CE/CZ 3001:
Advanced Computer Architecture

(Module 4: Instruction Level Parallelism(ILP))

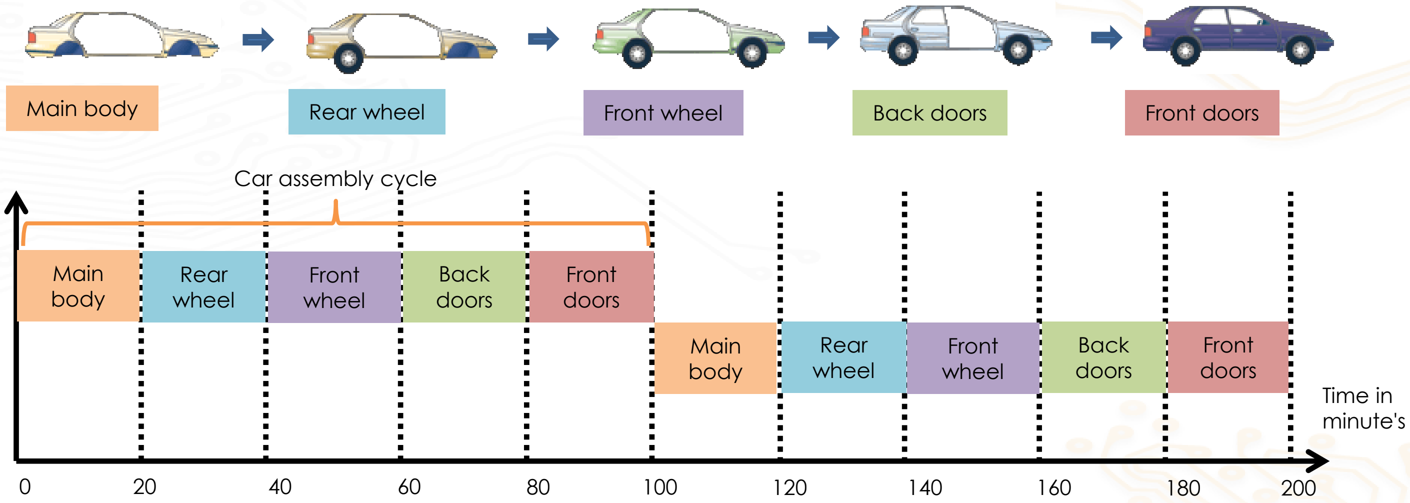
Dr Smitha K. G.
School of Computer Science
And Engineering

Overview

- Pipeline data-path
- Challenges in ILP realization
- Data-dependence
- Pipeline hazards and their solutions
- Out-of-order execution
- Branch prediction
- Dynamic scheduling
- VLIW and superscalar processors

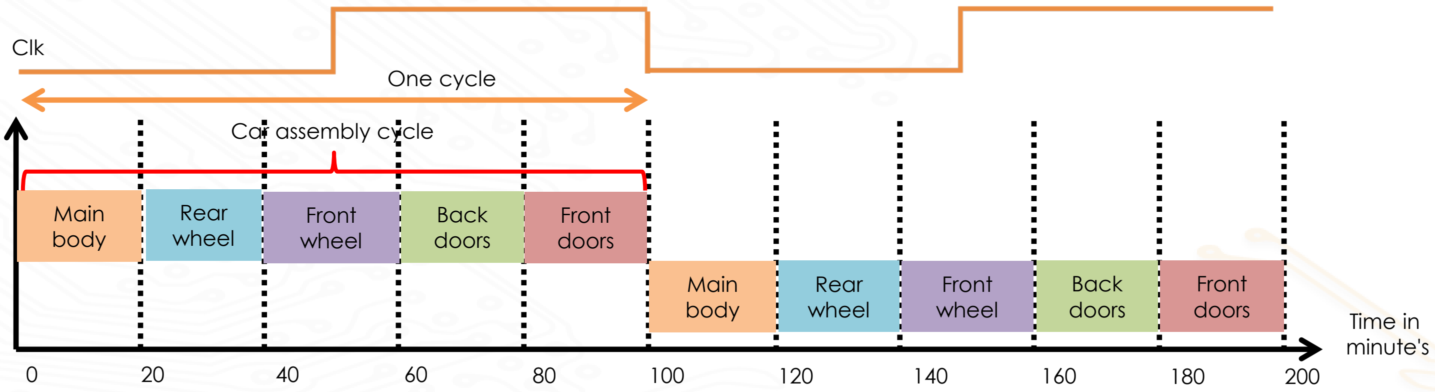
Building a Car

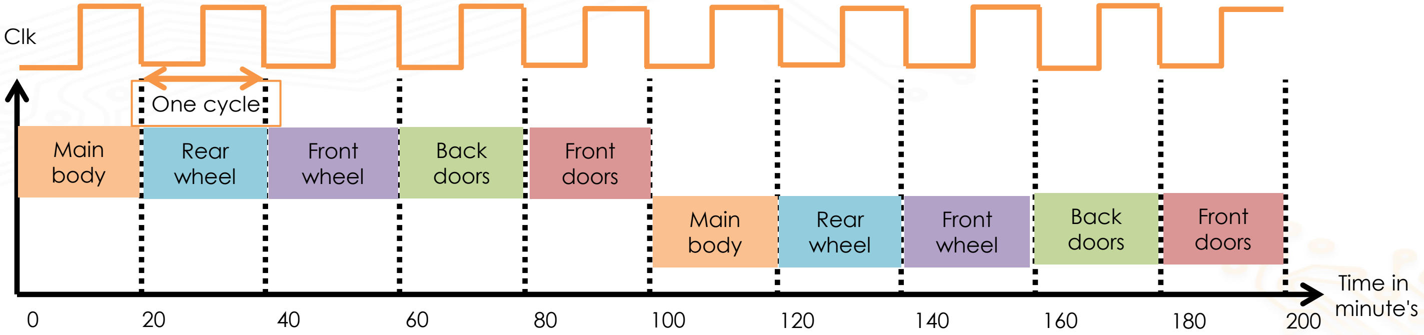
- Non pipeline example – car assembly line
  - Note that in each time unit, only **one** sub-task is active (other 4 sub-tasks are idle)



Advantage: Low CPI(1)
Disadvantage: Long clock period(to accommodate slowest instruction)

Single cycle and multi-cycle

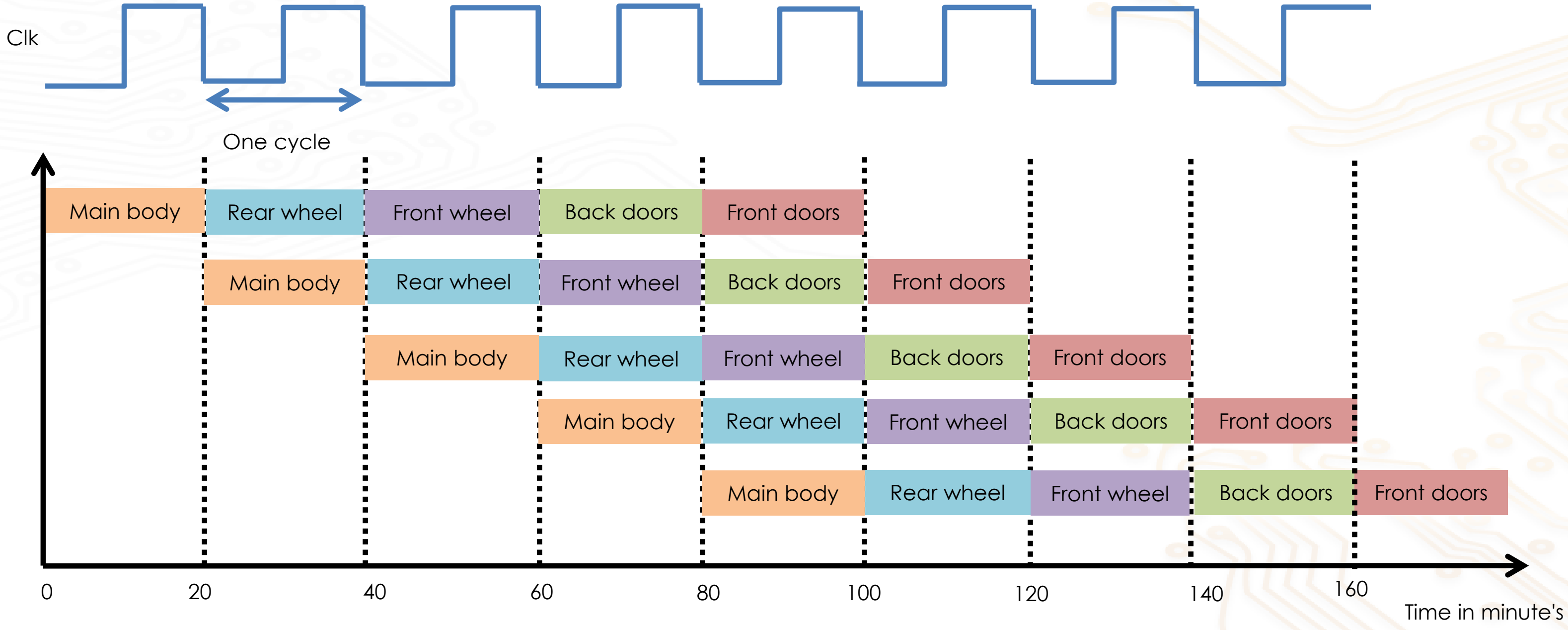




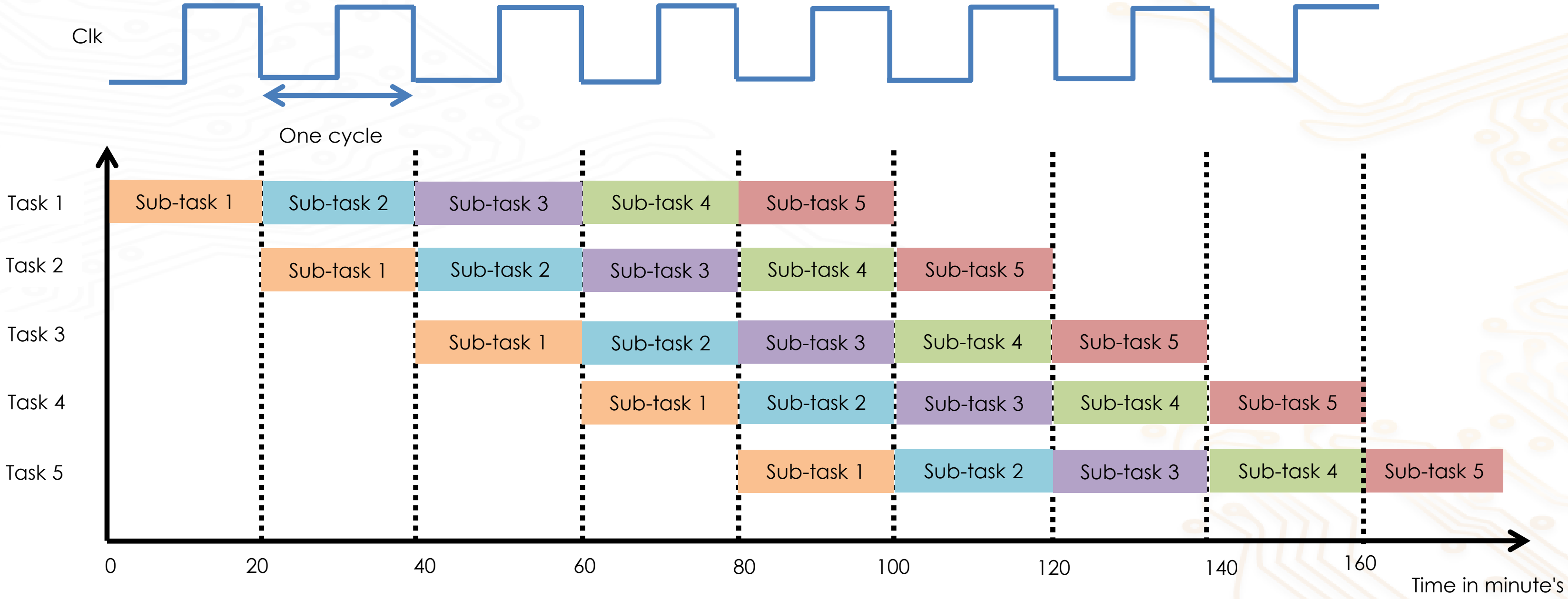
Advantage: Short clock period
Disadvantage: High CPI

Pipeline datapath (Part 1/3)

- **Pipelining** allows **multiple** sub-tasks to be carried out **simultaneously using independent resources**
- Need to **balance** time taken by each sub-task

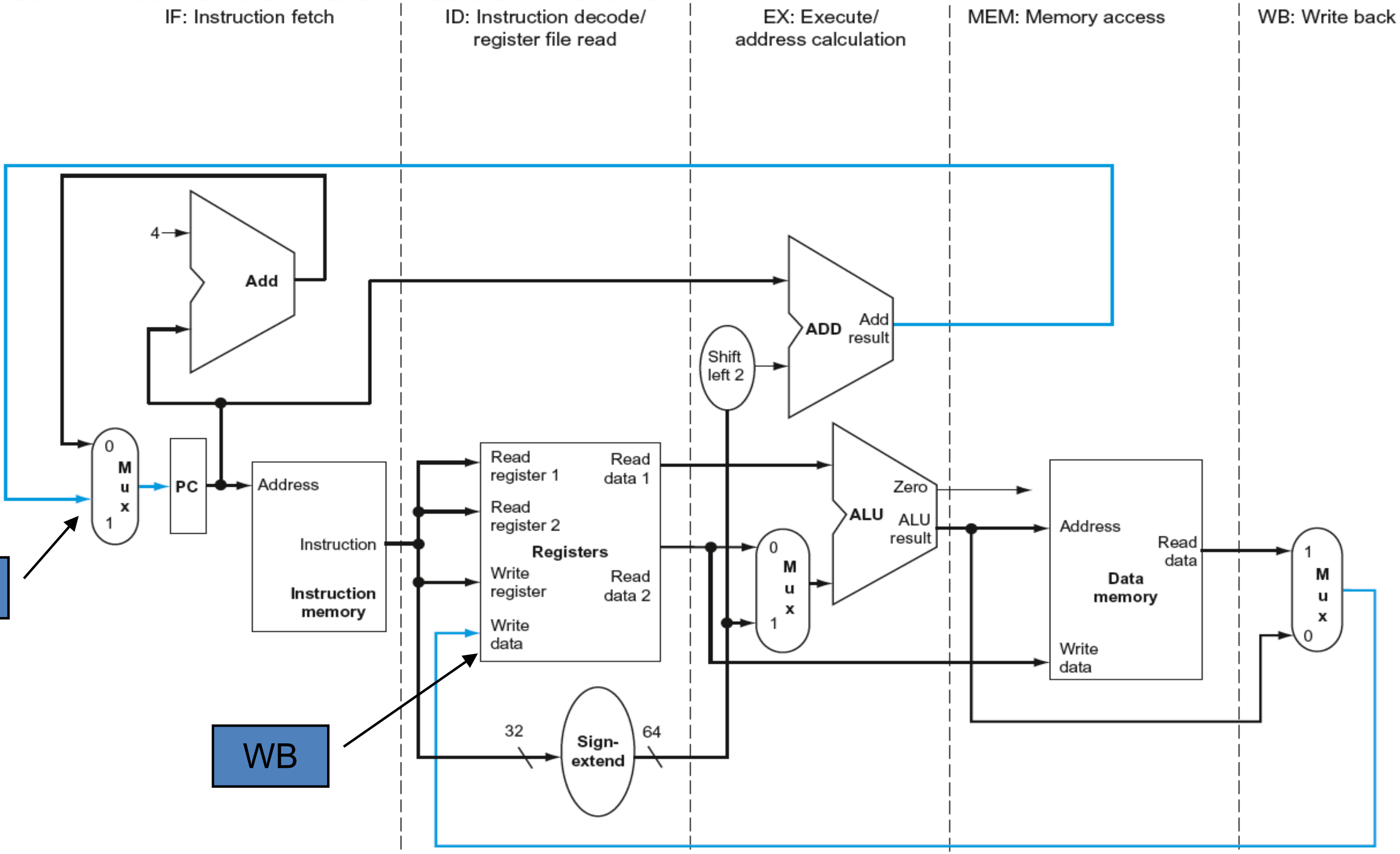


Pipeline datapath (Part 2/3)



