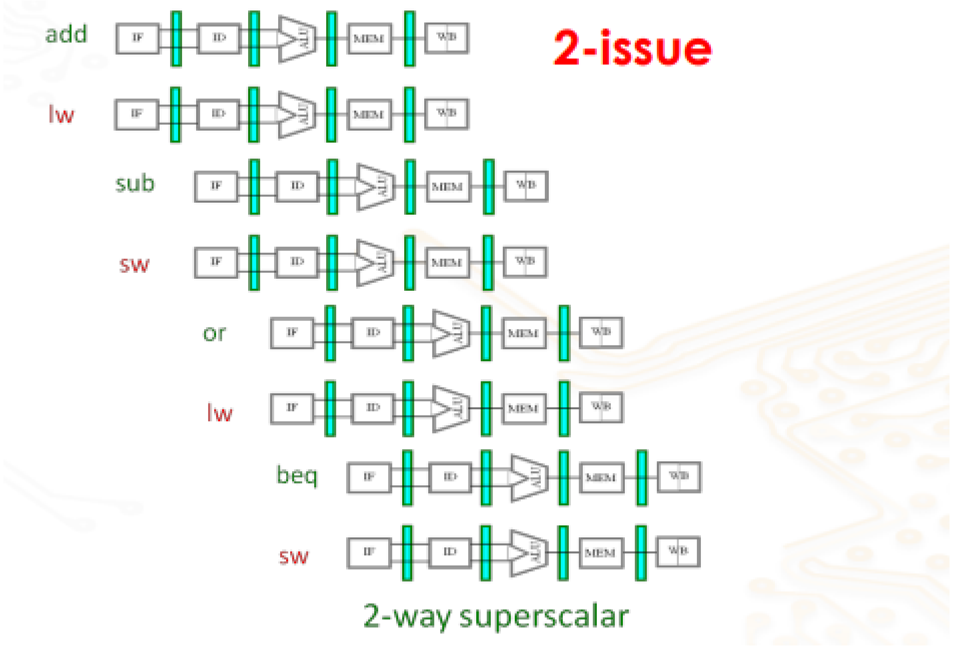


Scheme 2: 2 bit prediction uses the result of last two branches to predict instead of just the last branch.

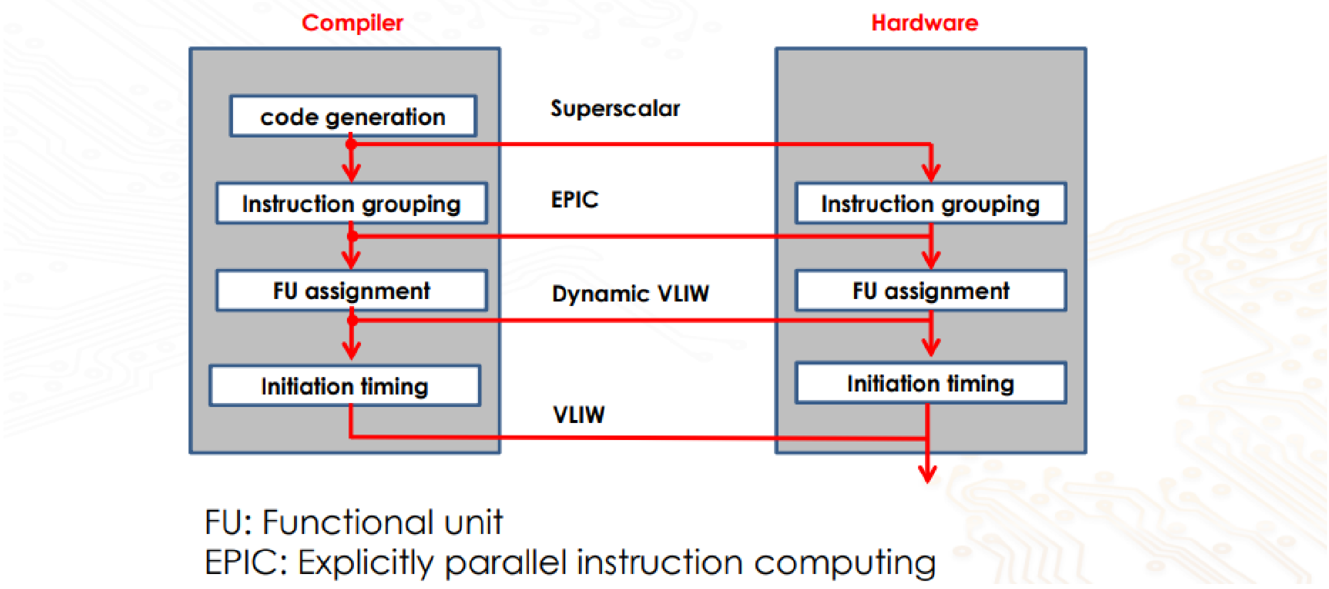
- TNTNTNTNTNTNTNTNTNTN → 50% accuracy
- (assuming initial to probable (weakly taken)
- Disadvantage: More hardware

