UNIVERSITÀ DEGLI STUDI DI TRIESTE

Dipartimento di Ingegneria e Architettura

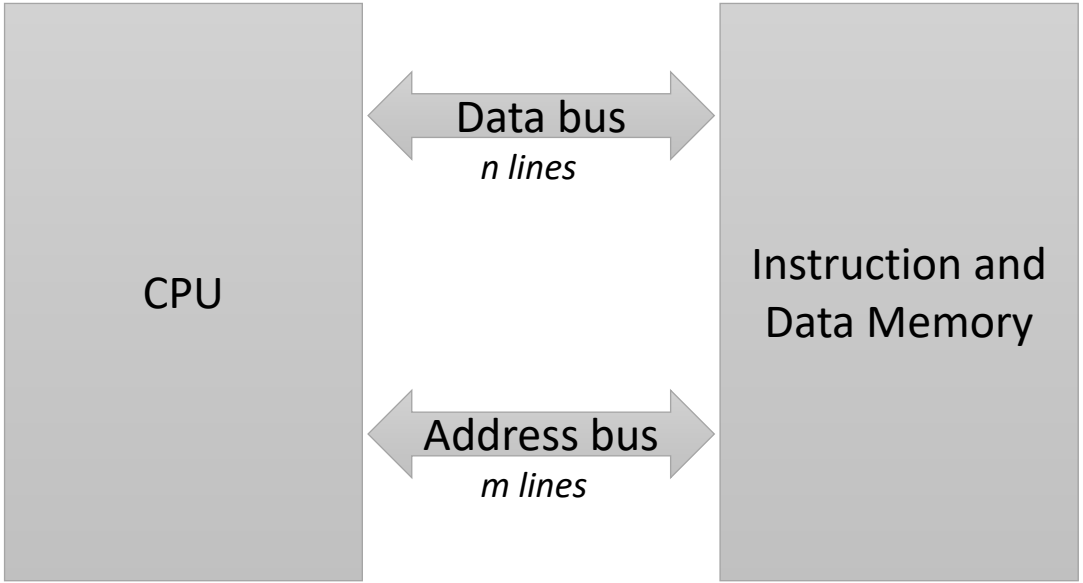
The processor

A. Carini – Digital System Architectures

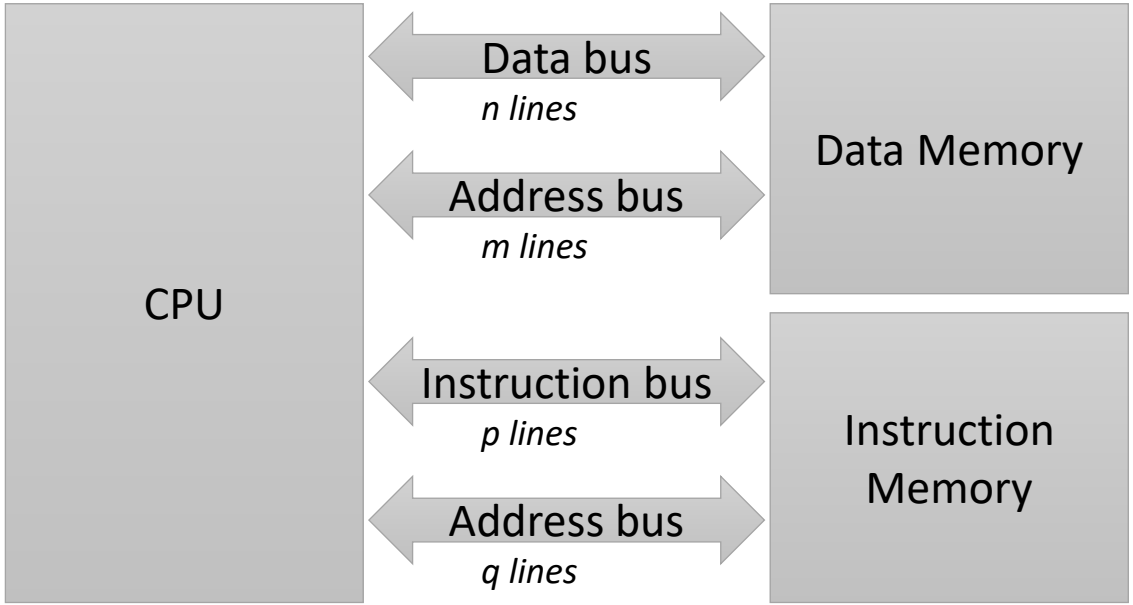
Introduction

- We have seen that the performance of a computer is determined by three key factors:
  - instruction count,
  - clock cycle time, and
  - clock cycles per instruction (CPI).
- The compiler and the instruction set architecture determine the instruction count required for a given program.
- The implementation of the processor determines both the clock cycle time and the number of clock cycles per instruction.
- We can have different organizations of the processor
  - Harward or Von Neumann
- And different implementation strategies
  - Single cycle; Multi cycle; Pipelined
- We will examine different LEGv8 implementations
  - A simplified version (Single cycle)
    - A multi cycle version, but for a different processor
  - A more realistic pipelined version

Von Neumann organization



Harward organization



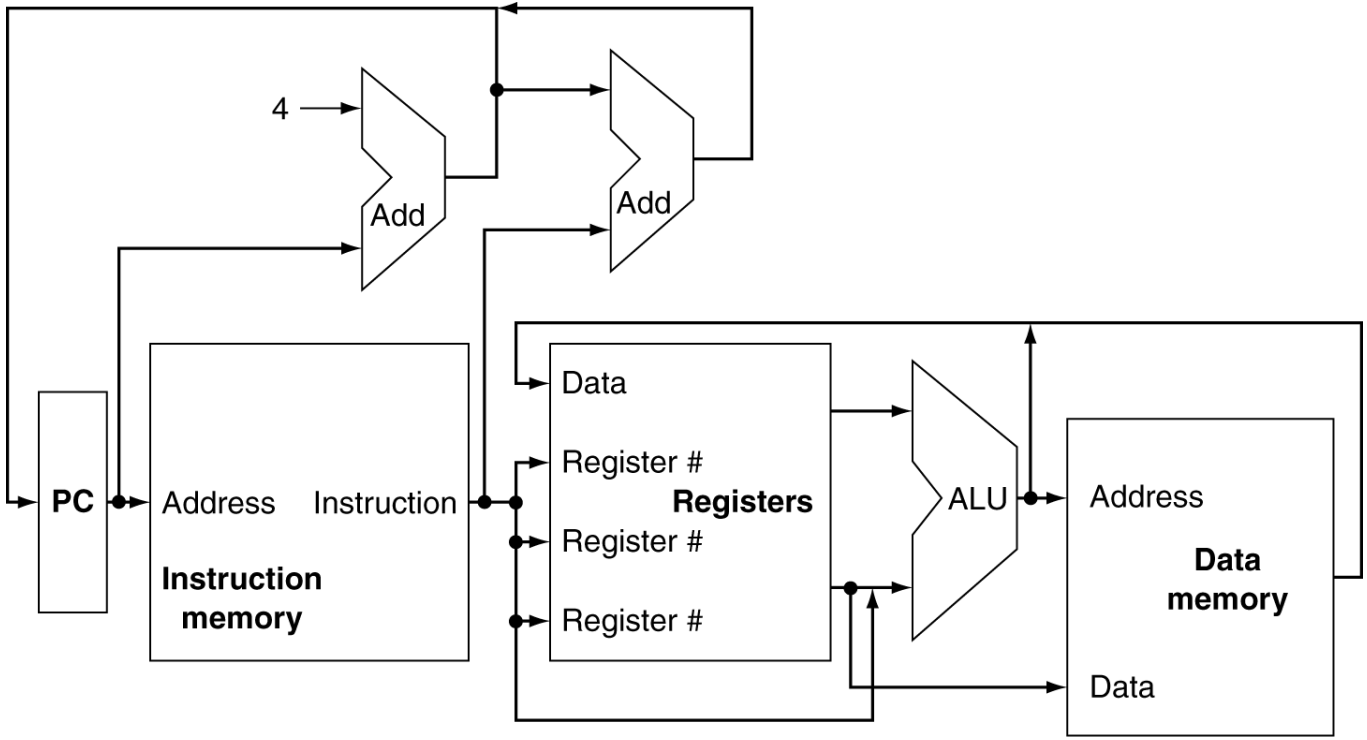
A Basic LEGv8 Implementation

- We will be examining an implementation that includes a subset of the core LEGv8 instruction set:
  - The memory-reference instructions *load register unscaled* (**LDUR**) and *store register unscaled* (**STUR**)
  - The arithmetic-logical instructions **ADD**, **SUB**, **AND**, and **ORR**
  - The instructions *compare and branch on zero* (**CBZ**) and *branch* (**B**)
- It illustrates the key principles used in creating a datapath and designing the control.
- We will have the opportunity to see
  - how the instruction set architecture determines many aspects of the implementation, and
  - how the choice of various implementation strategies affects the clock rate and CPI.

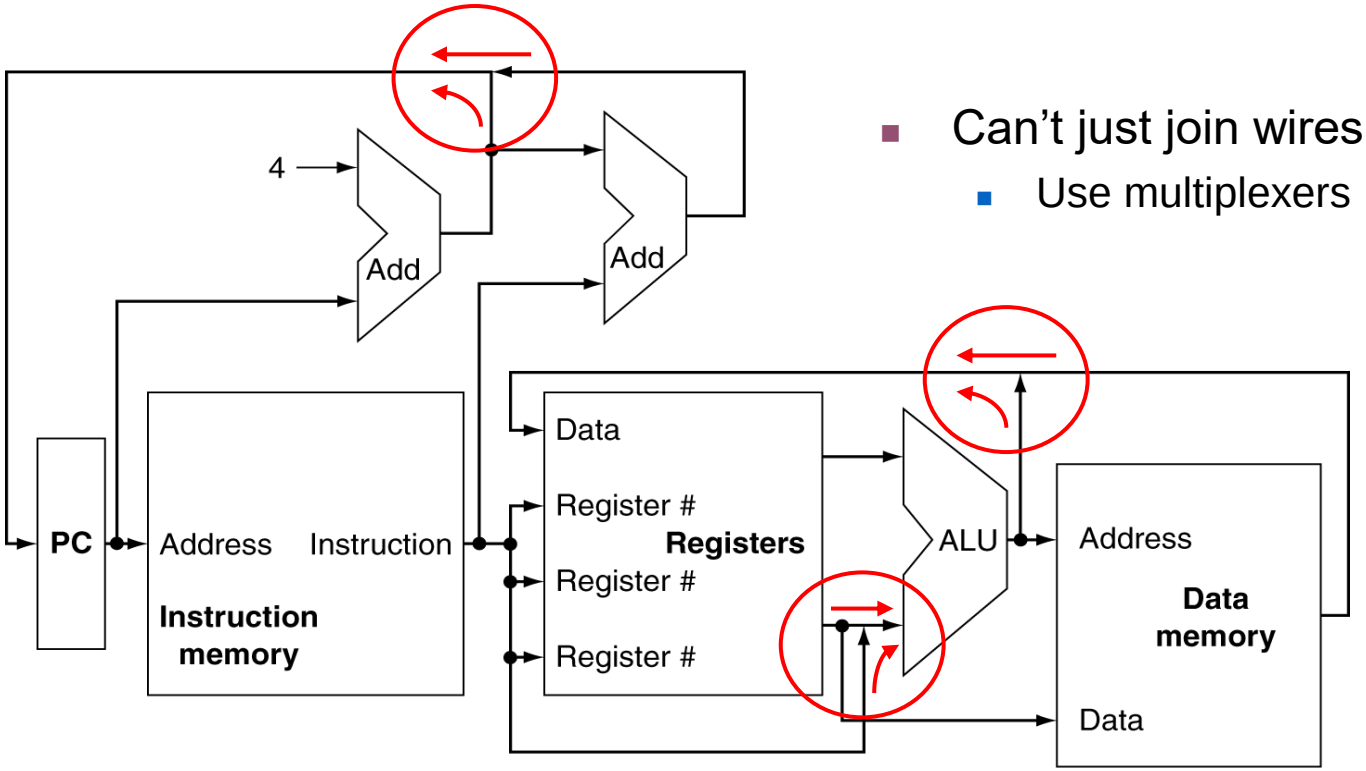
Instruction Execution

- Much of what needs to be done to implement these instructions is the same, independent of the exact class of instruction.
- For every instruction, the first two steps are identical:
  1. Send the *program counter* (PC) to the memory that contains the code and fetch the instruction from that memory.
  2. Read one or two registers, using fields of the instruction to select the registers to read.
- After these, the actions required to complete the instruction depend on the instruction class.
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Comparison with zero in branch
  - Access data memory for load/store
  - PC ← target address or PC + 4

CPU Overview

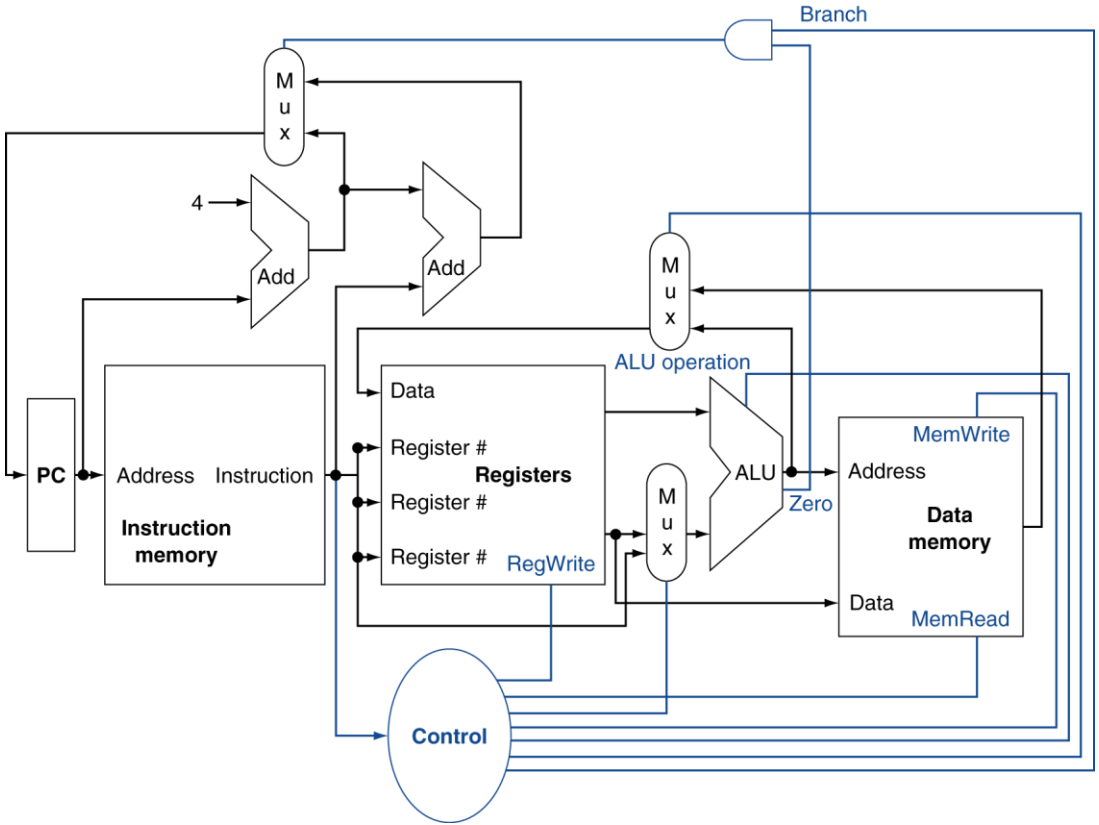


Multiplexer



- Can't just join wires together
  - Use multiplexers

Control

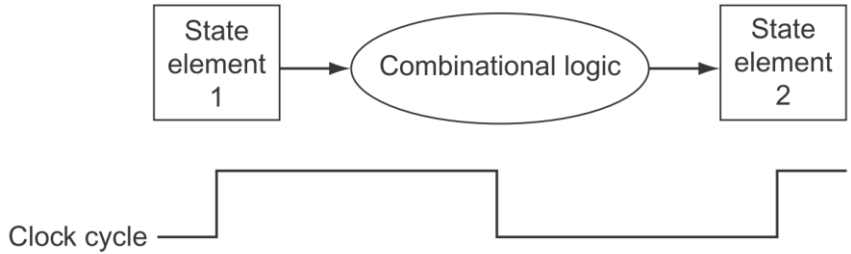

Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

Logic Design Conventions

- The datapath elements in the LEGv8 implementation consist of two different types of logic elements:
  - Combinational elements
    - Operate on data
    - Output is a function of input
  - State (sequential) elements
    - Store information
    - E.g, registers and memories
    - We call these elements **state elements** because, if we pulled the power plug on the computer, we could restart it accurately by loading the state elements with the values they contained before we pulled the plug.
    - A state element has at least two inputs and one output:
      - Data input, clock.
      - The output is the value that was written in an earlier clock cycle.
  - The clock is used to determine when the state element should be written.

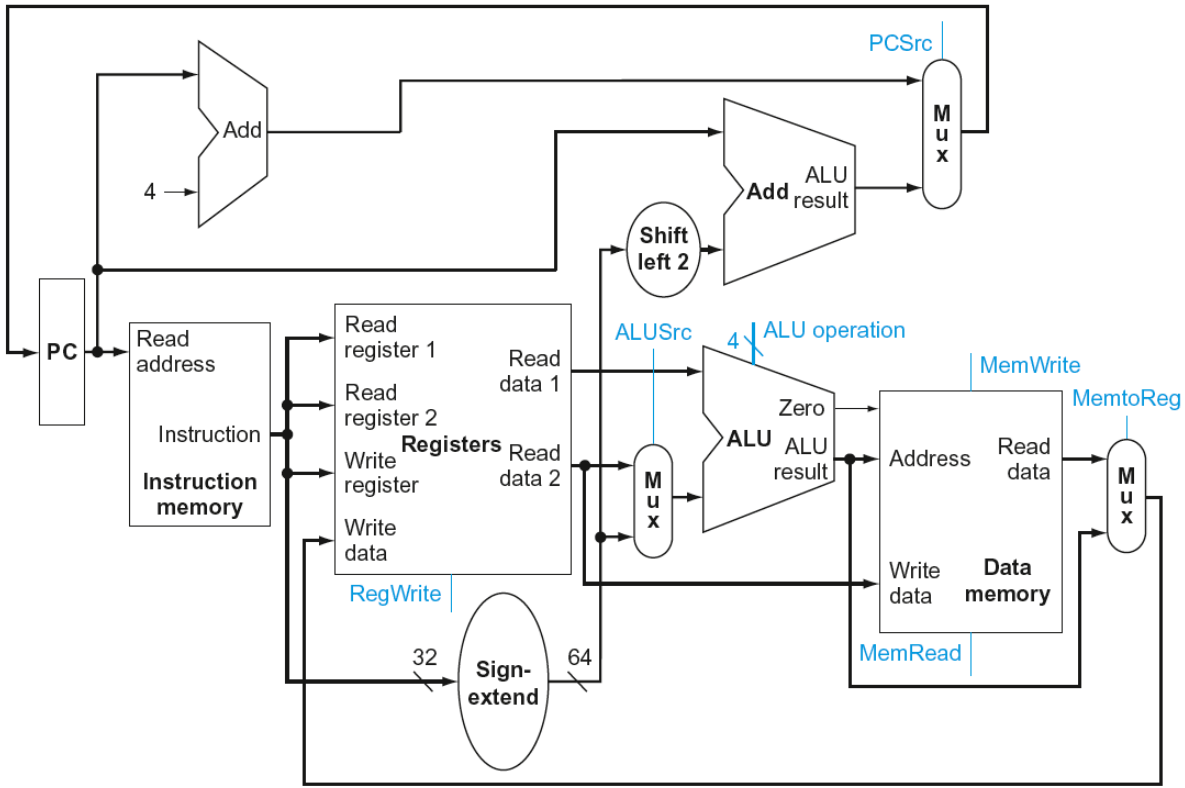
Clocking Methodology

- A **clocking methodology** defines when signals can be read and when they can be written.
- We will assume an **edge-triggered** clocking methodology.
  - Any values stored in a sequential logic element are updated only on a clock edge.

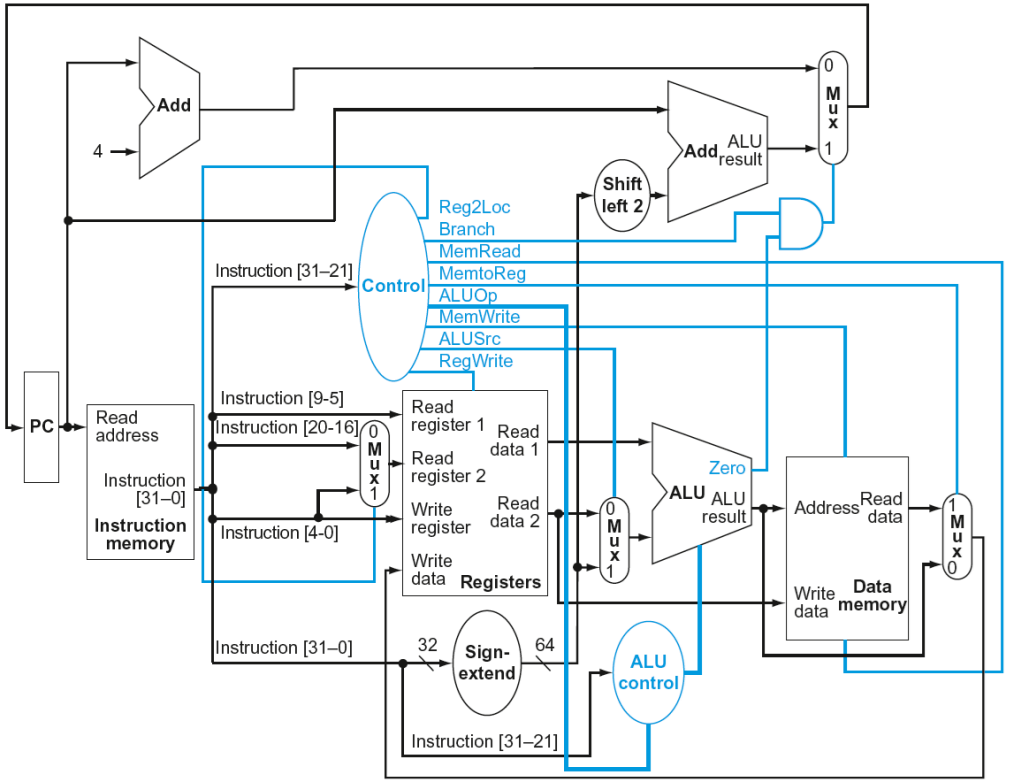


- All signals must propagate from state element 1, through the combinational logic, and to state element 2 in the time of one clock cycle.
- The time necessary for the signals to reach state element 2 defines the length of the clock cycle.
- If a state element is not updated on every clock, then an explicit write **control signal** is required.
  - The state element is changed only when the control signal is **asserted** and a clock edge occurs.

The Simple Datapath



The Simple Datapath with Control



ALU Control

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	pass input b
1100	NOR

- ALU used for
  - Load/Store: F = add
  - Branch: F = compare
  - R-type: F depends on opcode
- Assume 2-bit ALUOp derived from opcode:
  - 00 - add 00 for loads and stores,
  - 01 - pass input b for CBZ, or
  - 10 - determined by opcode field for R-type

Instruction	ALUOp	Instruction operation	Opcode field	Desired ALU action	ALU control input
LDUR	00	load register	XXXXXXXXXXX	add	0010
STUR	00	store register	XXXXXXXXXXX	add	0010
CBZ	01	compare and branch on zero	XXXXXXXXXXX	pass input b	0111
R-type	10	ADD	10001011000	add	0010
R-type	10	SUB	11001011000	subtract	0110
R-type	10	AND	10001010000	AND	0000
R-type	10	ORR	10101010000	OR	0001

Instruction format

- We will see the implementation of three instructions:

```
ADD X1, X2, X3
LDUR X1, [X2, offset]
CBZ X1, offset
```

Field	opcode	Rm	shamt	Rn	Rd
Bit positions	31:21	20:16	15:10	9:5	4:0

a. R-type instruction

Field	1986 or 1984	address	0	Rn	Rt
Bit positions	31:21	20:12	11:10	9:5	4:0

b. Load or store instruction

Field	180	address	Rt
Bit positions	31:24	23:5	4:0

c. Conditional branch instruction

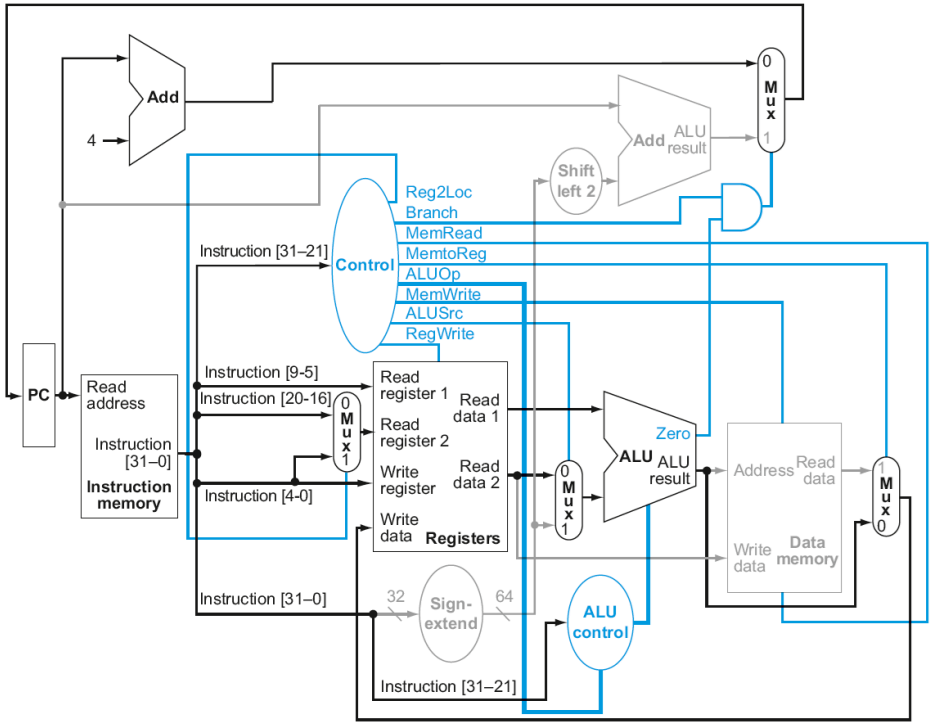
Instruction format

- There are several major observations about this instruction format that we will rely on:
  - The opcode field is between 6 and 11 bits wide and found in bits 31:26 to 31:21.
  - The first register operand is always in bit positions 9:5 (Rn) for both R-type instructions and for the base register for load and store instructions.
  - The other register operand is in one of two places. It is in bit positions 20:16 (Rm) for R-type instructions and it is in bit positions 4:0 (Rt) for the register to be written by a load. That is also the field that specifies the register to be tested for zero for compare and branch on zero.
  - Another operand can also be a 19-bit offset for compare and branch on zero or a 9-bit offset for load and store.
  - The destination register for R-type instructions (Rd) and for loads (Rt) is in bit positions 4:0.