Task 1	Sub-task 1	Sub-task 2	Sub-task 3	Sub-task 4	Sub-task 5				
Task 2		Sub-task 1	Sub-task 2	Sub-task 3	Sub-task 4	Sub-task 5			
Task 3			Sub-task 1	Sub-task 2	Sub-task 3	Sub-task 4	Sub-task 5		
Task 4				Sub-task 1	Sub-task 2	Sub-task 3	Sub-task 4	Sub-task 5	
Task 5					Sub-task 1	Sub-task 2	Sub-task 3	Sub-task 4	Sub-task 5

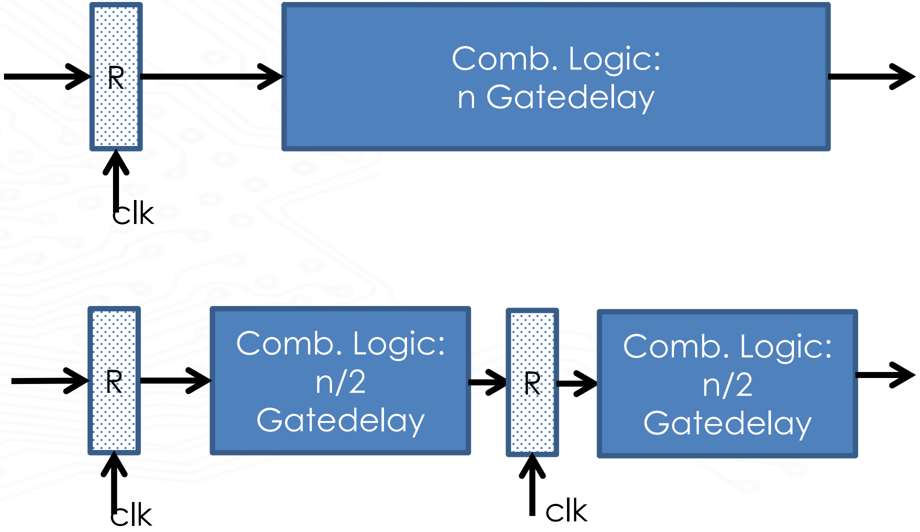
Pipeline datapath (Part 3/3)



MEM

WB

Ideal Pipelining



R

Comb. Logic:
n Gatedelay

clk

Frequency of Operation = ~(1/n)

R

Comb. Logic:
n/2
Gatedelay

R

Comb. Logic:
n/2
Gatedelay

clk

clk

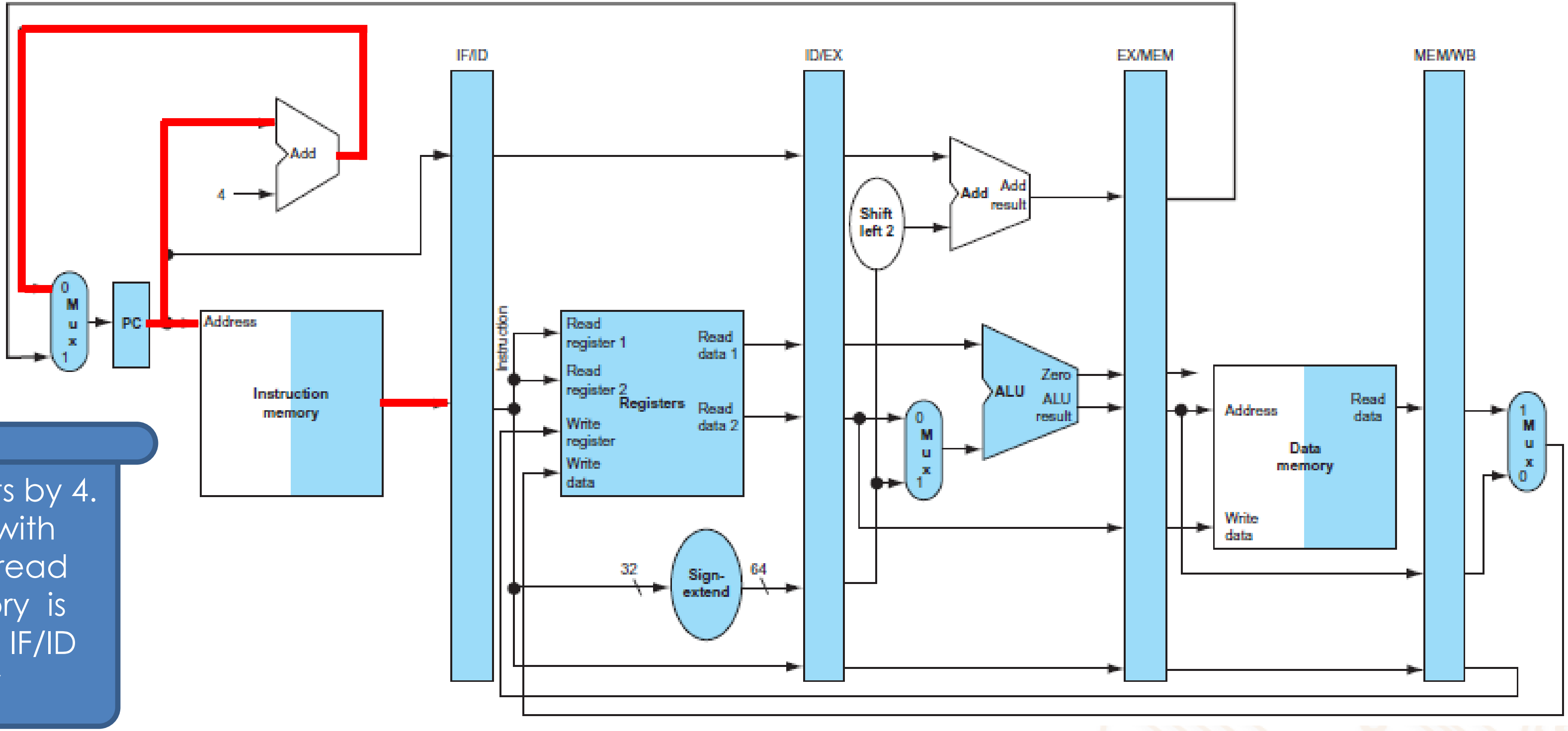
Frequency of Operation = ~(2/n)

- So an n-stage pipeline means that the CPU can operate at a maximum of n times faster than without a pipeline (ideally!).
- There is an increase in latency due to register delays.

Pipelined datapath – example 1 (Part 1/5)

LDUR X0, [X1,#100]

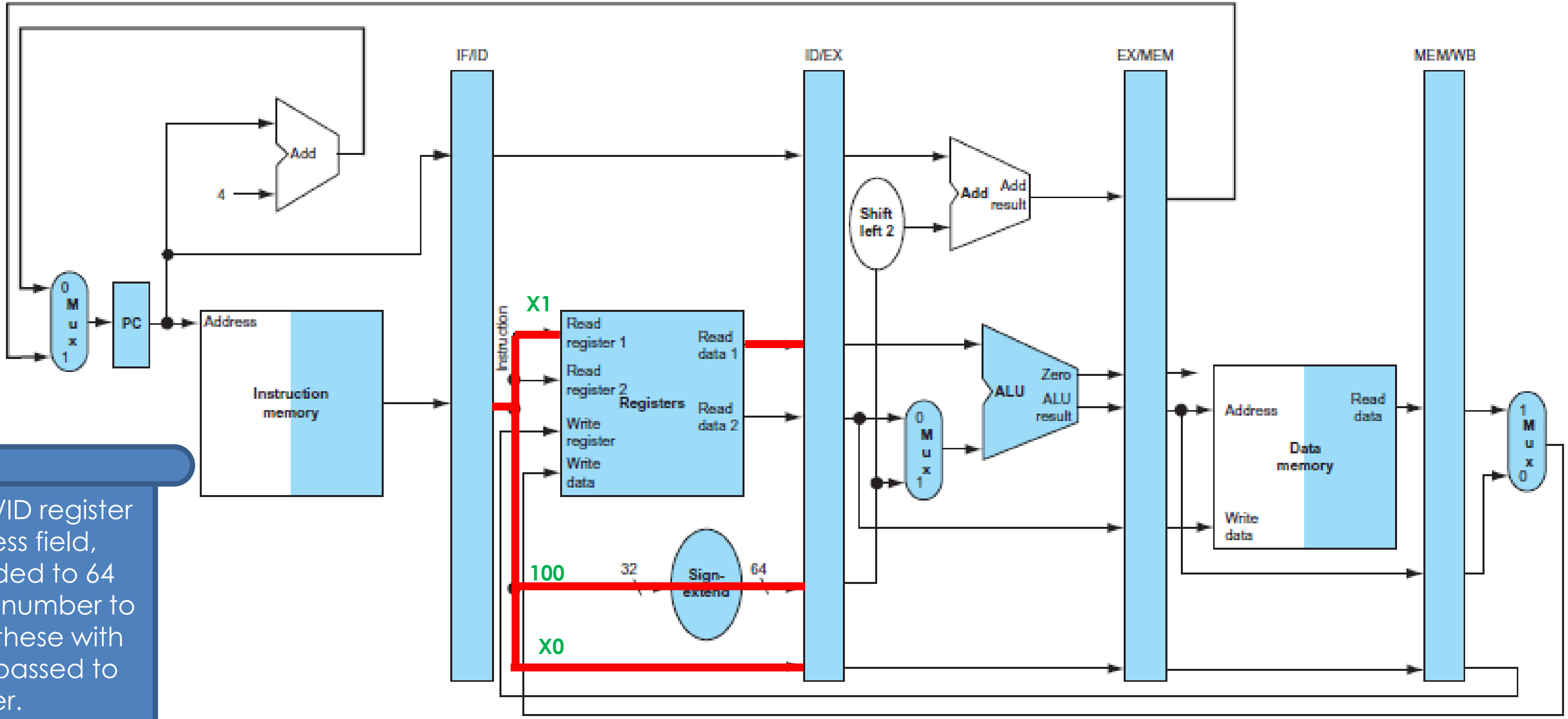
Representation scheme: write on first half cycle of clock and read on second half cycle of clock



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

Pipelined datapath – example 1 (Part 2/5)

LDUR X0, [X1,#100]

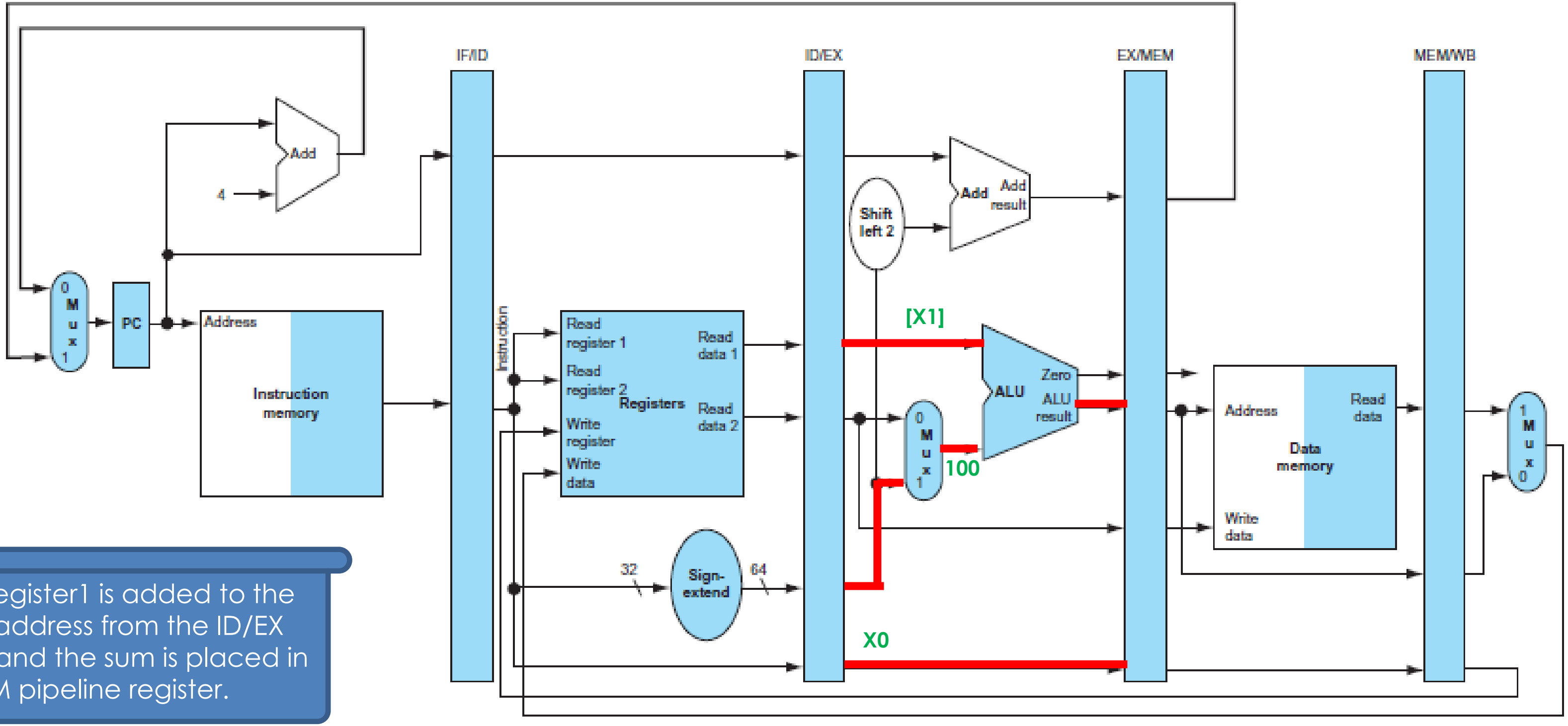


Instruction word in IF/ID register supply's the address field, which is sign extended to 64 bits, and the register number to read register. Both these with incremented PC is passed to ID/EX register.

