01076114
องค์ประกอบและสถาปัตยกรรมคอมพิวเตอร์
Computer Organization and Architecture

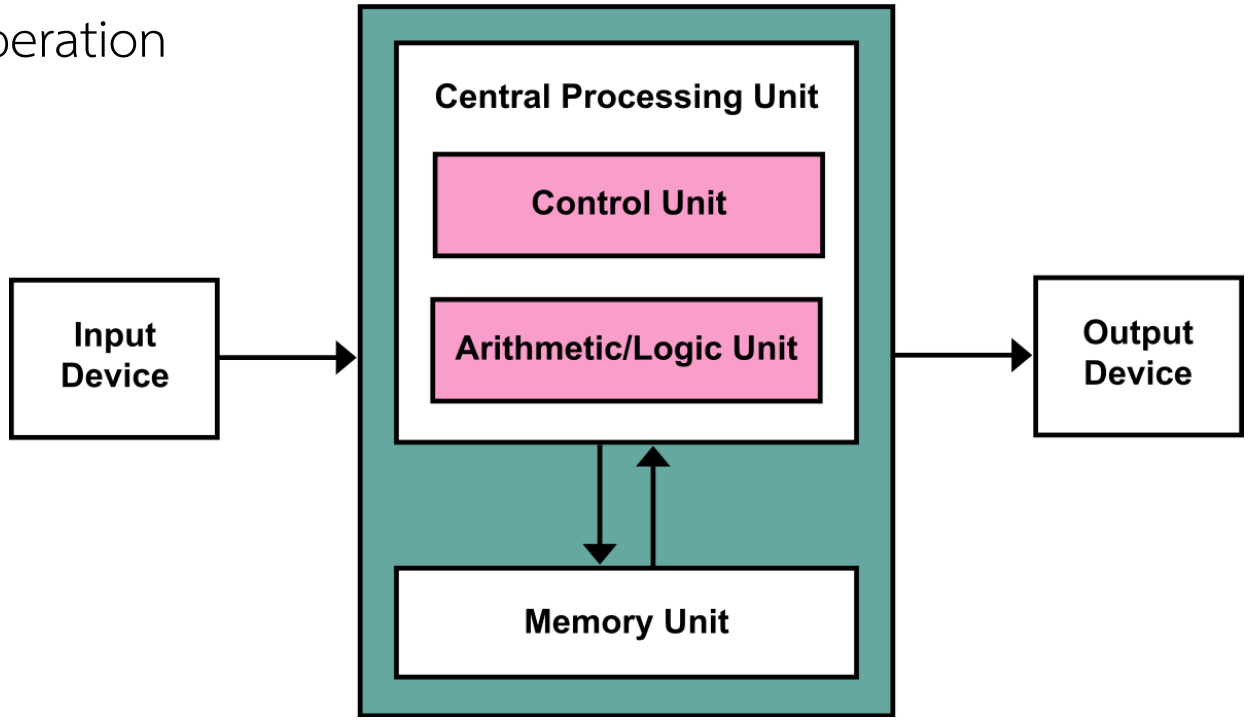
The Processor

Basic ARM Instruction

- ในบทนี้ (บทที่ 4) เราจะทำความเข้าใจกับการทำงานของ CPU, clock, state โดยแสดงการทำงานของ CPU อย่างง่ายที่มีการทำงานไม่มาก เช่น
  - การคำนวณพื้นฐาน เช่น ADD, SUB, AND, OR
  - การใช้งานหน่วยความจำ LDR, STR
  - Branch เช่น BEQ, B

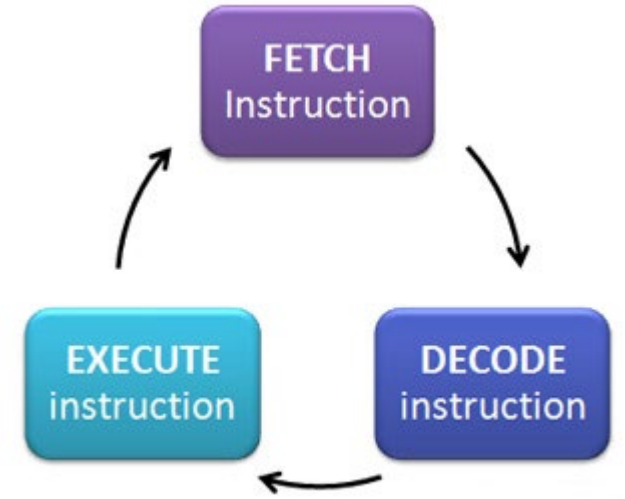
Implementation Overview

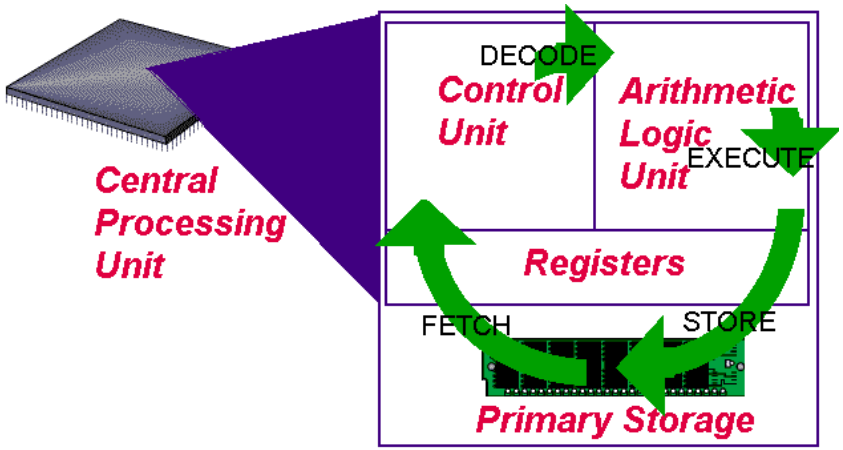
- องค์ประกอบพื้นฐานที่ CPU ต้องการ
  - Memory สำหรับเก็บคำสั่ง และ เก็บข้อมูล
  - Register สำหรับเก็บข้อมูลชั่วคราวระหว่างประมวลผล
  - ALU สำหรับทำ Operation
  - Control Logic



Implementation Overview

- โดยทั่วไป การทำงานของ CPU จะซ้ำเป็นรอบๆ
  - PC (Program Counter) จะชี้ตำแหน่งของคำสั่งในหน่วยความจำ จากนั้นอ่านคำสั่งเข้ามาใน Instruction Register ขั้นตอนนี้จะเรียกว่า Fetch
  - จากนั้นจะเป็นการตีความคำสั่งที่อยู่ใน Instruction Register ขั้นตอนนี้เรียกว่า Decode (คือการแยกส่วนคำสั่งแต่ละส่วนที่มีเป้าหมายการทำงาน)
  - สุดท้ายคือ การทำงานตามคำสั่ง เรียกว่า Execute