ADD X1, X2, X3

- Although everything occurs in one clock cycle, we can think of four steps to execute the instruction;
- these steps are ordered by the flow of information:
    1. The instruction is fetched, and the PC is incremented.
    2. Two registers, X2 and X3, are read from the register file; also, the main control unit computes the setting of the control lines during this step.
    3. The ALU operates on the data read from the register file, using portions of the opcode to generate the ALU function.
    4. The result from the ALU is written into the destination register (X1) in the register file.

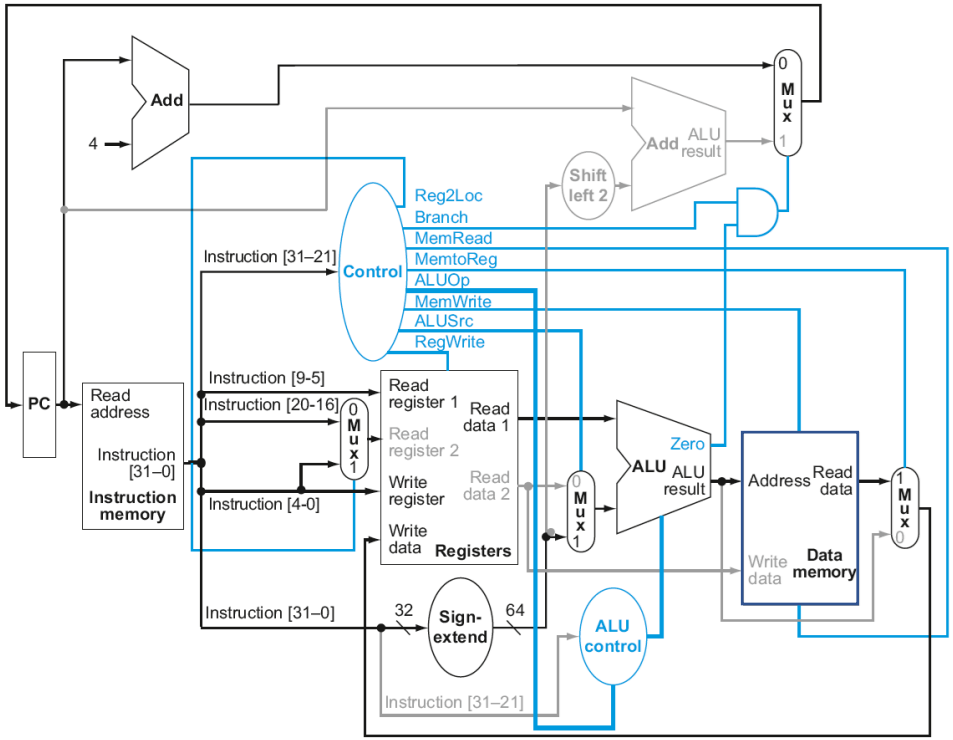
ADD X1, X2, X3


Add
4
0
M
u
x
1
Add
ALU result
Shift left 2
Reg2Loc
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–21]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [9-5]
Instruction [20-16]
Instruction [4-0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Zero
ALU
ALU result
Address
Read data
Write data
Data memory
Instruction [31–0]
32
Sign-extend
64
ALU control
Instruction [31–21]

LDUR X1, [X2,offset]

- We can think of a load instruction as operating in five steps :
  1. An instruction is fetched from the instruction memory, and the PC is incremented.
  2. A register (X2) value is read from the register file.
  3. The ALU computes the sum of the value read from the register file and the sign-extended 9 bits of the instruction (offset).
  4. The sum from the ALU is used as the address for the data memory.
  5. The data from the memory unit is written into the register file (X1).

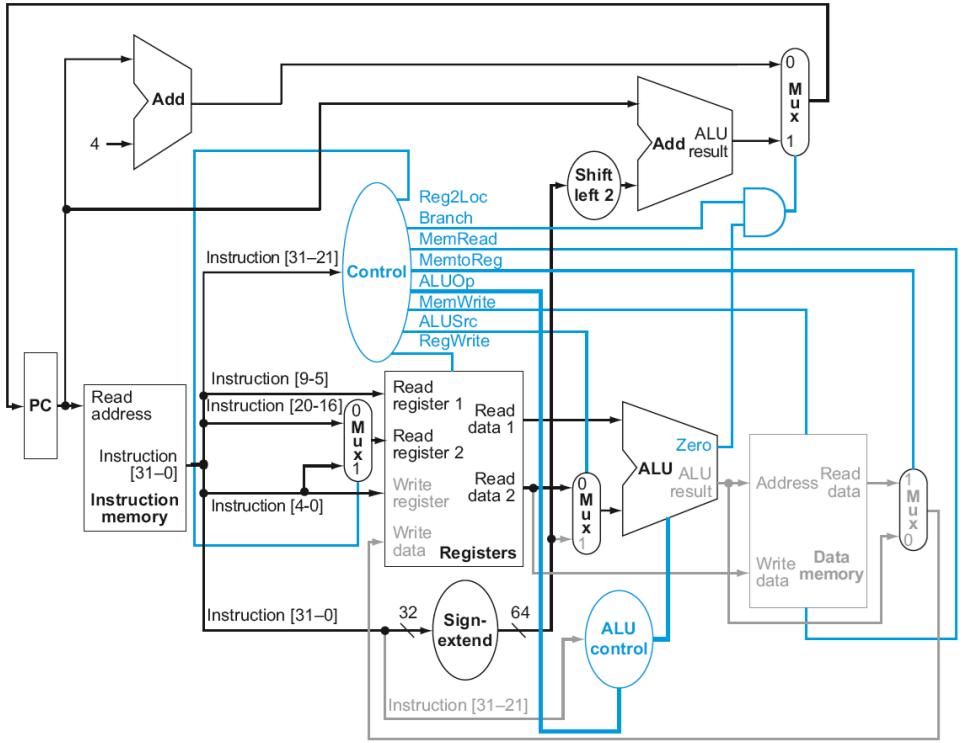
LDUR X1, [X2,offset]



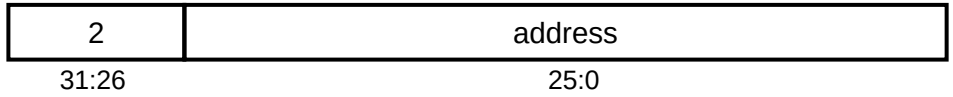
CBZ X1, offset

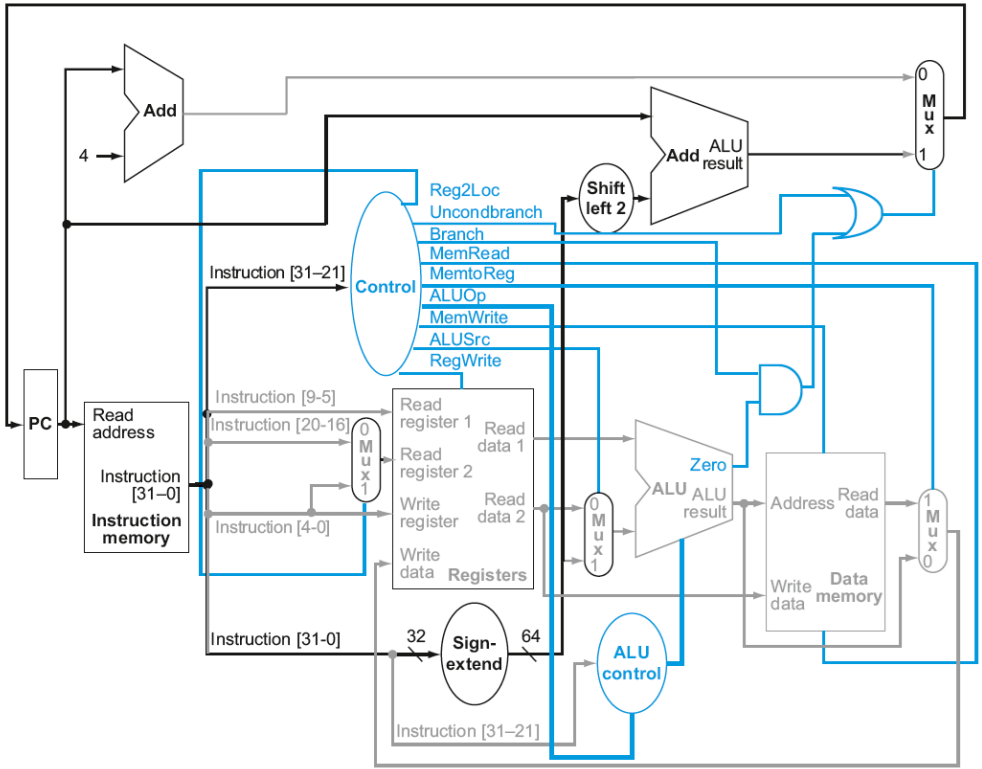
- It operates much like an R-format instruction, but the ALU output is used to determine whether the PC is written with PC + 4 or the branch target address.
- We can think of four steps in the execution:
    1. An instruction is fetched from the instruction memory, and the PC is incremented.
    2. The register X1 is read from the register file using bits 4:0 of the instruction (Rt).
    3. The ALU passes the data value read from the register file. The value of PC is added to the sign-extended, 19 bits of the instruction (offset) are shifted left by two; the result is the branch target address.
    4. The Zero status information from the ALU is used to decide which adder result to store in the PC.

CBZ X1, offset



Unconditional Branch





Why a Single-Cycle Implementation is not Used Today

- Although the single-cycle design will work correctly, it is too inefficient to be used in modern designs.
- Notice that the clock cycle must have the same length for every instruction in this single-cycle design.
- The longest possible path in the processor determines the clock cycle.
  - This path is most likely a load instruction, which uses five functional units in series:
  - the instruction memory, the register file, the ALU, the data memory, and the register file.
- Although the CPI is 1, the overall performance of a single-cycle implementation is likely to be poor, since the clock cycle is too long.

- Historically, early computers with very simple instruction sets did use this implementation technique.
- However, if we tried to implement the floating-point unit or an instruction set with more complex instructions, this single-cycle design wouldn't work well at all.

Multi-cycle Control strategies

R1 ← R1 + M[num]

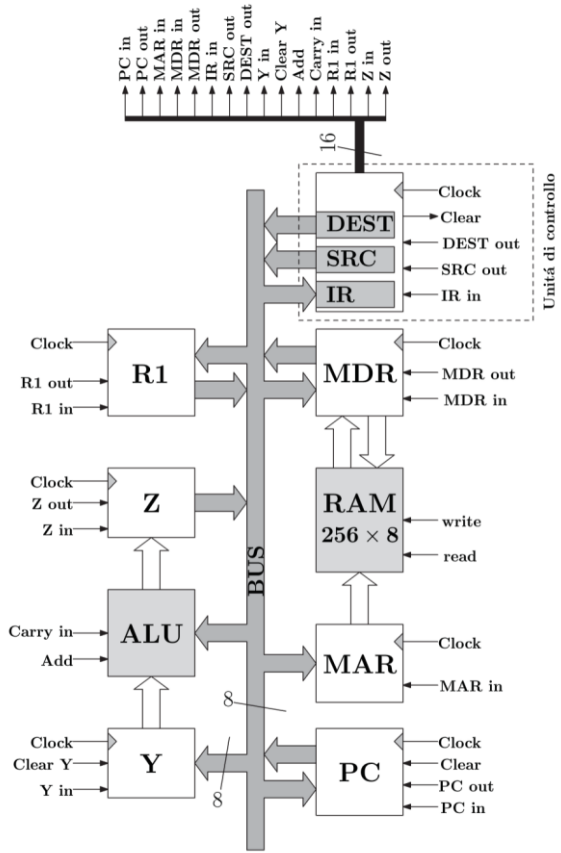


Figura 5.13: Un ipotetico processore a 8 bit.

Multi-cycle Control strategies

- Let us consider the hypothetical processor of the previous slide.
- The control unit decode and execute the instructions, and update the program counter (PC), fetching the next instruction.
- The control unit is here composed of three register:
  - IR has the OpCode of the instruction
  - SRC can contain a parameter included in the IW or a memory address, often expressed in relative terms as a PC increment.
  - DEST can contain an address or the pointer to a register where to write the result.
- From the analysis of the OpCode, the control unit must provide the sequence of control signals necessary for
  1. Fetch and update of the PC
  2. Execution of the current instruction.

Multi-cycle Control strategies

- Historically, two possible approaches have been followed for the control unit:
  - *Microprogrammed* approach
  - *Cabled* approach
  - Microcontrollers with old architectures are microprogrammed. More recent ones, especially RISC ones, are cabled.
- Two possible clocking strategies:
  - *Multi-cycle control*
    - Fetch, decode, execute performed with multiple clock periods.
  - *Single-cycle control*
    - Fetch, decode, execute performed in a single period.
- Single-cycle control strategy is employed only in cabled controls.
- Multi-cycle control strategy is used both in all microprogrammed controls and in some cabled controls.

Microprogrammed control

- Is implemented with a control unit that replicates the structure of a simple CPU with a memory, a PC, an ALU, called **microcode engine**.
- Each macro-instruction corresponds to a microcode, composed by some words. Microcodes are stored in a ROM memory.
- Two possibilities:
  - *Horizontal microprogramming*: the control unit execute the code strictly sequentially, starting from the address pointed by the OpCode.
  - *Vertical microprogramming*: jumps are possible and allow to repeat microcode segments, i.e., the introduction of micro-subroutines.
- The *microcode* is composed of words, whose bits directly assert/negate specific control signals.
- The microcode *wordlenght* depends on the number of control signals.

Microprogrammed control

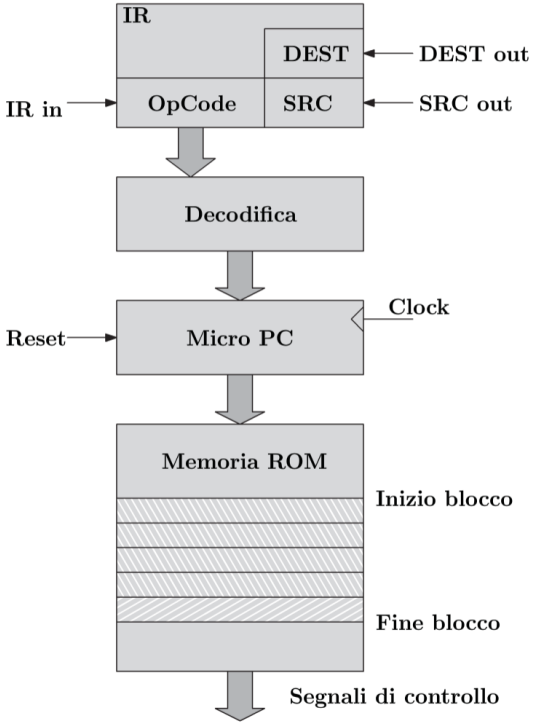


Figura 5.14: Schema a blocchi semplificato di una unità di controllo microprogrammata.

Cabled control

- The microcode engine is replaced by a *combinatorial logic circuit* that generates directly the control signals form the OpCode of the current instruction, managing also temporizations.
- Includes a *secondary clock generator*, whose purpose is the time distribution of control signal activations.
- Let us assume that a single instruction is executed in 7 periods.
- The secondary clock generator is a clock divider by 7 that generates 1 pulse every 7 clock periods, and feeds a shift register of 6 FlipFlops.
- The decoder activates one output line for each OpCode.
- The combinatorial network, composed by AND and OR gates, feeds the control lines on the basis of the OpCode and of secondary clock state (from 1 to 7).
- The solution provides *very fast response*, with *little silicon area occupation,* but *lack flexibility*, and could require *nop* cycles to manage shorter instructions.

Cabled control

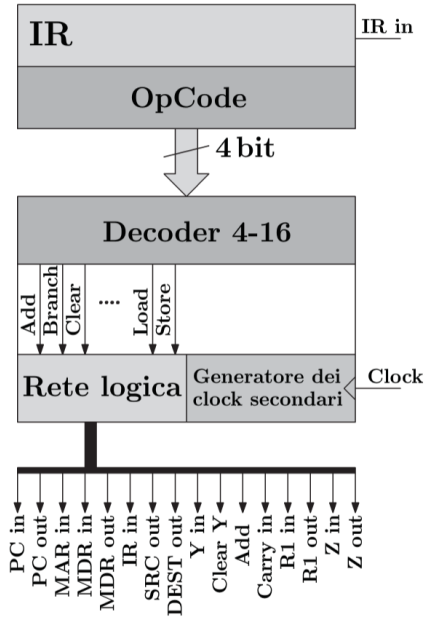


Figura 5.15: Unità di controllo cablata: è visibile un decodificatore delle istruzioni (a 4 bit), un generatore dei clock secondari e una rete logica combinatoria che asserisce i segnali di controllo del processore, elaborando sia i clock secondari che le linee del bus gestito dal decodificatore.

Cabled control

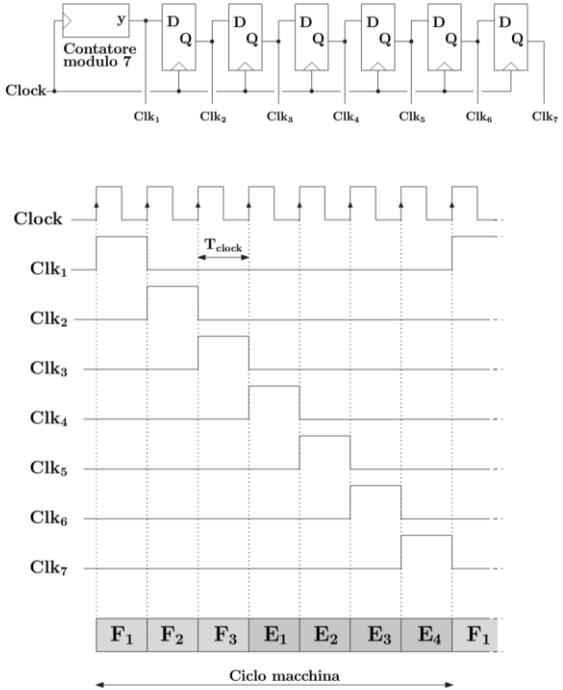


Figura 5.16: Circuito per la generazione dei segnali di clock secondari richiesti da una unità di controllo cablata.

Example

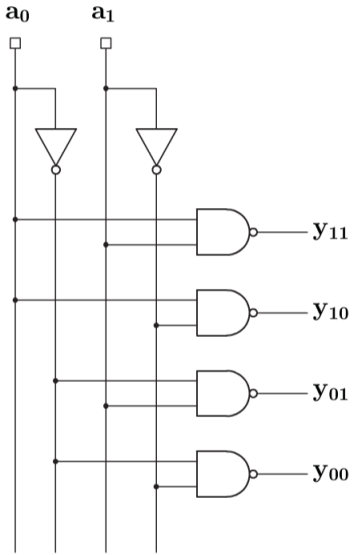


Figura 5.17: Schema di principio di un decodificatore binario, o demultiplexer, a 2 bit. Ciascuna delle 4 combinazioni di ingresso attiva, portandola, in questo caso, a livello logico *basso*, una e una sola delle 4 uscite.

Cabled control

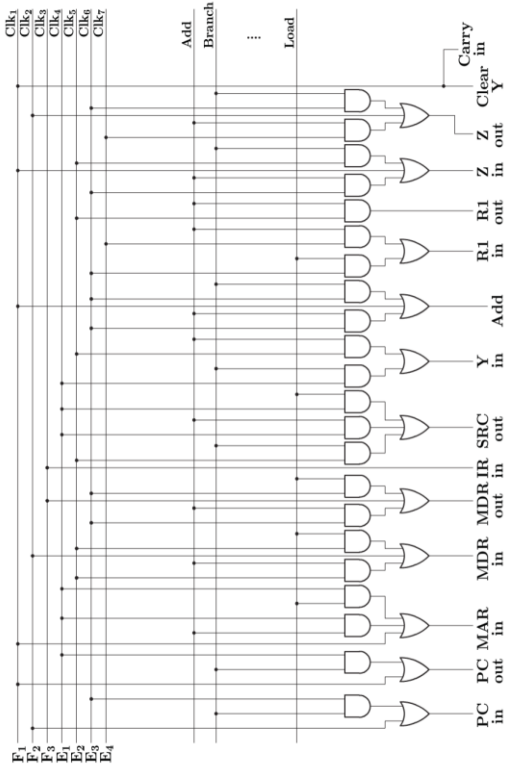


Figura 5.18: Una parte della rete combinatoria richiesta dal controllo del processore di Fig. 5.13.

Multi-cycle vs Single-cycle Organization

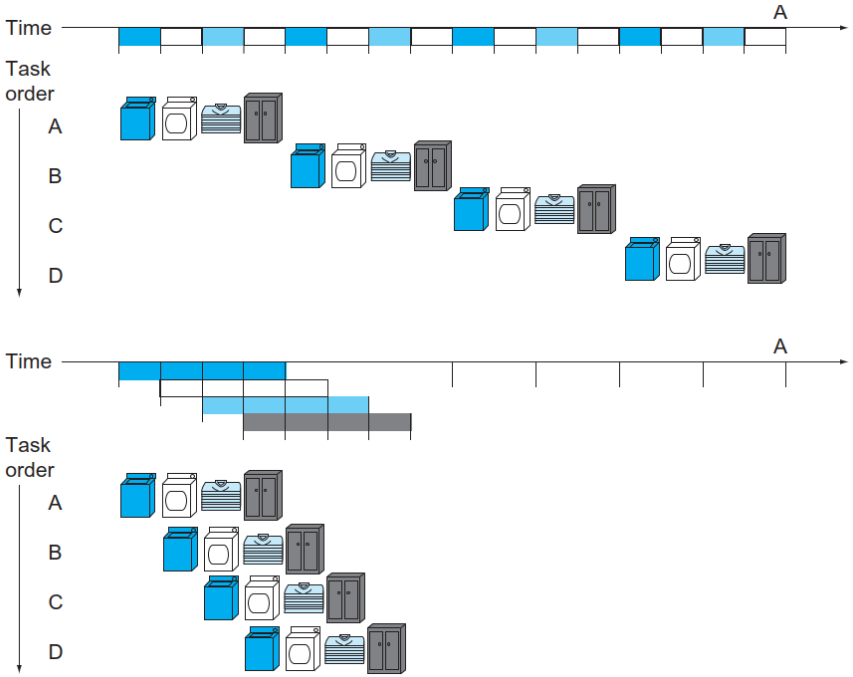
- Only the availability of multiple resources allow a single cycle temporization.
- It requires at least that the fetch phase is performed simultaneously to decode and execute.
- It imposes the following system requirements:
  - Separate data memory and instruction memory;
  - Separate ALU for PC increment;
  - Flexible PC increment for managing jumps without main ALU intervention.
- Unless the instruction set is very simple, the single cycle organization is often *inefficient*. It is the most onerous instruction that determine the clock period.
- On the contrary, in multi-cycle organization, it is the slowest functional unit (ALU or memory) that determines the minimum period.
- It is possible to combine advantages of both, using a *pipeline* organization.

An Overview of Pipelining

- **Pipelining** is an implementation technique in which multiple instructions are overlapped in execution.
  - Today, pipelining is nearly universal.

Pipelining analogy

- Pipelined laundry overlapping execution
  - Parallelism improves performance



- Four loads:
  - Speedup
    = 16/7 = 2.3
- Non-stop:
  - Speedup
    = 4n/1n + 3 ≈ 4
    = number of stages
- The pipelining paradox is that the time for processing a single laundry load is not shorter for pipelining.
- But more loads are process per hour.
- Pipelining improves *throughput* of our laundry system.

Single-Cycle versus Pipelined Performance

- LEGv8 instructions classically take five steps:

1. Fetch instruction from memory.
2. Read registers and decode the instruction.
3. Execute the operation or calculate an address.
4. Access an operand in data memory (if necessary).
5. Write the result into a register (if necessary).

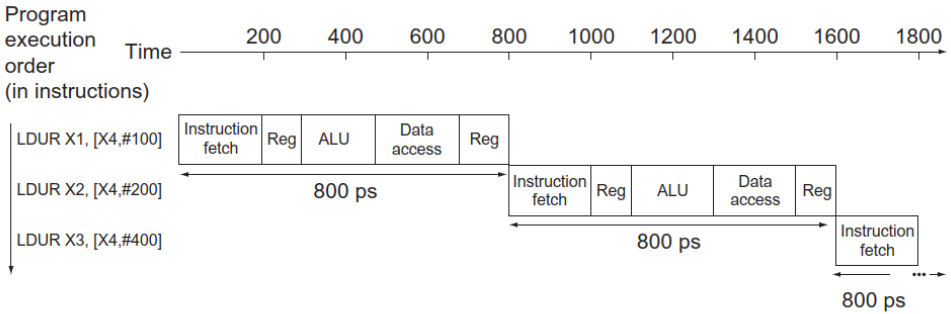
- Hence, the LEGv8 pipeline we consider has five stages.