LDUR X0, [X1,#100]

Pipelined datapath – example 1 (Part 3/5)

LDUR X0, [X1,#100]

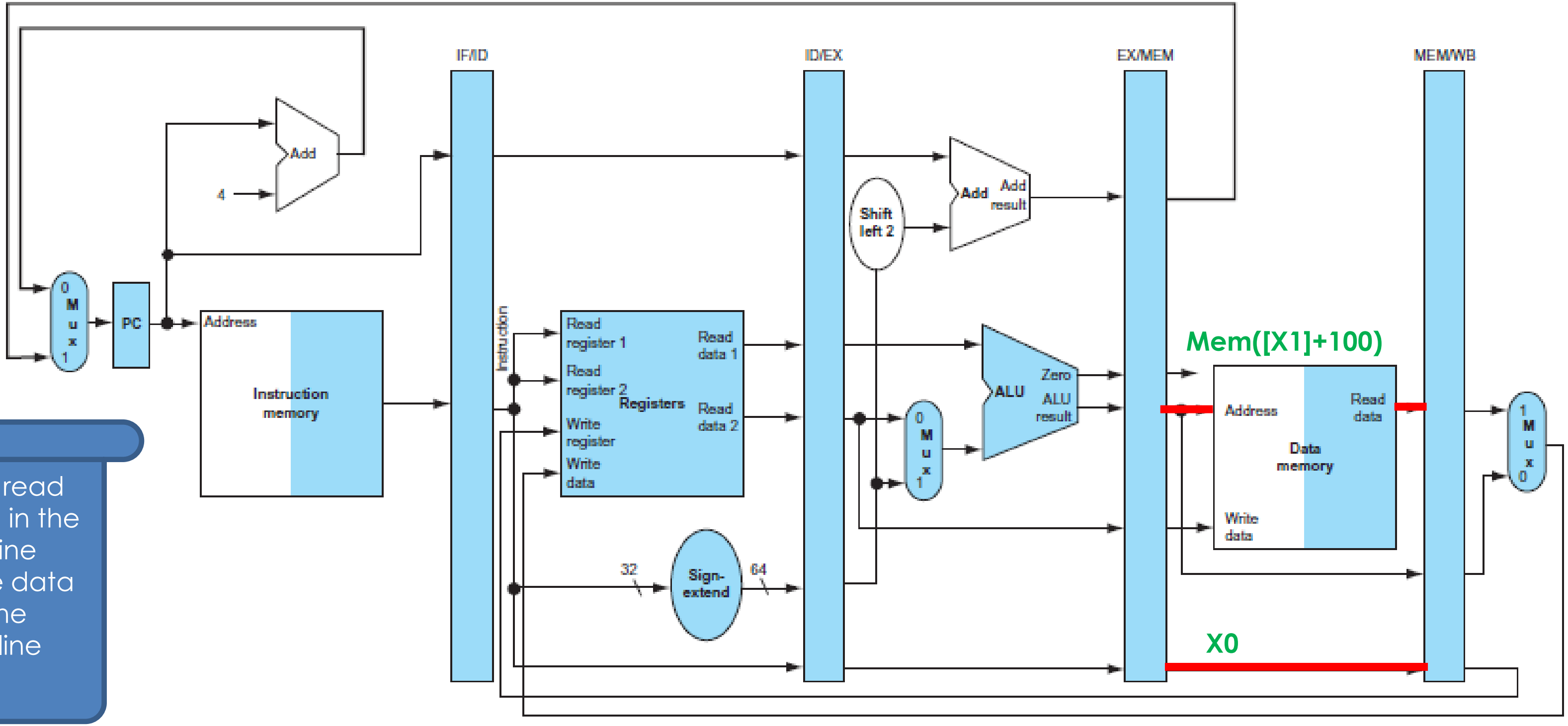


The content of register1 is added to the sign-extended address from the ID/EX pipeline register, and the sum is placed in the EX/MEM pipeline register.

LDUR X0, [X1,#100]

Pipelined datapath – example 1 (Part 4/5)

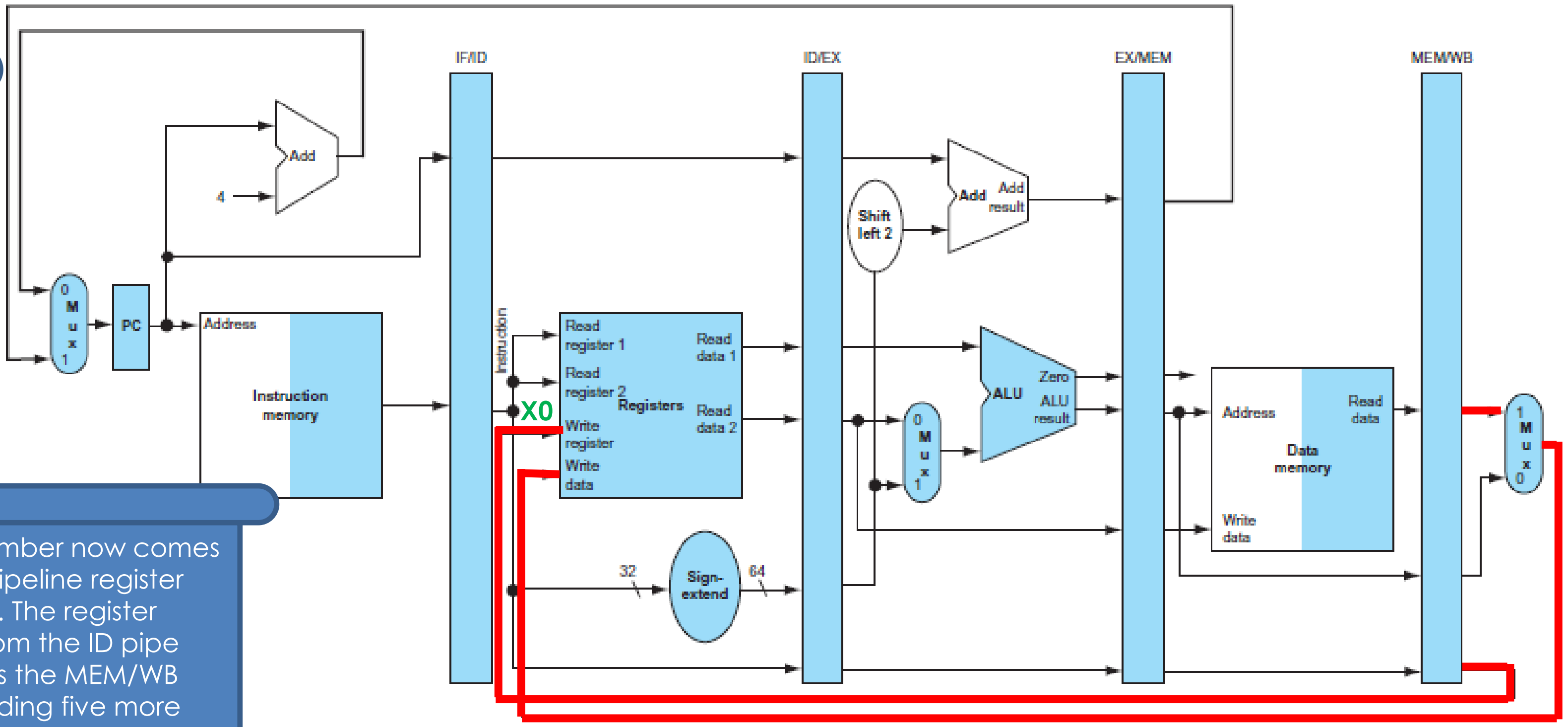
LDUR X0, [X1,#100]



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

Pipelined datapath – example 1 (Part 5/5)

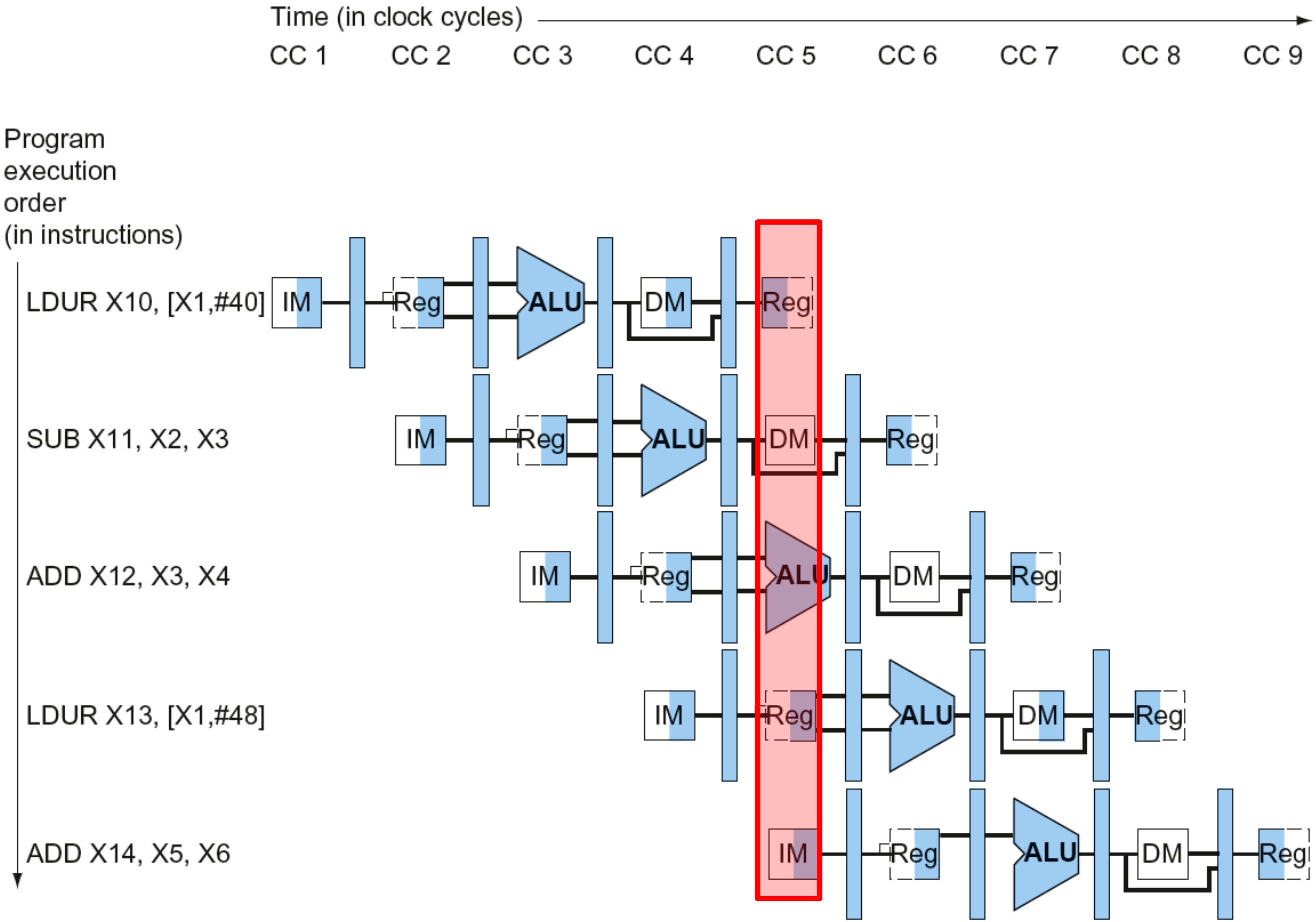
Data is read from the MEM/WB pipeline register and written into the register file in the middle of the datapath



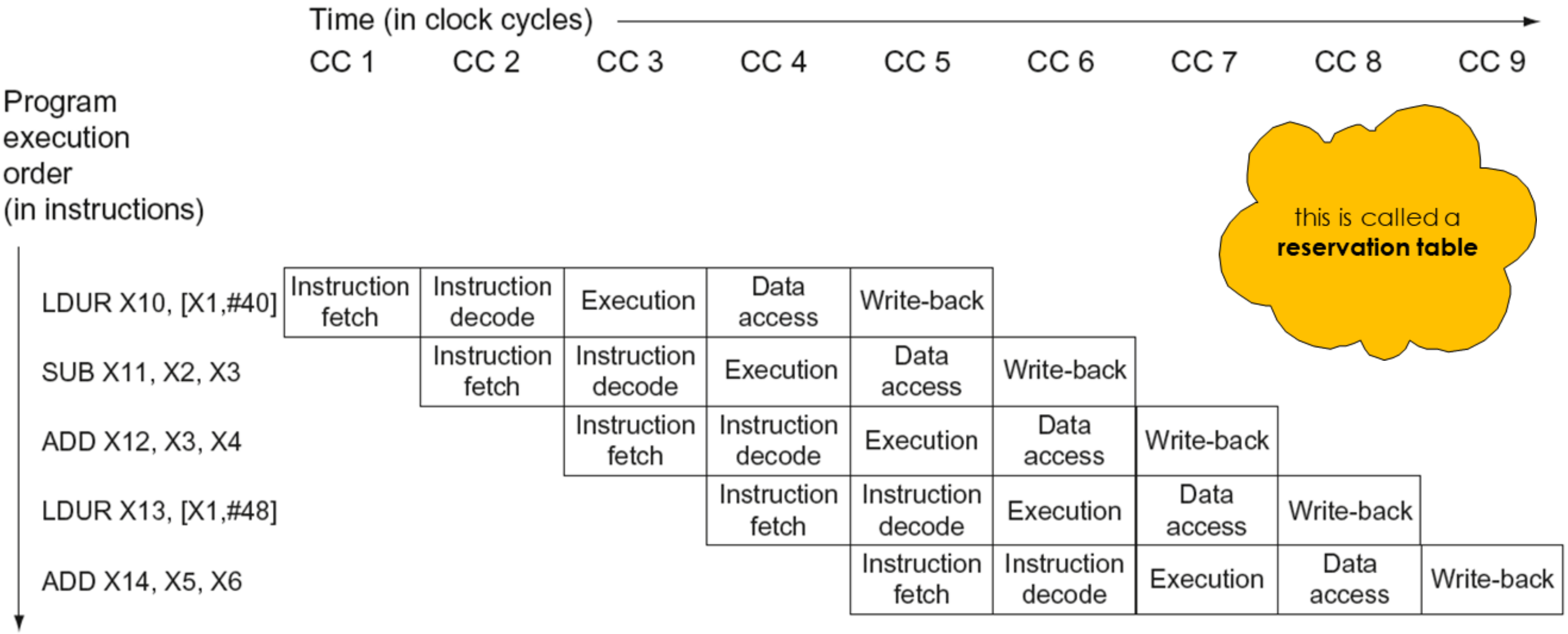
The write register number now comes from the MEM/WB pipeline register along with the data. The register number is passed from the ID pipe stage until it reaches the MEM/WB pipeline register, adding five more bits to the last three pipeline registers.

Mem ([X1]+100)