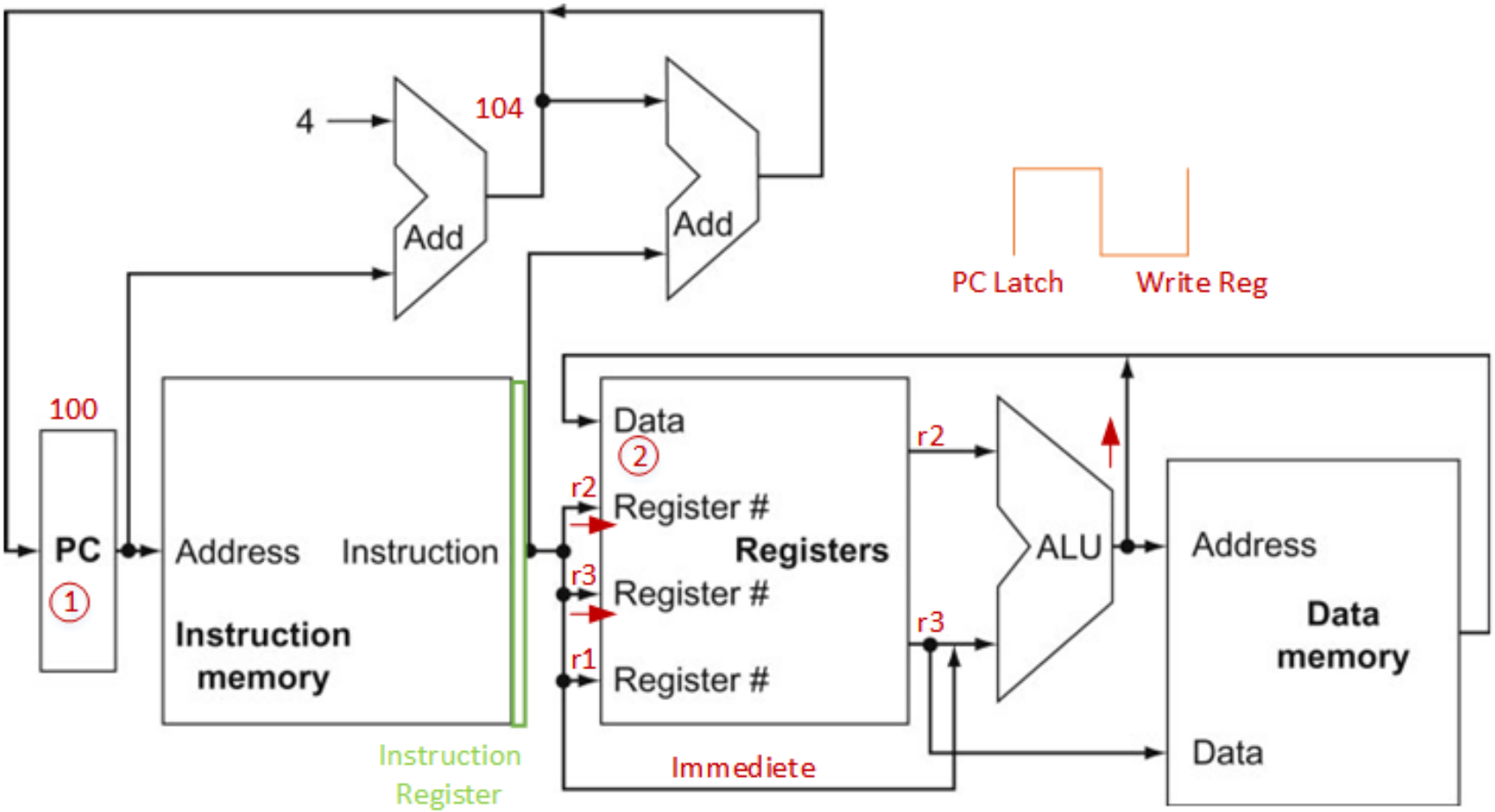




View from 30,000 Feet

- ADD r1,r2,r3

Cond	F	I	Opcode	S	Rn	Rd	Operand2
4 bits	2 bits	1 bits	4 bits	1 bits	4 bits	4 bits	12 bits

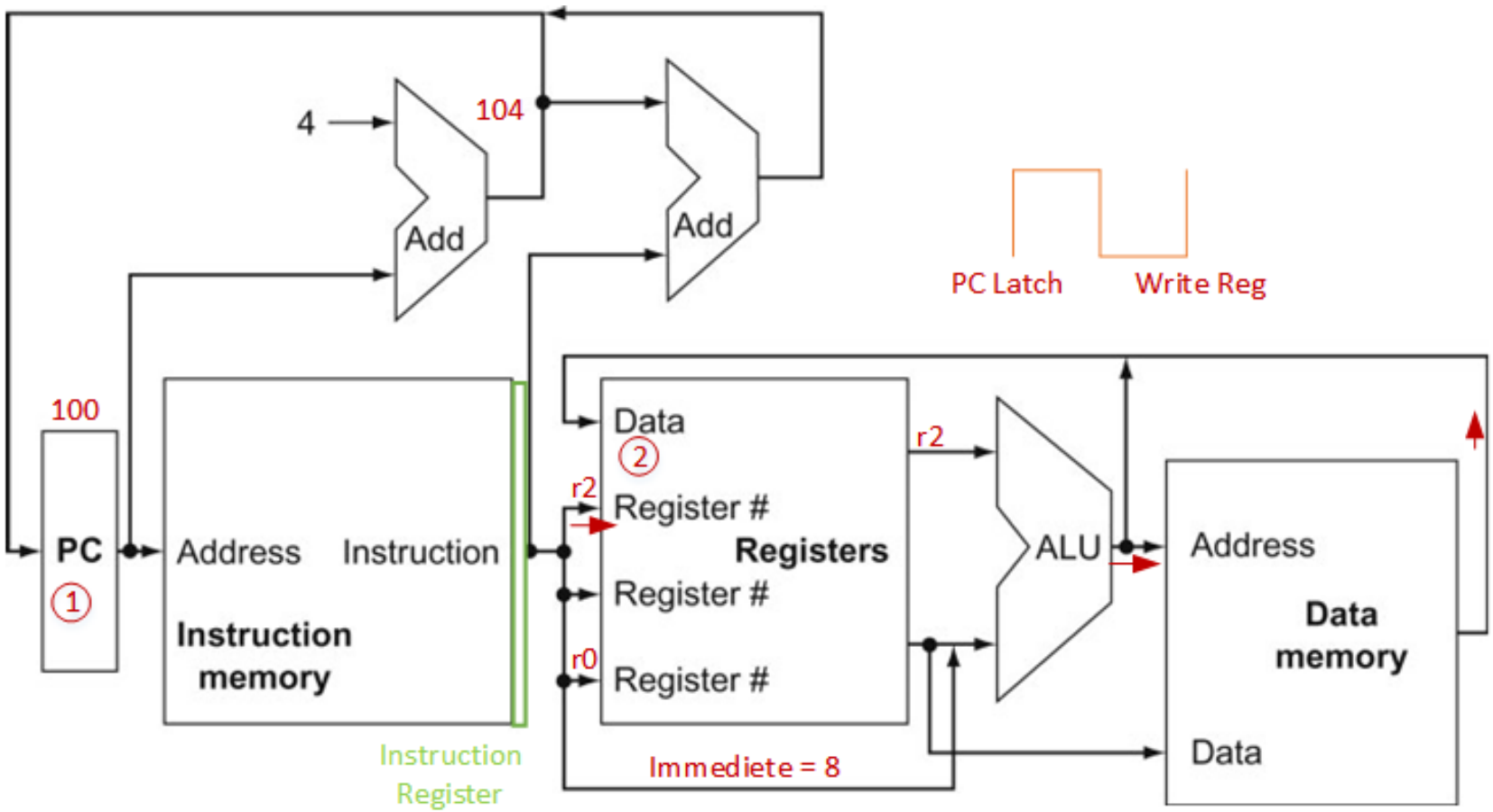


View from 30,000 Feet

- จากรูป PC จะให้ Address กับ Instruction Memory (หมายถึง หน่วยความจำที่เก็บคำสั่ง โดยจะเป็นที่เดียวกับ Data Memory หรือไม่ก็ได้)
- ข้อมูลในหน่วยความจำในตำแหน่งที่ PC ชี้ จะปล่อยออกมาและ Latch ไว้ที่ Instruction Register
- Instruction Register จะแบ่งข้อมูลในคำสั่งออกเป็นส่วนๆ ตาม Instruction Format และส่งไปตามส่วนต่างๆ เช่น Rn จะส่งไปยัง Register In 1
- เมื่อ register file ได้รับหมายเลข register ก็จะส่งข้อมูลใน register ออกทาง output ของ register file ซึ่งต่อกับ ALU หรือหน่วยความจำ
- เมื่อ ALU ได้รับข้อมูล และได้รับรหัสคำสั่ง ก็จะดำเนินการบวก และส่ง Output มาที่ทางออกของ ALU ซึ่งจะส่งเข้าไปเก็บใน register file อีกที