Single-Cycle versus Pipelined Performance

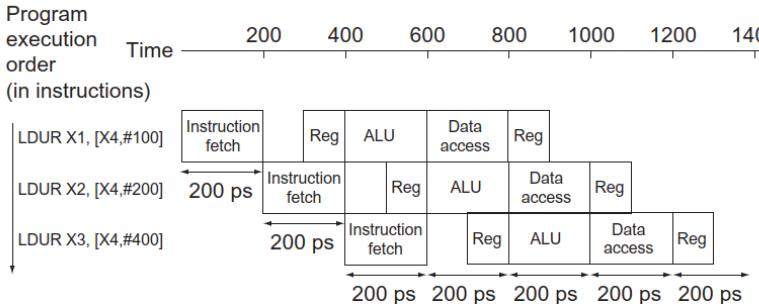
- We limit our attention to seven instructions: load register (**LDUR**), store register (**STUR**), add (**ADD**), subtract (**SUB**), AND (**AND**), OR (**ORR**), and compare and branch on zero (**CBZ**).
- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load register (LDUR)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store register (STUR)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (ADD, SUB, AND, ORR)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (CBZ)	200 ps	100 ps	200 ps			500 ps

Single-Cycle versus Pipelined Performance



Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)

Pipeline Speedup

- If the stages are perfectly balanced, then

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

- Under ideal conditions and with a large number of instructions, the speed-up from pipelining is approximately equal to the number of pipeline stages.
- If the stages are not balanced, the speedup is less.
  - Moreover, we will see pipelining involves some overhead.

- In reality, in our case the total execution time for the three instructions is 1400 ps versus 2400 ps.
- But if we add 1,000,000 instructions:

$$\frac{800,002,400 \text{ ps}}{200,001,400 \text{ ps}} \simeq \frac{800 \text{ ps}}{200 \text{ ps}} \simeq 4.00$$

- Pipelining improves performance by *increasing instruction throughput, in contrast to decreasing the execution time of an individual instruction*.

Designing Instruction Sets for Pipelining

- LEGv8 was designed for pipelined execution:
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Memory operands only appear in loads or stores
    - We can use the execute stage to calculate the memory address and then access memory in the following stage

Pipeline Hazards

- There are situations in pipelining when the next instruction cannot execute in the following clock cycle.
- These events are called **hazards**, and there are three different types:

- **Structure hazards**
  - When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.
- **Data hazard**
  - When a planned instruction cannot execute in the proper clock cycle because data that are needed to execute the instruction are not yet available.
- **Control hazard,** also called **branch hazard**
  - When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.

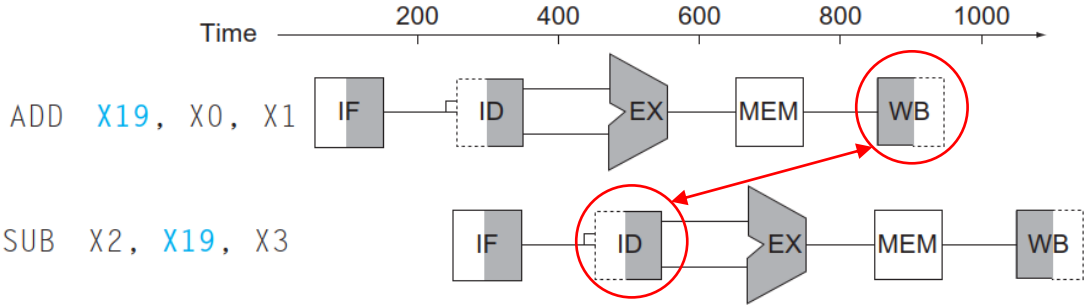
Structural Hazards

- When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.
- The LEGv8 instruction set was designed to be pipelined, making it fairly easy for designers to avoid structural hazards when designing a pipeline.
- Suppose, however, that we had a single memory instead of two.
  - Load/store requires data access
  - Instruction fetch would have to ***stall*** for that cycle
    - Would cause a pipeline "***bubble***"
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

Data Hazards

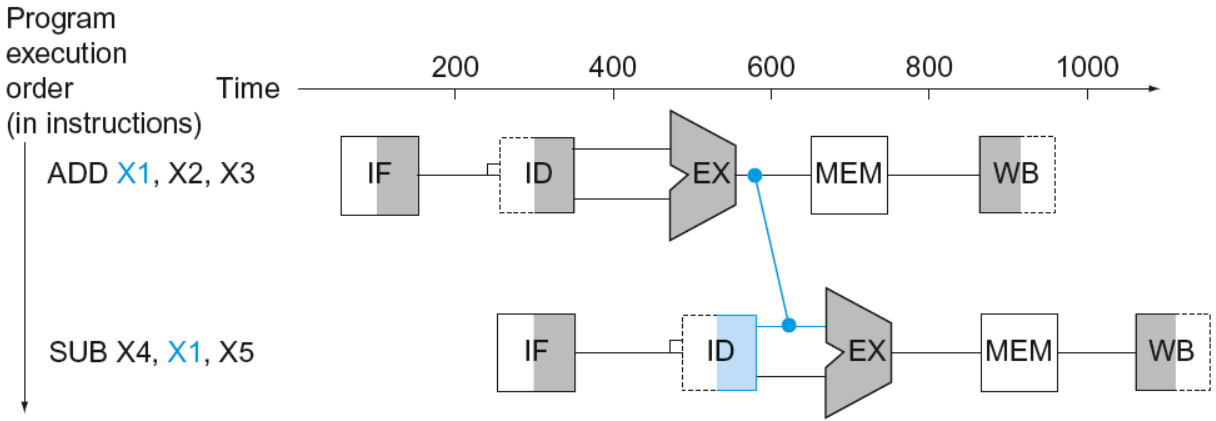
- Data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline.
- For example:

```
ADD  X19, X0, X1
SUB  X2, X19, X3
```



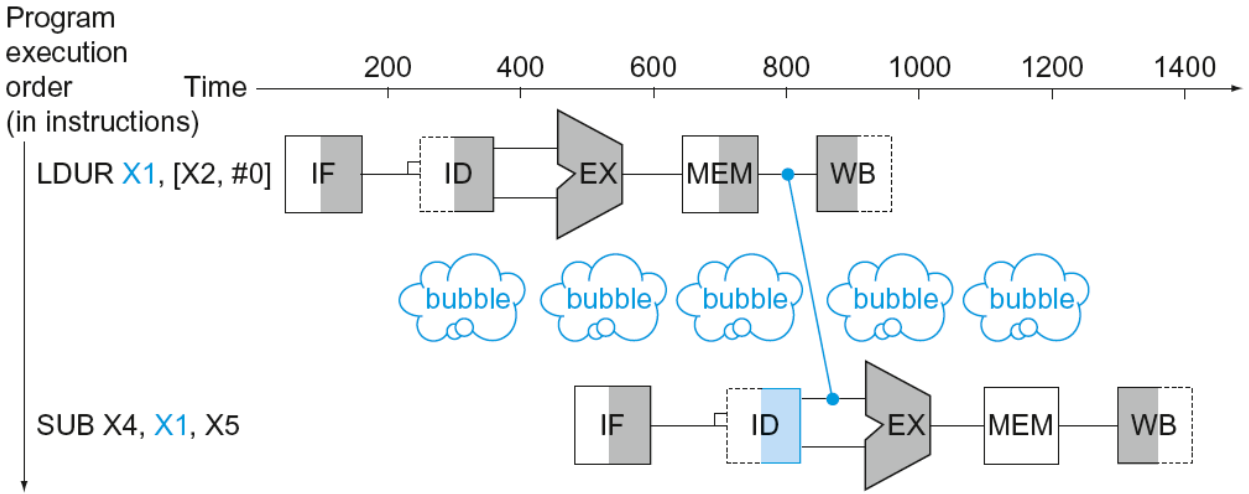
Forwarding (aka Bypassing)

- We could try to rely on compilers to remove all such hazards, but the results would not be satisfactory: these dependences happen just too often.
- The primary solution is based on **forwarding:**
- As soon as the ALU creates the sum for the add, we can supply it as an input for the subtract.
  - Requires extra connections in the datapath



Load-Use Data Hazard

- Forwarding cannot prevent all pipeline stalls.
- Suppose the first instruction was a load of X1 instead of an add.
- The desired data would be available only after the *fourth stage* of the first instruction in the dependence, which is *too late for* the input of *the third stage* of SUB.



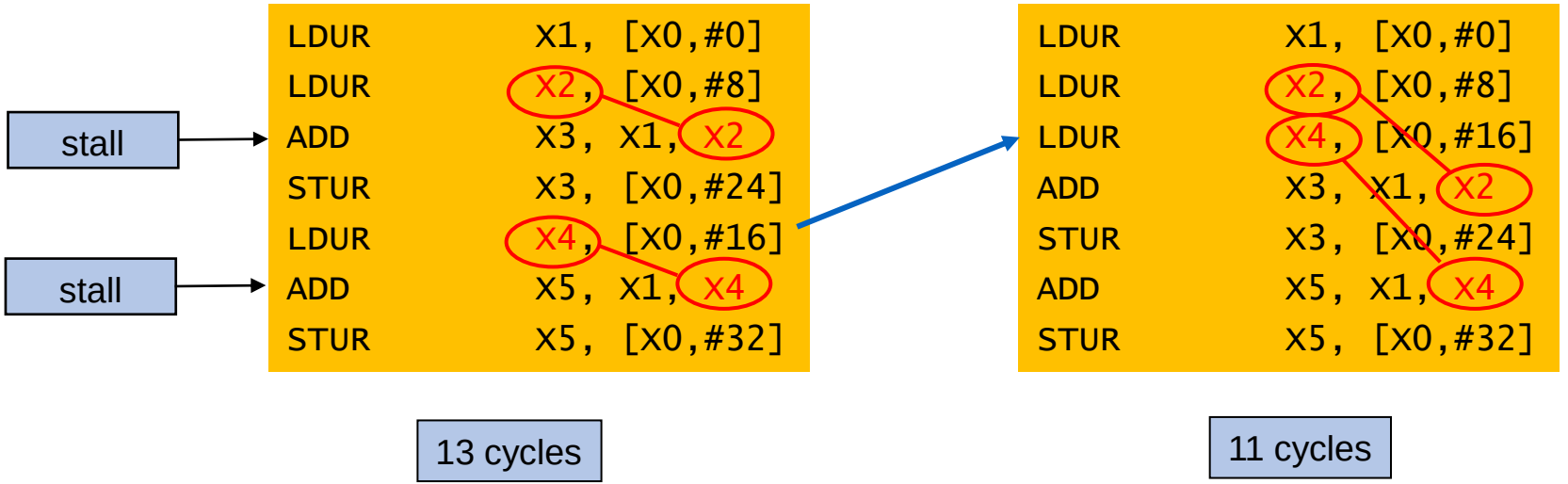
Load-Use Data Hazard

- **load-use data hazard** A specific form of data hazard in which the data being loaded by a load instruction have not yet become available when they are needed by another instruction.

- **pipeline stall** Also called **bubble**. A stall initiated in order to resolve a hazard.

Reordering Code to Avoid Pipeline Stalls

- We can prevent Load-Use Data Hazard by reordering the code to *avoid the use of the load result in the next instruction.*
- C code for A = B + E; C = B + F;



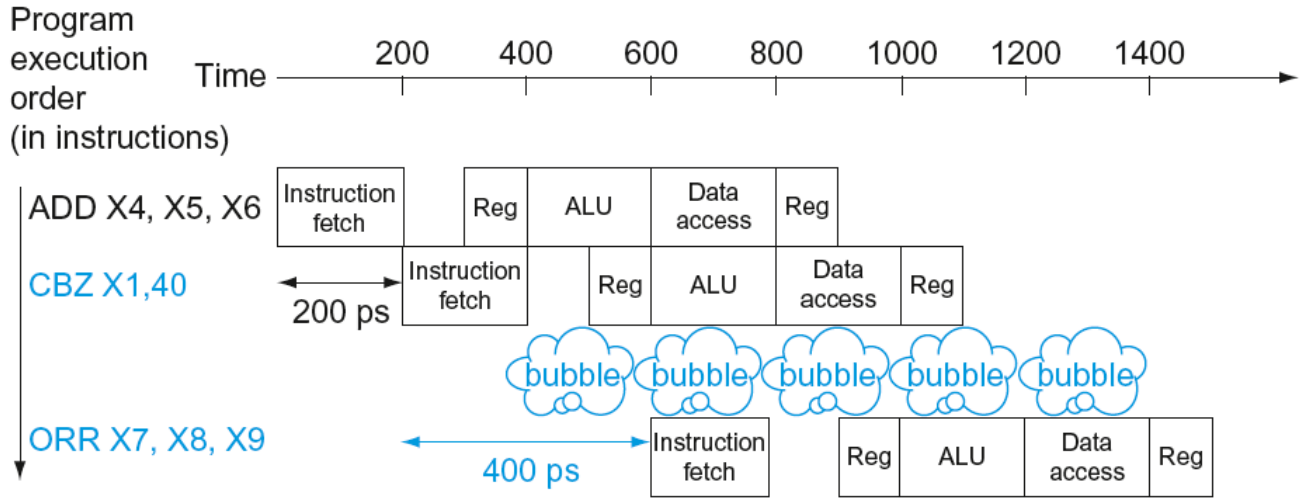
Control Hazards

- **Control hazard** also called **branch hazard**. When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.
- In the **conditional branch instruction**, we must begin fetching the instruction following the branch on the following clock cycle….
- Nevertheless, *the pipeline cannot possibly know what the next instruction should be*, since it only just received the branch instruction from memory!

- In LEGv8 pipeline:
- We need to compare registers and compute the target early in the pipeline.
- Let’s assume that we put in enough extra hardware so that we can test a register, calculate the branch address, and update the PC during the second stage of the pipeline (ID stage).

Stall on Branch

- One possible solution is to stall immediately after we fetch a branch, waiting until the pipeline determines the outcome of the branch.



Performance of “Stall on Branch”

- Estimate the impact on the clock cycles per instruction (CPI) of stalling on branches.
- Assume all other instructions have a CPI of 1.

- Conditional branches are 17% of the instructions executed in SPECint2006.
- Since the other instructions (83%) run have a CPI of 1, and conditional branches took one extra clock cycle for the stall, then we would see a CPI of

$$CPI = 0.83*1+0.17*2 = 1.17$$

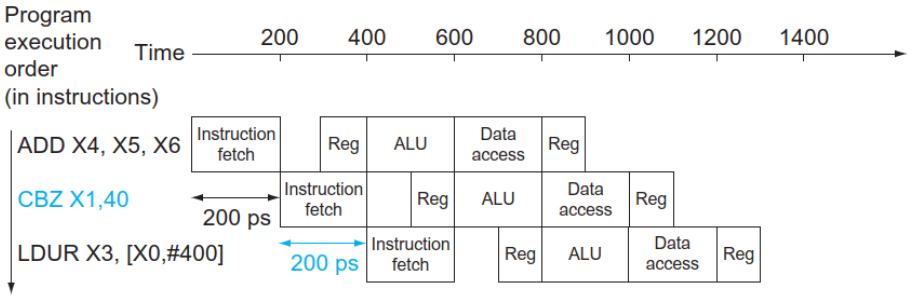
- Hence a slowdown of 1.17 versus the ideal case.

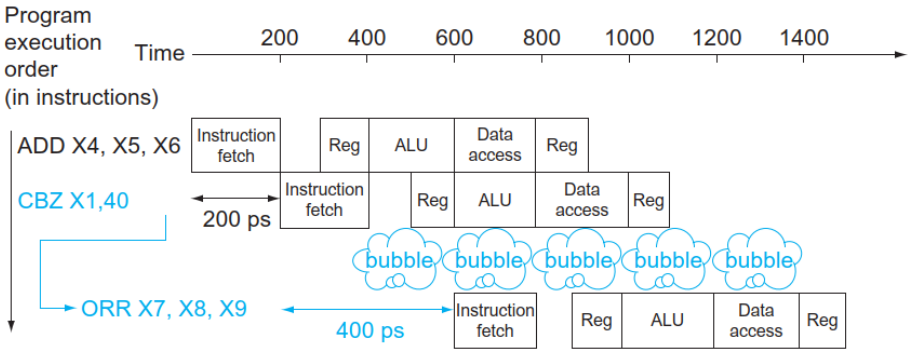
Branch Prediction

- If we cannot resolve the branch in the second stage, as is often the case for *longer pipelines*, then we'd see an *even larger slowdown* if we stall on conditional branches.
- The cost of this option is too high for most computers to use and motivates a second solution to the control hazard: **predict** the outcome of branch.
- This option does not slow down the pipeline when you are correct.
- When you are wrong: you need to redo the load that was washed while guessing the decision (with the creation of a **bubble**).

- One simple approach is to predict always that conditional branches will be untaken.
- When you're right, the pipeline proceeds at full speed.
- Only when conditional branches are taken does the pipeline stall.

Branch Prediction





More-Realistic Branch Prediction

- A more sophisticated version of **branch prediction** would have some conditional branches predicted as taken and some as untaken.
- **Static** branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- **Dynamic** branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

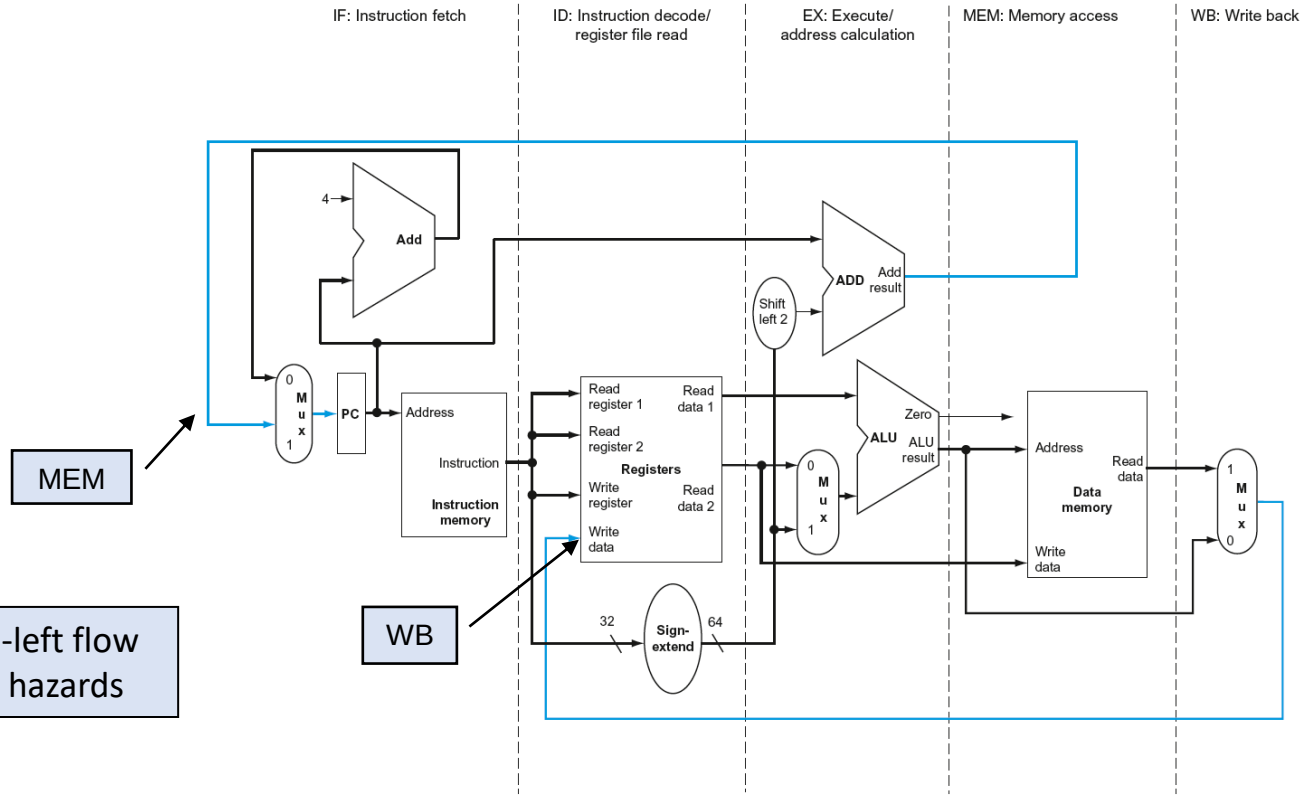
Third approach: delayed branch

- There is a third approach to the control hazard, called a ***delayed branch***.
  - Used in MIPS, TI C54, and many other processors
- The delayed branch *always executes the next sequential instruction*, with the branch taking place after that one instruction delay.
- It is hidden from the MIPS assembly language programmer because the assembler can automatically arrange the instructions to get the branch behavior desired by the programmer.
  - MIPS software will place an instruction immediately after the delayed branch instruction that is not affected by the branch.
  - A taken branch changes the address of the instruction that follows this safe instruction.

Pipeline Summary: the BIG picture

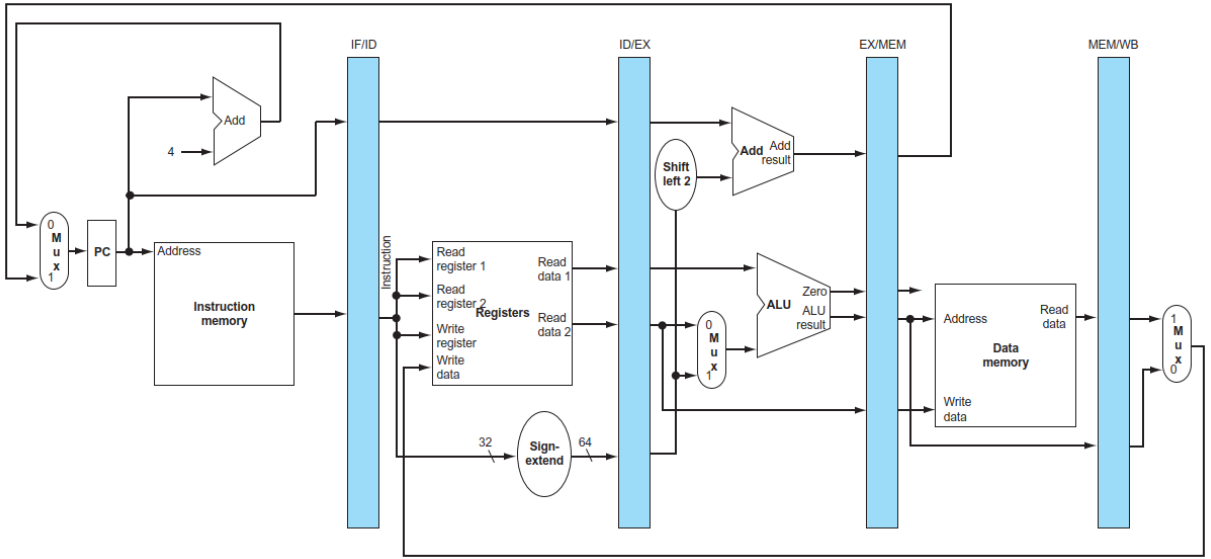
- Pipelining increases the number of simultaneously executing instructions and the rate at which instructions are started and completed.
- Pipelining does not reduce the time it takes to complete an individual instruction, i.e., the *latency*.
  - For example, the five-stage pipeline still takes five clock cycles for the instruction to complete.
- Pipelining *improves instruction throughput* rather than individual instruction execution time.
- Instruction sets can either make life harder or simpler for pipeline designers, who must already cope with *structural, control, and data hazards*.
- Branch prediction and forwarding help make a computer fast while still getting the right answers.