Example of ideal pipelining



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

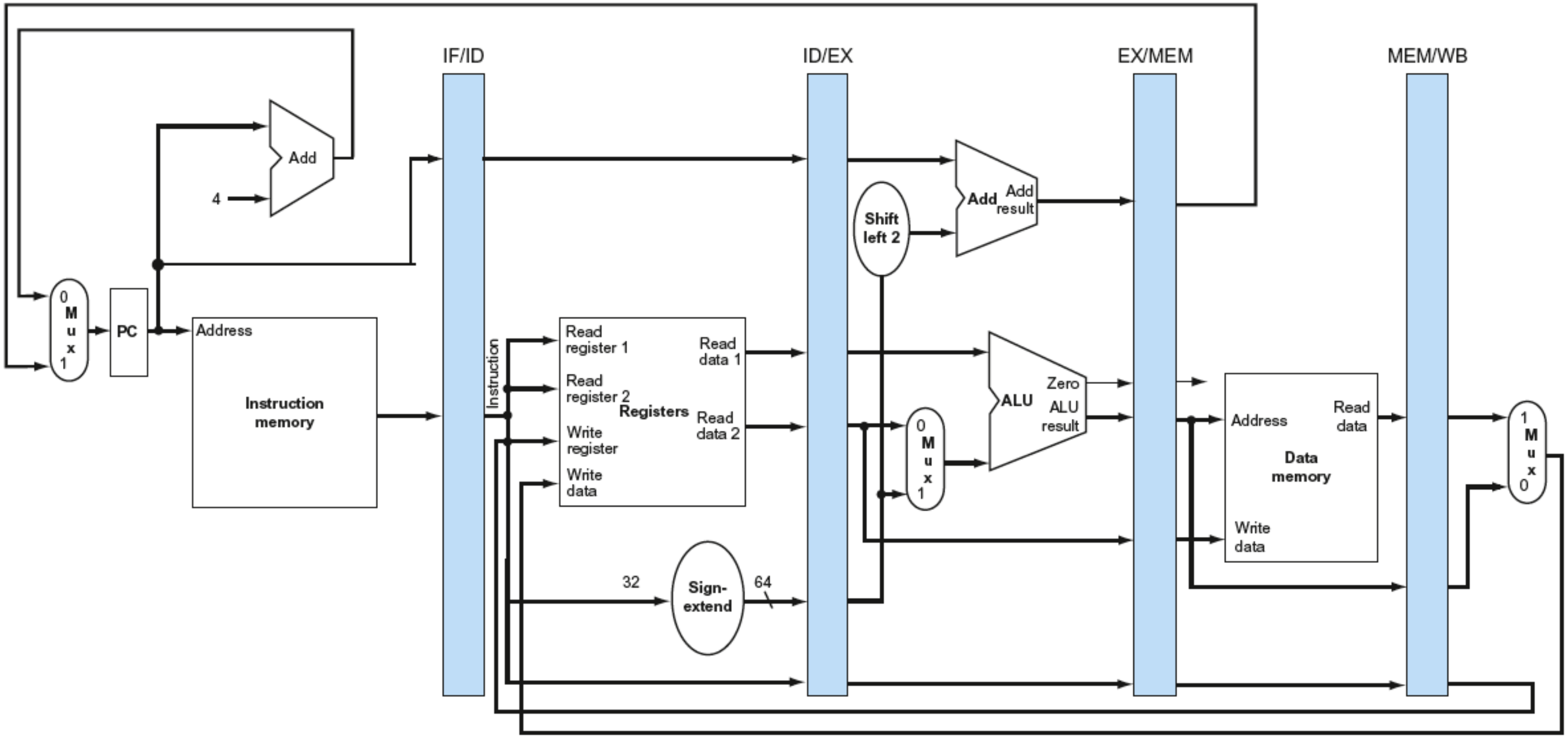
this is called a **reservation table**

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

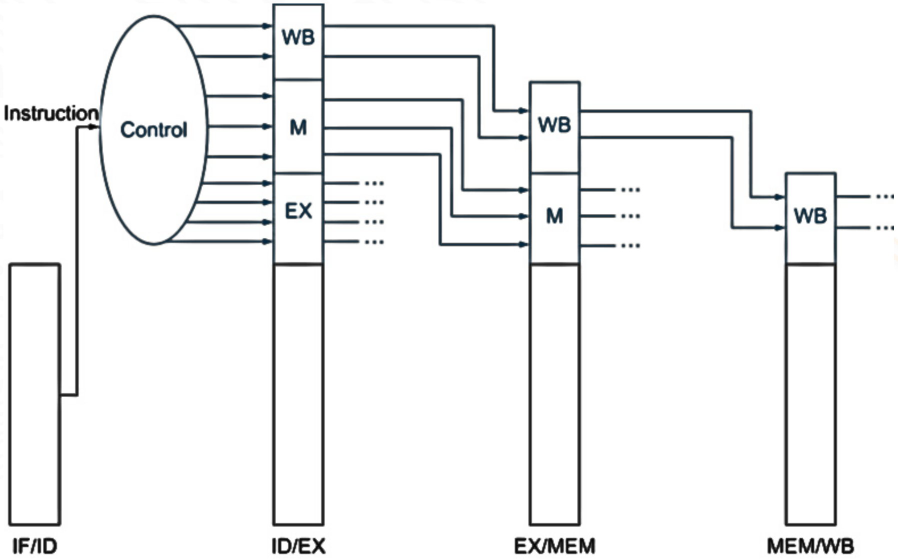
Why this is ideal?

Pipelined datapath – example 2

ADD X14, X5, X6	LDUR X13, [X1,48]	ADD X12, X3, X4	SUB X11, X2, X3	LDUR X10, [X1,40]
Instruction fetch	Instruction decode	Execution	Memory	Write-back

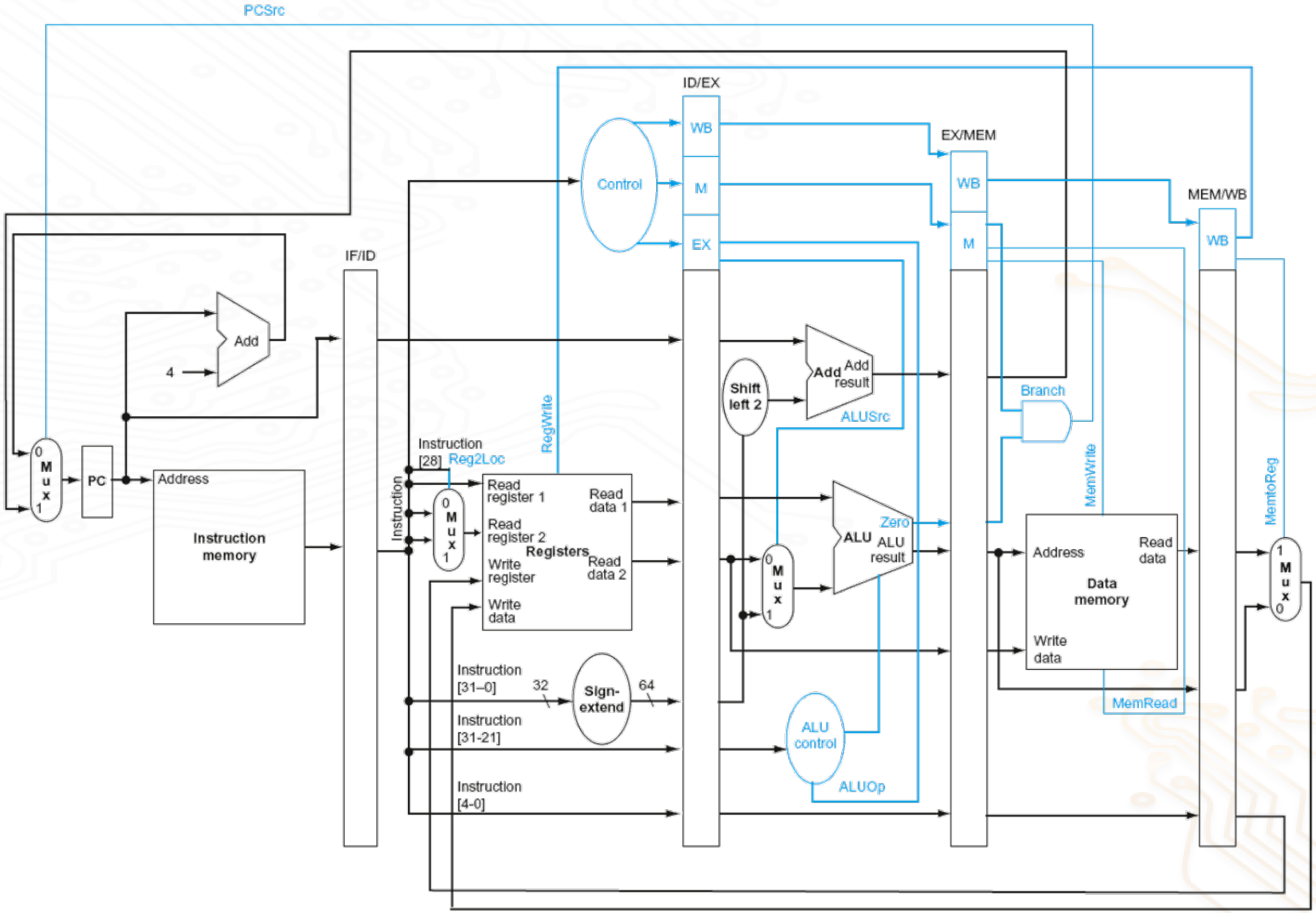


Synchronization of the control signals in pipelined datapath



Instruction	Instruction decode stage control lines	Execution/address calculation stage control lines			Memory access stage control lines			Write-back stage control lines	
	Reg2Loc	ALUOp1	ALUOp0	ALUSrc	Branch	MemRead	MemWrite	RegWrite	MemtoReg
R-format	0	1	0	0	0	0	0	1	0
LDUR	X	0	0	1	0	1	0	1	1
STUR	1	0	0	1	0	0	1	0	X
CBZ	1	0	1	0	1	0	0	0	X

Pipelined datapath and control



Challenges in instruction pipelining realisation

Ideal pipeline

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

So is pipeline always moving?

Real pipeline

- Except load instruction all other instructions do not need all stages
- Not all pipeline stages involve the same time to complete their respective sub-tasks
- The execution of one instruction may depend on one of the preceding instructions

Complications

- Datapath
  - **Many instructions in flight**
- Control
  - **Must correspond to multiple instructions**
- Instructions may have
  - **data and control flow *dependences***
  - **One may have to stall and wait for another**

Fundamental issues in pipelining

- Balancing work in pipeline stages
  - How many stages and what is done in each stage
- Keeping the pipeline **correct, moving, and full** in the presence of events that disrupt pipeline flow
  - Handling dependences
    - **Data**
    - **Control**
  - Handling resource contention
  - Handling resource contention
- Advanced: Improving pipeline throughput
  - Minimizing stalls

Pipeline Hazards (Part 1/2)

- Stalling of the pipeline – drop in efficiency **(hazard).**
- Various types of hazards are **data hazard, control/instruction hazard and structural hazard.**
- **Structural hazard**: If two instructions require to use of a given hardware resource at the same time that will lead to a stall in the pipeline (one instruction has to wait at least for a clock cycle).

  Example: Consider we have only one memory. For a case when write stage access the memory for writing the result and the instruction fetch stage tries to fetch the instruction from the memory at the same time.

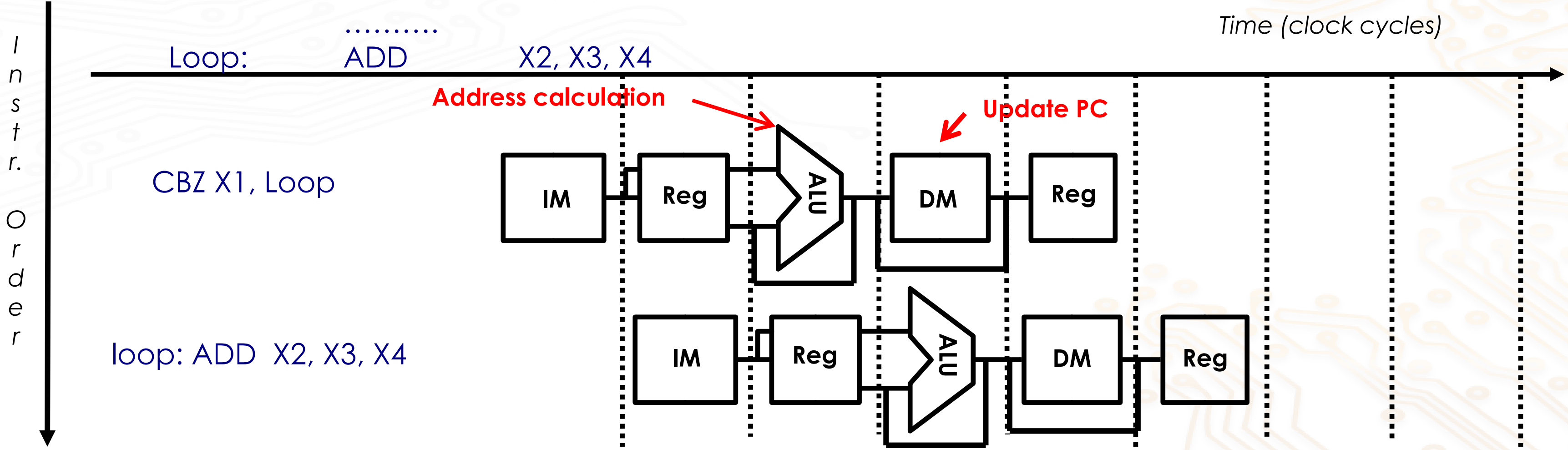
  Without two memories –structural hazard.
- **Data hazard**: either the source or destination register of an instruction are not available at the time expected in the pipeline. Hence results in pipeline stalls.

Pipeline Hazards (Part 2/2)

- **Control/Instruction hazard:** Conditional and unconditional jumps, subroutine calls, and other program control instructions can stall a pipeline because of a delay in the availability of an instruction.

If missed, the instruction has to be fetched from main memory which introduces delay in the instruction fetch and the pipeline stalls.

```
        CBZ      X1,Loop
        ORR      X7,X8, X9
        ..........
Loop:   ADD      X2, X3, X4
```



Solutions to solve pipeline hazard

- **Structural hazard**
  - Can be solved by additional hardware elements
- **Data hazard**
  - Need to know the dependencies between data in the program and eliminate them
- **Control Hazard**
  - Need to predict the location of conditional /unconditional branch to avoid stalling

Data Dependencies

Types of data dependences

- True/flow dependence (**RAW** – read after write) – *j* cannot execute until *i* produces its result

  (*i* and *j* are two different instructions)

- Output dependence (**WAW** – write after write) – *j* cannot write its result until *i* has written its result

- Anti dependence (**WAR** – write after read) – *j* cannot write its result until *i* has read its sources

Data Dependencies (example-1)

True/RAW (read-after-write) dependency arise when one instruction relies upon the output from the previous instruction:

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

Output/WAW (write-after-write) dependency arise when two nearby instructions must write to the same location:

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

The SUBI must wait until ADD finishes writing.

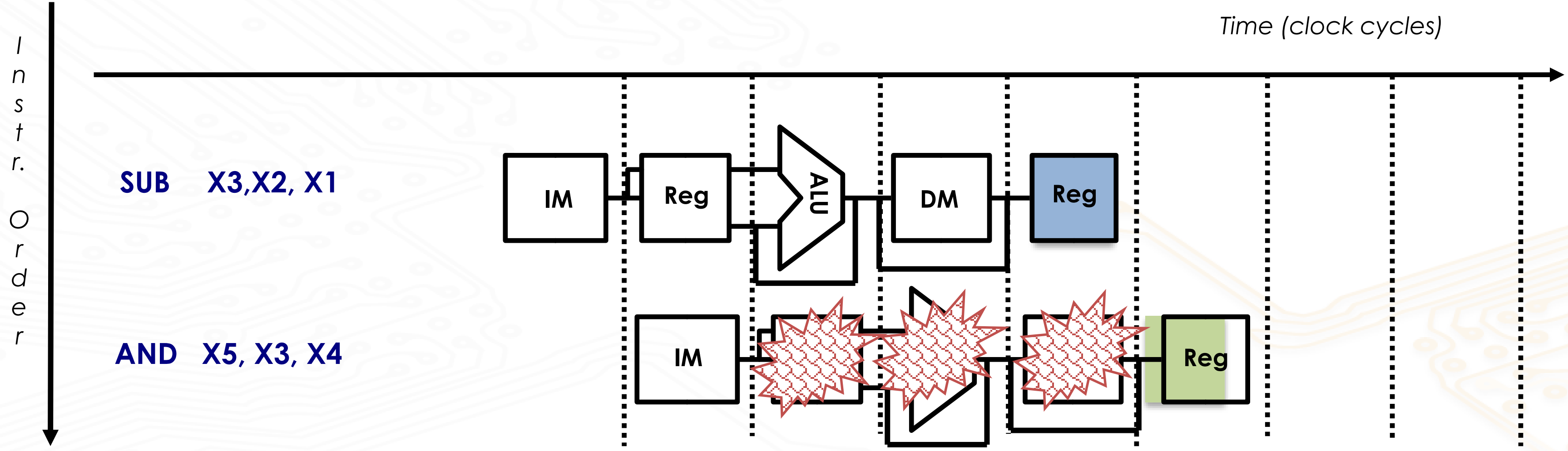
Anti/WAR (write-after-read) dependency in the bellow example, because the ANDI must not change the value of X1 before the ADD has used the value that is in X1.