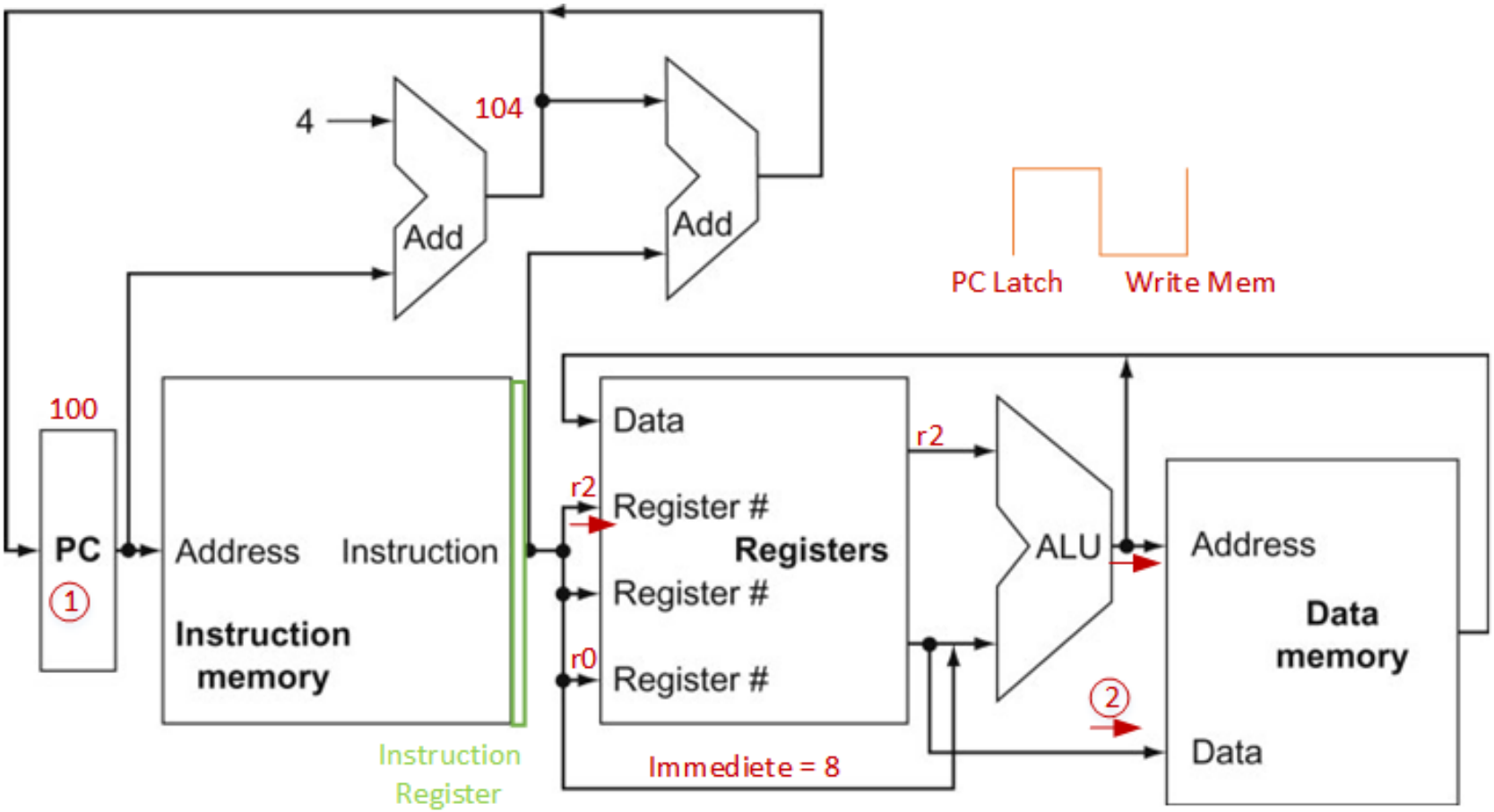
View from 30,000 Feet

- LDR r0, [r2,#8]



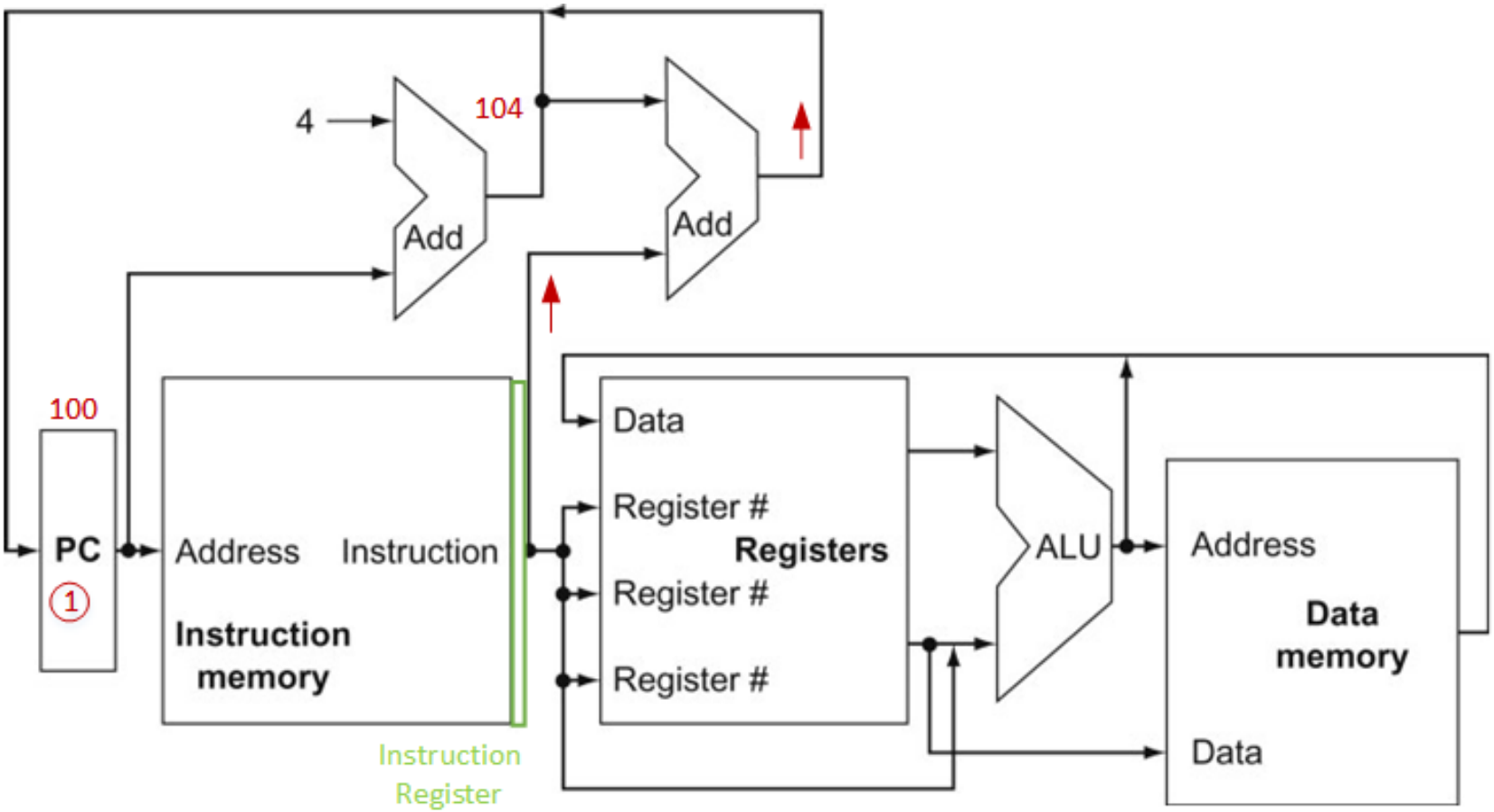
View from 30,000 Feet

- STR r0, [r2,#8]