LEGv8 Pipelined Datapath



Pipeline registers

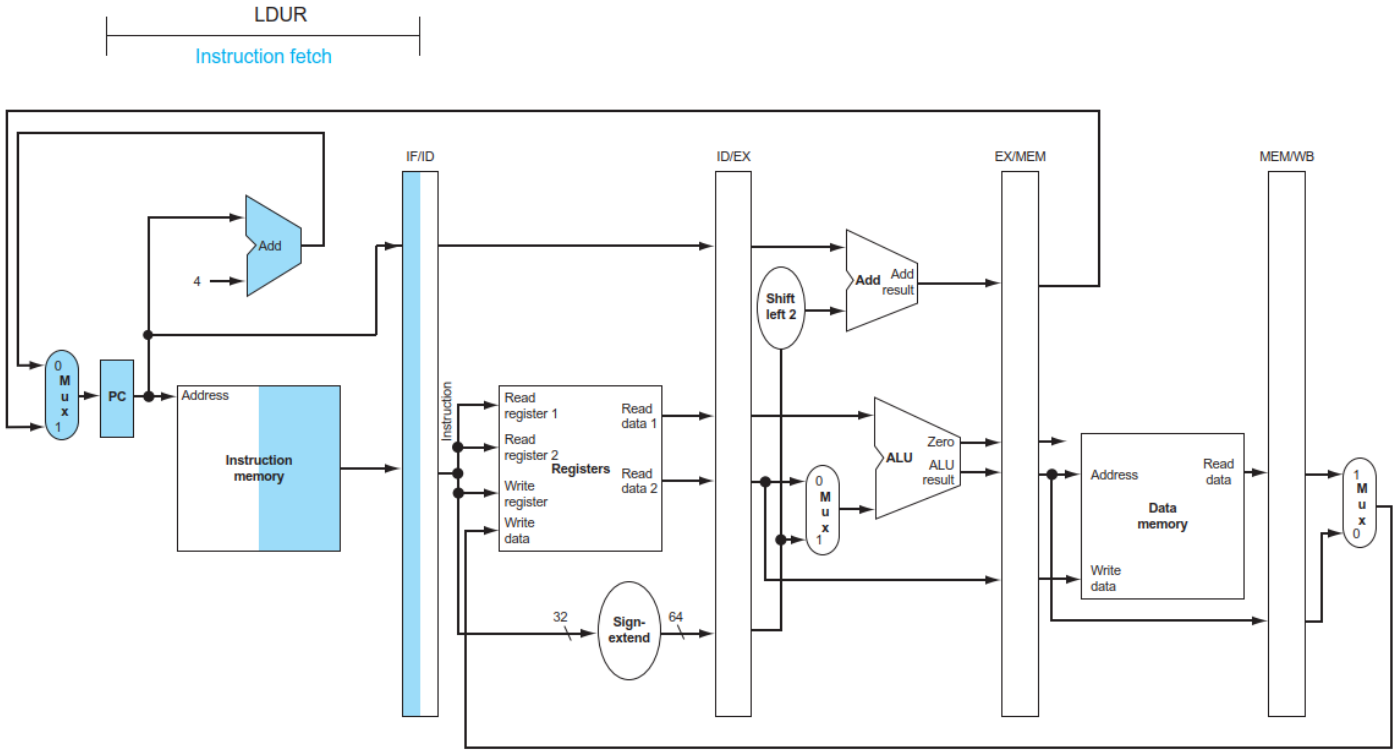
- Need registers between stages
  - To hold information produced in previous cycle



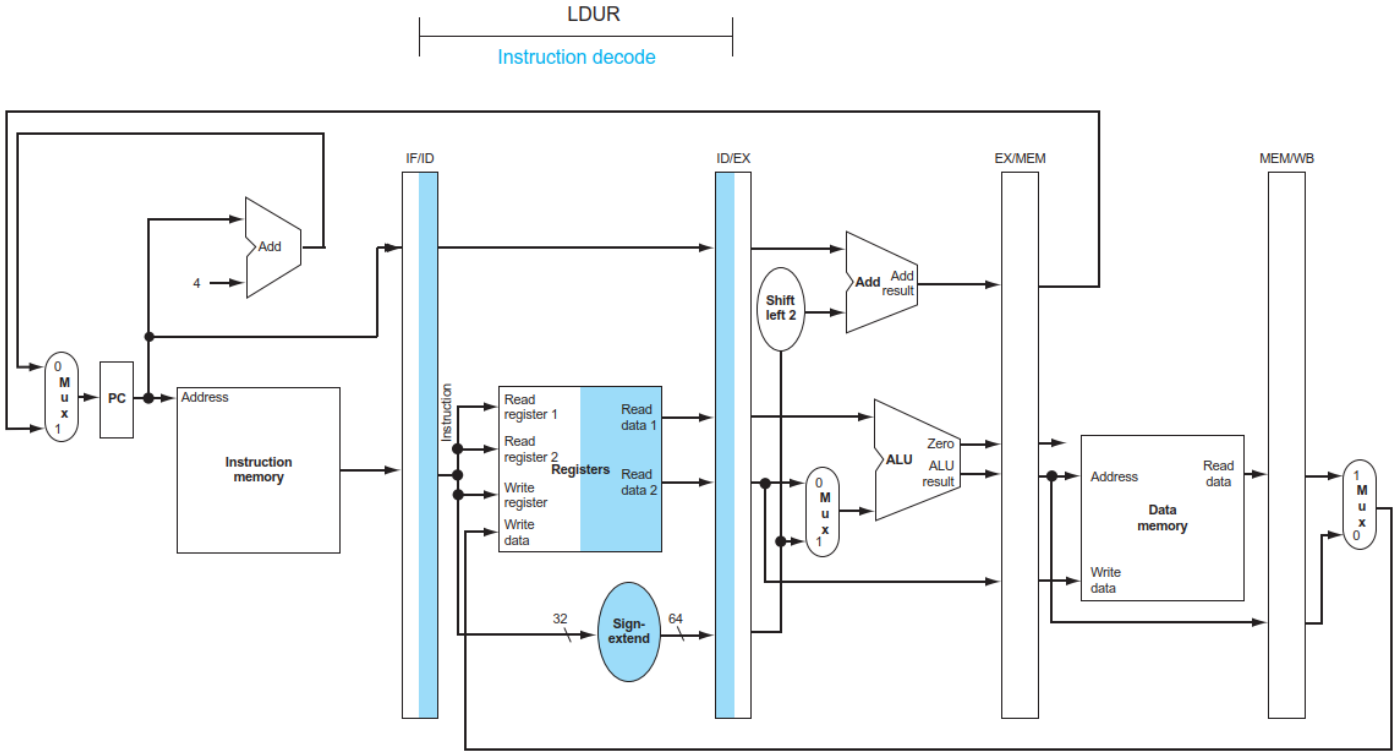
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - “Single-clock-cycle” pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. “multi-clock-cycle” diagram
    - Graph of operation over time
- We’ll look at “single-clock-cycle” diagrams for load & store

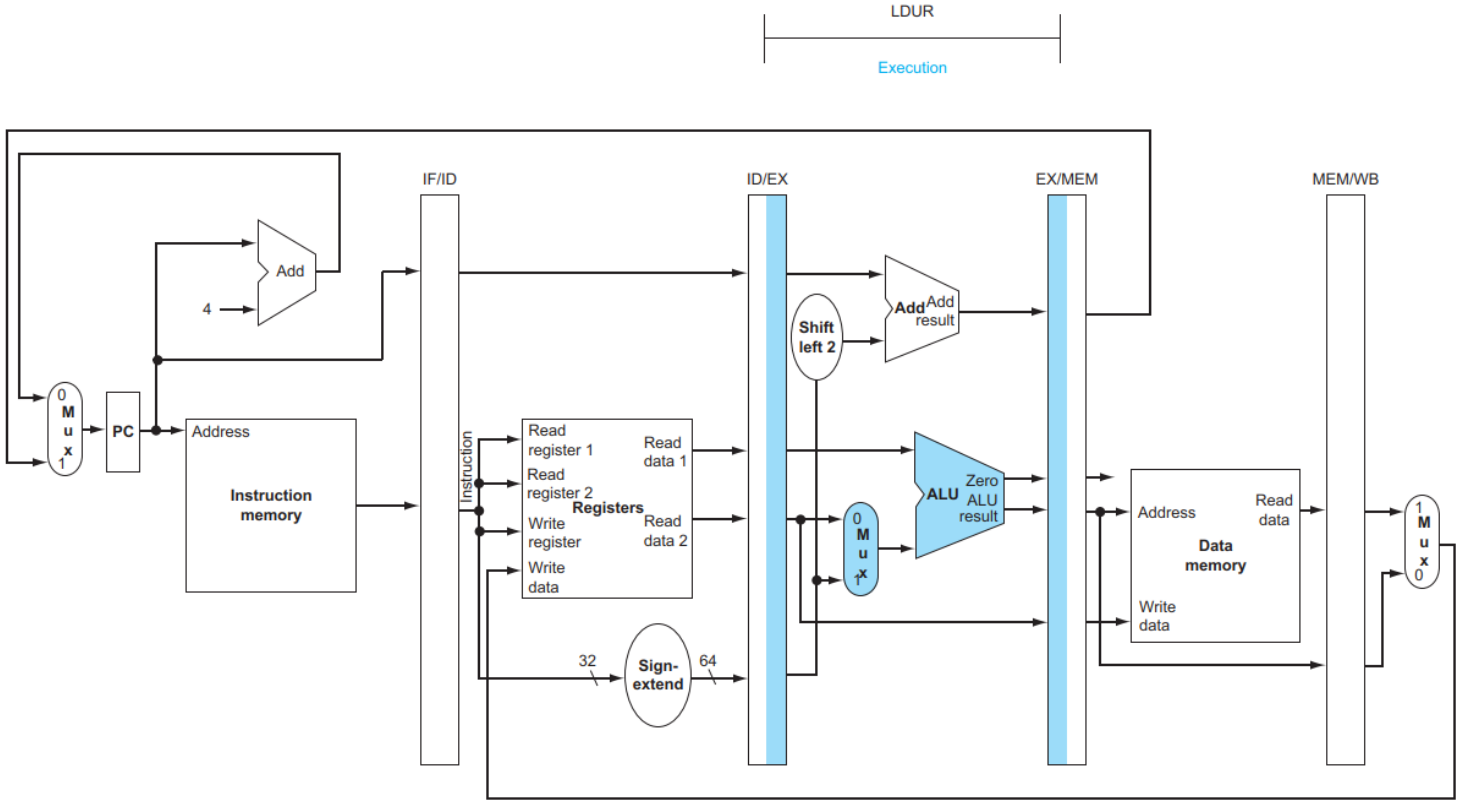
IF for Load, Store, …



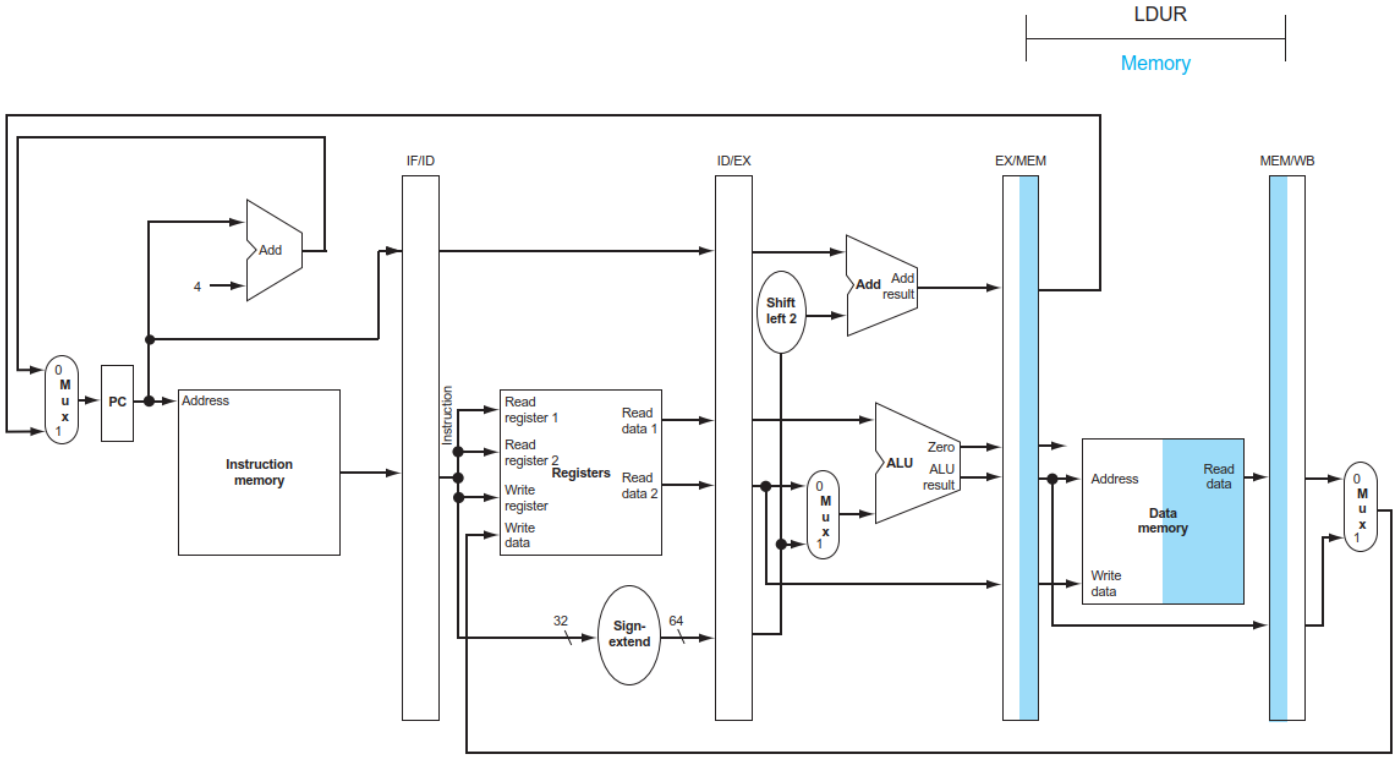
ID for Load, Store, …



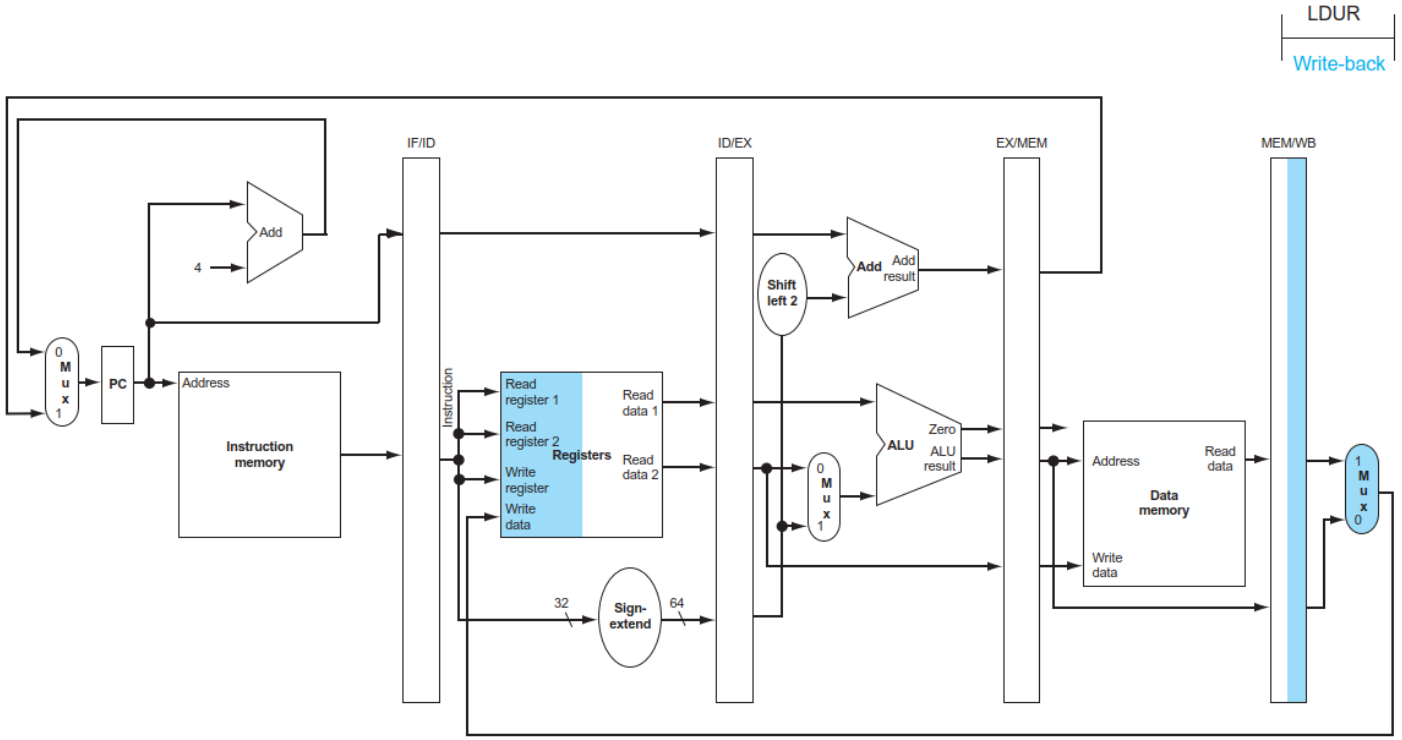
EX for Load


LDUR
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
32
Sign-extend
64

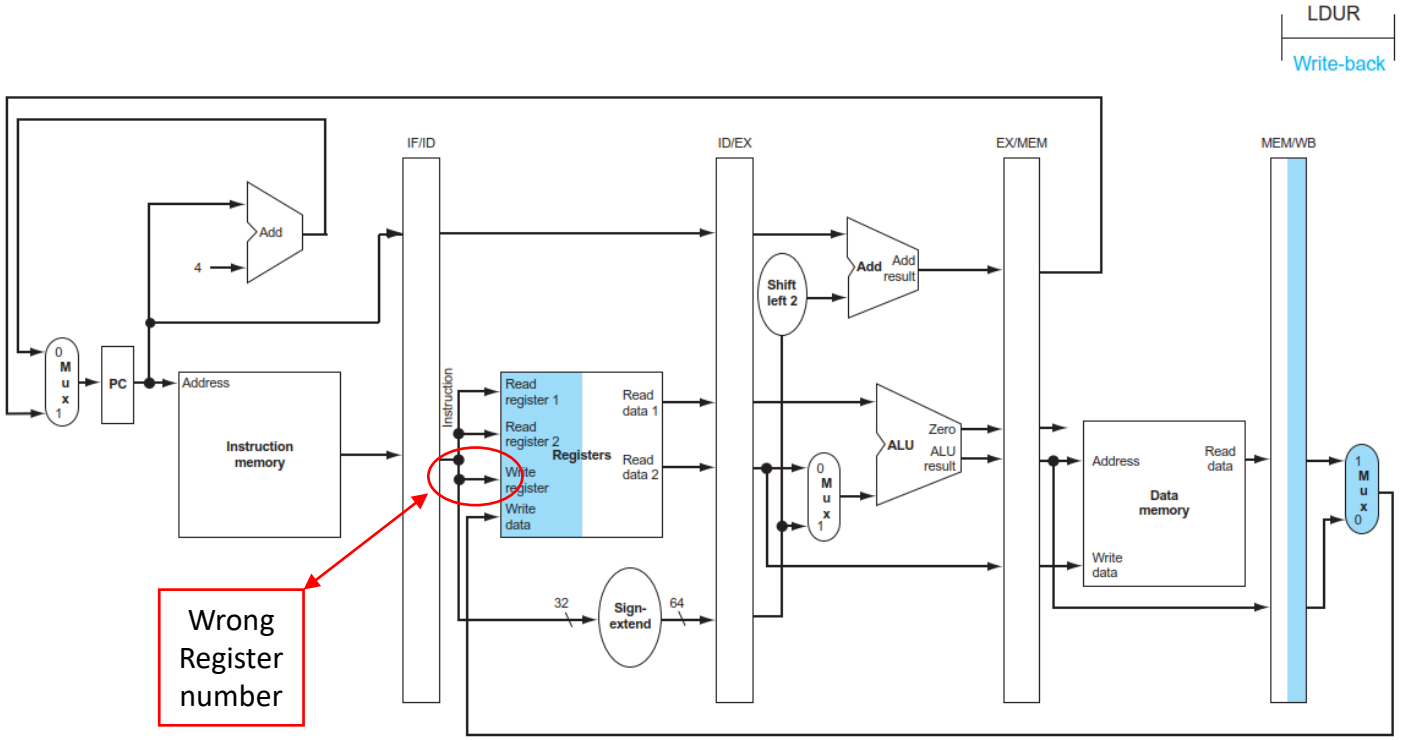
MEM for Load


LDUR
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
32
Sign-extend
64

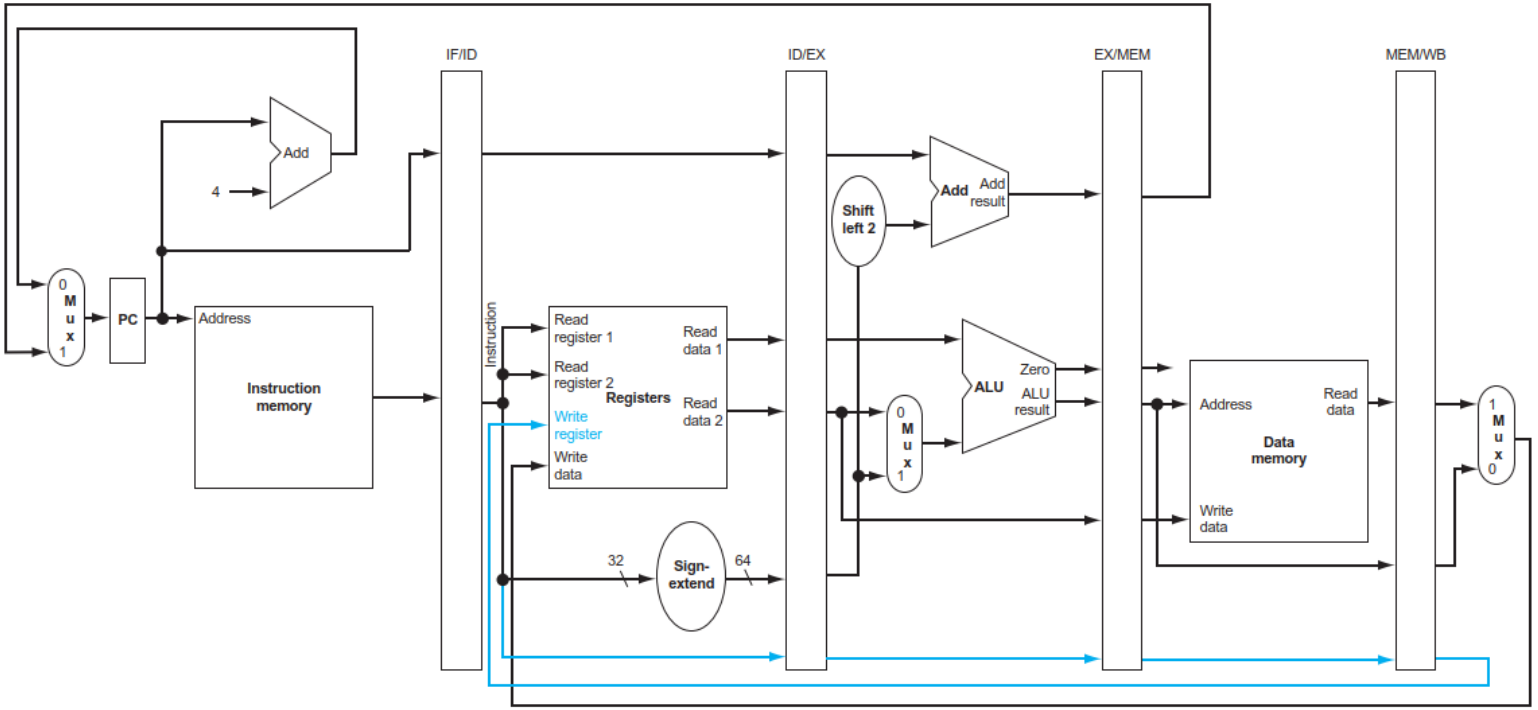
WB for Load


LDUR
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
32
Sign-extend
64

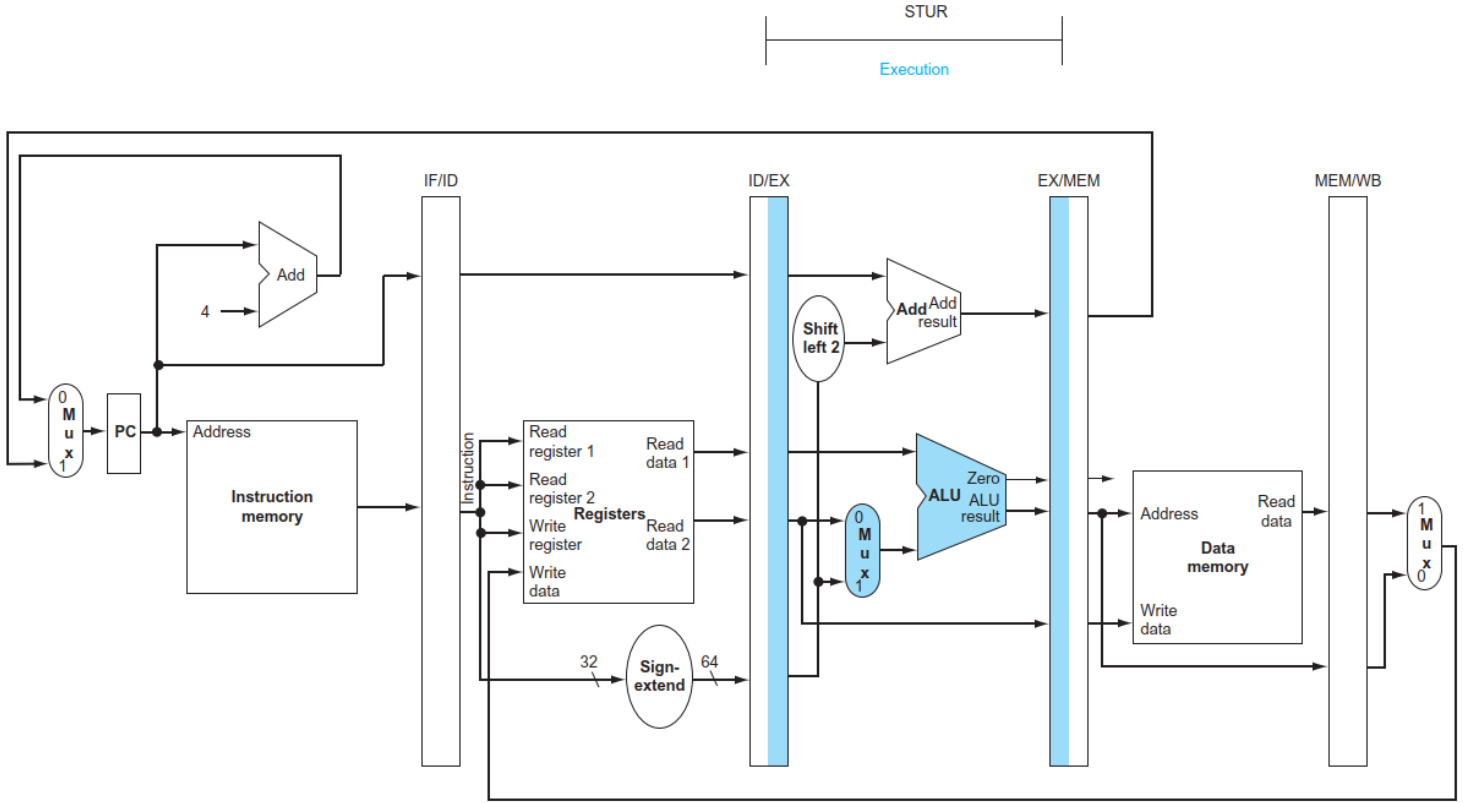
WB for Load


LDUR
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
32
Sign-extend
64
Wrong Register number