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

How to handle data dependencies

- Anti and output dependences are easier to handle
- True (Flow or RAW)dependences are more difficult to handle as they constitute true dependence on a value
- **Five fundamental ways of handling True dependences**
    - Detect and wait until value is available in register file
        - Stall the program. (HARDWARE)
        - Compiler can also plug in the NOP instructions in between. (SOFTWARE)
    - Detect and forward / bypass data to dependent instruction
    - Detect and eliminate the dependence at the software level
        - No need for the hardware to detect dependence
    - Predict the needed value(s), execute "speculatively", and verify
    - Do something else (fine-grained multithreading)
        - No need to detect

Detect and wait



SUB X3,X2, X1

AND X5, X3, X4

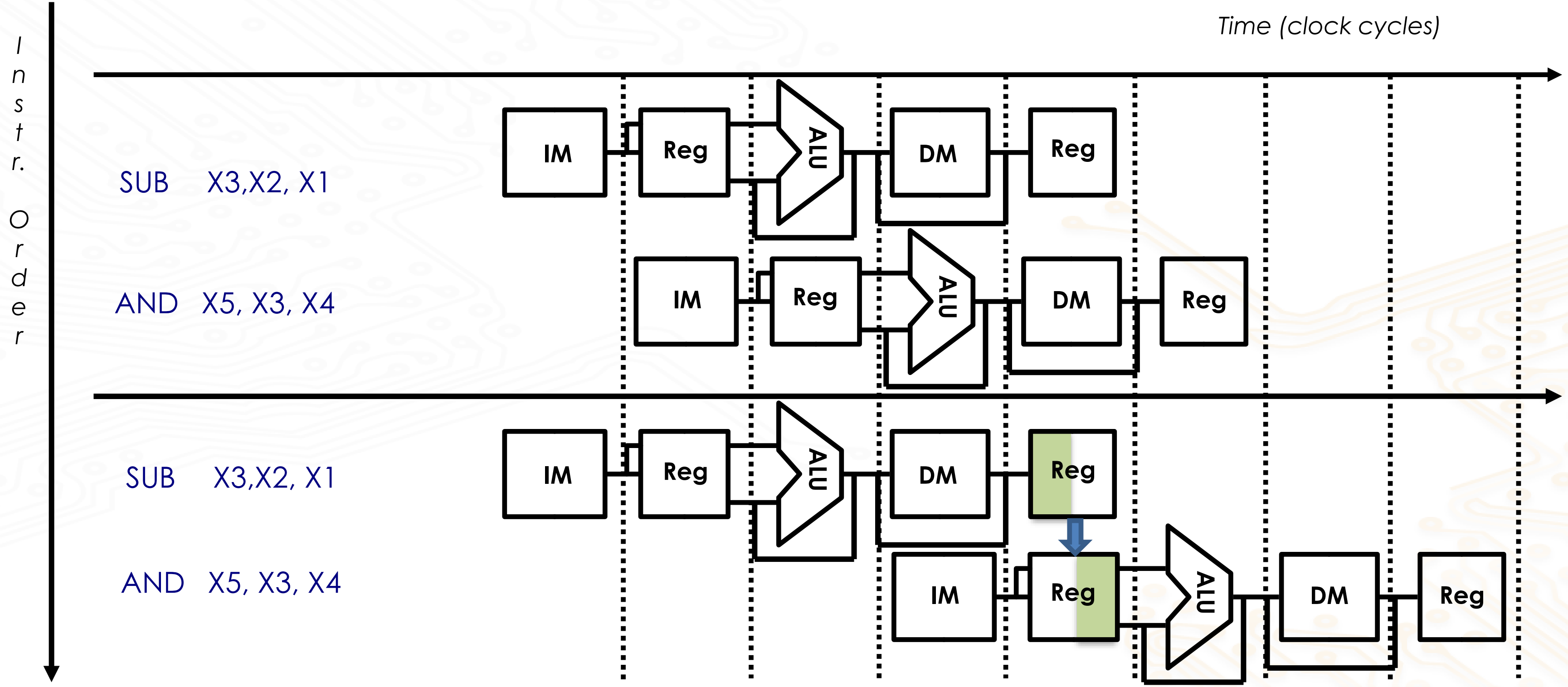
Hardware stall

Instr.	1	2	3	4	5	6	7	8	9
I1	IF	ID	EX	M	WB				
I2		IF	S	S	S	ID	EX	M	WB

Software inserting NOPs

Instr.	1	2	3	4	5	6	7	8	9
I1	IF	ID	EX	M	WB				
3NOPs									
I2					IF	ID	EX	M	WB

Data Forwarding – through register



Solution: write and read in the same cycle
Most processors have this as it is easy to implement

What happens when same register is read and written during same clock cycle?

Slide from last module

- The register read happens combinational (no clock)
- The register will be valid during the time it is read.
- The value returned will be the value written in the earlier clock cycle.
- The write of the register file occurs on the clock edge.
- If we want to read to return the value currently being written- we need additional logic.

Regfile program from lab2

```
reg [`DSIZE-1:0] regdata [0:`NREG-1];

integer i;
    always@(posedge clk)
        begin
            if(rst)
                begin
                    for (i=0; i<`NREG; i=i+1)
                    regdata[i] <=0;
                end
            else
        regdata[waddr] <=((wen == 1)) ? wdata : regdata[waddr];

        end

assign rdata1 = ((wen) && (waddr == raddr1)) ? wdata : regdata[raddr1];
assign rdata2 = ((wen) && (waddr == raddr2)) ? wdata : regdata[raddr2];
```

Detect and Forward / bypass

From pipeline stage registers to function units

LDUR X1, [X2, #2]

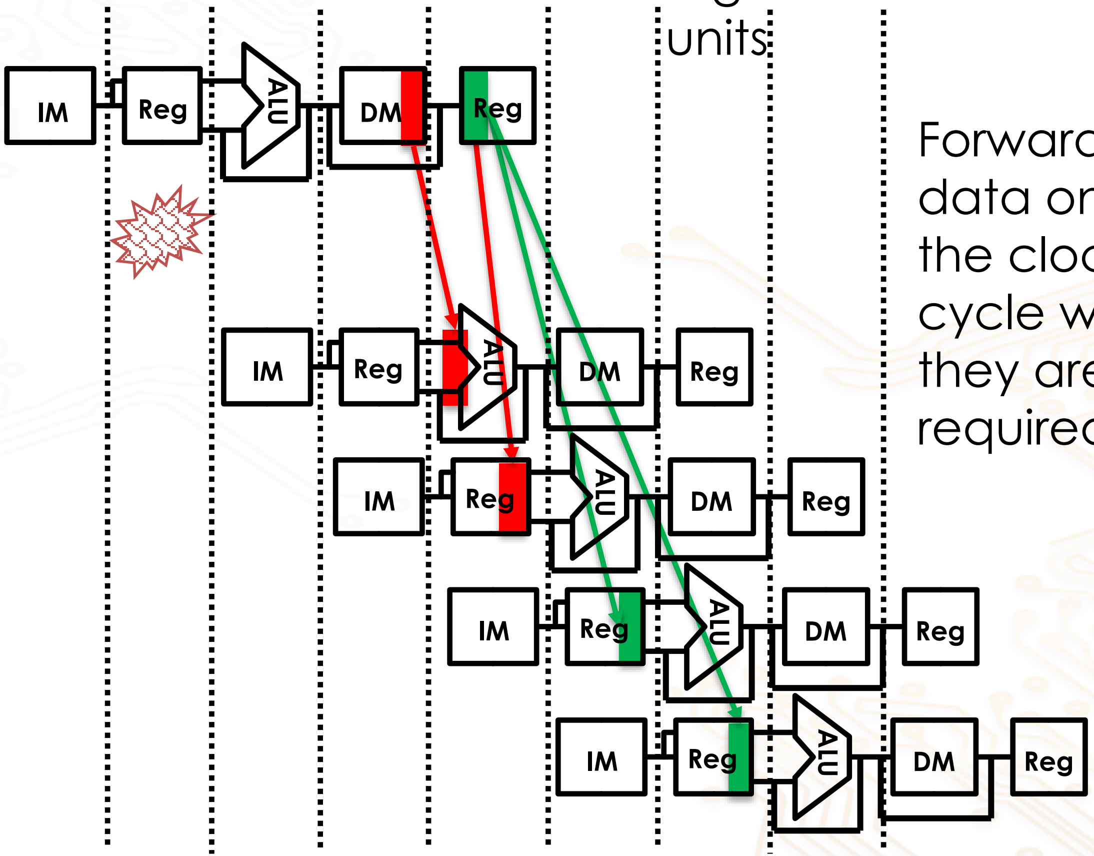
stall

SUB X4, X1, X5

AND X6, X1, X7

ORR X8, X1, X9

XOR X4, X1, X5



Forward data only in the clock cycle when they are required

Data forwarding – example 2

I1: ADD X1, X2, X3

I2: SUB X2, X1, X2

I3: AND X6, X7, X1

Without forwarding
(writeback and decode can happen simultaneously)

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

In-order and out of order execution

- In-order instruction execution
  - instructions are fetched, executed & completed in compiler generated order
  - one stalls, they all stall
  - instructions are statically scheduled
- Out-of-order instruction execution
  - instructions are fetched in compiler-generated order
  - but they may be executed in some other order
  - independent instructions behind a stalled instruction may pass it
  - instructions are dynamically scheduled during run time

Dynamic Scheduling

Out-of-order processors:

After instruction decode

- check for structural hazards:-
  - an instruction can be issued when a functional unit is available
  - an instruction stalls if no appropriate functional unit is free in that clock cycle
- check for data hazards:-
  - an instruction can execute if its operand has been loaded in the register
  - an instruction stalls if operands are not available
- don’t wait for previous instructions to execute if this instruction does not depend on them, i.e., independent ready instructions can execute before earlier instructions that are stalled

in-order processors

```
LDUR X1, [X4, #100]
ADD X2, X1, X4
(need to wait till the value is loaded)
SUB X5, X6, X7
```

out-of-order processors

```
LDUR X1, [X4, #100]
SUB X5, X6, X7
ADD X2, X1, X4(no stalls)
```

Instruction Reordering

```
LDUR    X1, [X0, #0]
LDUR    X2, [X0, #4]
ADD     X3, X1, X2
STUR    X3, [X0, #12]
LDUR    X4, [X0, #8]
ADD     X5, X1, X4
STUR    X5, [X0, #16]
```

→

```
LDUR    X1, [X0, #0]
LDUR    X2, [X0, #4]
LDUR    X4, [X0, #8]
ADD     X3, X1, X2
STUR    X3, [X0, #12]
ADD     X5, X1, X4
STUR    X5, [X0, #16]
```

ready instructions can execute before earlier instructions that are stalled
– Key idea behind out of order execution

Any other techniques to reduce stalling?

Register Renaming

```
WAR    ADD     X4, X2, X1;      X4 ← X2 + X1
       ANDI    X1, X0, #2;      X1 ← X0 & 2
             ↓ Rename X1 to X3
       ADD     X4, X2, X1;      X4 ← X2 + X1
       AND     X3, X0, #2;      X3 ← X0 & 2

WAW    ADD     X0, X2, X1;      X0 ← X2 + X1
       SUB     X0, X3, X5;      X0 ← X3 – X5
             ↓ Rename X0 to X4
       ADD     X0, X2, X1;      X0 ← X2 + X1
       SUB     X4, X3, X5;      X4 ← X3 – X5
```

More register resources will be needed

Loop unrolling

- loop unrolling leads to multiple replications of the loop body
  - unrolling creates longer code sequences
  - goal is 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];
}
```

- greater demand for registers
  - higher register pressure : more concurrency demands for more resources

How can loop unrolling help us to reduce the stalls and to improve CPI?

Loop unrolling example (Part 1/3)

```
Loop:   LDUR    X0, [X1, #0];       load to X0 from mem[0+X1]
        ADD     X4, X0, X2;         add [X0]+[X2]
        STUR    X4, [X1, #0];       store X4 to mem[0+X1]
        SUBI    X1, X1, #8;         decrement pointer 8
        CBNZ    X1,Loop;             branch X1!=zero
```

```
1. Loop:  LDUR    X0, [X1, #0]
2.        stall
3.        ADD     X4, X0, X2
4.        STUR    X4, [X1, #0]
5.        SUBI    X1, X1, #8;
6.        CBNZ    X1,Loop;
```

If full data forwarding is allowed then we will have one stall for data dependency

CPI = (No of instructions + no of stall) / No. of instruction = (5 + 1) / 5 = 1.2

Loop unrolling example (Part 2/3)

```
Loop:   LDUR    X0, [X1, #0]
        ADD     X4, X0, X2
        STUR    X4, [X1, #0]        ;drop SUBI & CBNZ
        LDUR    X6, [X1, #-8]
        ADD     X8, X6, X2
        STUR    X8, [X1, #-8]       ;drop SUBI & BNEZ
        LDUR    X10, [X1,#-16]
        ADD     X12, X10, X2
        STUR    X12, [X1, #-16]     ;drop SUBI & BNEZ
        LDUR    X14, [X1, #-24]
        ADD     X16, X14, X2
        STUR    X16, [X1, #-24]
        SUBI    X1, X1, #32         ;alter to 4*8
        CBNZ    X1,LOOP
```

1 stall here

full data forwarding is allowed

Rewrite loop to minimize stalls?

STRAIGHT FORWARED UNROLLING,
CPI= (14+4)/14=1.286

Loop unrolling example (Part 3/3)

Branch is not handled here

```
Loop:   LDUR    X0, [X1, #0]
        ADD     X4, X0, X2
        STUR    X4, [X1, #0]
        LDUR    X6, [X1, #-8]
        ADD     X8, X6, X2
        STUR    X8, [X1, #-8]
        LDUR    X10, [X1,#-16]
        ADD     X12, X10, X2
        STUR    X12, [X1, #-16]
        LDUR    X14, [X1,#-24]
        ADD     X16, X14, X2
        STUR    X16, [X1, #-24]
        SUBI    X1, X1, #32
        CBNZ    X1,LOOP
```

STRAIGHT FORWARED UNROLLING, CPI= (14+4)/14=1.286