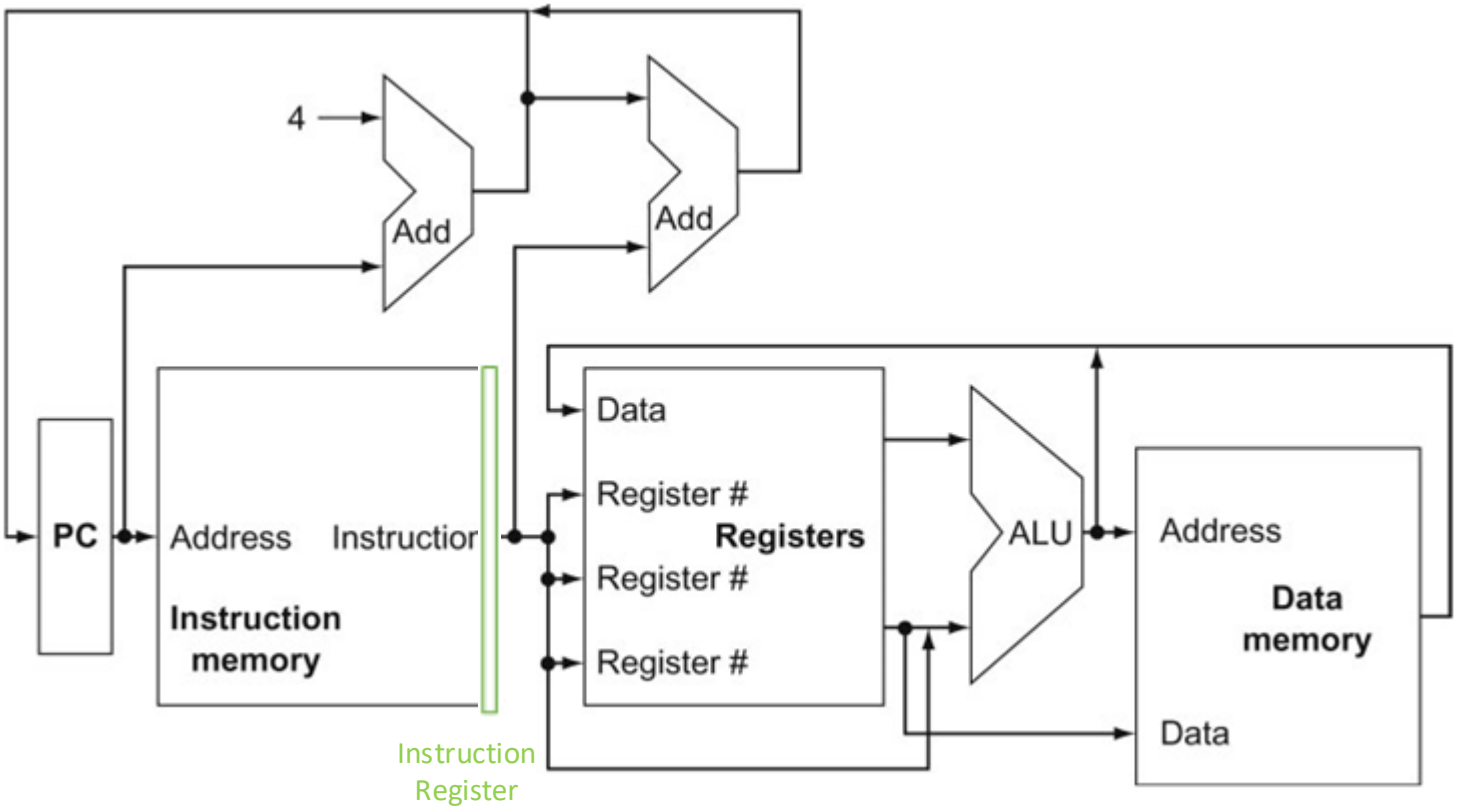
View from 30,000 Feet

- B loop

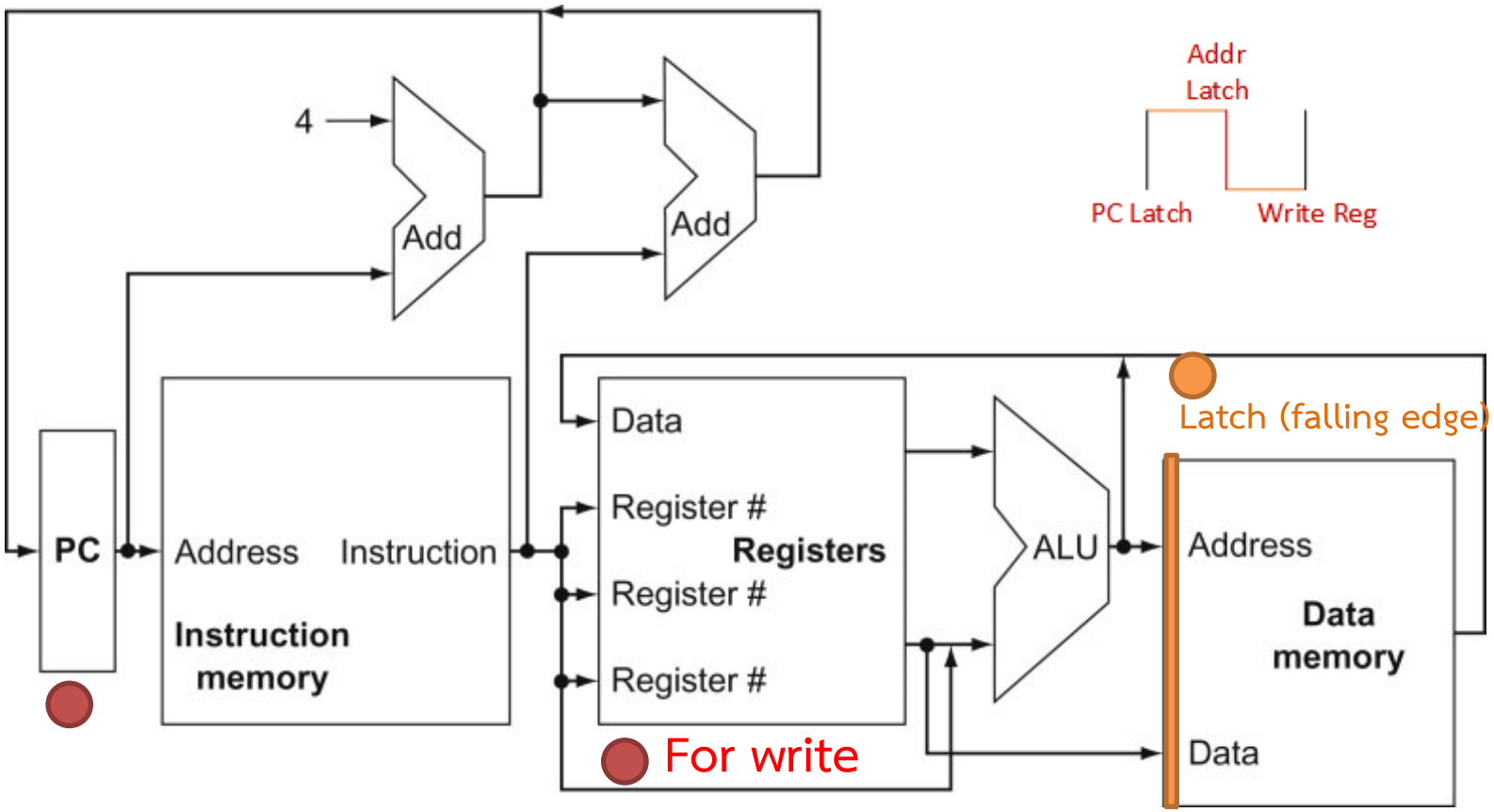


Exercise

- จงเขียนการทำงานโดยย่อ ของคำสั่ง SUB r1, r2, #3



Clocking Methodology

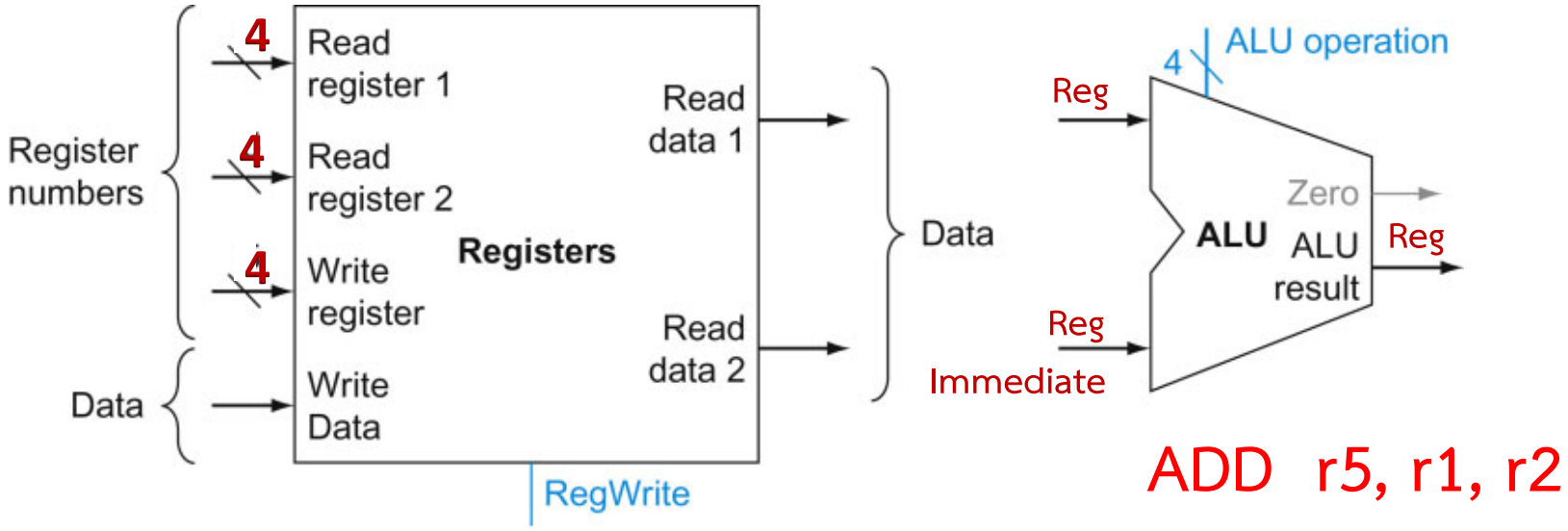


- Unit ใดที่ต้องการ clock
- ส่วนใดที่ต้อง Latch ที่ Rising Edge (ขอบขาขึ้น)