4. Instruction Level Parallelism ILP

- Involves the execution of more than one instruction at the same time
- **Multi-issue datapath** allows execution of two or more instructions in parallel
- **N-issue architecture** $\Longrightarrow$ Executes N instructions at the same time



- Things to consider
  - Which instructions to be executed in parallel (need to check data dependency)
  - Where should instructions be executed
  - When should instructions be executed
- Two approaches to achieve ILP is by using a superscalar processor (hardware) or a Very Long Instruction Word (VLIW) processor (software)

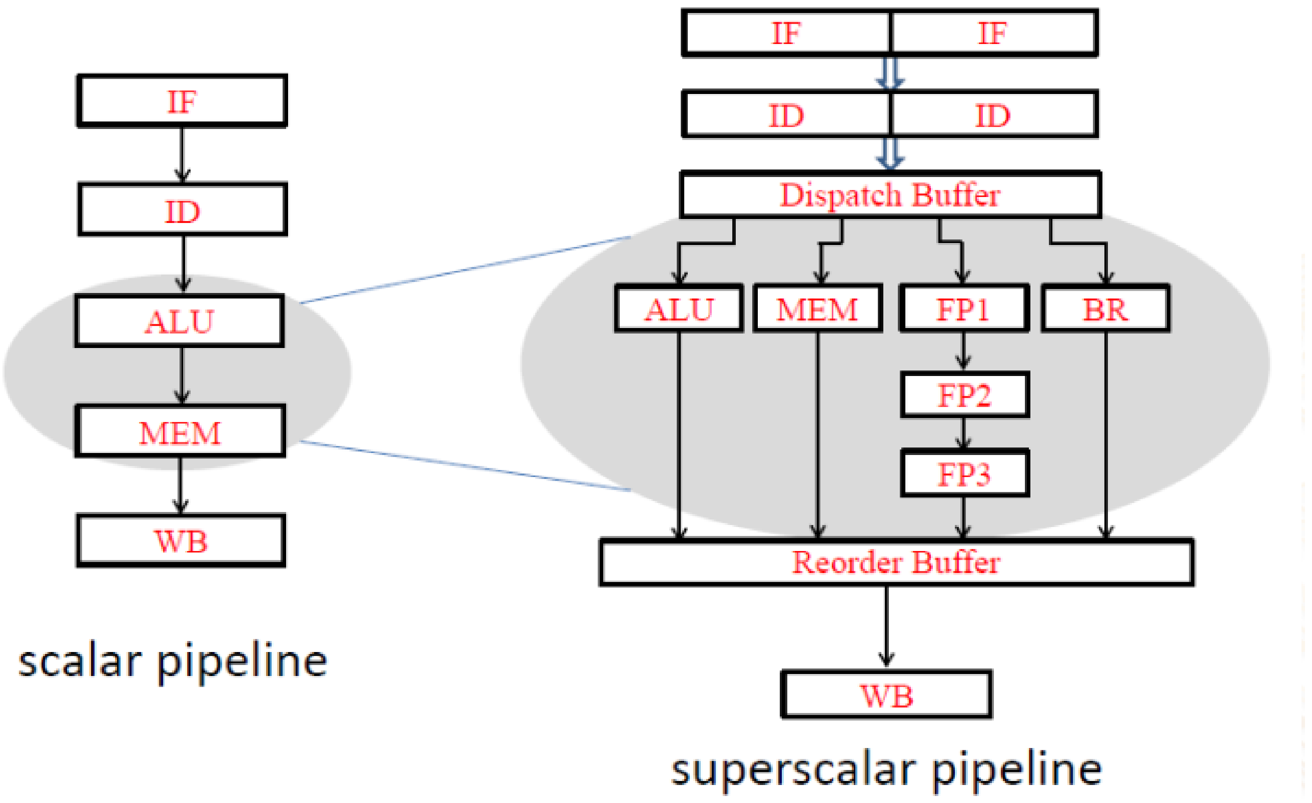


Superscalar Processing

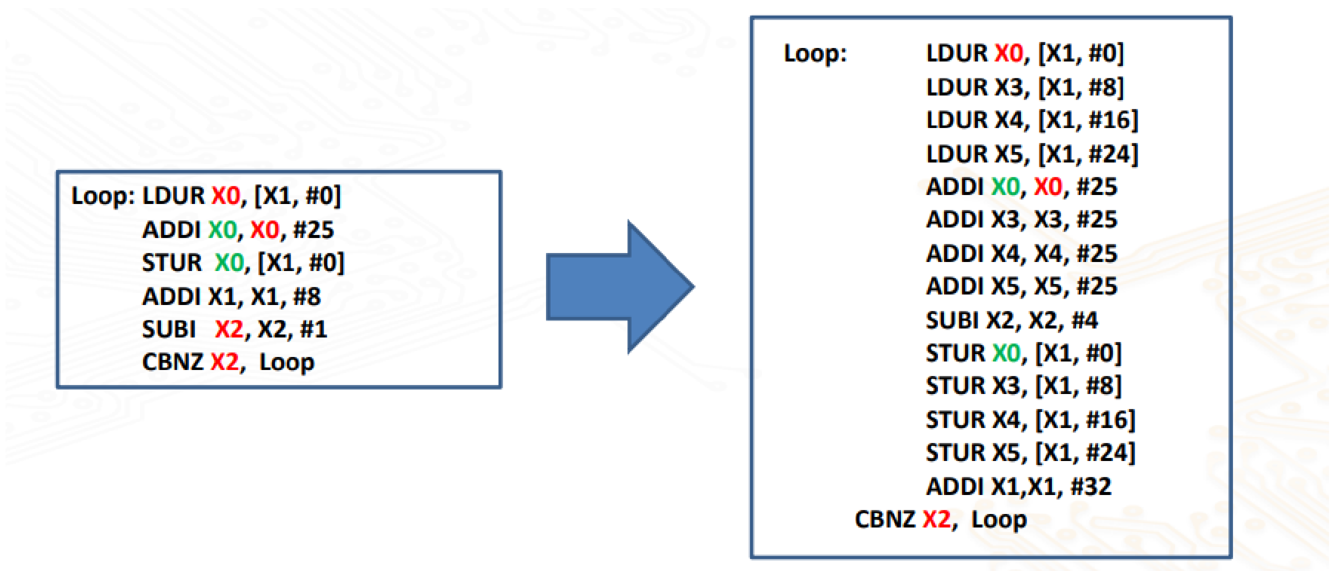
1. Multiple instructions are fetched in parallel
2. Instructions are decoded in parallel

3. Instructions dispatched to multiple functional units for execution in a clock cycle

- **Requirements**
  - Multiple FUs
  - More than 1 independent instructions
  - Specialised hardware needed to issue multiple independent instructions for simultaneous execution