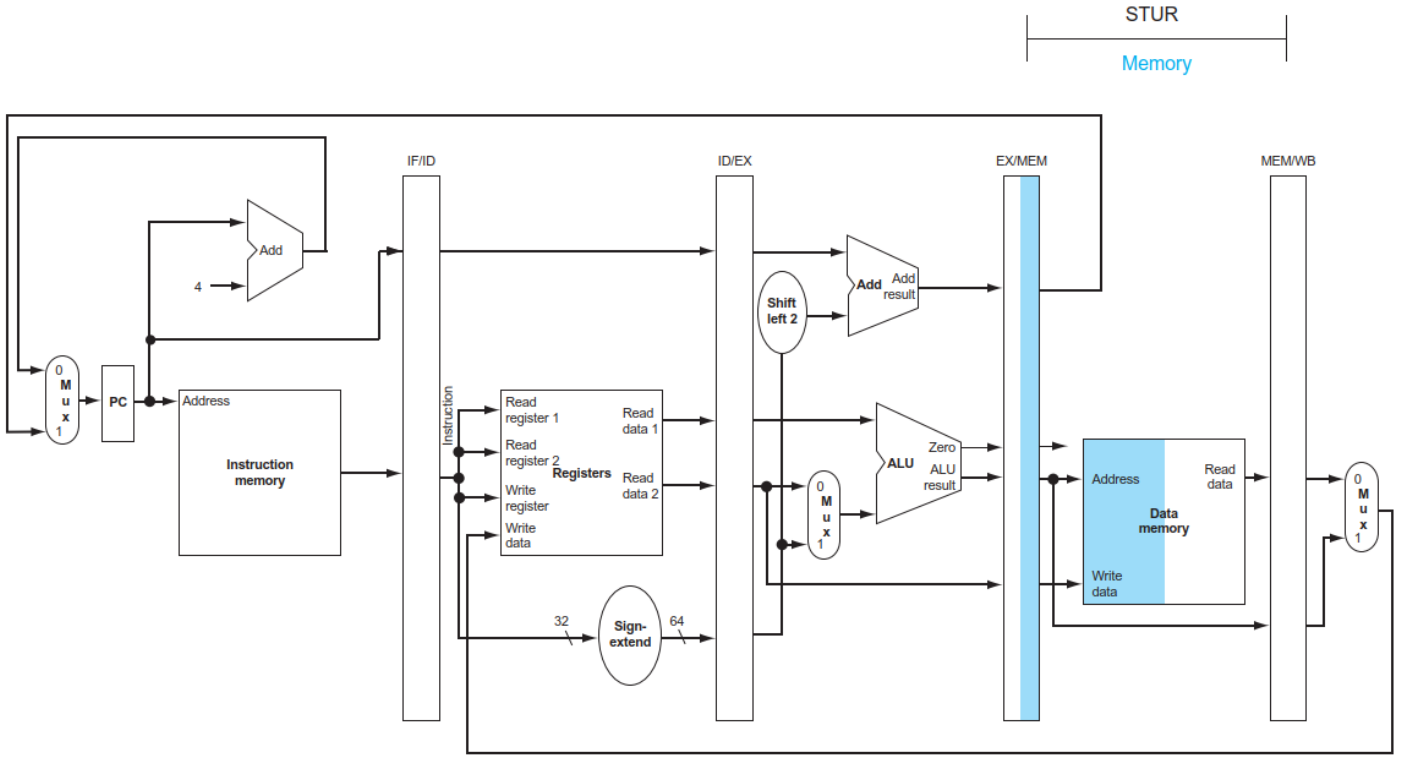
Corrected Datapath for Load



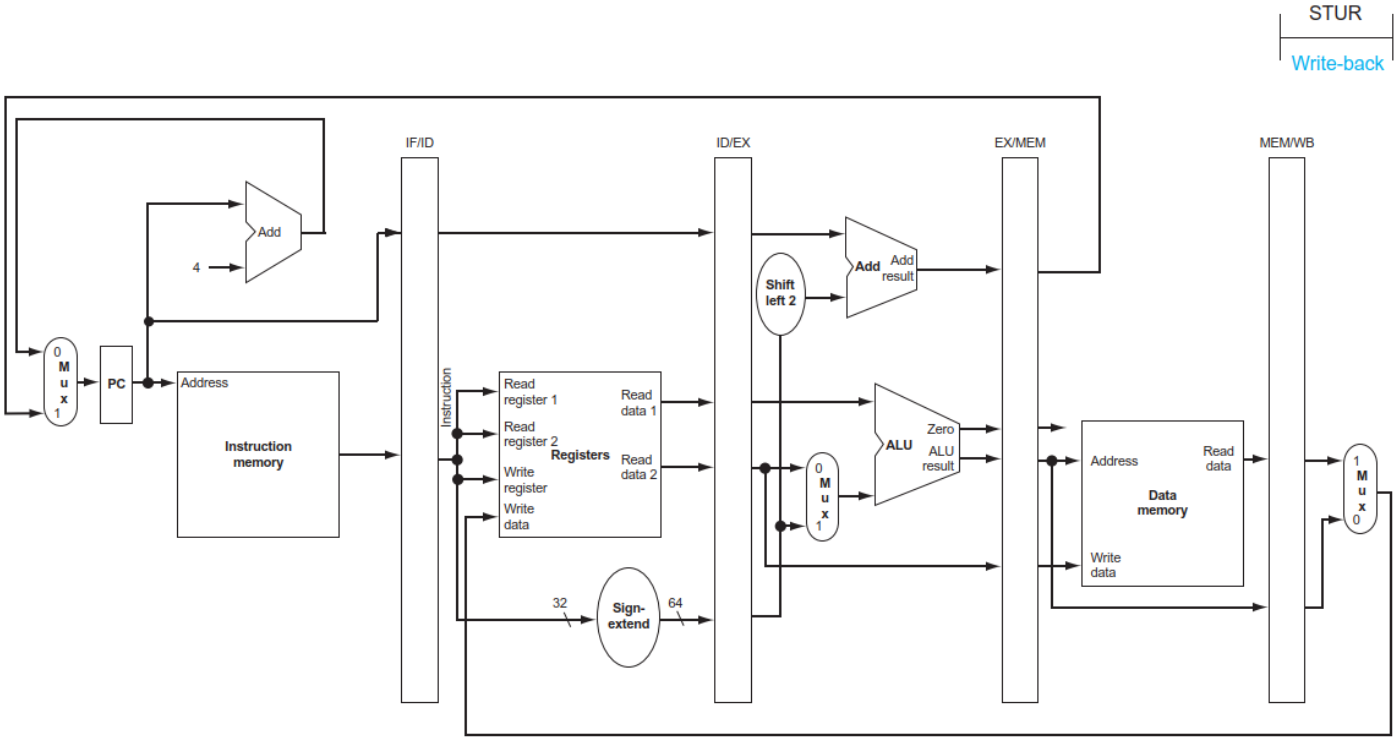
EX for Store


STUR
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
0
M
u
x
1
32
Sign-extend
64
Address
Data memory
Read data
Write data
1
M
u
x
0

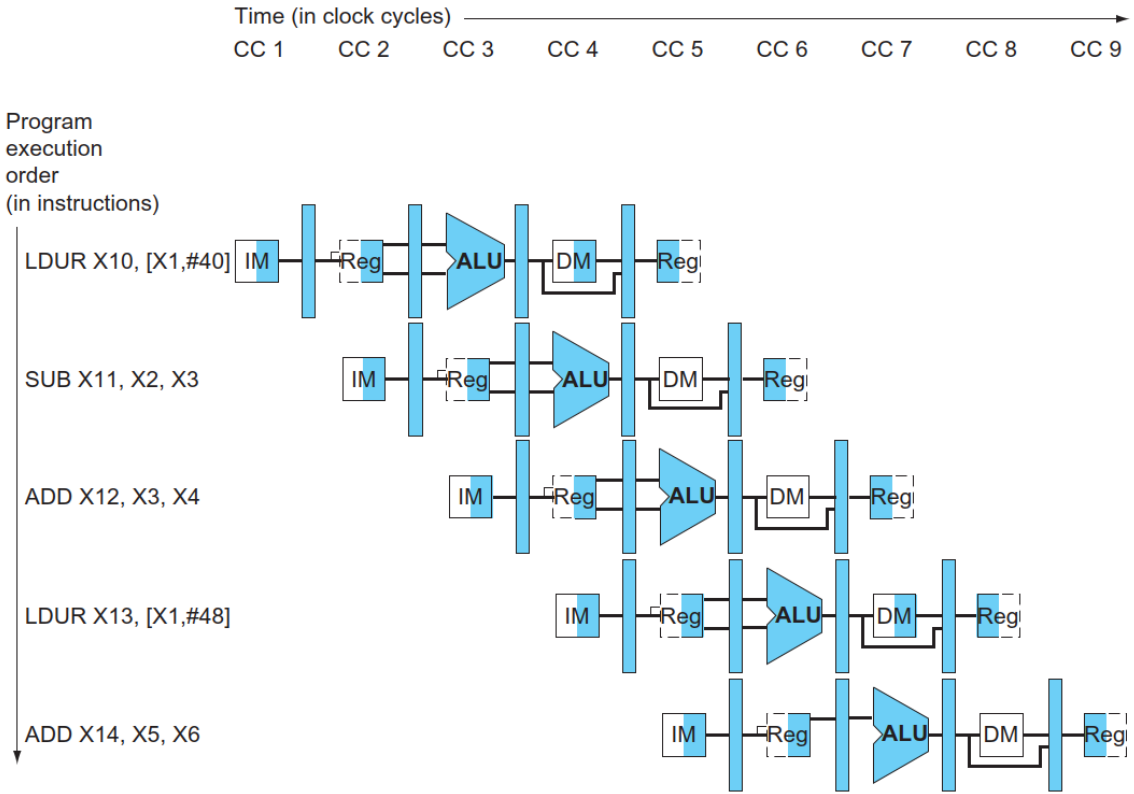
MEM for Store



WB for Store

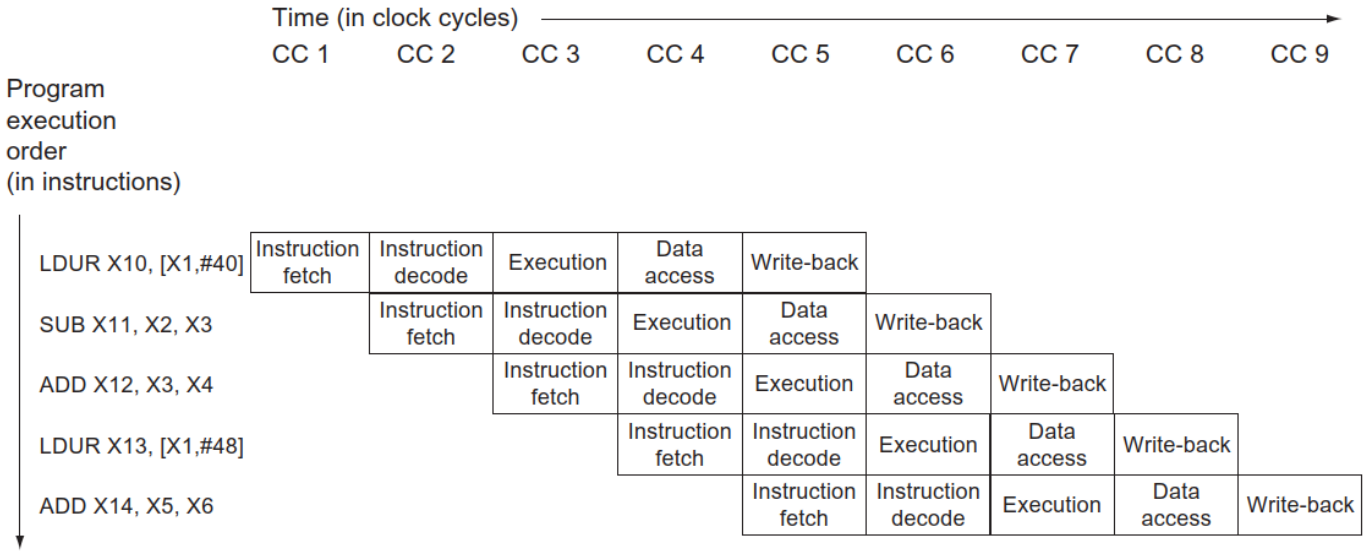

STUR
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
32
Sign-extend
64

Multi-Cycle Pipeline Diagram



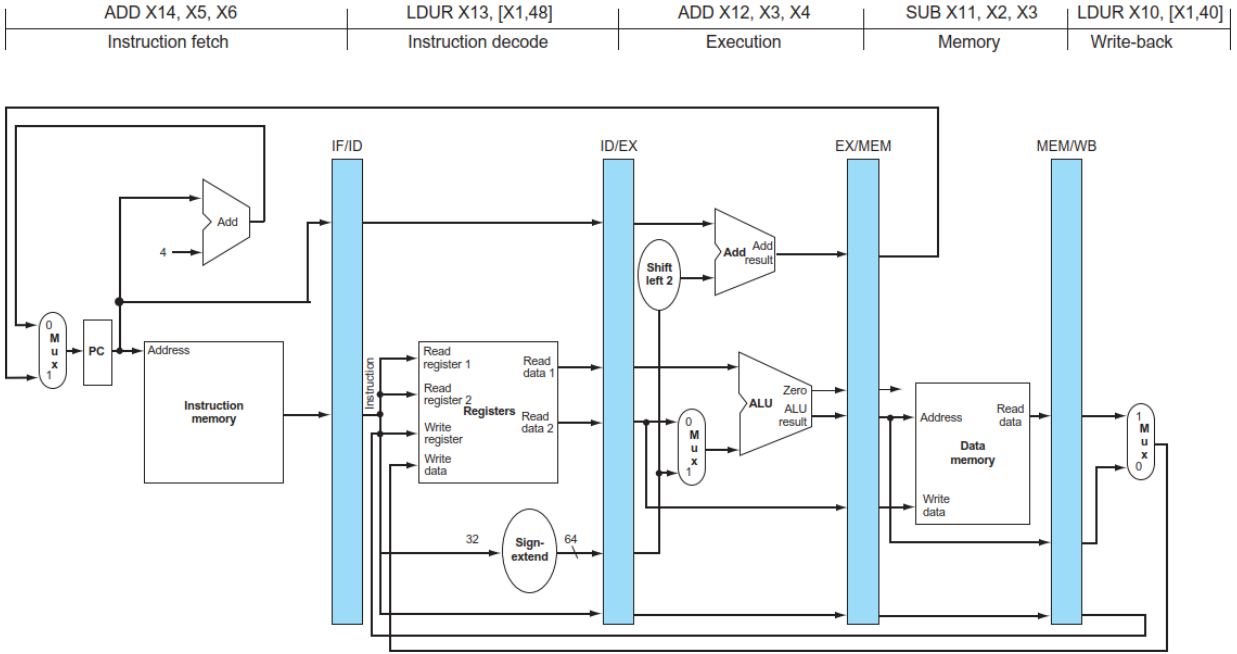
Multi-Cycle Pipeline Diagram

- Traditional form:

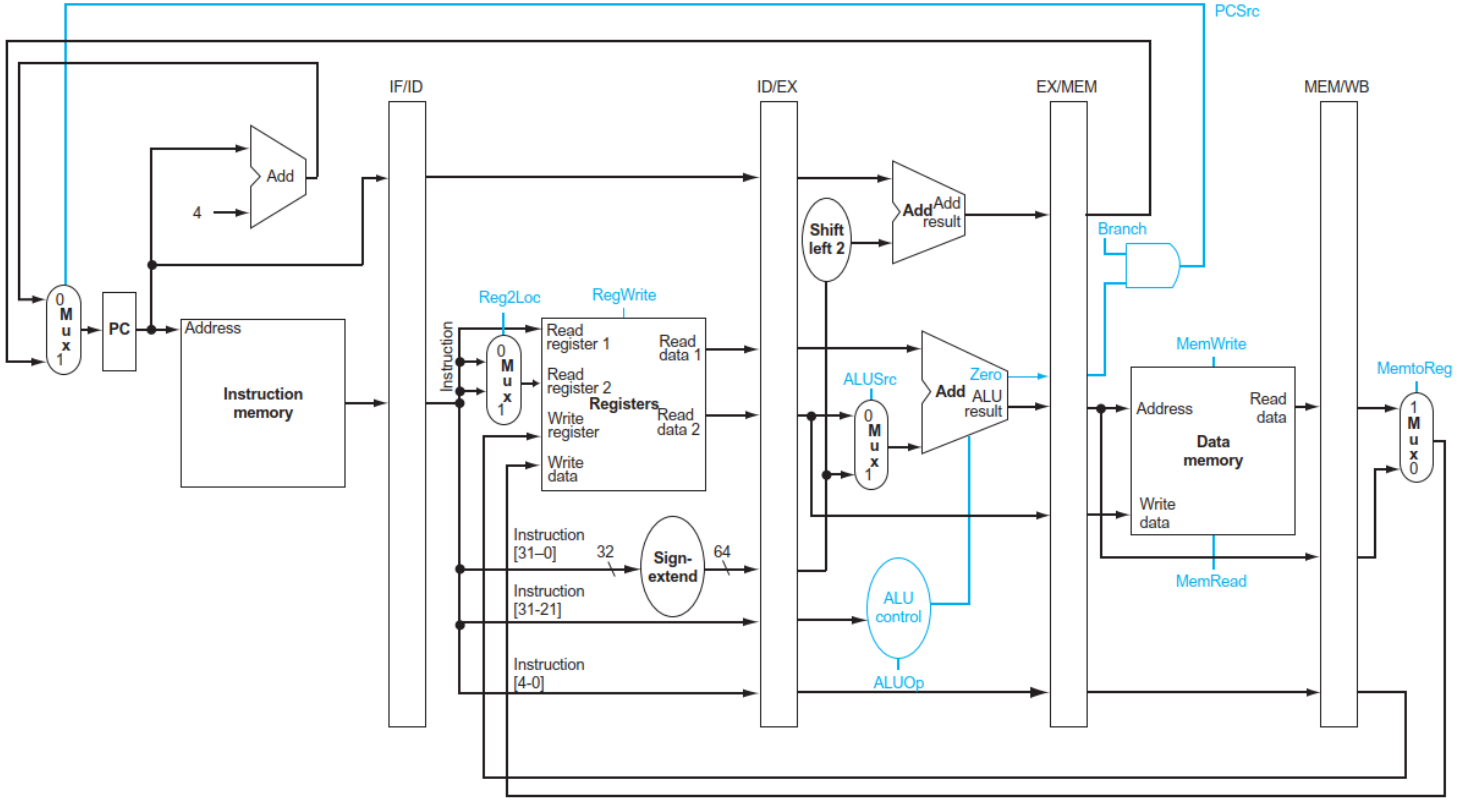


Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle.

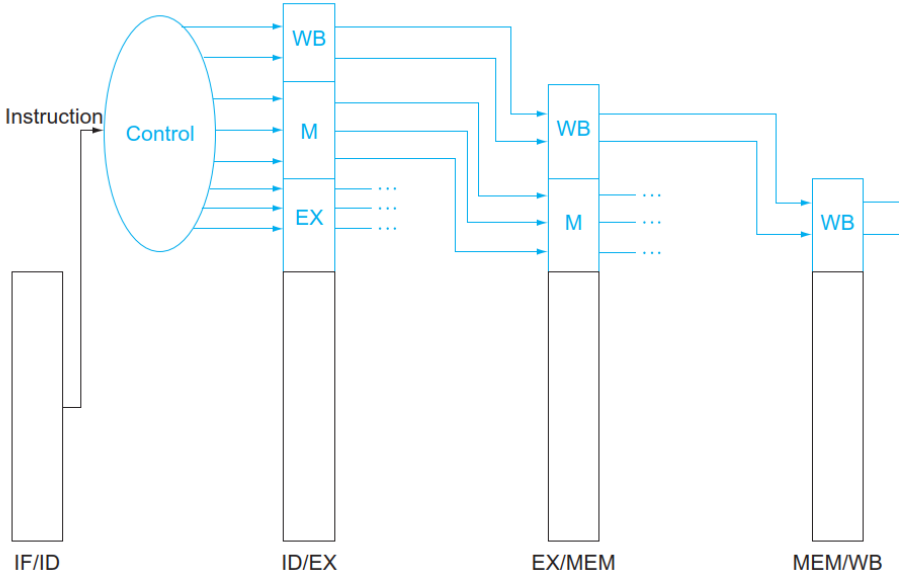


Pipelined Control (Simplified)

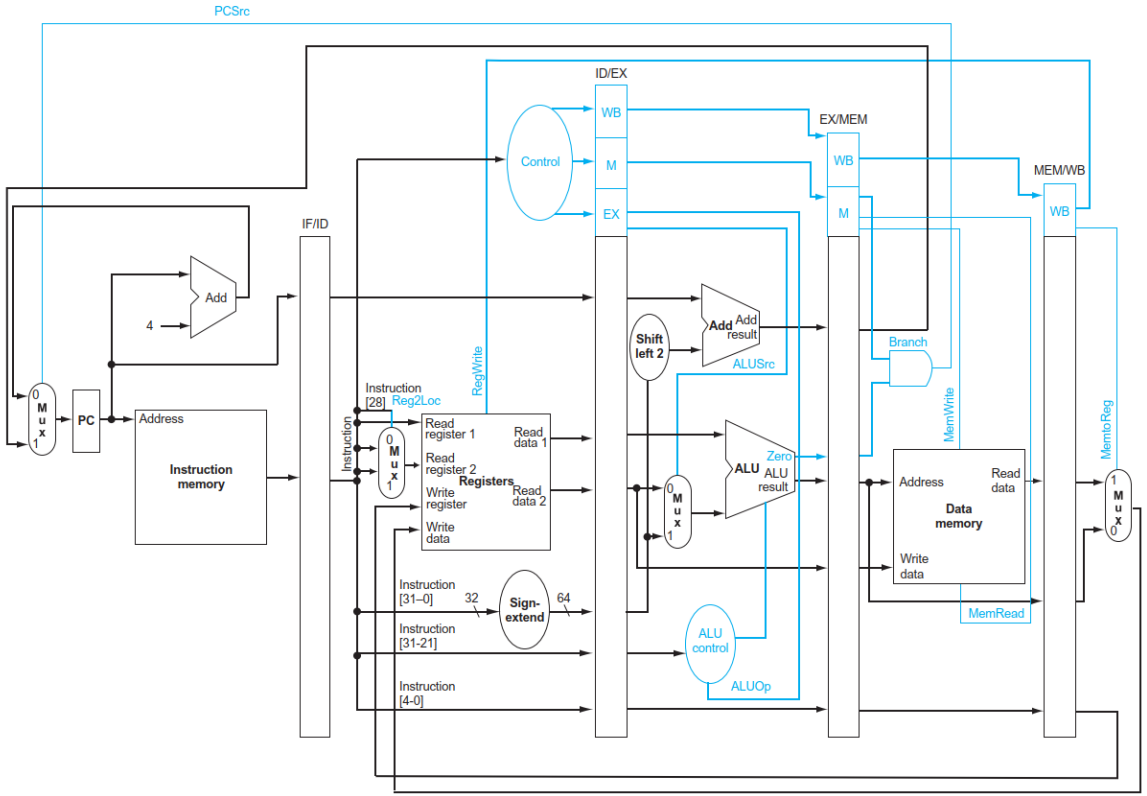


Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation



Pipelined Control



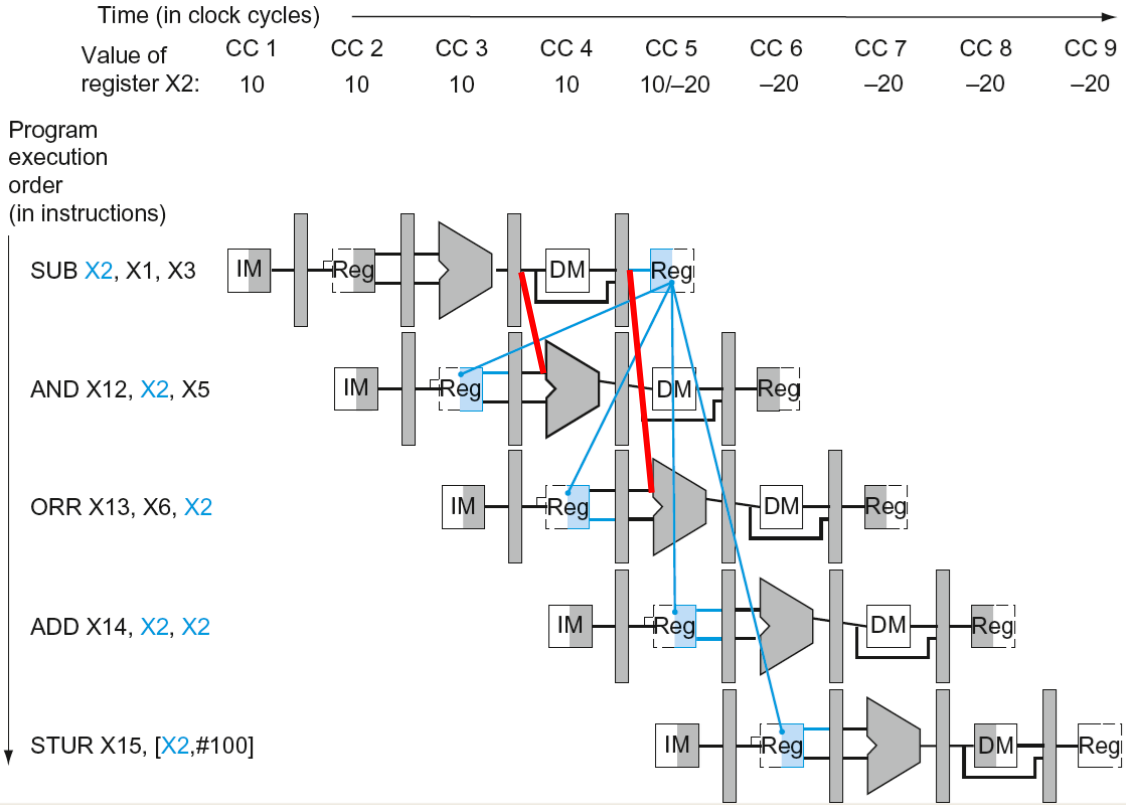
Data Hazards in ALU Instructions

- Consider this sequence:

```
SUB   X2, X1,X3      // Register X2 written by SUB
AND   X12,X2,X5      // 1st operand(X2) depends on SUB
OR    X13,X6,X2      // 2nd operand(X2) depends on SUB
ADD   X14,X2,X2      // 1st(X2) & 2nd(X2) depend on SUB
STUR  X15,[X2,#100] // Base (X2) depends on SUB
```

- We can resolve hazards with forwarding
  - How do we detect when to forward?