Implementing R-type Instructions

- Registers บางครั้งเรียกว่า register file (เพราะมีหลายตัวรวมกัน) จากรูปแบบคำสั่ง จะมี register สูงสุด 3 ตัว คือ Rn, Rd และ Operand2 มีข้อมูลออก และรับผลลัพธ์สำหรับ write register
- ALU มีเพียง Control ที่กำหนด Operation

Cond	F	I	Opcode	S	Rn	Rd	Operand2
4 bits	2 bits	1 bits	4 bits	1 bits	4 bits	4 bits	12 bits

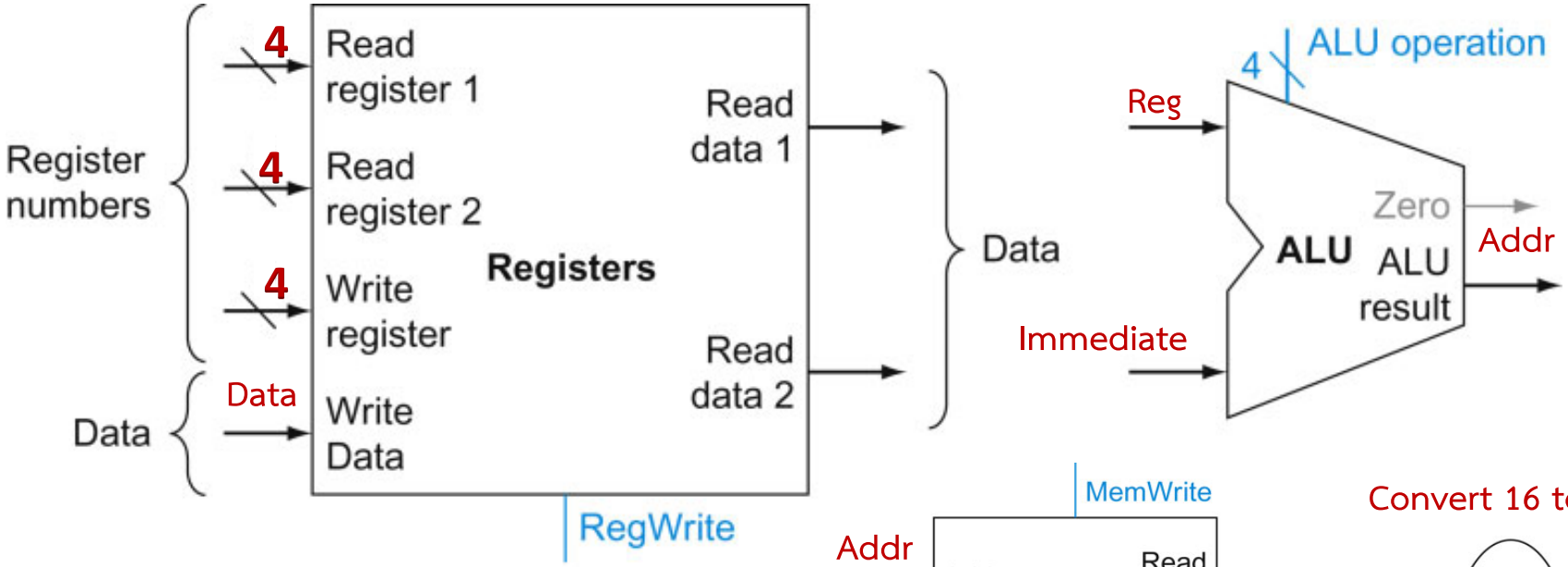


a. Registers

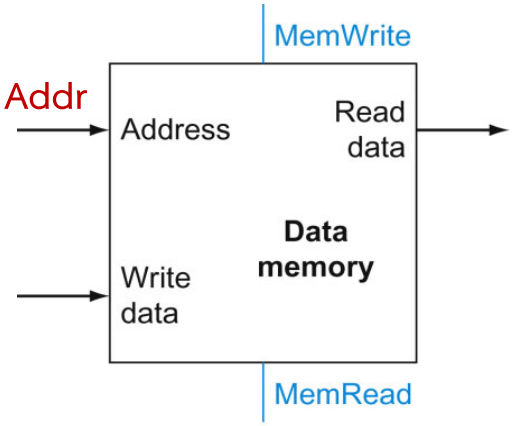
b. ALU

Implementing Load/Stores

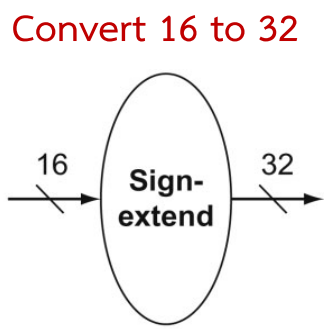
- LDR r0, [r2, #8]