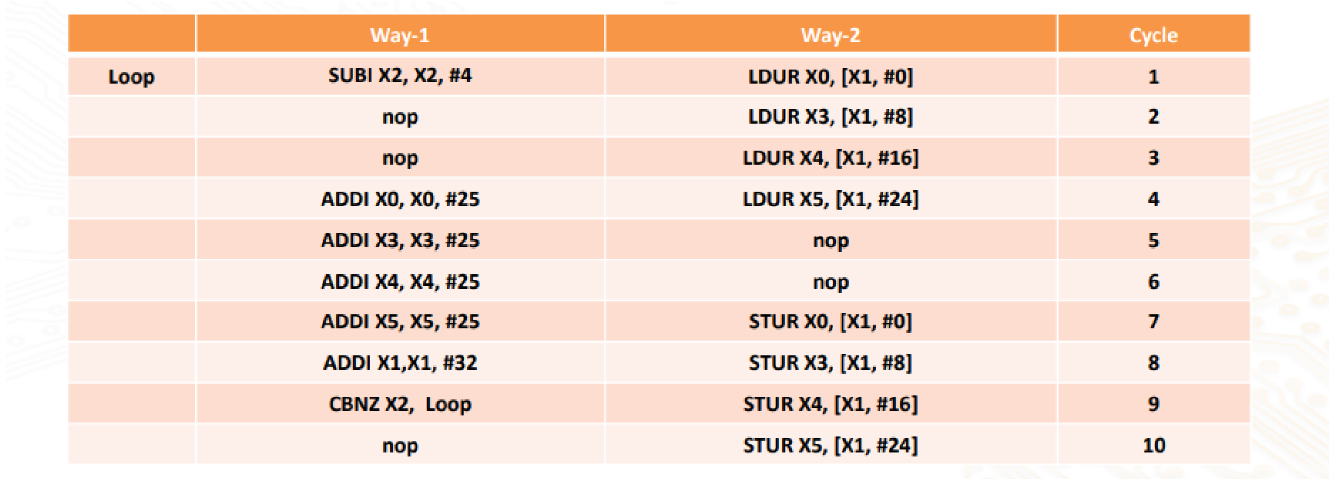
- **Out of Order execution**
  - Instructions are first fetched and decoded in compiler generated order and placed in a dispatch buffer which dispatches instructions into different FUs
  - Instructions are executed in random orders; independent instructions behind a stalled instruction can be executed prior to stalled instruction
  - Instruction completion **may be** in order after going through a reorder buffer
- More parallelism can be achieved by
  - Reordering instructions if it does not violate data dependence
  - Execute Speculatively without knowing if an instruction needs to be executed (branching)
  - Scheduling dynamically by executing instructions as soon as dependencies are satisfied and FUs are available
- Loop unrolling leads to multiple replications of the loop body which can be independent instructions that can be executed in parallel



```
Loop: LDUR X0, [X1, #0]
      ADDI X0, X0, #25
      STUR  X0, [X1, #0]
      ADDI X1, X1, #8
      SUBI   X2, X2, #1
      CBNZ X2,  Loop
```

```
Loop:        LDUR X0, [X1, #0]
             LDUR X3, [X1, #8]
             LDUR X4, [X1, #16]
             LDUR X5, [X1, #24]
             ADDI X0, X0, #25
             ADDI X3, X3, #25
             ADDI X4, X4, #25
             ADDI X5, X5, #25
             SUBI X2, X2, #4
             STUR X0, [X1, #0]
             STUR X3, [X1, #8]
             STUR X4, [X1, #16]
             STUR X5, [X1, #24]
             ADDI X1,X1, #32
        CBNZ X2,  Loop
```

after loop unrolling by 4 and instruction reordering



	Way-1	Way-2	Cycle
Loop	SUBI X2, X2, #4	LDUR X0, [X1, #0]	1
	nop	LDUR X3, [X1, #8]	2
	nop	LDUR X4, [X1, #16]	3
	ADDI X0, X0, #25	LDUR X5, [X1, #24]	4
	ADDI X3, X3, #25	nop	5
	ADDI X4, X4, #25	nop	6
	ADDI X5, X5, #25	STUR X0, [X1, #0]	7
	ADDI X1,X1, #32	STUR X3, [X1, #8]	8
	CBNZ X2, Loop	STUR X4, [X1, #16]	9
	nop	STUR X5, [X1, #24]	10