Dependencies & Forwarding



Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRn1, ID/EX.RegisterRm2
- Data hazards when

1a. EX/MEM.RegisterRd = ID/EX.RegisterRn1
1b. EX/MEM.RegisterRd = ID/EX.RegisterRm2

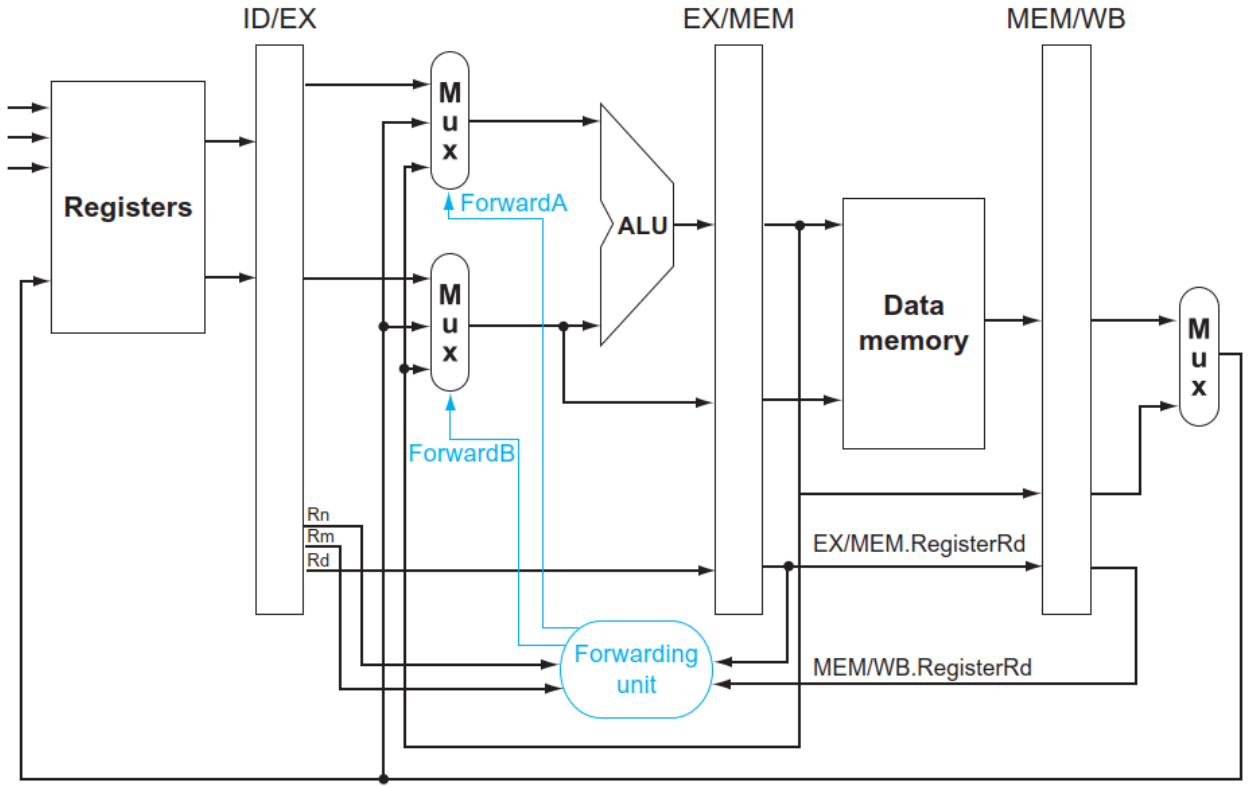
→ Fwd from EX/MEM pipeline reg

2a. MEM/WB.RegisterRd = ID/EX.RegisterRn1
2b. MEM/WB.RegisterRd = ID/EX.RegisterRm2

→ Fwd from MEM/WB pipeline reg

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not XZR
  - EX/MEM.RegisterRd ≠ 31, MEM/WB.RegisterRd ≠ 31

Forwarding Paths



Forwarding Conditions

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Detection Conditions

1. *EX hazard:*

```
if  (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 31)
and (EX/MEM.RegisterRd = ID/EX.RegisterRn1)) ForwardA = 10

if  (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 31)
and (EX/MEM.RegisterRd = ID/EX.RegisterRm2)) ForwardB = 10
```

2. *MEM hazard:*

```
if  (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 31)
and (MEM/WB.RegisterRd = ID/EX.RegisterRn1)) ForwardA = 01
if  (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 31)
and (MEM/WB.RegisterRd = ID/EX.RegisterRm2)) ForwardB = 01
```

Double Data Hazard

- Consider the sequence:

```
add X1,X1,X2
add X1,X1,X3
add X1,X1,X4
```

- Both hazards occur
  - We want to use the most recent data
- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn't true

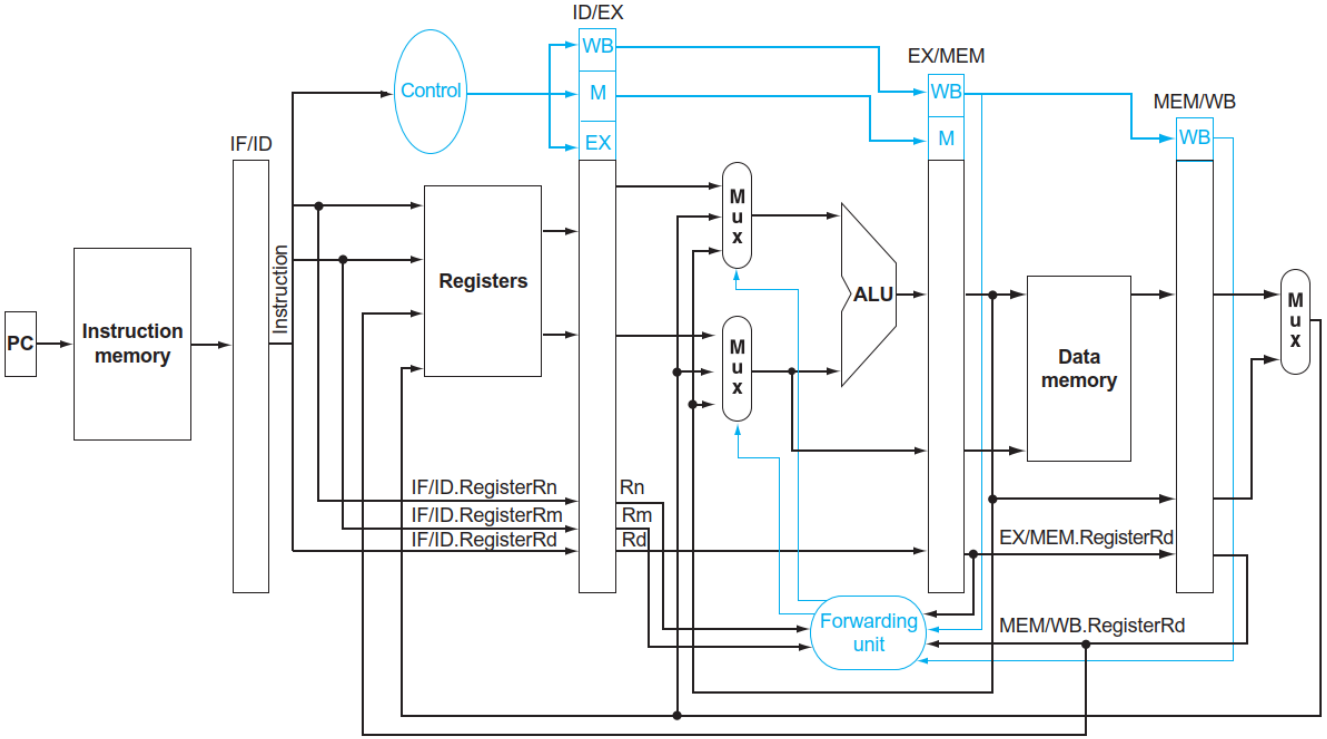
Double Data Hazard

- MEM hazard revised:

```
if   (MEM/WB.RegWrite
and  (MEM/WB.RegisterRd ≠ 31)
and  not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 31)
        and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRn1))
and  (MEM/WB.RegisterRd = ID/EX.RegisterRn1)) ForwardA = 01

if   (MEM/WB.RegWrite
and  (MEM/WB.RegisterRd ≠ 31)
and  not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 31)
        and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRm2))
and  (MEM/WB.RegisterRd = ID/EX.RegisterRm2)) ForwardB = 01
```

Datapath with Forwarding



Load-Use Hazard Detection

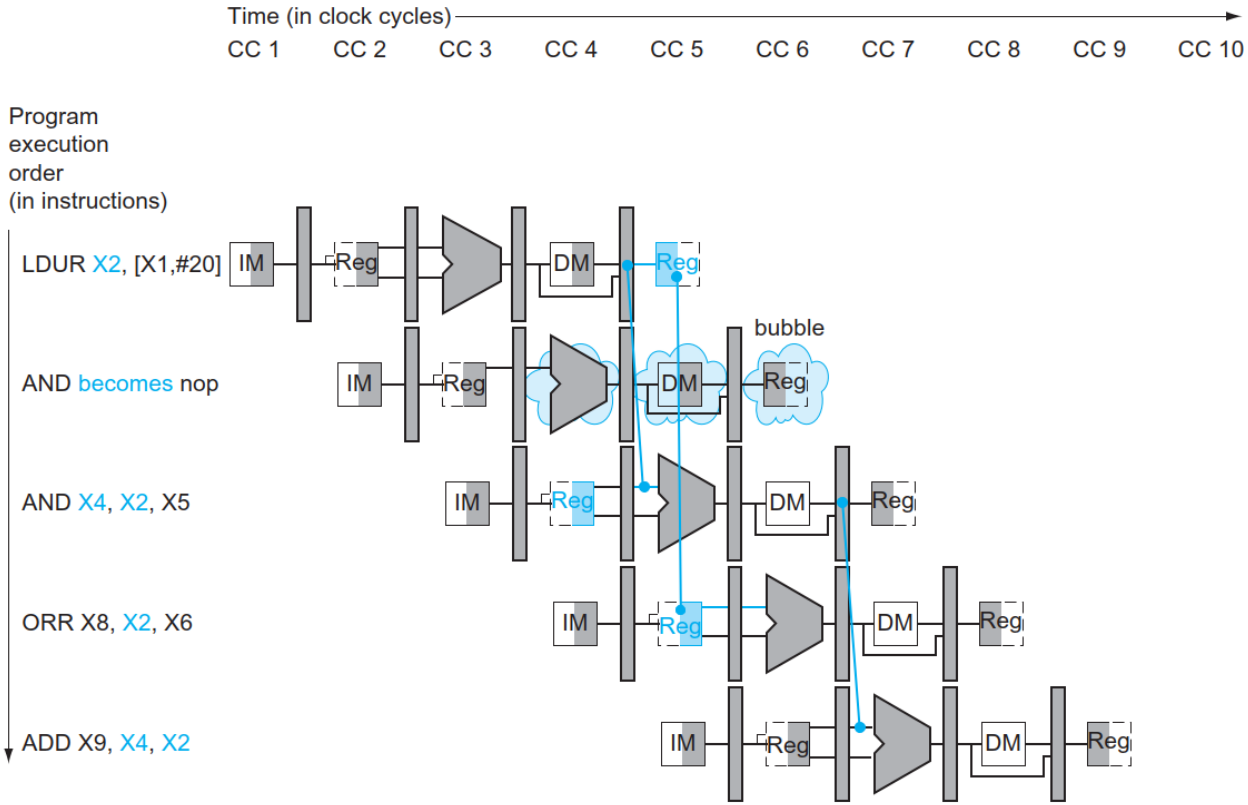
- One case where forwarding cannot save the day is when an instruction tries to read a register following a load instruction that writes the same register.
- In that case it is necessary to **stall** the pipeline.

- We check this condition when the using instruction is decoded in ID stage
- The ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRn1, IF/ID.RegisterRm2
- We have a **Load-Use hazard** when
  - ID/EX.MemRead and
    ((ID/EX.RegisterRd = IF/ID.RegisterRn1) or
    (ID/EX.RegisterRd = IF/ID.RegisterRm2))
- If detected, we must stall and insert bubble.

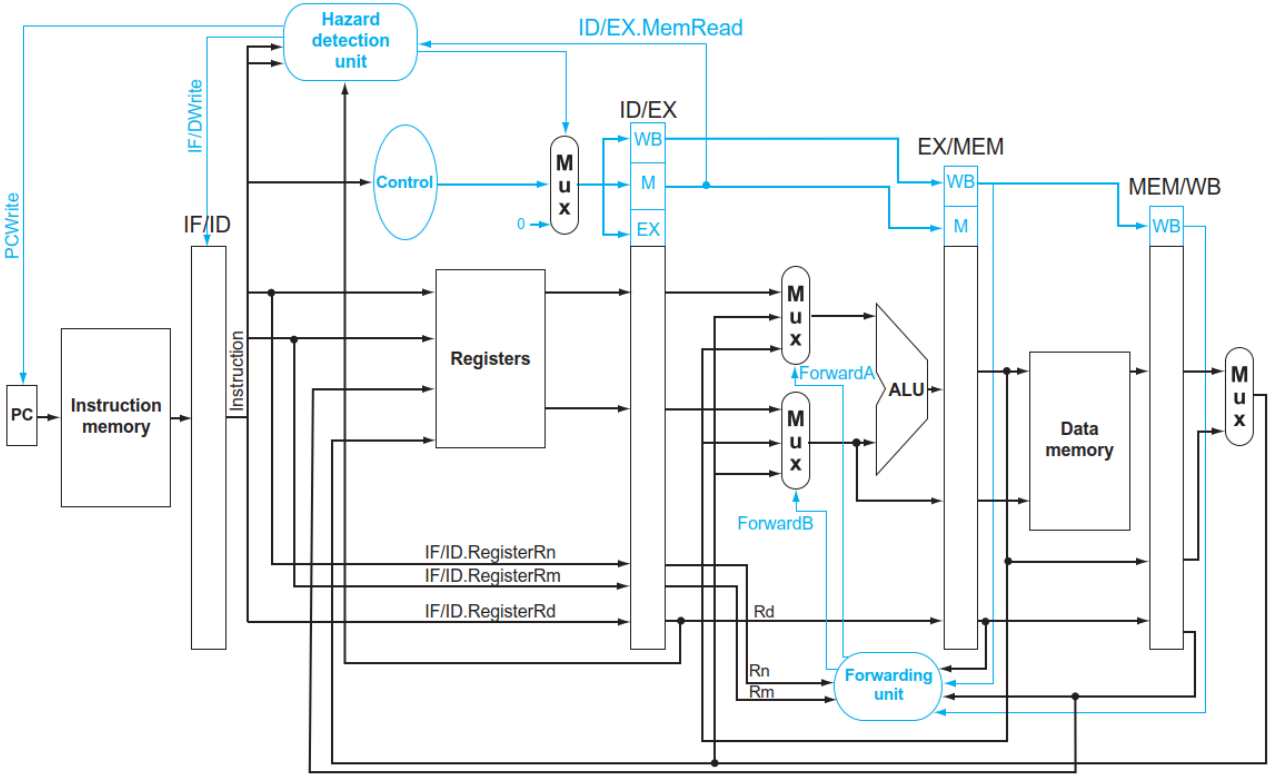
How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - The decoded instruction is decoded again
  - The following instruction is fetched again
- 1-cycle stall allows MEM to read data for LDUR
  - Can subsequently forward to EX stage

Load-Use Hazard



Datapath with Hazard Detection



Stalls and Performance

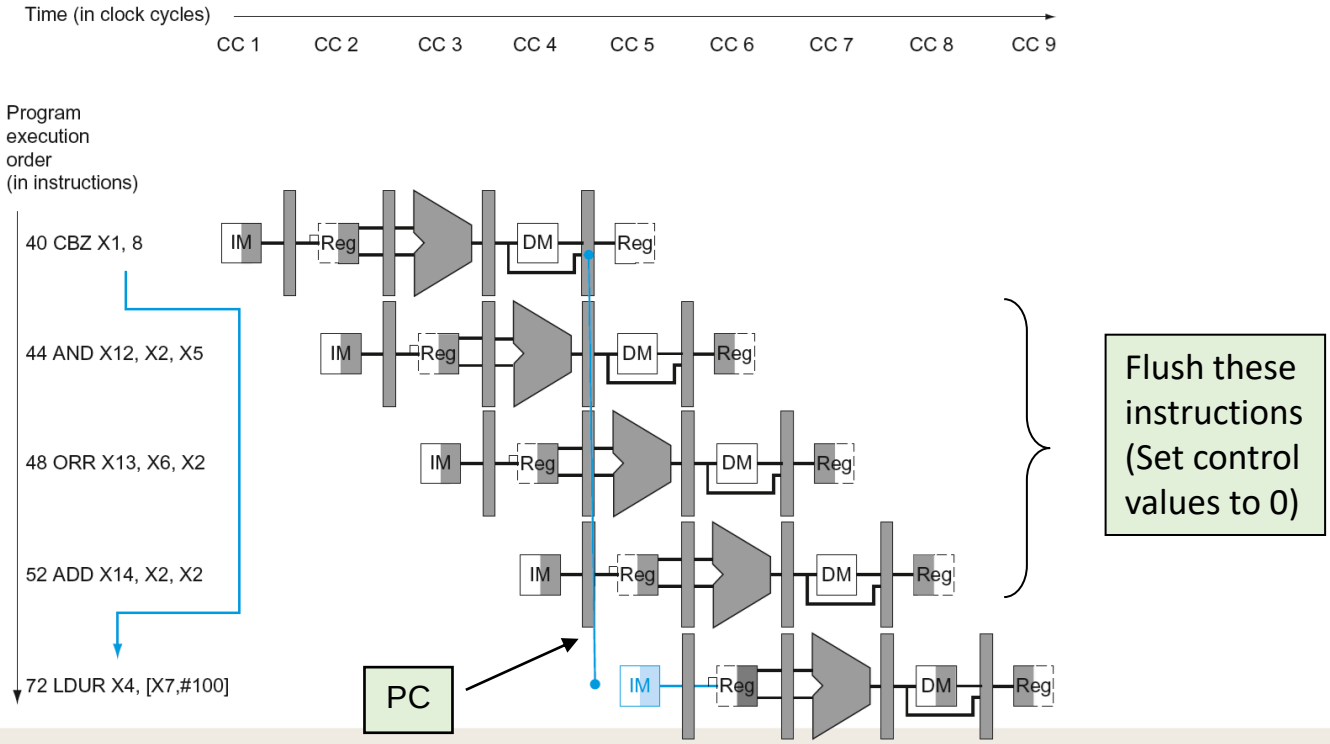
The BIG Picture

- Although the compiler generally relies upon the hardware to resolve hazards and thereby ensure correct execution, the compiler must understand the pipeline to achieve the best performance.
- Otherwise, unexpected stalls will reduce the performance of the compiled code.

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Branch Hazards

- If branch outcome determined in MEM



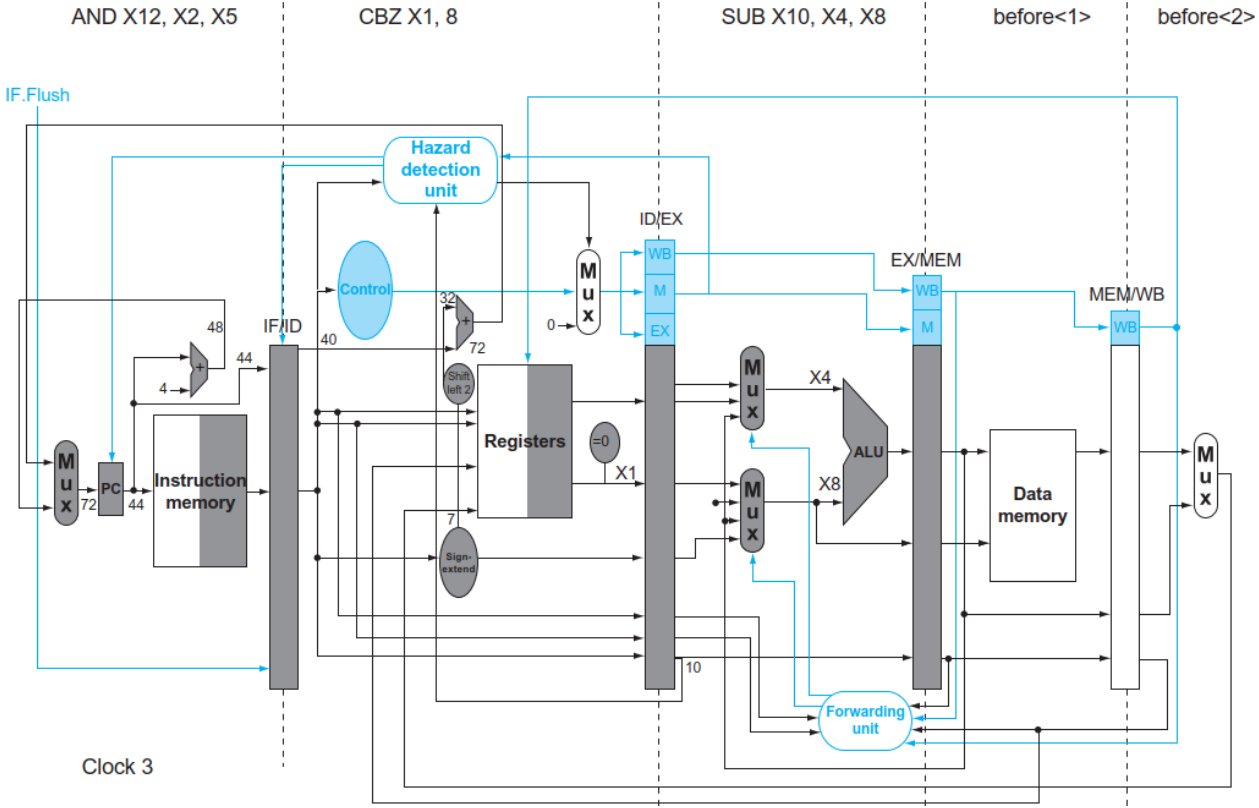
Reducing Branch Delay

- If we move the conditional branch execution earlier in the pipeline, then fewer instructions need be flushed.
- Moving the branch decision up requires two actions to occur earlier:
  - computing the branch target address and
  - evaluating the branch decision.
- Moving the branch test to the ID stage implies **additional forwarding** and **hazard detection** hardware.
- During ID, we must decode the instruction, decide whether a bypass to the zero test unit is needed, and complete the zero test so that if the instruction is a branch, we can set the PC to the branch target address. The bypassed source operands of a branch can come from either the ALU/MEM or MEM/WB pipeline latches.
- Because the value in a branch comparison is needed during ID but may be produced later in time, it is possible that a data hazard can occur and a stall will be needed.
  - If an **ALU** instruction immediately preceding a branch produces the operand for the test in the conditional branch, **a stall** will be required.
  - If a **load** is immediately followed by a conditional branch that depends on the load result, **two stall** cycles will be needed.

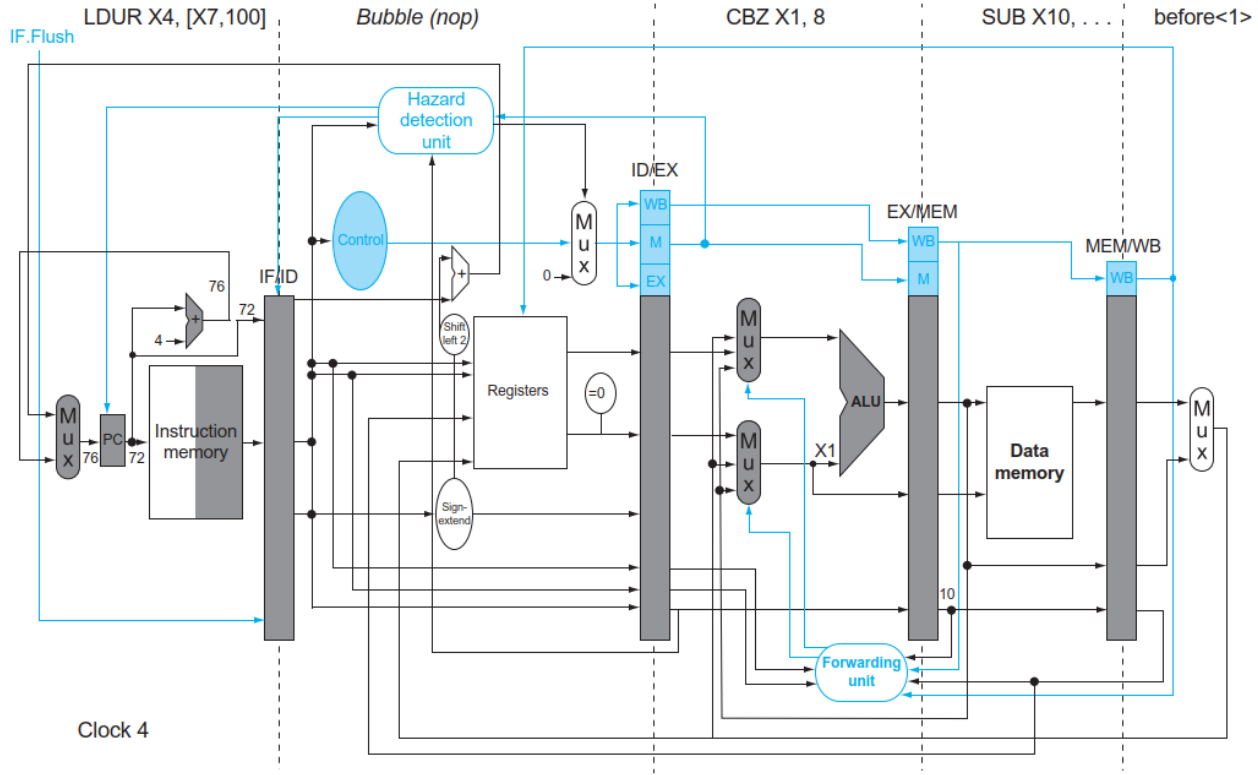
Example: branch taken

```
36:   SUB  X10, X4, X8
40:   CBZ  X1,  X3, 8
44:   AND  X12, X2, X5
48:   ORR  X13, X2, X6
52:   ADD  X14, X4, X2
56:   SUB  X15, X6, X7
      ...
72:   LDUR X4, [X7,#50]
```

Example: branch taken



Example: branch taken

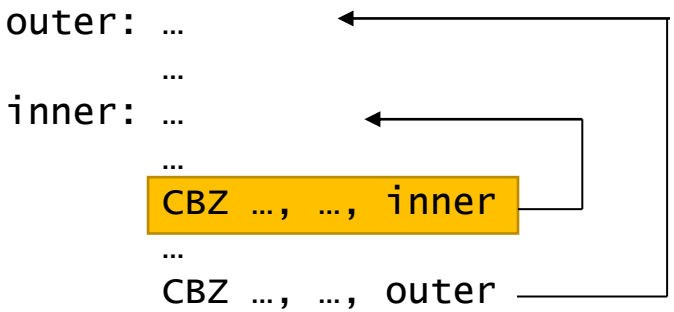


Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - *Branch prediction buffer* (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

1-Bit Predictor: Shortcoming

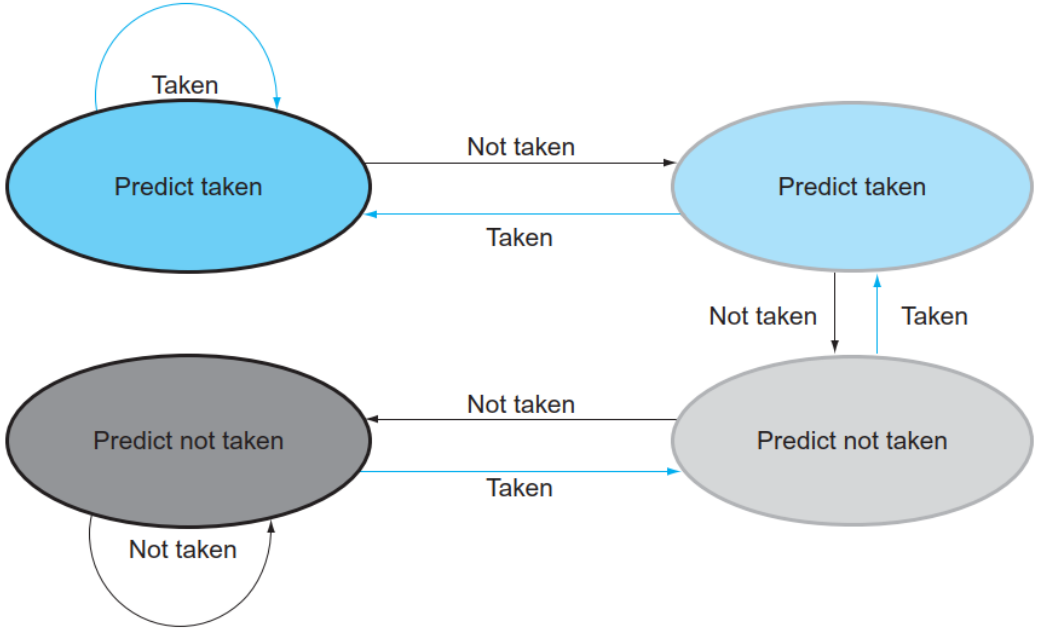
- Inner loop branches mispredicted twice!