a. Registers



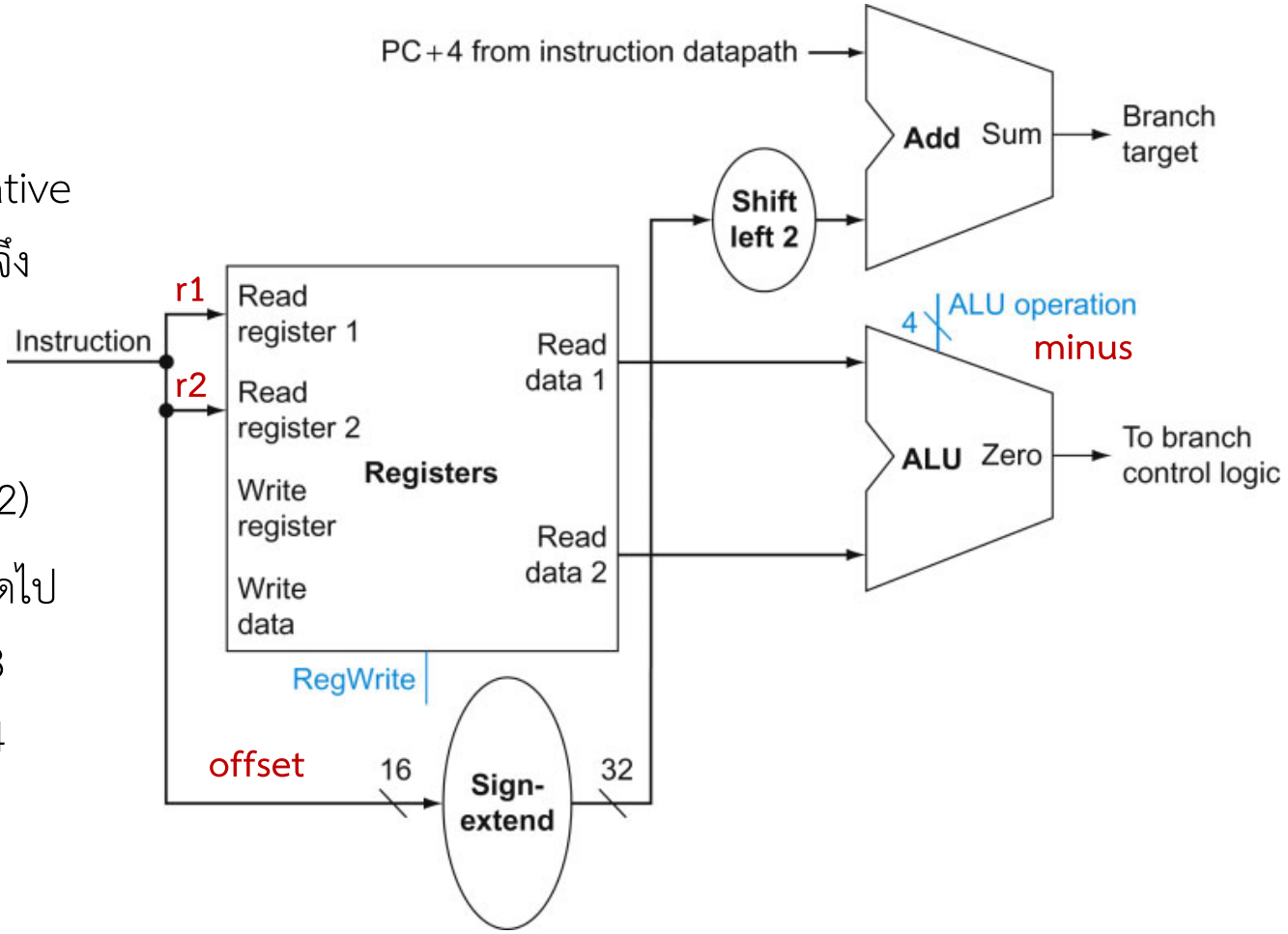
a. Data memory unit



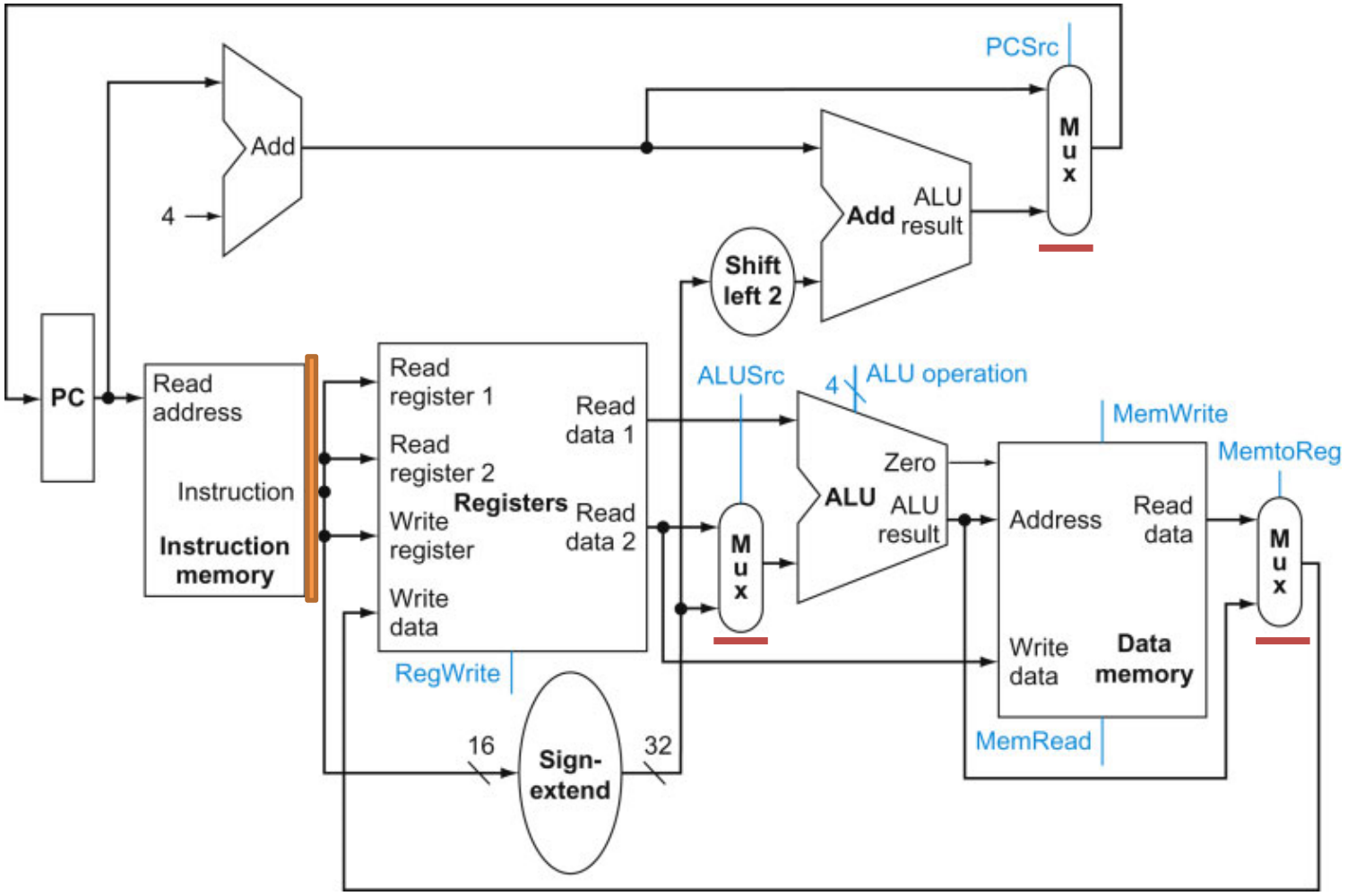
b. Sign extension unit

Implementing B-type Instructions

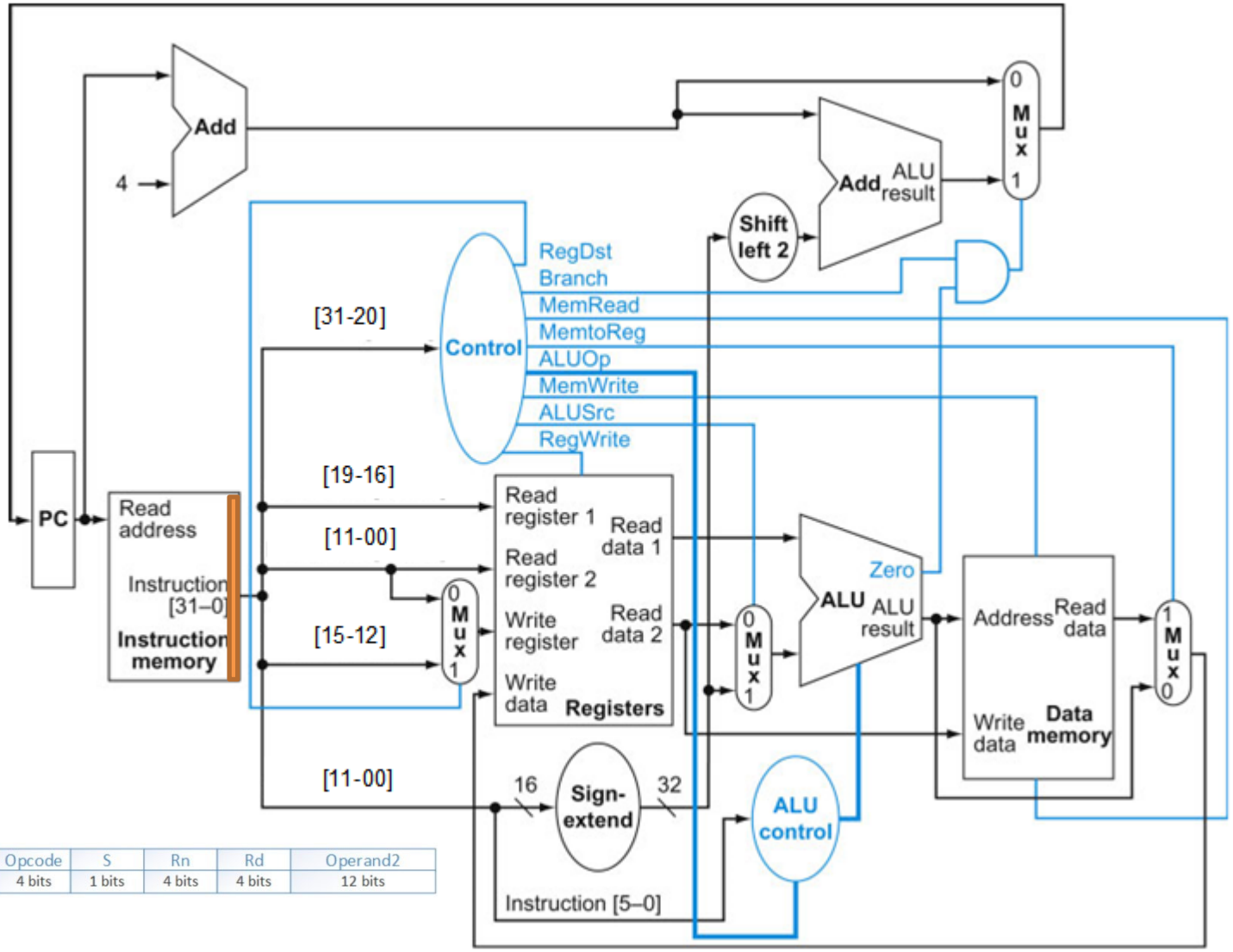
- CMP r0,r1
- BEQ Loop
- Loop จะเก็บเป็น Relative และมีขนาด < 32 บิต จึงต้อง extend ให้เป็น 32 บิต
- จากนั้น x4 (shift left 2)
- เช่น 2 (หมายถึงกระโดดไปอีก 2 คำสั่ง) ก็จะเป็น 8 แล้วจึงไปบวกกับ pc+4