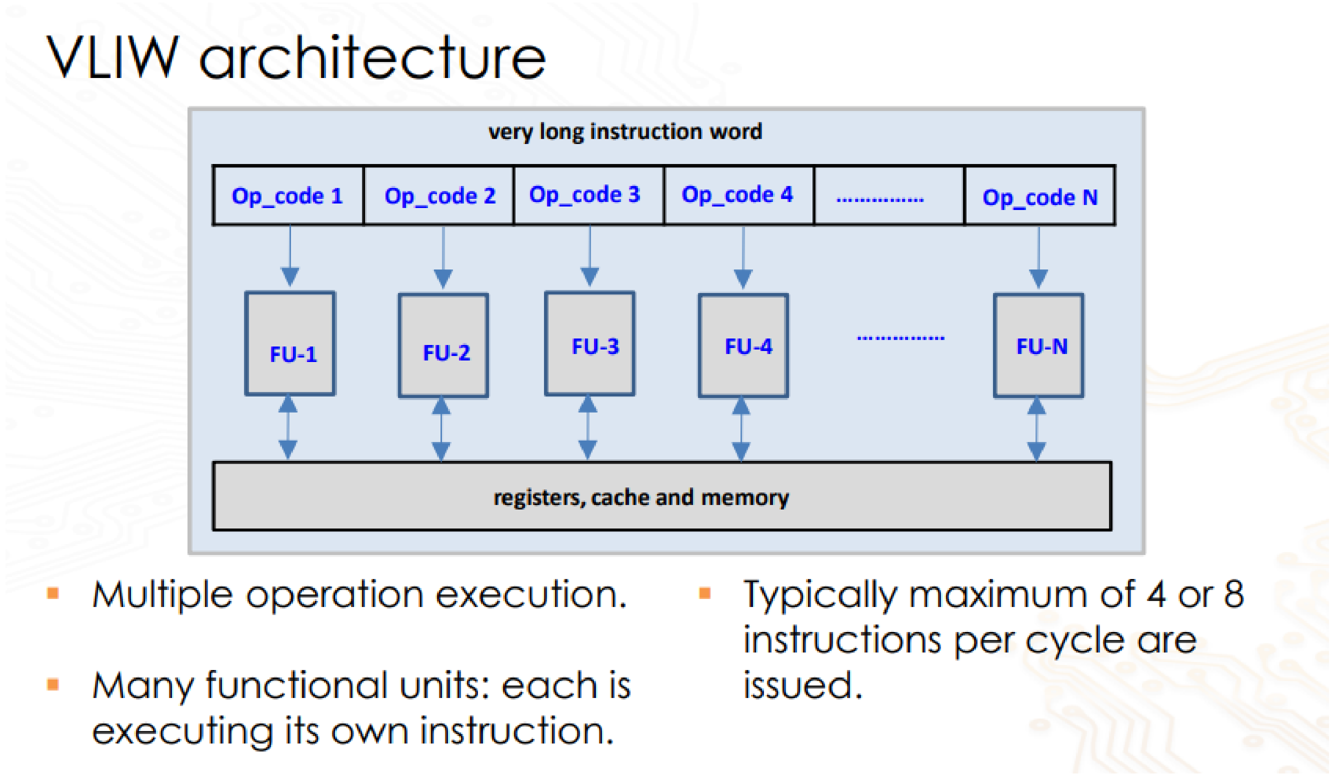
2-way super-scalar after 4-unrolling and reordering : CPI = 10/15= 2/3

- **Stages in superscalar pipeline**
  - Multiple instructions are fetched in every clock cycle and stored in instruction buffer
  - Instructions in the instruction buffer are decoded and sent to the dispatch buffer
  - When operands and respective FUs are available, dispatcher will dispatch instructions into FUs to be executed and marked 'finished' after execution and sent to the reorder buffer
  - Instructions exit the reorder buffer in order and results are written in the processor registers in order
  - Update memory if any

VLIW Processing

- Packs multiple independent operations into one instruction
- Compiler breaks the program instruction into basic operations that can be performed by VLIW processor (which contains multiple FUs) in parallel

- CPI will be reduced significantly if available slots (FUs) are filled



VLIW architecture

- Multiple operation execution.
- Many functional units: each is executing its own instruction.
- Typically maximum of 4 or 8 instructions per cycle are issued.

- Compiler prepares fixed packets of multiple operations
  - Dependencies are determined by compiler which is used to schedule according to the latencies of corresponding FUs
  - FUs are assigned by compiler and correspond to the position within the instruction packet / instruction slot
  - Compiler produces fully-scheduled, hazard-free code, means hardware does not need to check for dependencies during scheduling
- Compatibility across implementations is a major problem
  - VLIW code cannot run properly with different number of FUs or FUs with different latencies
  - Unscheduled events (cache misses) can stall entire processor
- Code density is an important concern
  - NOPs increase with branching
  - Reduce NOPs by compression