  - Mispredict as taken on last iteration of inner loop
  - Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor

- Only change prediction on two successive mispredictions



Branch Target Buffer and Other Branch Predictors

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- **Branch target buffer**
  - Cache of target addresses
  - Indexed by PC when instruction fetched
  - If hit and instruction is branch predicted taken, can fetch target immediately

- **Correlating predictor**
  - A branch predictor that combines local behavior of a particular branch and global information about the behavior of some recent number of executed branches.

- **Tournament branch predictor**
  - A branch predictor with multiple predictions for each branch and a selection mechanism that chooses which predictor to enable for a given branch.

Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, …
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard

Type of event	From where?	ARMv8 terminology
System reset	External	Exception
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Floating-point arithmetic overflow or underflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

Handling Exceptions

- Save PC of offending (or interrupted) instruction
  - In LEGv8: **Exception Link Register** (ELR) 64bit

- Save indication of the problem
  - In LEGv8: **Exception Syndrome Register** (ESR) 32bit, e.g.:
    - 8 representing an undefined instruction,
    - 10 representing arithmetic overflow or underflow, and
    - 12 representing hardware malfunction.

An Alternate Mechanism

- Vectored Interrupts
  - The handler address we jump to is determined by the cause
- Exception vector address to be added to a vector table base register:
  - Unknown Reason: 00 0000two
  - Overflow: 10 1100two
  - …: 11 1111two
- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler

Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use ELR to return to program
- Otherwise
  - Terminate program
  - Report error using ELR, cause, …

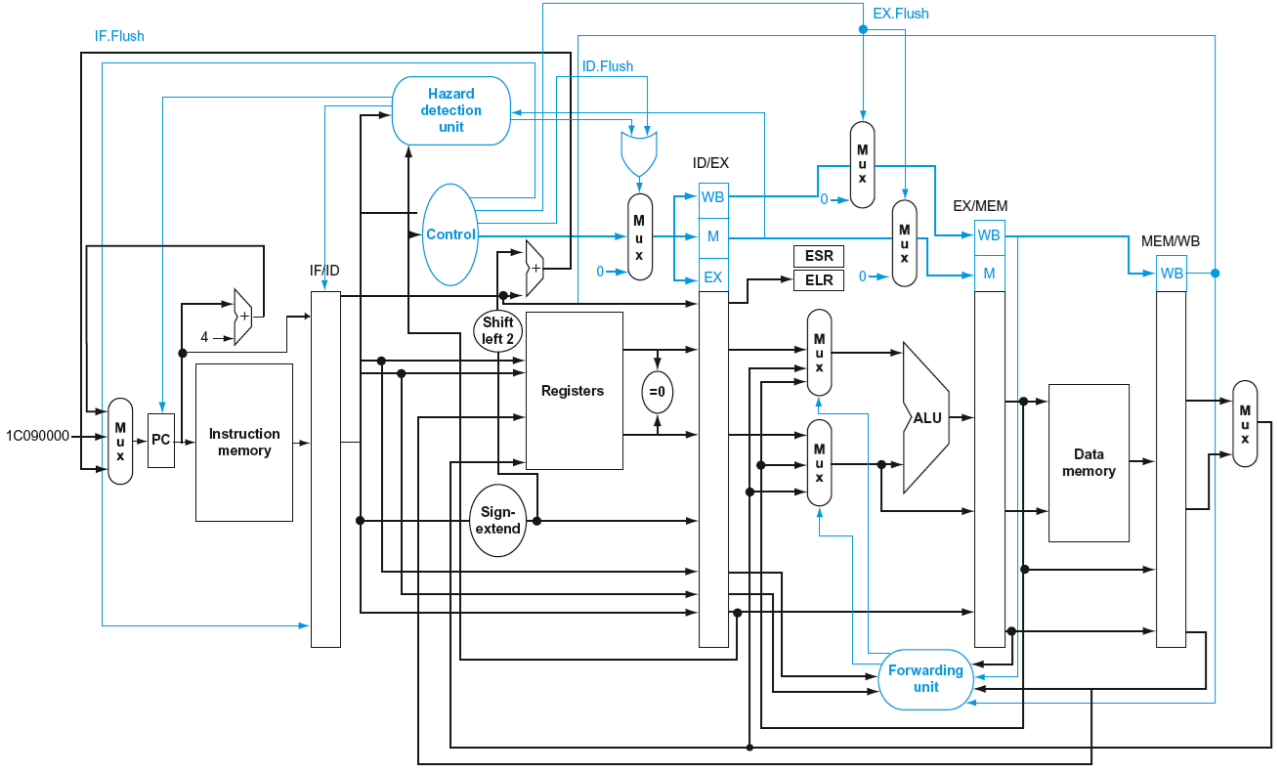
Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow or hardware mulfunction on add in EX stage

  `ADD X1, X2, X1`

  - Prevent X1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set ESR and ELR register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

Pipeline with Exceptions



Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
  - Refetched and executed from scratch
- PC saved in ELR register
  - Identifies causing instruction
  - Actually PC + 4 is saved
  - Handler must adjust

Exception Example

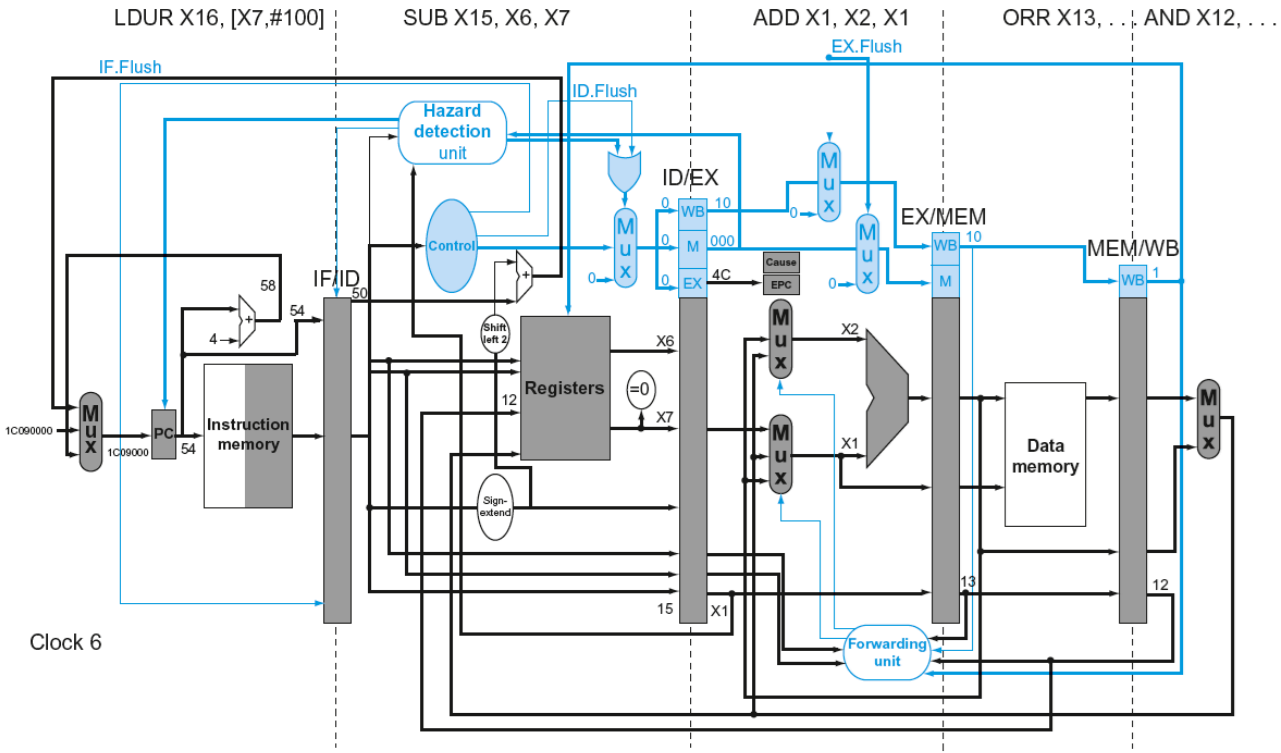
- Exception on ADD in

```
40        SUB  X11, X2, X4
44        AND  X12, X2, X5
48        ORR  X13, X2, X6
4C        ADD  X1,  X2, X1
50        SUB  X15, X6, X7
54        LDUR X16, [X7,#100]
…
```

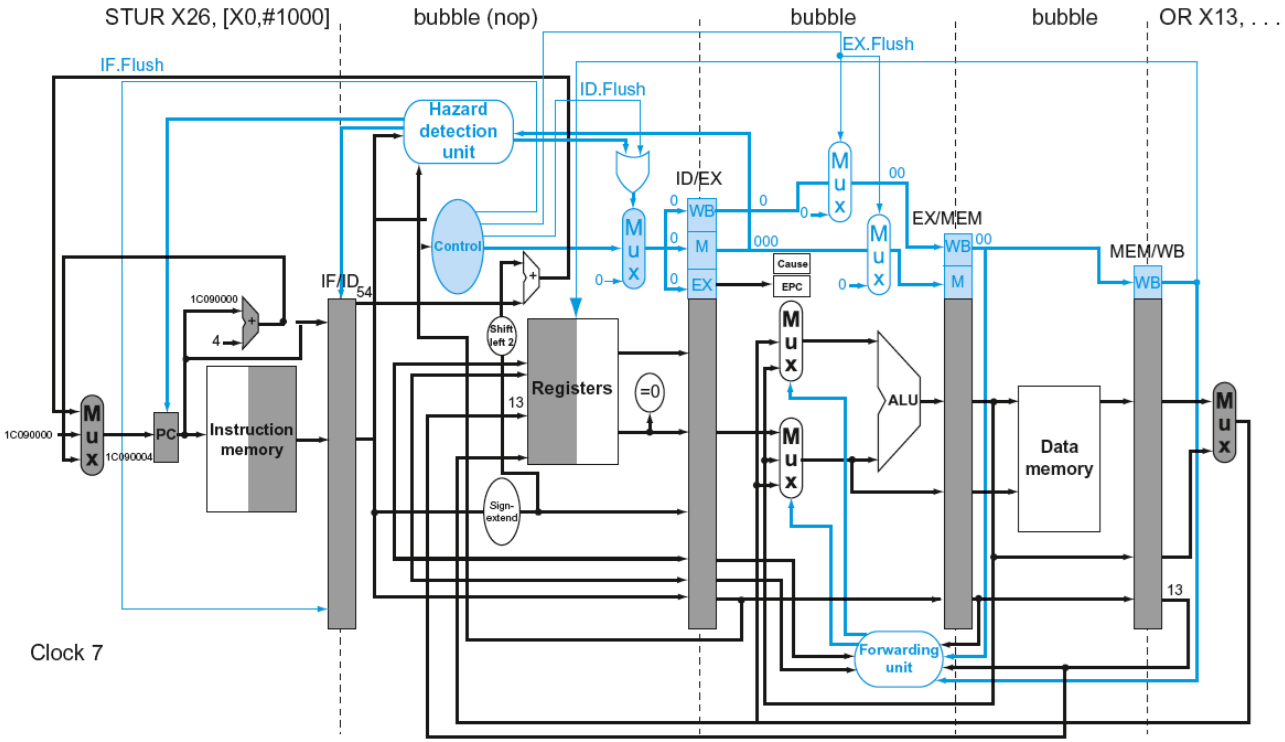
- Handler

```
80000180  STUR X26, [X0,#1000]
80000184 STUR X27, [X0,#1008]
…
```

Exception Example



Exception Example



Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - “Precise” exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!

Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require “manual” completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

HW/SW interface

- The hardware and the operating system must work in conjunction so that exceptions behave as you would expect.
- The hardware contract is normally to
  - stop the offending instruction in midstream,
  - let all prior instructions complete,
  - flush all following instructions,
  - set a register to show the cause of the exception,
  - save the address of the offending instruction,
  - and then branch to a prearranged address.
- The operating system contract is to look at the cause of the exception and act appropriately.
  - For an undefined instruction or hardware failure, the operating system normally kills the program and returns an indicator of the reason.
  - For an I/O device request or an operating system service call, the operating system saves the state of the program, performs the desired task, and, at some point in the future, restores the program to continue execution.

Parallelism via Instructions

- Pipelining exploits the potential parallelism among instructions.
- This parallelism is called **instruction-level parallelism** (ILP).
- There are two primary methods for increasing the potential amount of instruction-level parallelism:
  1. **Increase** the **depth** of the pipeline to overlap more instructions.
     - Performance is potentially greater since the clock cycle can be shorter.
  2. **Replicate the internal components** of the computer so that it can launch **multiple instructions** in every pipeline stage.
     - The general name for this technique is **multiple issue**.
     - It allows the instruction execution rate to exceed the clock rate or the CPI to be less than 1.
     - E.g., 4GHz 4-way multiple-issue
       - 16 BIPS, peak CPI = 0.25, peak IPC = 4
     - But dependencies reduce this in practice

Multiple Issue

- There are two main ways to implement a multiple-issue processor, with the major difference being the division of work between the compiler and the hardware:
  - **Static multiple issue**
    - An approach to implementing a multiple-issue processor where many decisions are made by the compiler before execution.
  - **Dynamic multiple issue**
    - An approach to implementing a multiple-issue processor where many decisions are made during execution by the processor.
- Two primary and distinct responsibilities must be dealt with in a multiple-issue pipeline:
  1. Packaging instructions into issue slots.
     - How does the processor determine how many instructions and which instructions can be issued in a given clock cycle?
  2. Dealing with data and control hazards.

The Concept of Speculation

- **Speculation** is an approach that allows the compiler or the processor to “guess” what to do with an instruction, to start execution of other instructions that may depend on the speculated instruction.
- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated
- The difficulty with **speculation** is that it **may be wrong**.
  - Any speculation mechanism must include both a method to check if the guess was right and a method to unroll or back out the effects of the instructions that were executed speculatively.
- Speculation may be done in the compiler or by the hardware.
  - Common to static and dynamic multiple issue

Compiler/Hardware Speculation

- Compiler can reorder instructions
  - e.g., move an instruction across a branch or a load across a store.
  - Can include “fix-up” instructions to recover from incorrect guess.
- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed.
  - Flush buffers on incorrect speculation and re-execute the correct instruction sequence.

Static Multiple Issue

- Static multiple-issue processors use the compiler to assist with packaging instructions and handling hazards.
- In a static issue processor, you can think of the set of instructions issued in a given clock cycle, the **issue packet**, as *one large instruction* with multiple operations.
- Since a static multiple-issue *processor usually restricts* what *mix of instructions* can be initiated in a given clock cycle, it is useful to think of the issue packet as a single instruction allowing several operations in certain predefined fields.
- This view led to the original name for this approach: ***Very Long Instruction Word (VLIW)***.

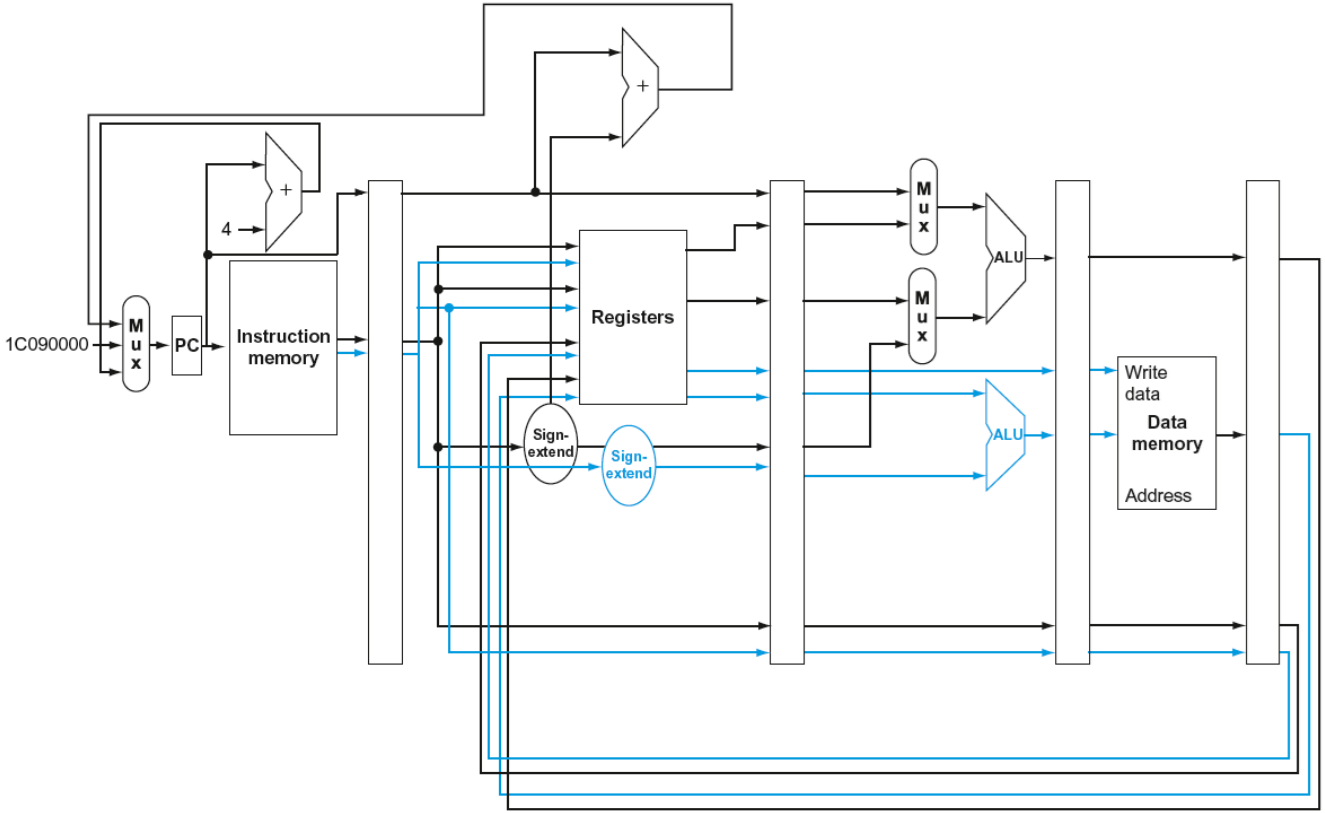
- The compiler must remove some/all **hazards**
  - Reorder instructions into issue packets
  - No dependencies within a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary

LEGv8 with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

LEGv8 with Static Dual Issue



LEGv8 with Static Dual Issue

- Clearly, this two-issue processor can improve performance by up to a factor of two!
- Doing so, however, requires that twice as many instructions be overlapped in execution, and this additional overlap increases the relative performance loss from data and control hazards.
- For example,
  - Now we can't use ALU result in load/store in same packet

```
ADD  X0, X0, X1
LDUR X2, [X0,#0]
```

  - We have to split into two packets (effectively a stall).

  - In our five-stage pipeline, loads have a **use latency** of one clock cycle, which prevents one instruction from using the result without stalling.
  - In the two-issue, five-stage pipeline the result of a load instruction cannot be used on the next clock cycle. This means that the *next two instructions* cannot use the load result without stalling.

- To effectively exploit the parallelism, a more aggressive scheduling and a more ambitious compiler is needed.

Scheduling Example

- How would this loop be scheduled on a static two-issue pipeline for LEGv8?

```
Loop: LDUR X0, [X20,#0]  // X0=array element
      ADD  X0,X0,X21     // add scalar in X21
      STUR X0, [X20,#0]  // store result
      SUBI X20,X20,#8     // decrement pointer
      CMP  X20,X22       // compare to loop limit
      BGT  Loop          // branch if X20 > X22
```

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		LDUR X0,[X20,#0]	1
	SUBI X20, X20, #8		2
	ADD X0, X0, X21		3
	CMP X20, X22		4
	BGT Loop	STUR X0,[X20,#8]	5

IPC = 6/5 = 1.2 (c.f. peak IPC = 2)

Loop Unrolling for Multiple-Issue Pipelines

- An important compiler technique to get more performance from loops is **loop unrolling**, where multiple copies of the loop body are made.
- After unrolling, there is more ILP available by overlapping instructions from different iterations.
- During the unrolling process, the compiler generally introduces additional registers.
- The goal of this process, called **register renaming**, is to eliminate dependences that are not true data dependences, the so called **antidependences** or **name dependences.**

- **Antidependence**, also called name dependence, is an ordering forced by the reuse of a name, typically a register, rather than by a true dependence that carries a value between two instructions.

Loop Unrolling Example

- Loop unrolling with factor 4 of the previous loop:

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:	SUBI X20, X20,#32	LDUR X0, [X20,#0]	1
		LDUR X1, [X20,#24]	2
	ADD X0, X0, X21	LDUR X2, [X20,#16]	3
	ADD X1, X1, X21	LDUR X3, [X20,#8]	4
	ADD X2, X2, X21	STUR X0, [X20,#32]	5
	ADD X3, X3, X21	STUR X1, [X20,#24]	6
	CMP X20, X22	STUR X2, [X20,#16]	7
	BGT Loop	STUR X3, [X20,#8]	8

IPC = 15/8 = 1.875

Closer to 2, but at cost of registers and code size

Dynamic Multiple Issue

- Dynamic multiple-issue processors are also known as **superscalar** processors.
- In the **simplest** superscalar processors, **instructions issue in order**, and the processor decides whether zero, one, or more instructions can issue in a given clock cycle.
    - Achieving good performance on such a processor still requires the compiler to try to schedule instructions to move dependences apart and thereby improve the instruction issue rate.
    - Big difference with VLIW: now the code, whether scheduled or not, is guaranteed by the hardware to execute correctly.
    - Compiled code will always run correctly independent of the issue rate or pipeline structure of the processor.