```
Loop:   LDUR    X0, [X1, #0]
        LDUR    X6, [X1, #-8]
        LDUR    X10, [X1,#-16]
        LDUR    X14, [X1,#-24]
        ADD     X4, X0, X2
        ADD     X8, X6, X2
        ADD     X12, X10, X2
        ADD     X16, X14, X2
        STUR    X4, [X1, 0]
        STUR    X8, [X1, #-8]
        STUR    X12, [X1, #-16]
        STUR    X16, [X1, #-24]
        SUBI    X1, X1, #32
        CBNZ    X1,LOOP
```

Rearranging after unrolling, CPI=(14/14)=1

Full dataforwarding

How data hazards can be eliminated?

Summary

- **Detect and wait (stalling unnecessarily)**
- **Data forwarding**
- **Reordering (out of order)**
- **Renaming (to remove WAR and WAW hazard)**
- **Loop unrolling and reordering**

Control Hazards

- Branching instructions
  - pipeline to stall as it changes the execution sequence of instructions.
- Unconditional and conditional branches: In decode stage of branch the decoding unit comes to know that there is an alteration in the program control flow.
  - Incorrectly fetched instruction by the instruction fetch unit must be discarded.
  - New instruction specified by the conditional or unconditional branch must be fetched by the instruction fetch unit. This causes stalls (bubble) in the pipeline. Example: B, CBZ
- The time lost as a result of a branch instruction is often referred to as the branch penalty.

How to tackle control hazards

- **Conservative way**
  - Stall the pipeline immediately after we fetch a branch instruction, waiting until the pipeline determines the outcome of the branch and knows the address of next instruction to be fetched
  - 3 stalls without forwarding and 2 stalls with forwarding
- **Branch prediction**
  - Simple way is to predict always that the branches will be untaken
  - Only when branches are taken does the pipeline stall

Control Hazards (stall)

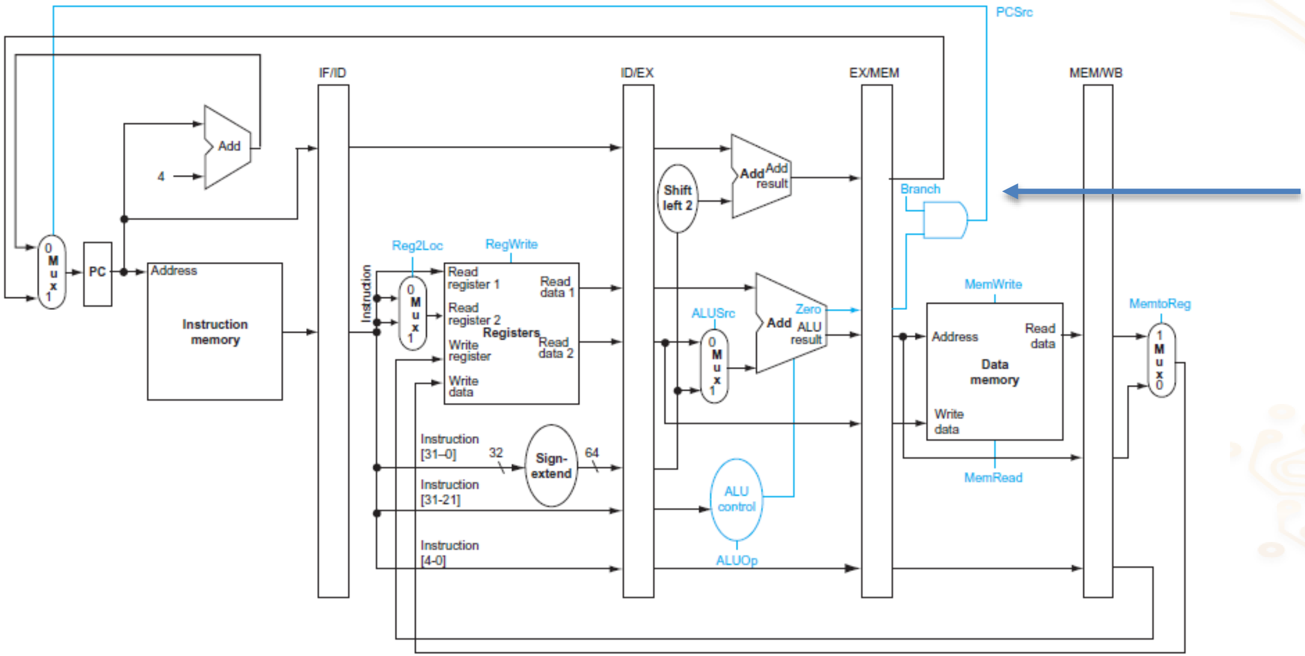
Example

CBZ X1, loop

……….

Loop: LDUR X3, [X0, #30]

Clock	1	2	3	4	5	6	7	8	9
I1	IF	ID	EX	M	WB				
I2		nop	nop	nop	nop	nop			
I3			nop	nop	nop	nop	nop		
I4				nop	nop	nop	nop	nop	
I5					IF	ID	EX	M	WB



CBZ updating PC in MEM stage
Branch penalty=
3 stalls

If data forwarding allowed:
Then CBZ can update PC in EXE stage.
Branch penalty=
2 stalls

Early Evaluation of PC for reducing branch stalls

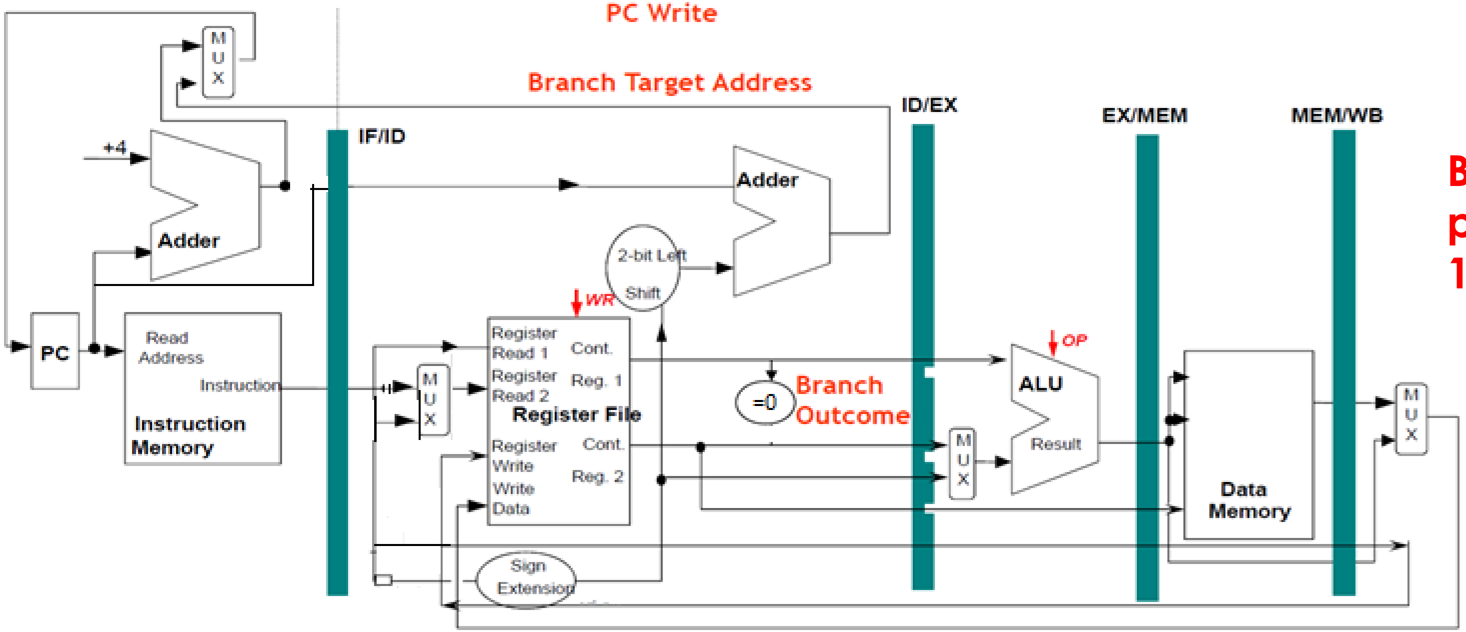
Example

CBZ X1, loop

……….

Loop: LDUR X3, [X0, #30]

Clock	1	2	3	4	5	6	7	8	9	10	11
I1	IF	ID	EX	M	WB						
I2		nop	nop	nop	nop	nop					
I5			IF	ID	EX	M	WB				



Branch penalty= 1 stalls

Need for branch prediction

Modified LEGv8 pipeline with branch address and outcome calculated in second stage. (needs only one stall)

- With the branch decision made during ID stage, there is a reduction of the cost associated with each branch **(branch penalty):**
- We need only **one-clock-cycle stall** after each branch
- Or a **flush** of only **one** instruction following the branch
- One-cycle-delay for every branch still yields a performance loss of 10% to 30% depending on the branch frequency:
- **Pipeline Stall Cycles per Instruction due to Branches = Branch frequency x Branch Penalty**

Branch Prediction

Branch prediction deals with branch penalty by continuing executing the instructions following the branch speculatively.

- If the branch does not occur then there was no pipeline stall.
- If, however, the branch does occur, the pipeline must be flushed of the speculative instructions.
- On average, this reduces pipeline stalls by 50%.

Static branch prediction techniques:- The actions for the branch are fixed for each branch during the entire execution. (when behaviour is highly predictable).

Dynamic branch prediction techniques:- The prediction decision may change depending on the execution history (when behaviour is not predictable).

Static Prediction (Part 1/2)

Static branch prediction is used in processors where the expectation is that the branch behaviour is highly predictable at compile time.

a) **Branch Always Not Taken – Predict-Not-Taken (Speculation)**
   - Execute successive instructions in sequence.
   - Flush the pipeline and read correct instructions if branch actually taken.

b) **Branch Always Taken – Predict-Taken**
   - The predicted-taken scheme makes sense for pipelines where the branch target address is known before the branch outcome.
   - But in LEGv8 pipeline we haven't calculated yet branch target address, still incurs branch penalty.

c) **Delayed branch**

Static Prediction (Part 2/2)

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

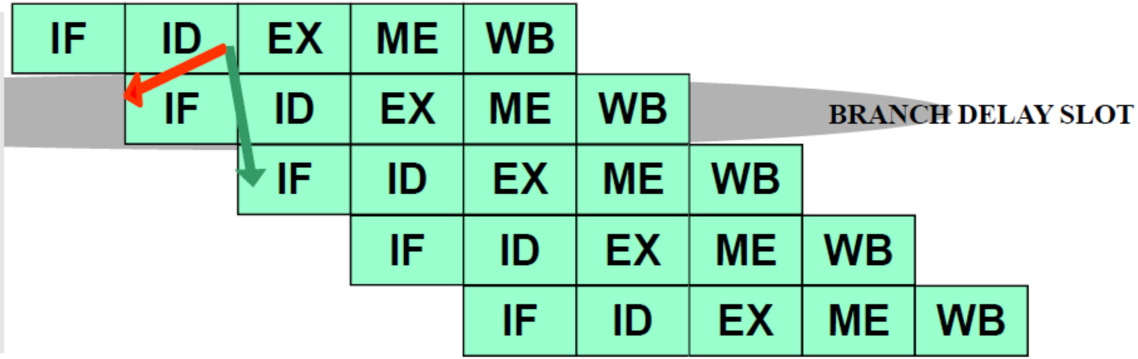
Branch not taken

Static Prediction – Delayed Branch (Part 1/3)

- The compiler statically schedules an independent instruction in the **branch delay slot**.
- The instruction in the branch delay slot is executed whether or not the branch is taken.
- If we assume a branch delay of one-cycle (as for modified LEGV8) ->we have only **one-delay slot**.
- The LEGv8 compiler always schedules a branch independent instruction after the branch.

Static Prediction – Delayed Branch (Part 2/3)