View from 10,000 Feet

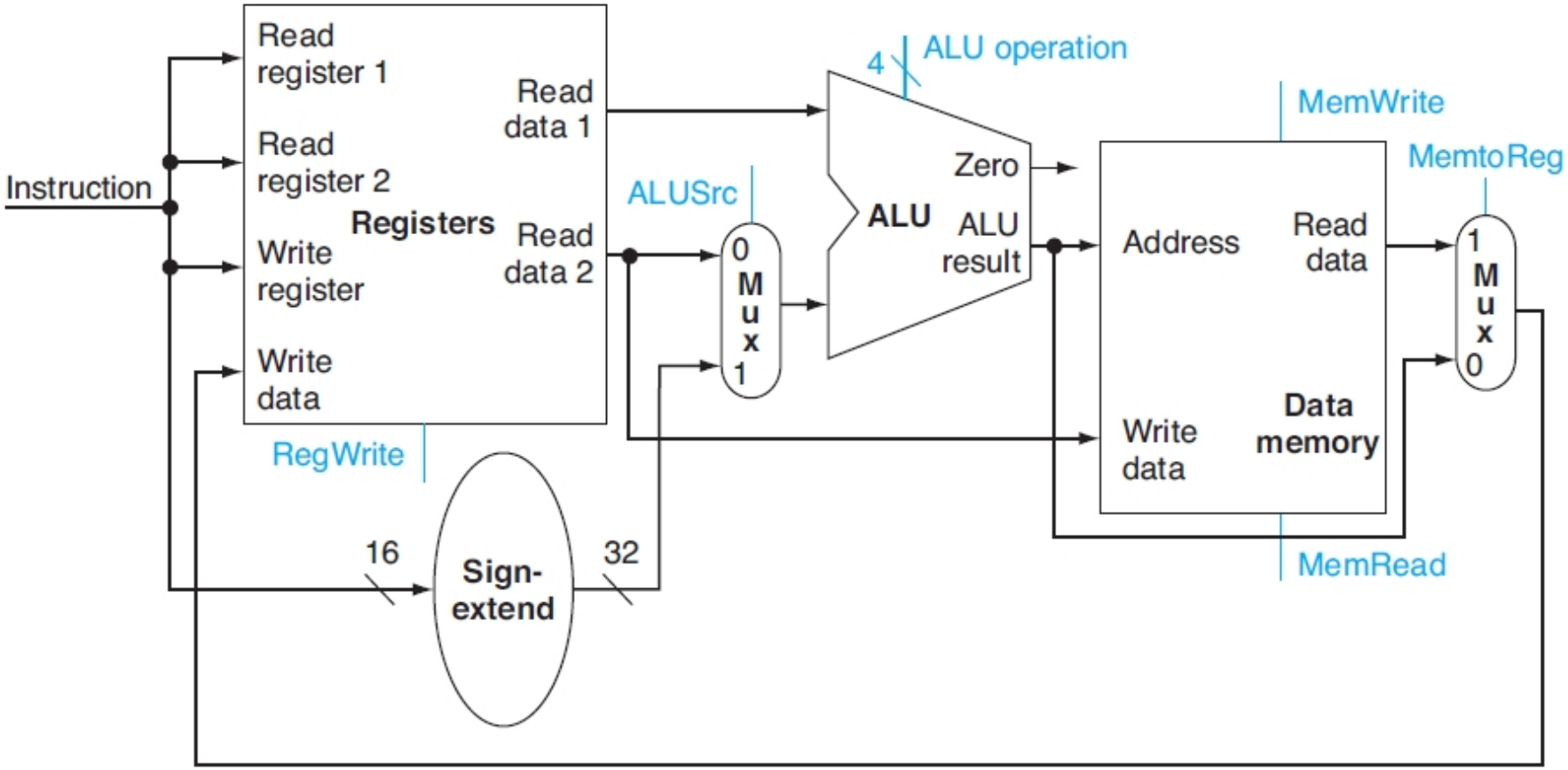


View from 5,000 Feet



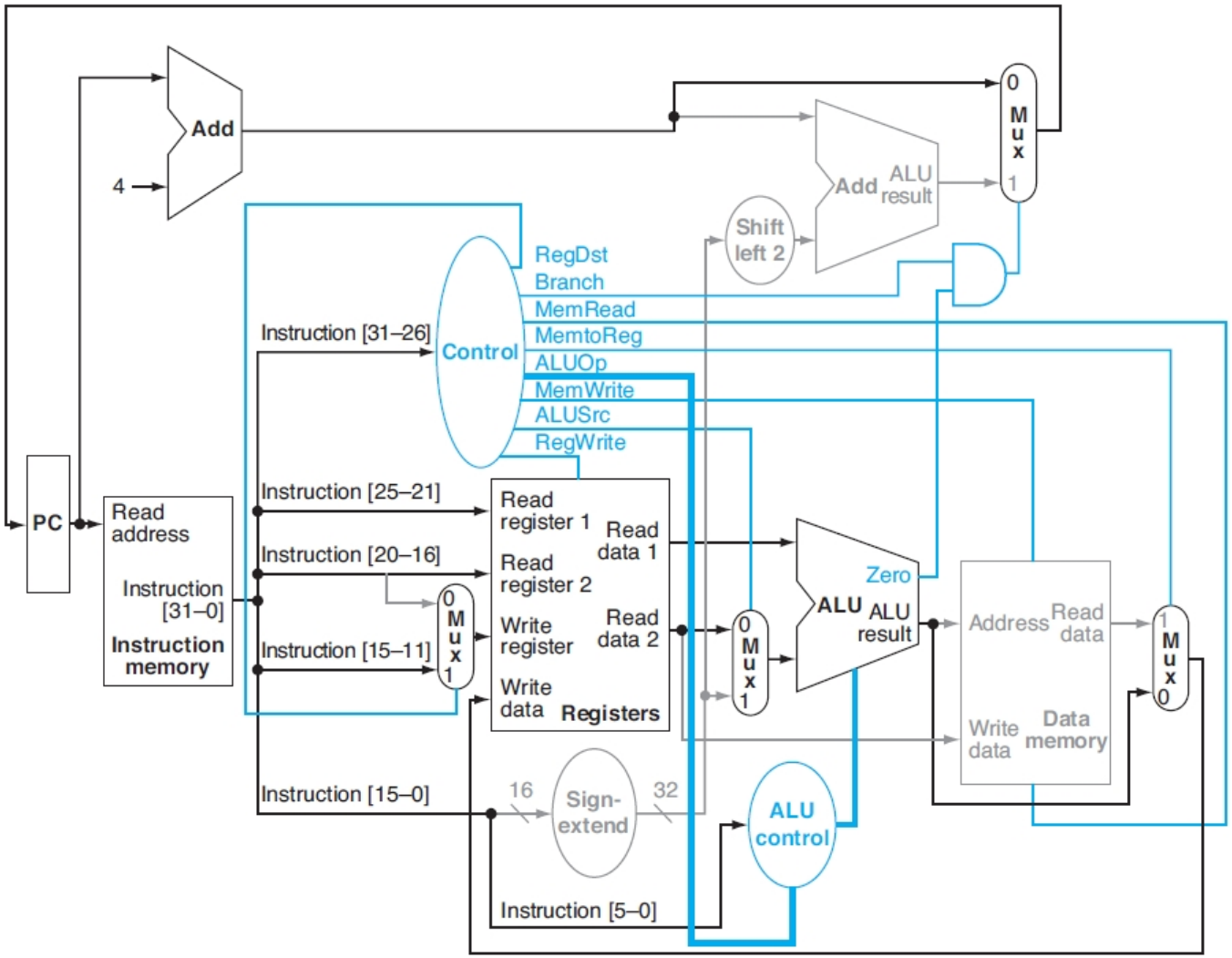
Cond	F	I	Opcode	S	Rn	Rd	Operand2
4 bits	2 bits	1 bits	4 bits	1 bits	4 bits	4 bits	12 bits