Dynamic Pipeline Scheduling

- Many superscalars extend the multiple issue framework to include **dynamic pipeline scheduling**.
  - Hardware allow the CPU to execute instructions *out of order* to avoid stalls.
    - But commit result to registers in order
- Example

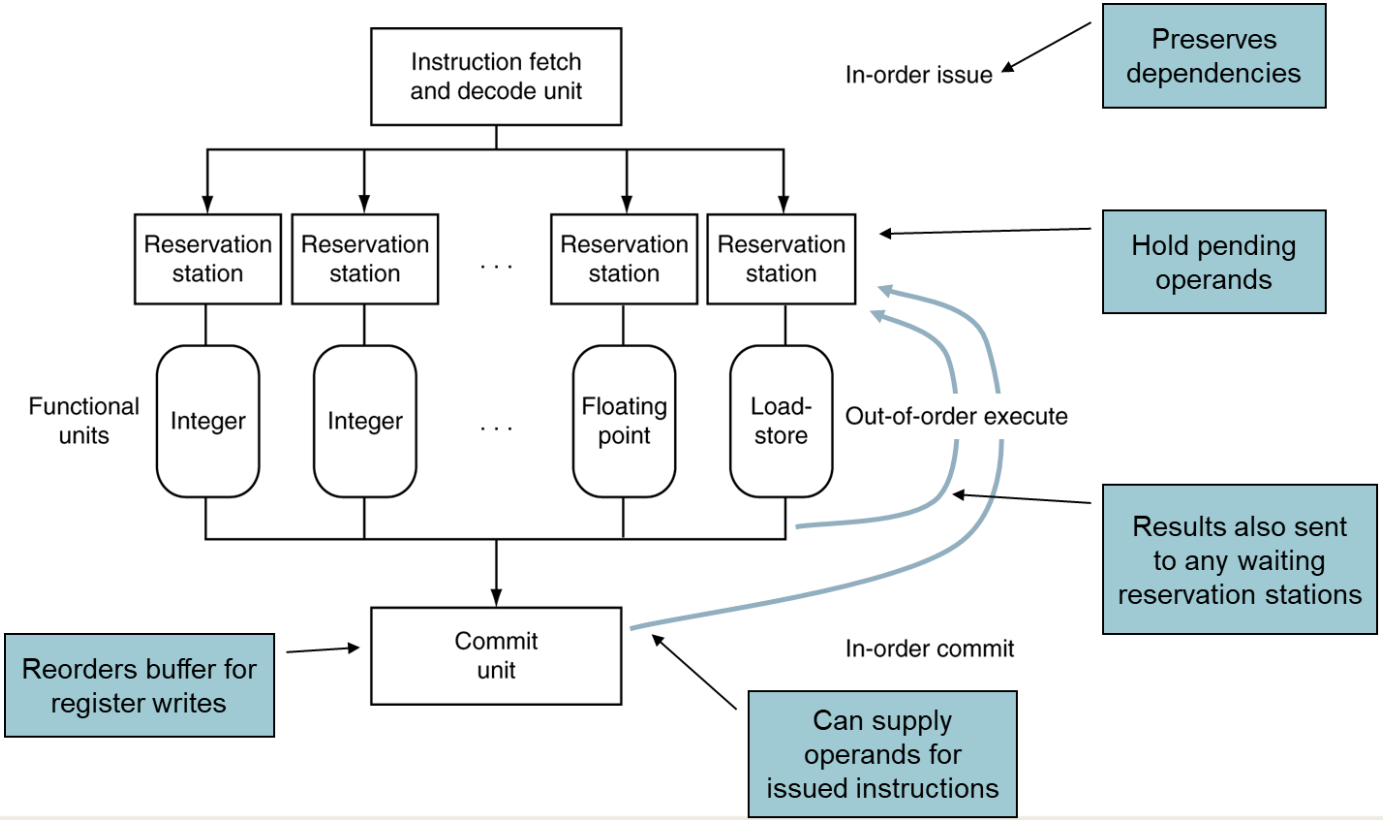
```
LDUR X0, [X21,#20]
ADD  X1, X0, X2
SUB  X23,X23,X3
ANDI X5, X23,#20
```

  - Even though the SUB instruction is ready to execute, it must wait for the LDUR and ADD to complete first, which might take many clock cycles if memory is slow.
  - Dynamic pipeline scheduling allows such hazards to be avoided.

Dynamic Pipeline Scheduling

- Dynamic pipeline scheduling chooses which instructions to execute next, possibly reordering them to avoid stalls.
- In such processors, the pipeline is divided into **three major units**:
  - an **instruction fetch and issue unit**,
  - **multiple functional units** (a dozen or more in high-end designs in 2015), and
  - a **commit unit**.
- The first unit fetches instructions, decodes them, and sends each instruction to a corresponding functional unit for execution.
- Each functional unit has buffers, called **reservation stations**, which hold the operands and the operation.
- As soon as the buffer contains all operands and the functional unit is ready to execute, the result is calculated.
- When the result is completed, it is sent to any reservation stations waiting for it and to the commit unit, which buffers the result until it is safe to put it into the register file or into memory. Its buffer is often called **reordering buffer** and is also used for forwarding operands.

Dynamically Scheduled CPU



Register Renaming

- Reservation stations and reorder buffer effectively provide register renaming
- On instruction issue to reservation station
  - If operand is available in register file or reorder buffer
    - Copied to reservation station
    - No longer required in the register; can be overwritten
  - If operand is not yet available
    - It will be provided to the reservation station by a function unit
    - Register update may not be required

Out-of-order Execution

- Conceptually, you can think of a dynamically scheduled pipeline as analyzing the data flow structure of a program.
- The processor then executes the instructions in some order that preserves the data flow order of the program.
- This style of execution is called an **out-of-order execution**, since the instructions can be executed in a different order than they were fetched.
- To make programs behave as if they were running on a simple in-order pipeline:
  - the instruction fetch and decode unit is required to **issue** instructions **in order**,
    - which allows dependences to be tracked,
  - and the commit unit is required to write results to registers and memory in program fetch order.
    - This conservative mode is called **in-order commit**.

Speculation

- Dynamic scheduling is often extended by including hardware-based speculation, especially for branches.
  - Because the instructions are committed in order, we know whether the branch was correctly predicted before any instructions from the predicted path are committed
- A speculative, dynamically scheduled pipeline can also support speculation on load addresses,
  - allowing load-store reordering, and
  - using the commit unit to avoid incorrect speculation.

Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?

- Not all stalls are predicable
  - e.g., cache misses
- Can’t always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

Does Multiple Issue Work?

- Yes, but not as much as we'd like.

- Both pipelining and multiple-issue execution increase peak instruction throughput and attempt to exploit instruction-level parallelism (ILP).
- Data and control dependences in programs, however, offer an upper limit on sustained performance because the processor must sometimes wait for a dependence to be resolved.
- Software-centric approaches to exploiting ILP rely on the ability of the compiler to find and reduce the effects of such dependences, while hardware-centric approaches rely on extensions to the pipeline and issue mechanisms.
- Speculation, performed by the compiler or the hardware, can increase the amount of ILP that can be exploited via prediction, although care must be taken since speculating incorrectly is likely to reduce performance.

Energy Efficiency and Advanced Pipelining

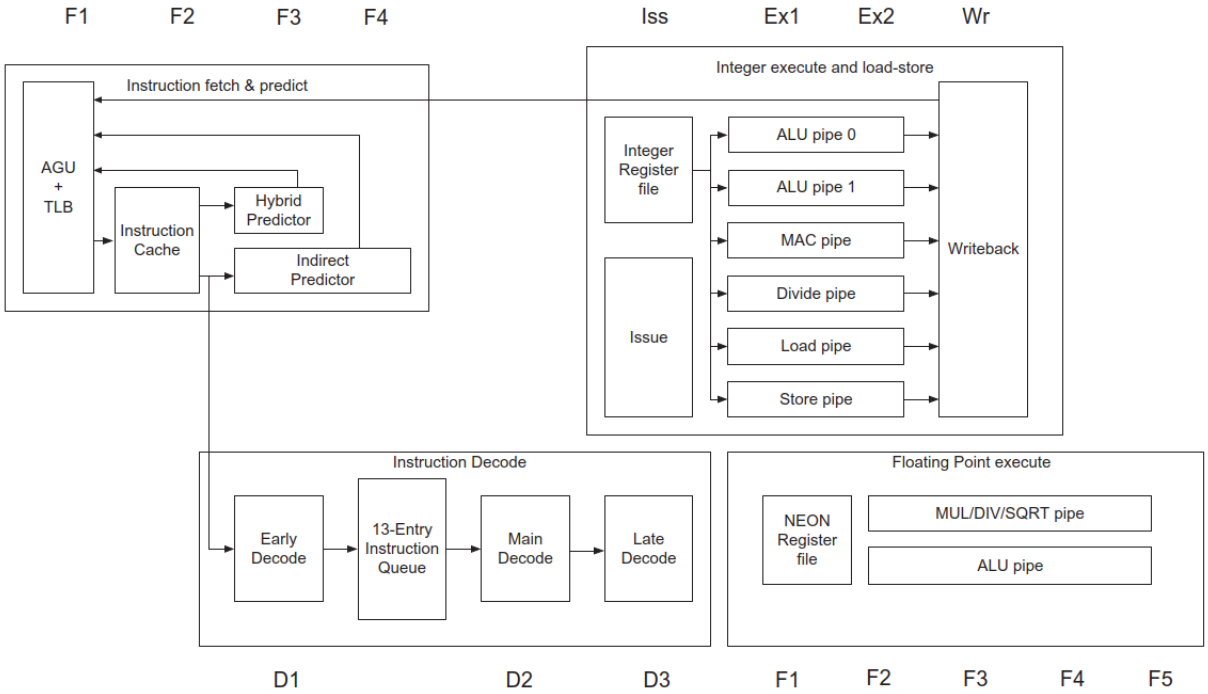
- The downside to the increasing exploitation of instruction-level parallelism via dynamic multiple issue and speculation is potential energy inefficiency.
- The current belief is that while the simpler processors are not as fast as their sophisticated brethren, they deliver better performance per Joule, so that they can deliver more performance per chip when designs are constrained more by energy than they are by the number of transistors.

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power	
Intel 486	1989	25 MHz	5	1	No	1	5	W
Intel Pentium	1993	66 MHz	5	2	No	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103	W
Intel Core	2006	2930 MHz	14	4	Yes	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	2–4	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77	W

Real Stuff: The ARM Cortex-A53 and Intel Core i7

Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, Cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple Issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline Stages	8	14
Pipeline schedule	Static In-order	Dynamic Out-of-order with Speculation
Branch prediction	Hybrid	2-level
1st level caches/core	16-64 KiB I, 16-64 KiB D	32 KiB I, 32 KiB D
2nd level cache/core	128–2048 KiB (shared)	256 KiB (per core)
3rd level cache (shared)	(platform dependent)	2–8 MiB

ARM Cortex-A53 pipeline



ARM Cortex-A53 pipeline

Key features of the microarchitecture of the Cortex-A53 core -2

- The Cortex-A53 core has a dual-issue in-order front end with 5 pipelines constituting the back end, as indicated in the next Figure.

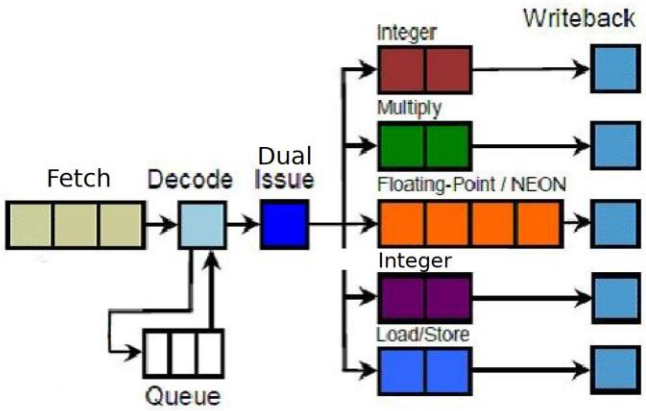


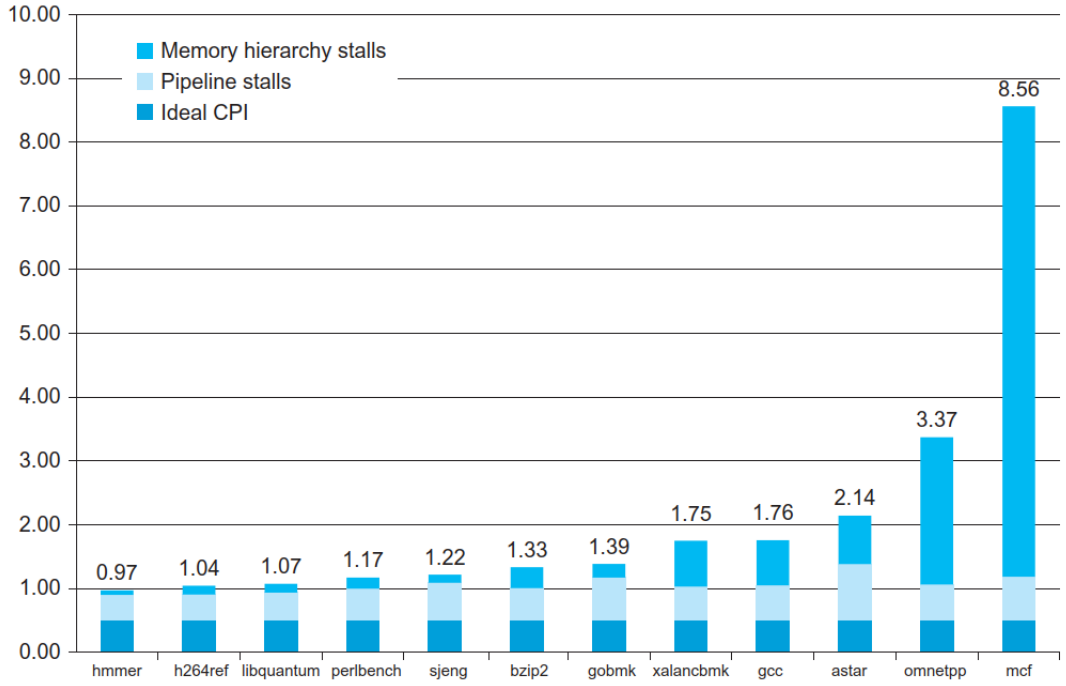
Figure: Pipeline stages of the Cortex-A53 [76]

- The pipeline for integer processing has 8 pipeline stages, NEON and FP processing has two additional pipeline stages, as seen in the Figure above.

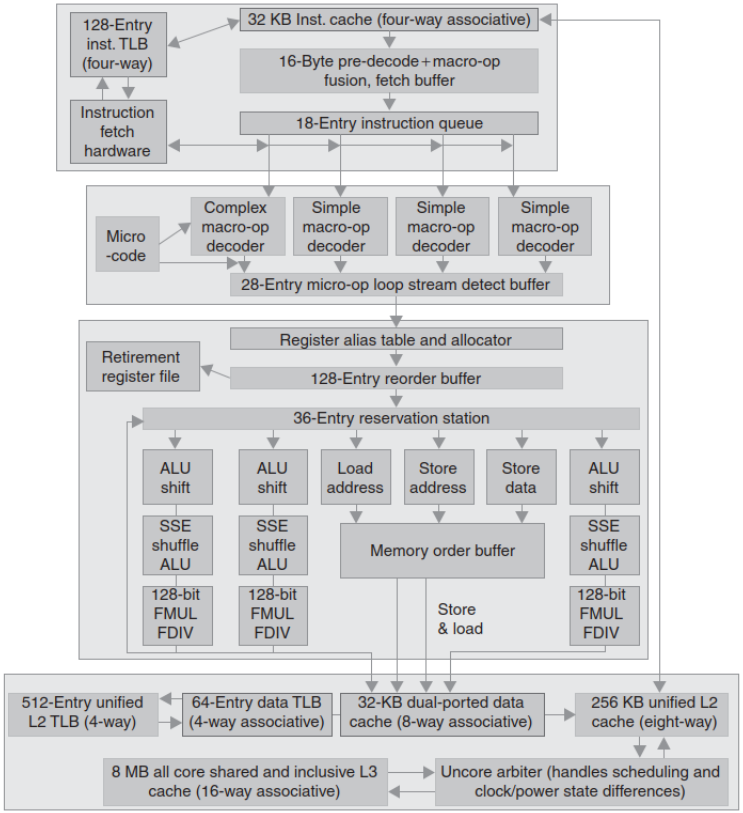
ARM Cortex-A53 pipeline

- The first three stages fetch two instructions at a time and try to keep a 13-entry instruction queue full. It uses a 6k-bit hybrid conditional branch predictor, a 256-entry indirect branch predictor, and an 8-entry return address stack to predict future function returns. The prediction of indirect branches takes an additional pipeline stage. When the branch prediction is wrong, it empties the pipeline, resulting in an **eight-clock cycle misprediction penalty**.
- The decode stages of the pipeline determine if there are dependences between a pair of instructions, which would force sequential execution, and in which pipeline of the execution stages to send the instructions.
- The instruction execution section primarily occupies three pipeline stages and provides one pipeline for load instructions, one pipeline for store instructions, two pipelines for integer arithmetic operations, and separate pipelines for integer multiply and divide operations. Either instruction from the pair can be issued to the load or store pipelines. The execution stages have full forwarding between the pipelines.
- Floating-point and SIMD operations add a two more pipeline stages to the instruction execution section and feature one pipeline for multiply/divide/square root operations and one pipeline for other arithmetic operations.

ARM Cortex-A53 performance



Intel Core i7 920 pipeline



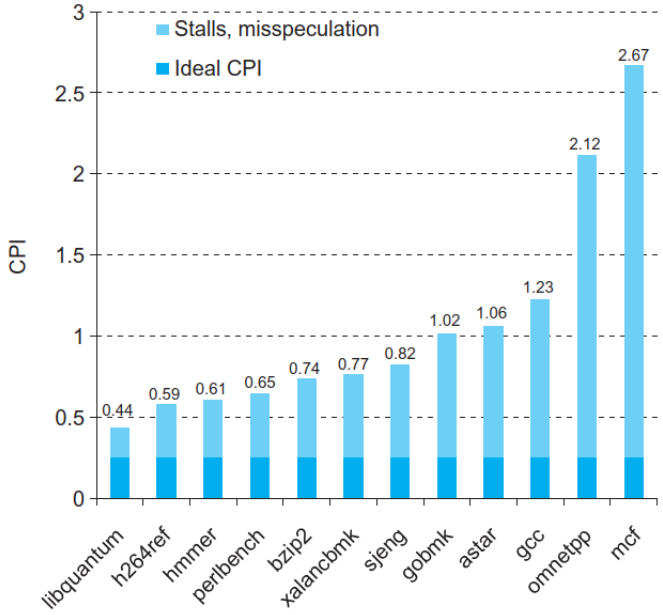
Intel Core i7 920 pipeline

1. Instruction fetch—The processor uses a multilevel branch target buffer to achieve a balance between speed and prediction accuracy. There is also a return address stack to speed up function return. Mispredictions cause a penalty of about 15 cycles. Using the predicted address, the instruction fetch unit fetches 16 bytes from the instruction cache.
2. The 16 bytes are placed in the predecode instruction buffer—The predecode stage transforms the 16 bytes into individual x86 instructions. This predecode is nontrivial since the length of an x86 instruction can be from 1 to 15 bytes and the predecoder must look through a number of bytes before it knows the instruction length. Individual x86 instructions are placed into the 18-entry instruction queue.
3. Micro-op decode—Individual x86 instructions are translated into micro-operations (micro-ops). Three of the decoders handle x86 instructions that translate directly into one micro-op. For x86 instructions that have more complex semantics, there is a microcode engine that is used to produce the micro-op sequence; it can produce up to four micro-ops every cycle and continues until the necessary micro-op sequence has been generated. The micro-ops are placed according to the order of the x86 instructions in the 28-entry micro-op buffer.

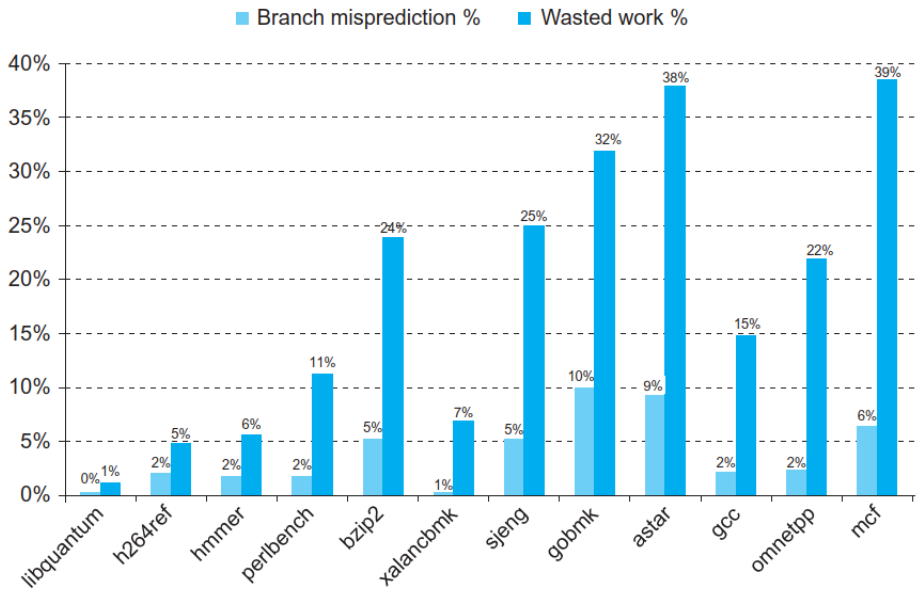
Intel Core i7 920 pipeline

4. The micro-op buffer performs *loop stream detection*—If there is a small sequence of instructions (less than 28 instructions or 256 bytes in length) that comprises a loop, the loop stream detector will find the loop and directly issue the micro-ops from the buffer, eliminating the need for the instruction fetch and instruction decode stages to be activated.
5. Perform the basic instruction issue—Looking up the register location in the register tables, renaming the registers, allocating a reorder buffer entry, and fetching any results from the registers or reorder buffer before sending the micro-ops to the reservation stations.
6. The i7 uses a 36-entry centralized reservation station shared by six functional units. Up to six micro-ops may be dispatched to the functional units every clock cycle.
7. The individual function units execute micro-ops and then results are sent back to any waiting reservation station as well as to the register retirement unit, where they will update the register state, once it is known that the instruction is no longer speculative. The entry corresponding to the instruction in the reorder buffer is marked as complete.
8. When one or more instructions at the head of the reorder buffer have been marked as complete, the pending writes in the register retirement unit are executed, and the instructions are removed from the reorder buffer.

Intel Core i7 920 performance



Intel Core i7 920 performance



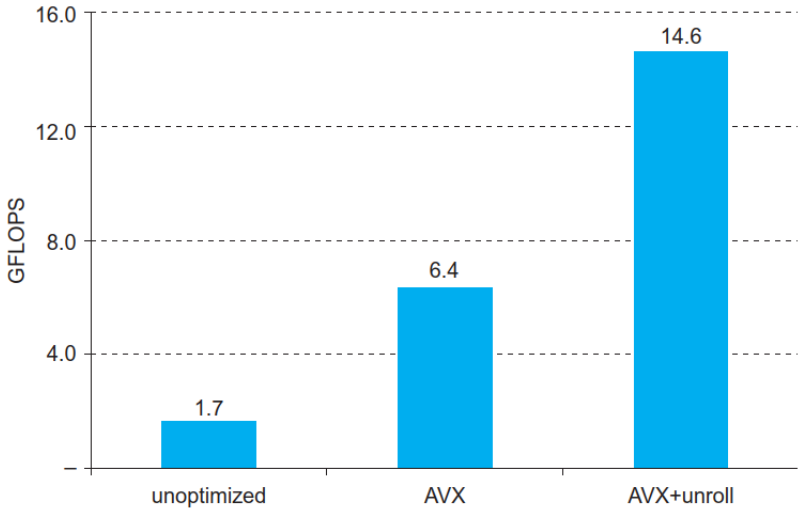
Going Faster: Instruction-Level Parallelism and Matrix Multiply

```
1 //include <x86intrin.h>
2 //define UNROLL (4)
3
4 void dgemm (int n, double* A, double* B, double* C)
5 {
6     for ( int i = 0; i < n; i+= UNROLL*4 )
7         for ( int j = 0; j < n; j++ ) {
8             __m256d c[4];
9             for ( int x = 0; x < UNROLL; x++ )
10                c[x] = _mm256_load_pd(C+i+x*4+j*n);
11
12            for( int k = 0; k < n; k++ )
13            {
14                __m256d b = _mm256_broadcast_sd(B+k+j*n);
15                for (int x = 0; x < UNROLL; x++)
16                    c[x] = _mm256_add_pd(c[x],
17                    _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
18            }
19
20            for ( int x = 0; x < UNROLL; x++ )
21                _mm256_store_pd(C+i+x*4+j*n, c[x]);
22      }
23 }
```

Matrix Multiply

```
1 vmovapd (%r11),%ymm4                  // Load 4 elements of C into %ymm4
2 mov %rbx,%rax                         // register %rax = %rbx
3 xor %ecx,%ecx                         // register %ecx = 0
4 vmovapd 0x20(%r11),%ymm3              // Load 4 elements of C into %ymm3
5 vmovapd 0x40(%r11),%ymm2              // Load 4 elements of C into %ymm2
6 vmovapd 0x60(%r11),%ymm1              // Load 4 elements of C into %ymm1
7 vbroadcastsd (%rcx,%r9,1),%ymm0       // Make 4 copies of B element
8 add $0x8,%rcx                         // register %rcx = %rcx + 8
9 vmulpd (%rax),%ymm0,%ymm5             // Parallel mul %ymm1,4 A
10 vaddpd %ymm5,%ymm4,%ymm4             // Parallel add %ymm5, %ymm4
11 vmulpd 0x20(%rax),%ymm0,%ymm5        // Parallel mul %ymm1,4 A
12 vaddpd %ymm5,%ymm3,%ymm3             // Parallel add %ymm5, %ymm3
13 vmulpd 0x40(%rax),%ymm0,%ymm5        // Parallel mul %ymm1,4 A
14 vmulpd 0x60(%rax),%ymm0,%ymm0        // Parallel mul %ymm1,4 A
15 add %r8,%rax                         // register %rax = %rax + %r8
16 cmp %r10,%rcx                        // compare %r8 to %rax
17 vaddpd %ymm5,%ymm2,%ymm2             // Parallel add %ymm5, %ymm2
18 vaddpd %ymm0,%ymm1,%ymm1             // Parallel add %ymm0, %ymm1
19 jne 68 <dgemm+0x68>                  // branch if %r8 != %rax
20 add $0x1,%esi                        // register % esi = % esi + 1
21 vmovapd %ymm4,(%r11)                 // Store %ymm4 into 4 C elements
22 vmovapd %ymm3,0x20(%r11)             // Store %ymm3 into 4 C elements
23 vmovapd %ymm2,0x40(%r11)             // Store %ymm2 into 4 C elements
24 vmovapd %ymm1,0x60(%r11)             // Store %ymm1 into 4 C elements
```

Matrix Multiply performance



References

- David A. Patterson and John L. Hennessy, “Computer organization and design ARM edition: the hardware software interface,” Morgan Kaufmann, 2016.
  - Chapter 4: 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 4.10, 4.11, 4.12

- Simone Buso, “ Introduzione alle applicazioni industriali di Microcontrollori e DSP”, Società editrice Esculapio, 2018
  - Chapter 5.3

Most of the text has been taken and adapted from “Computer Organization and Design ARM Edition: The Hardware Software Interface”.
If not differently indicated, all figures have been taken from the book or the material in the companion website of “Computer Organization and Design ARM Edition: The Hardware Software Interface”.

NOTA BENE:

Esercitazioni MCU per corso ASD

Gli studenti sono pregati di scaricare le slide che troveranno alla pagina

http://www2.units.it/carrato/didatt/ASD_web/index.html