```
     CBZ    X1 L1
     ADD    X4,X5,X6
L1   LDUR   X3, [X0, #300]
     LDUR   X7, [X0, #400]
     LDUR   X8, [X0, #500]
```



IF ID EX ME WB

IF ID EX ME WB — BRANCH DELAY SLOT

IF ID EX ME WB

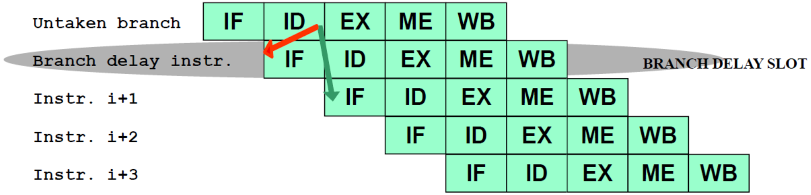
IF ID EX ME WB

IF ID EX ME WB

A previous **add** instruction with no effects on the branch is scheduled in the **Branch Delay Slot.**

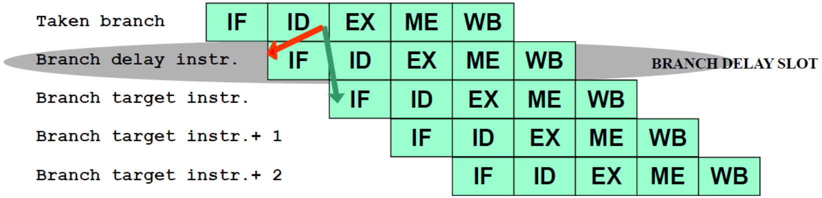
Static Prediction – Delayed Branch (Part 3/3)

- The behaviour of the delayed branch is the same whether or not the branch is taken.
- If the branch is **untaken** ➔ execution continues with the instruction after the branch.



Untaken branch	IF	ID	EX	ME	WB				
Branch delay instr.		IF	ID	EX	ME	WB			
Instr. i+1			IF	ID	EX	ME	WB		
Instr. i+2				IF	ID	EX	ME	WB	
Instr. i+3					IF	ID	EX	ME	WB

- If the branch is **taken** ➔ execution continues at the branch target.



Taken branch	IF	ID	EX	ME	WB				
Branch delay instr.		IF	ID	EX	ME	WB			
Branch target instr.			IF	ID	EX	ME	WB		
Branch target instr.+ 1				IF	ID	EX	ME	WB	
Branch target instr.+ 2					IF	ID	EX	ME	WB

Dynamic Prediction

Dynamic branch prediction usually relies on some measure of past behaviour to predict the future.

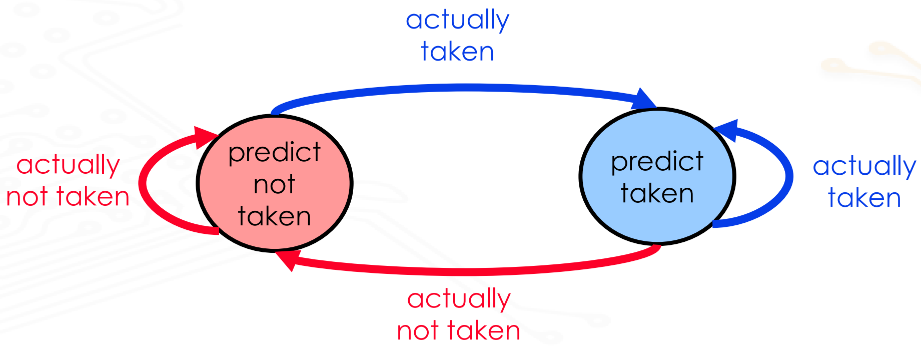
- When CPU sees a branch, it uses the hardware predictor to make a decision of which path to speculate on.

- Later on when the branch outcome is known, it updates the predictor.

Dynamic Prediction (One bit)

Scheme 1: Single T bit (Last time predictor)

- T is set to 1 when a branch is confirmed taken , 0 when not.



Example:

Assume single bit predictor for branch b1. Initial value of predictor to be 0. Then T_{b1} predicts that the branch is not taken.

branch b1 taken, T_{b1} =1 (predict next branch taken)
branch b1 not taken, T_{b1} =0 (predict next branch not taken)

Dynamic Prediction (One bit predictor)

- T is set to 1 when a branch is confirmed taken, 0 when not.
- Consider that you start from predict not taken
- TTTTTTTTTTNNNNNNNNNN → 90% accuracy
- Always mis-predicts the last iteration and the first iteration of a loop branch
- **TNTNTNTNTNTNTNTNTNTN** → 0% accuracy

Improving one bit predictor

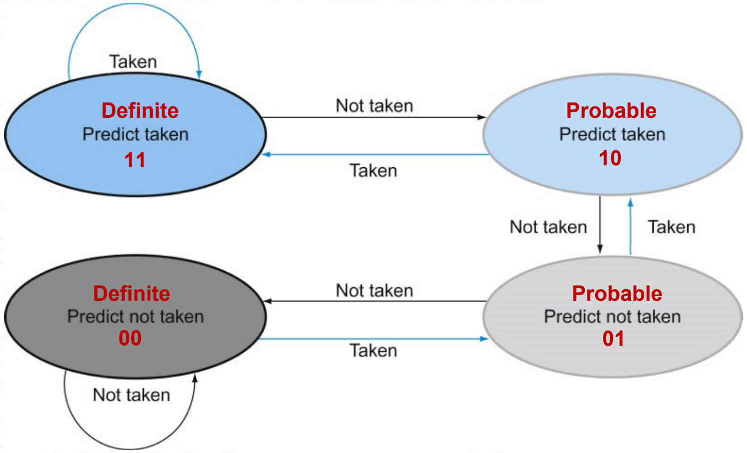
- Problem: A last-time predictor changes its prediction from Taken(T)→Not taken (N) or N→T too quickly
  - even though the branch may be mostly taken or mostly not taken.
- Solution Idea: Add hysteresis to the predictor so that prediction does not change on a single different outcome

  Use two bits to track the history of predictions for a branch instead of a single bit

  - Can have 2 states for T or N instead of 1 state for each.

Ref: Smith, "A Study of Branch Prediction Strategies," ISCA 1981.

Dynamic Prediction (Part 1/2)



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

Dynamic Prediction (Part 2/2)

- **Scheme 3: Bimodal prediction** uses a counter and the state of that counter determines the prediction.

- Generalized scheme of 3 bit predictor.

- The counter is incremented if branch is taken.

- The counter is decremented if branch not taken.

- Counters saturate (no wraparound). The speculation decision is based on the most significant bit: if MSB is 1, then counter is above half way.

000 → 001 → 010 → 011 → 100 → 101 → 110 → 111

Instruction-level parallelism (ILP)

- Execution of more than one instruction at the same time **(in parallel)** => **Instruction Level Parallelism (ILP)**

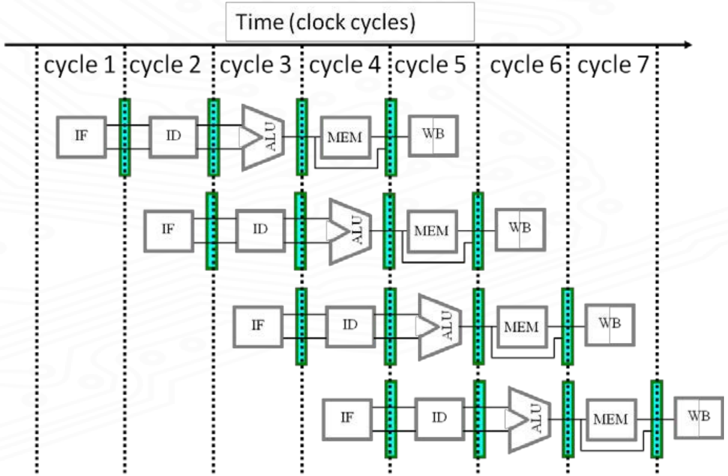
- **Multi-issue datapath** => Datapath that allow execution of two or more instructions in parallel

- ***N*-issue architecture** => Executes ‘***N***’ instructions at the same time

ILP and Multi-issue processors

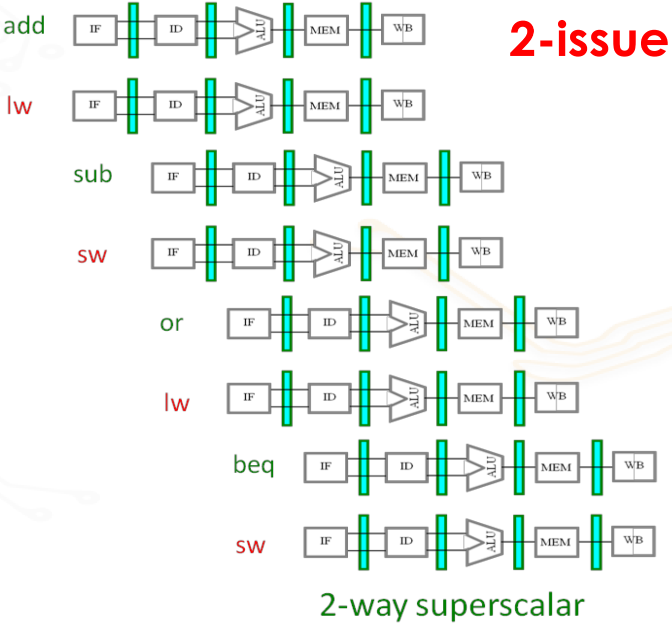
1 instruction issued in a clock cycle

=> single-issue processor



Pipelined processor

Scalar processor



2-issue

Multi-issue/super scalar processor =>2 or more instructions are issued in a clock cycle

Instruction-level parallelism (ILP) (Part 1/2)

Parallel execution of instructions requires **3 major tasks**:

1. **Which** instructions should be executed in parallel?
   Checking the data dependence between instructions to identify the instructions which can be **grouped** together for parallel execution.

2. **Where** to be executed?
   Assigning instructions to different functional units in the processor.

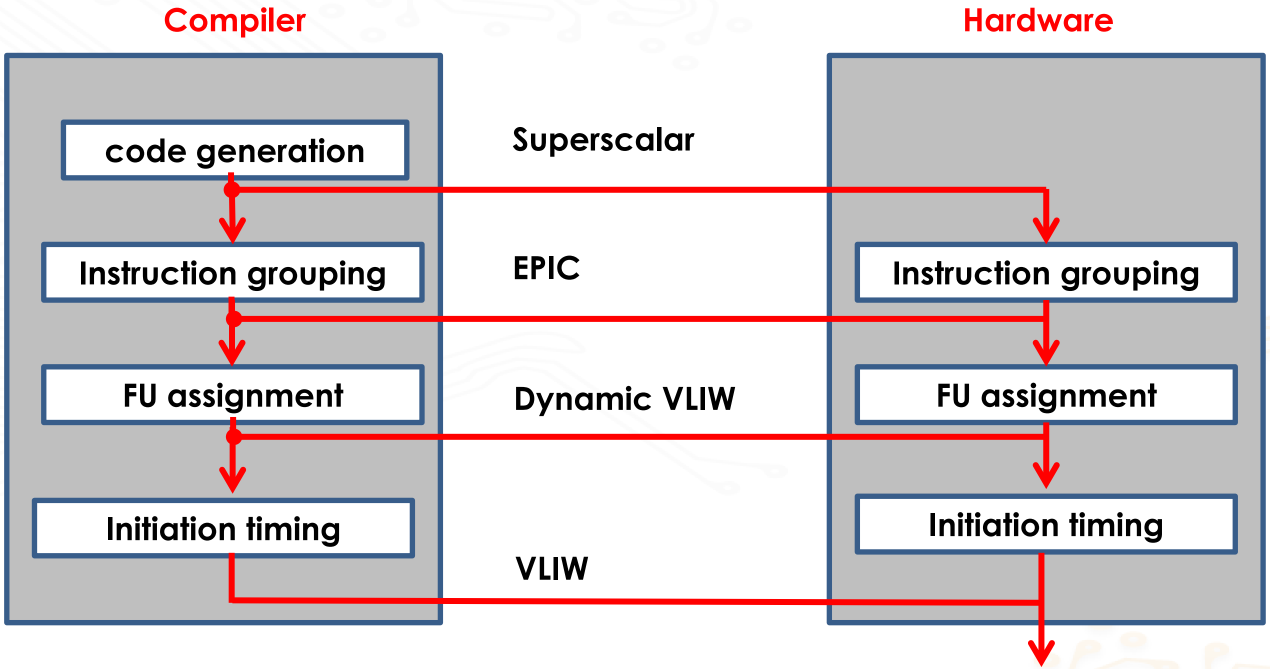
3. **When** to be executed?
   Determining when instruction execution is to be initiated.

Instruction-level parallelism (ILP) (Part 2/2)

Two approaches to achieve ILP

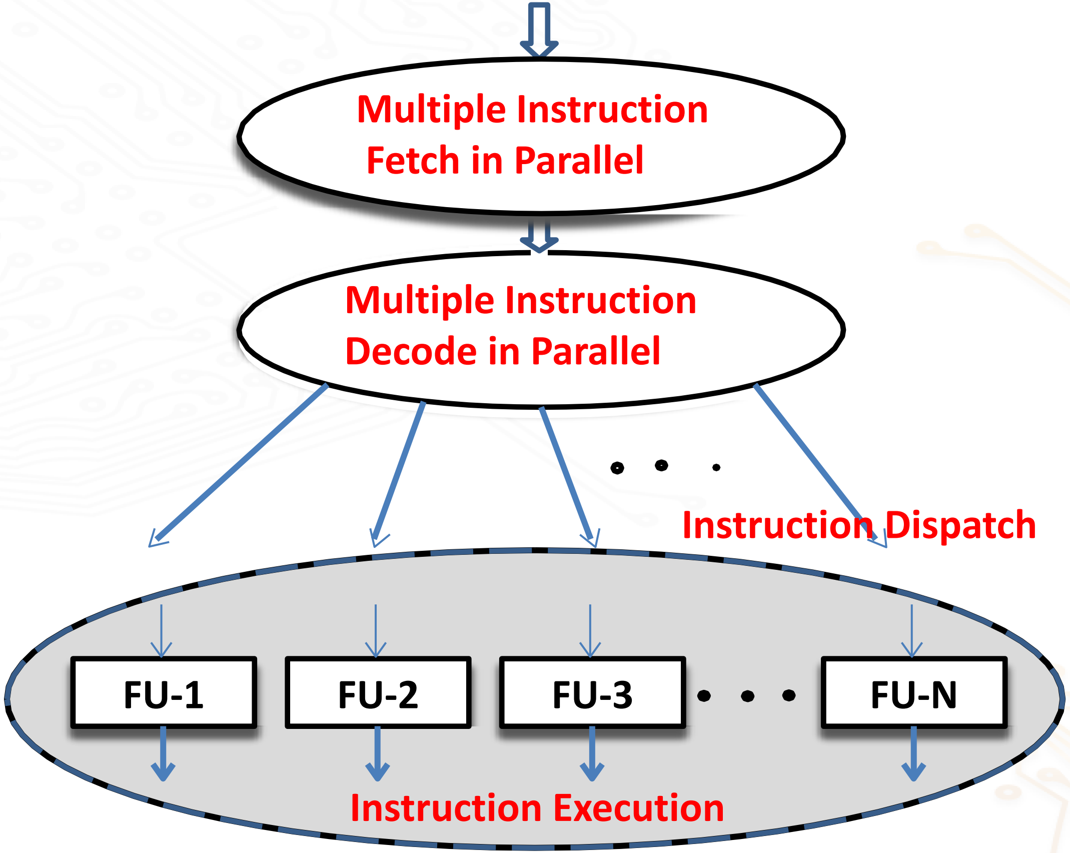
- Hardware Approach: **Superscalar** Processor
  - P5 Pentium, the first superscalar X86 processor
  - Most general purpose CPU's developed since 1998 are superscalar
- Software (or compiler-based) Approach: **Very Long Instruction Word(VLIW)** processor
  - VLIW CPU's contain multiple RISC like Functional Units(FUs). Typically have 4 to 8 FUs.

Hardware and Software Approaches



FU: Functional unit
EPIC: Explicitly parallel instruction computing

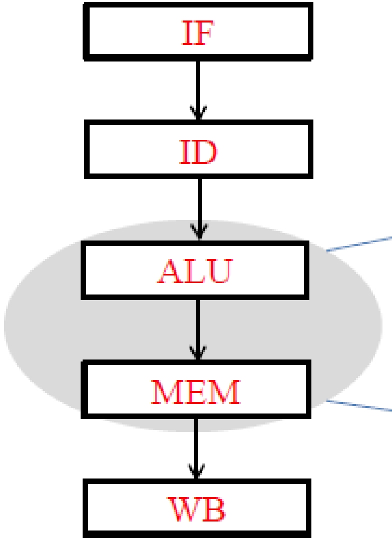
Concept of Superscalar Processing


Multiple Instruction Fetch in Parallel
Multiple Instruction Decode in Parallel
Instruction Dispatch
FU-1
FU-2
FU-3
FU-N
Instruction Execution

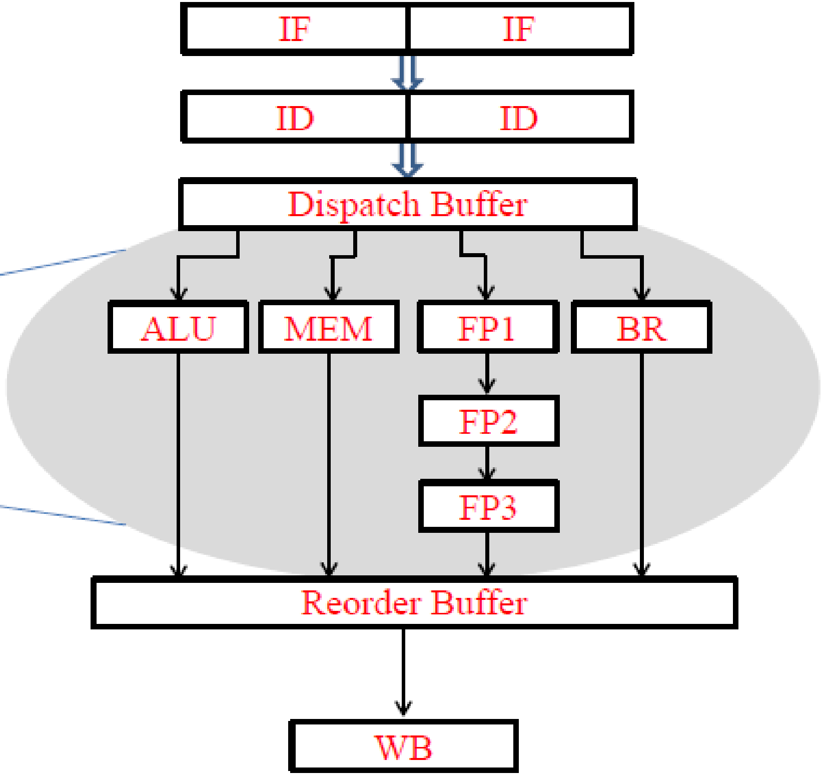
Requirement of superscalar processing

- Multiple functional units (FU) in a CPU are used to execute more than one instructions concurrently in a clock cycle.
- More than 1 independent instructions need to be available to be executed. (Makes use of ILP)
- Use specialized hardware to issue multiple independent instructions that can be executed simultaneously.

Scalar vs Superscalar



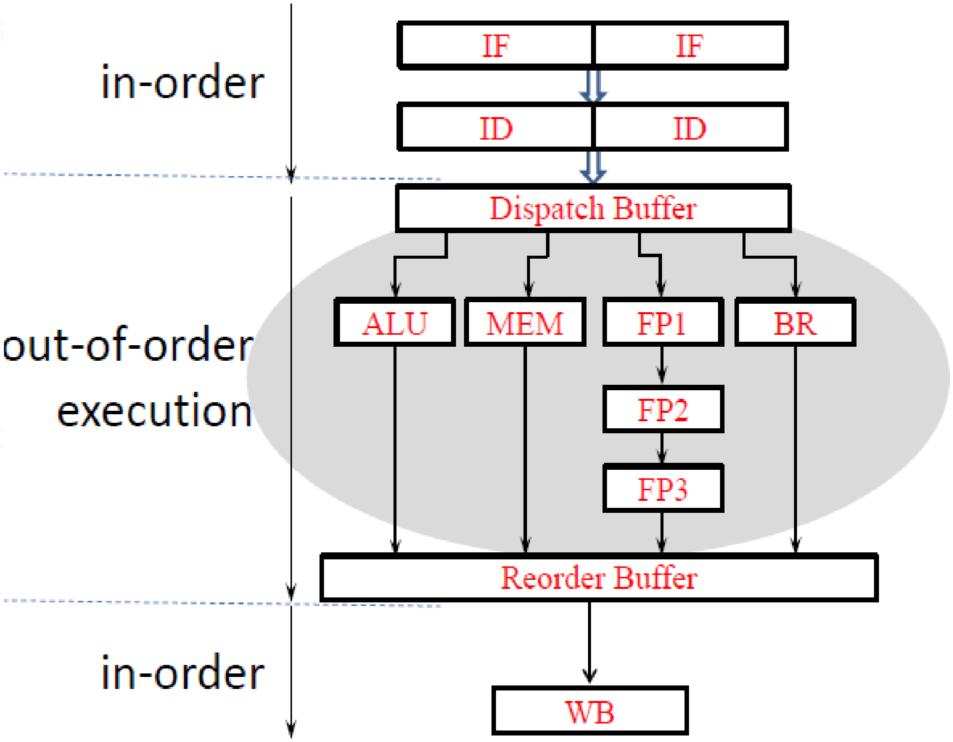
scalar pipeline



superscalar pipeline

Out-of-order execution in SS pipeline

- Instructions are fetched in compiler-generated order **(in-order).**
- Instruction completion may be also **in order**.
- Instructions are executed in some other order: **independent instructions behind a stalled instruction can be executed prior to the stalled instruction.**
- Dynamically scheduled: the order of execution of instructions is changed.



Methods to extract more parallelism

1. **Instruction reordering and out-of-order execution,**
2. **Speculative execution with dynamic scheduling,**
3. **Loop unrolling**

- Instruction Reordering: Change the order of execution of instruction if it does not violate the data dependence.
- Speculative Execution: To execute a instruction without exactly knowing if that need to be executed: ahead of branch outcome.
- Dynamic Scheduling: Execute instructions as soon as dependencies are satisfied and functional units are available.

Loop unrolling for ILP

- loop unrolling leads to multiple replications of the loop body
  - Multiple independent instructions that can be executed in parallel

- Example

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

Original loop

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];
}
```

Unrolled by a factor 2

How can loop unrolling help us to produce multiple independent instructions?

Example: Scheduling for superscalar (Part 1/2)

```
Loop:   LDUR X0, [X1, #0]        # load an array element
        ADDI X0, X0, #25         # add that with 25
        STUR  X0, [X1, #0]       # store back the result
        ADDI X1, X1, #8          # find address of the next element in dmem
        SUBI   X2, X2, #1        # find the remaining no. of iterations
        CNBZ X2,  Loop           # continue iterations if x2 != 0
```

Example: Scheduling for superscalar (Part 2/2)

```
Loop:   LDUR X0, [X1, #0]   (2 stalls)
        ADDI X0, X0, #25    (2 stalls)
        STUR  X0, [X1, #0]
        ADDI X1, X1, #8
        SUBI   X2, X2, #1   (2 stalls)
        CBNZ X2,  Loop      (1 stall)
```

Control hazard is removed in decode stage (1 stall cycle)

CPI = (6+7)/6=2.17 without reordering in a **scalar processor**

2-way super-scalar after instruction reordering CPI = 7/6 (No data-forwarding is done)

	Way-1	Way-2	Cycle
Loop	**SUBI X2, X2, #1**	**LDUR X0, [X1, #0]**	**1**
	nop	**nop**	**2**
	nop	**nop**	**3**
	ADDI X0, X0, #25	**nop**	**4**
	ADDI X1, X1, #8	**nop**	**5**
	CBNZ X2, Loop	**nop**	**6**
	nop	**STUR X0, [X1, #0]**	**7**

Loop unrolling and instruction reordering (Part 1/3)

```
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
```

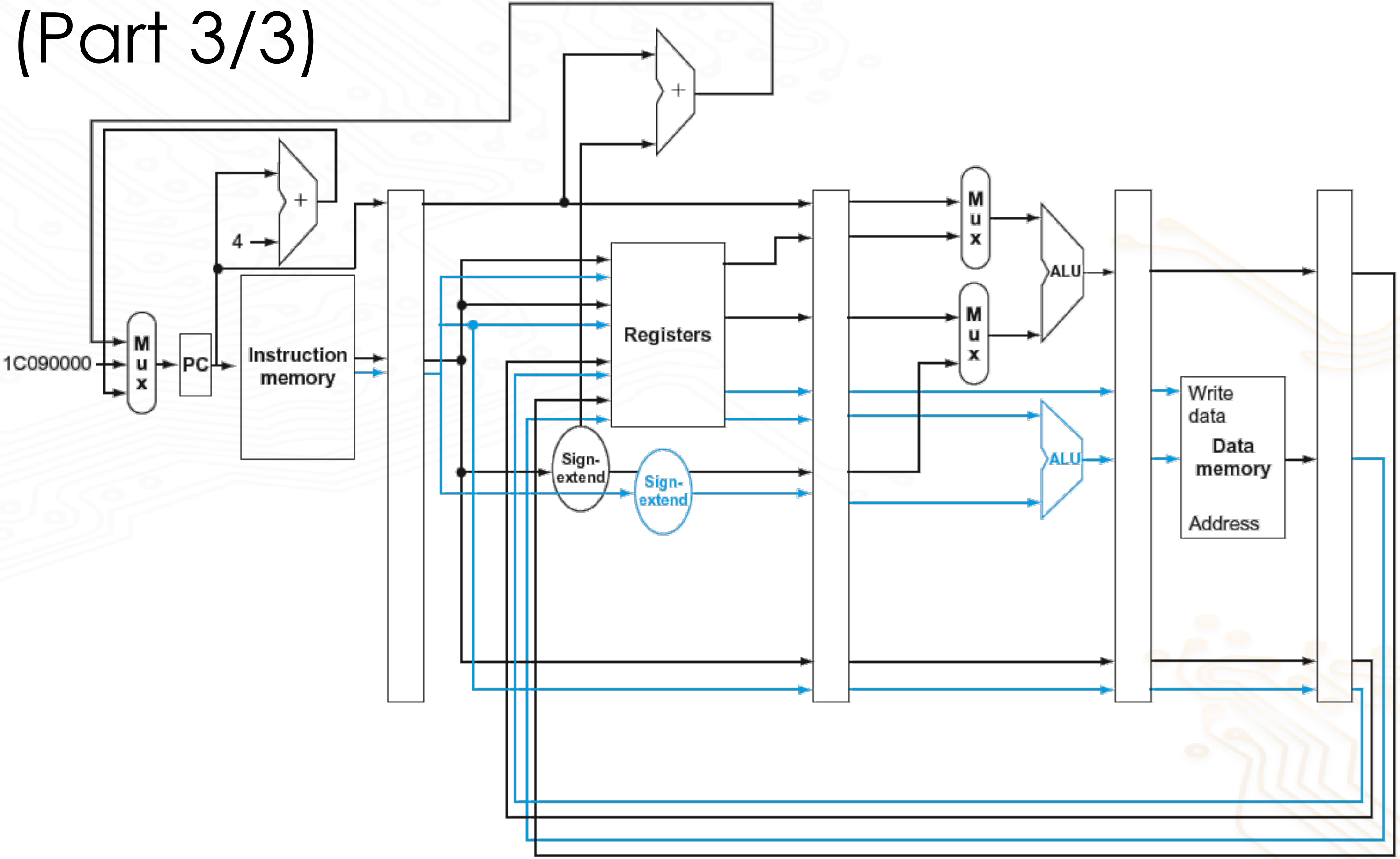
after loop unrolling by 4 and instruction reordering

Loop unrolling and instruction reordering (Part 2/3)

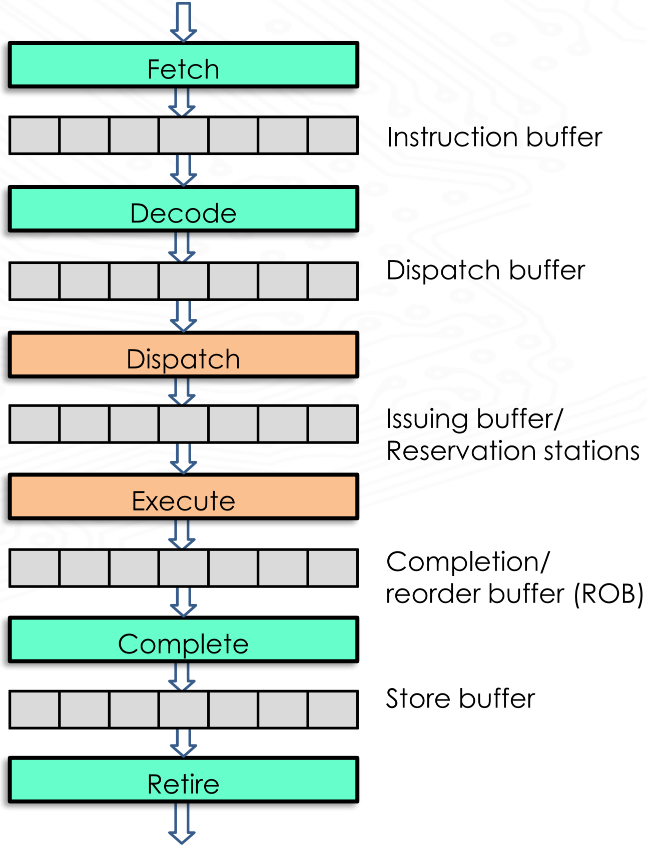
	Way-1	Way-2	Cycle
Loop	SUBI X2, X2, #4	LDUR X0, [X1, #0]	1
	nop	LDUR X3, [X1, #8]	2
	nop	LDUR X4, [X1, #16]	3
	ADDI X0, X0, #25	LDUR X5, [X1, #24]	4
	ADDI X3, X3, #25	nop	5
	ADDI X4, X4, #25	nop	6
	ADDI X5, X5, #25	STUR X0, [X1, #0]	7
	ADDI X1,X1, #32	STUR X3, [X1, #8]	8
	CBNZ X2, Loop	STUR X4, [X1,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

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

Loop unrolling and instruction reordering (Part 3/3)



Different stages in superscalar pipeline



Fetch multiple instructions in every clock cycle and store in instruction buffer(IB)

Decode the instructions available in the instruction buffer.

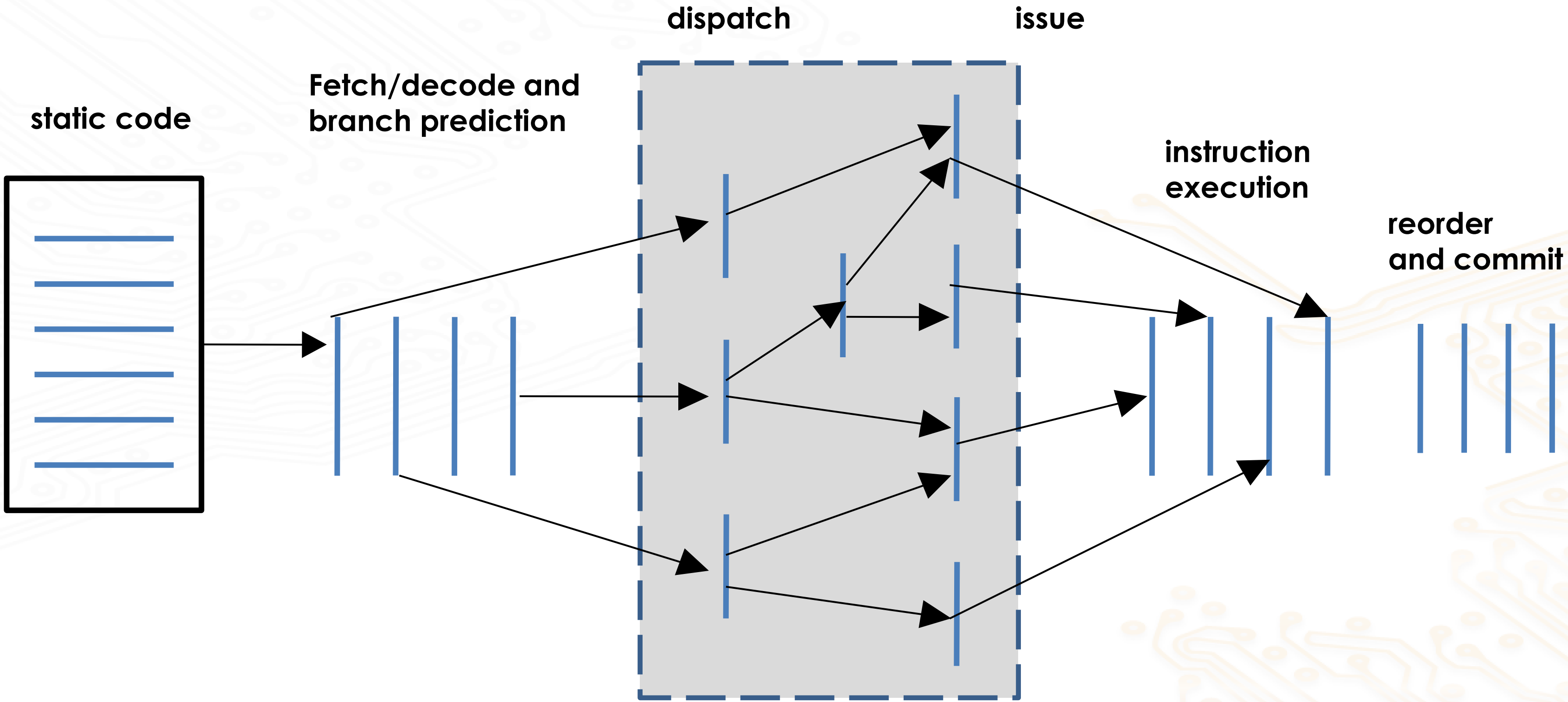
Instructions are issued into functional units as and when their operands and respective functional units are available.

Instructions are marked ‘finished’ when executed, and enter the reorder buffer (in out-of-order)

Instructions exit the ROB (in order). Results are written in the processor registers in order.

performs memory update if any (usually D-cache) .

Superscalar execution



Superscalar processor solves the data hazard, control hazard, and memory-latency problems

VLIW processing strategy

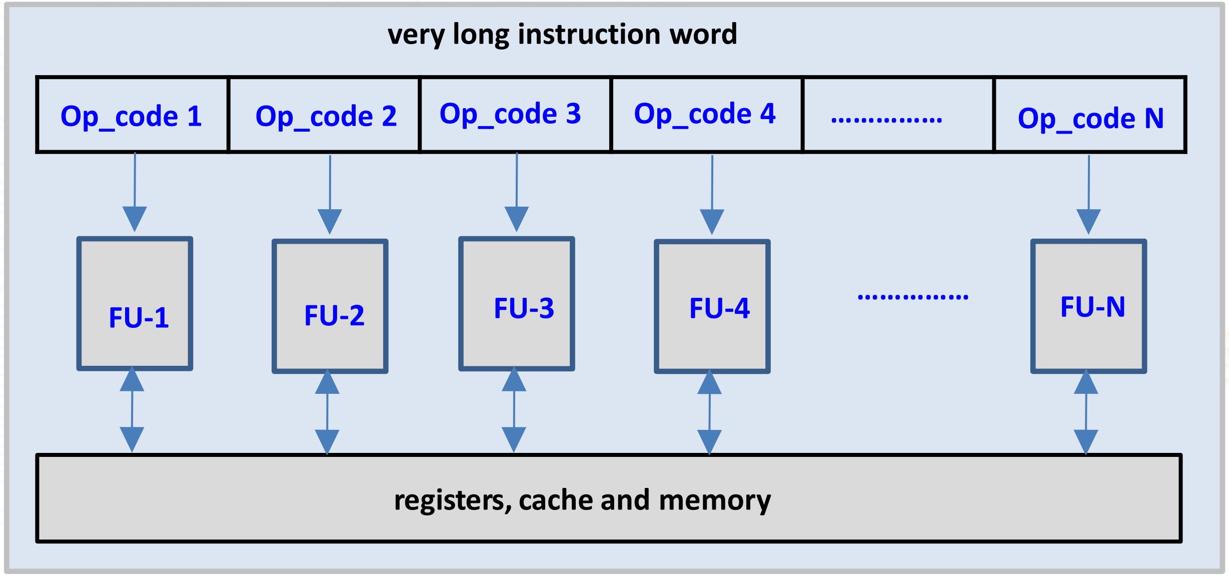
- Packs multiple independent operations into one instruction.

- VLIW processor needs a compiler to break the program instructions down into basic operations that can be performed by the processor in parallel.

- These operations are put into a very long instruction word which the processor (consisting of multiple functional units (FU)) gets executed in appropriate FUs.

- There must be enough parallelism in the program code to fill the available slots, to reduce the CPI significantly.

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.

Advantage and disadvantage of VLIW (Part 1/2)

- Compiler prepares fixed packets of multiple operations: gives the complete plan of execution.
  - Dependencies are determined by compiler and that is used to schedule according to the latencies of corresponding functional units.
  - Functional units are assigned by compiler, they correspond to the position within the instruction packet or instruction slot.
  - Compiler produces fully-scheduled, hazard-free code => hardware need not check dependencies for scheduling.

Advantage and disadvantage of VLIW (Part 2/2)

- Compatibility across implementations is a major problem.
  - VLIW code cannot run properly with different number of functional units or functional units with different latencies.
  - Unscheduled events (e.g., cache miss) can stall the entire processor.
- Code density is an important concern.
  - Slot utilization varies: nops increase with branching (control).
  - Reduce nops by compression ("flexible/variable-length VLIW").

Conclusions

- Basic of multi-issue processing.
- Limitation o f scalar pipeline and Motivation for superscalar processors.
- Concept of superscalar processing.
- Basics of superscalar pipeline.
- Instruction execution in a superscalar processor.
- Basics of VLIW processors.