Datapath for R-type and Mem



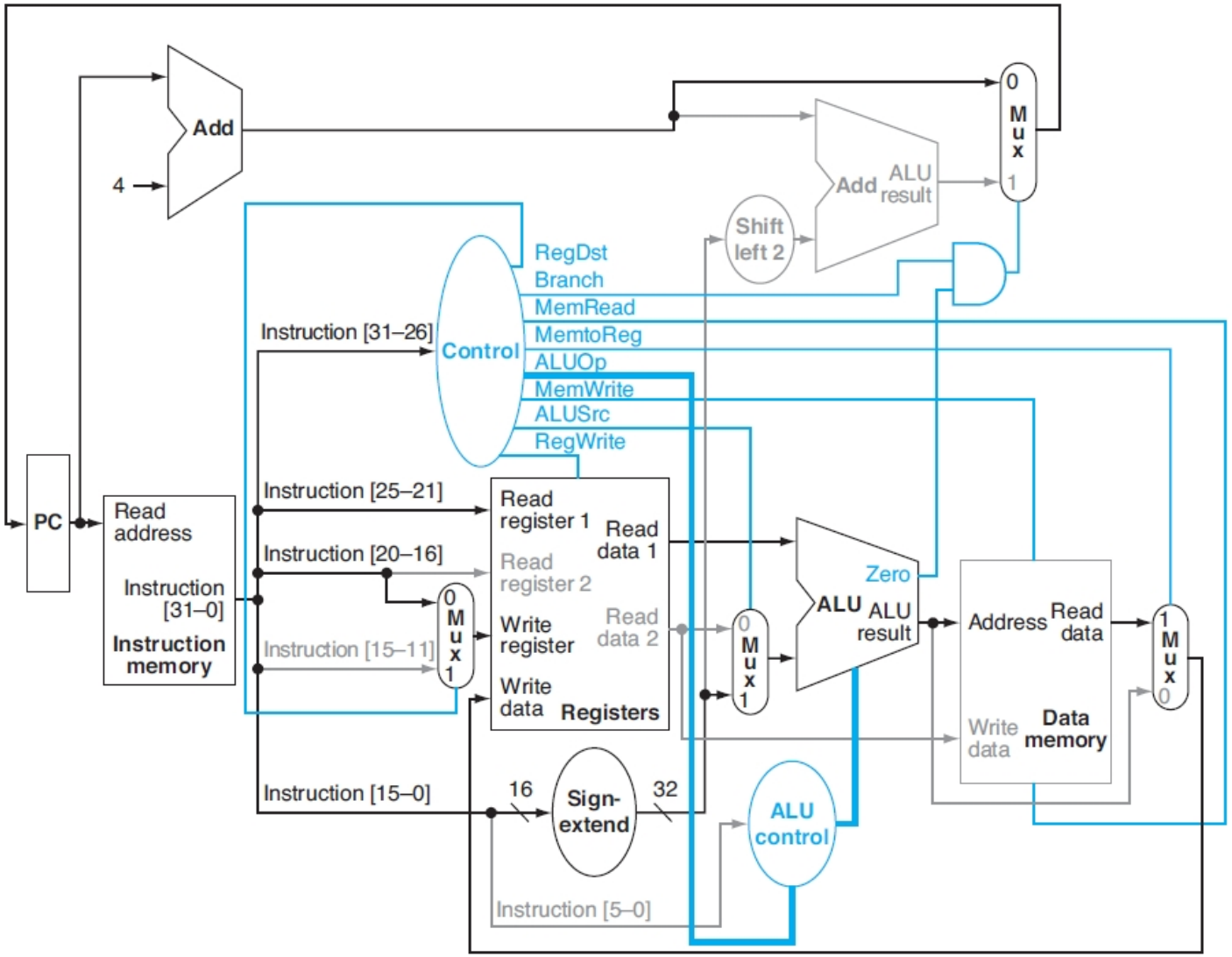
Datapath for R-type instruction

- ADD



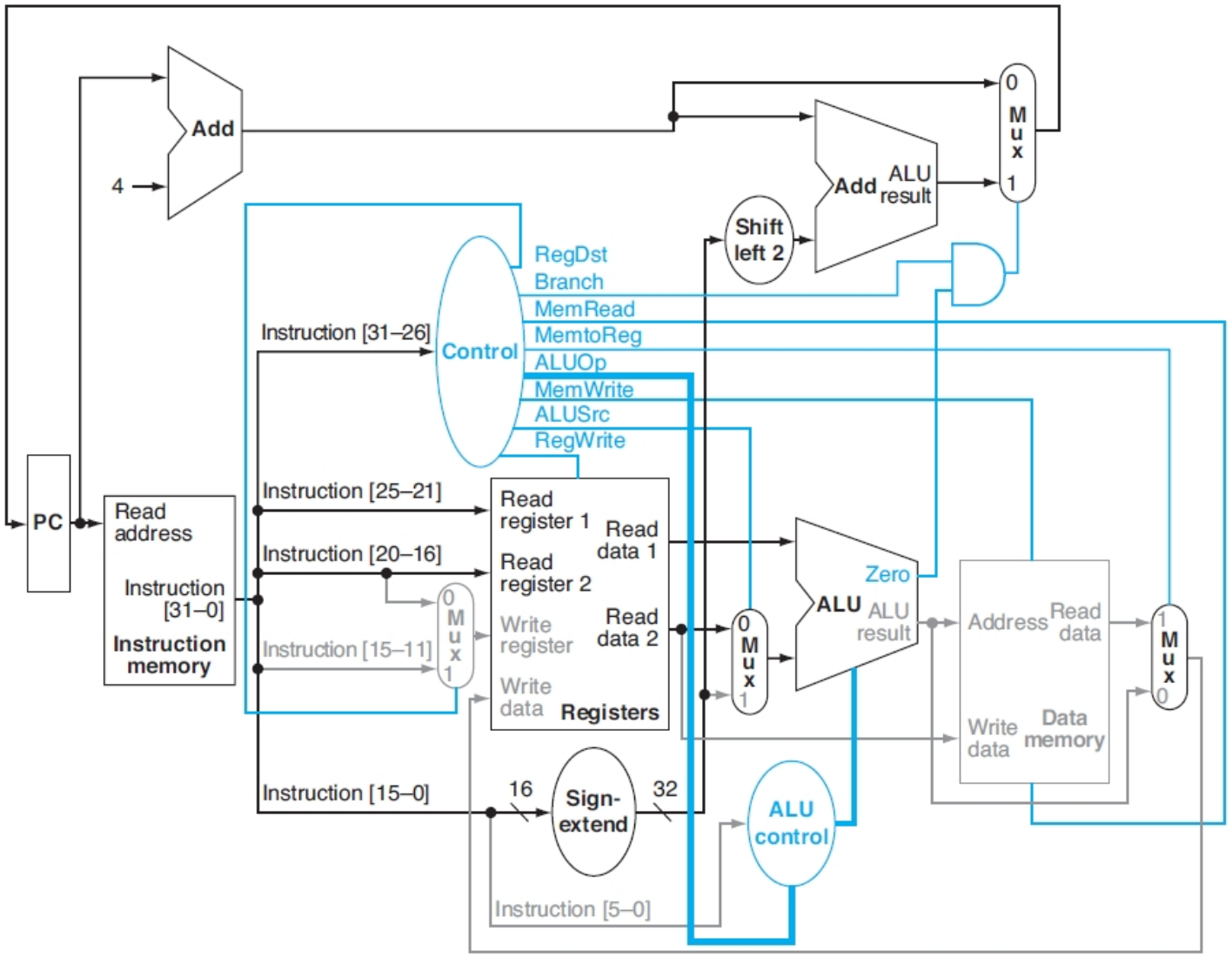
Datapath for Memory instruction

- LDR



Datapath for branch instruction

- BEQ



Exercise

- คำสั่งต่อไปนี้ใช้ Block ไหนบ้าง
  - AND r1, r2, #1
  - LDR r1, [r6, #4]
- ในแต่ละ Block ที่ระบุข้างต้น จะมี Input เป็นอะไร Output เป็นอะไร

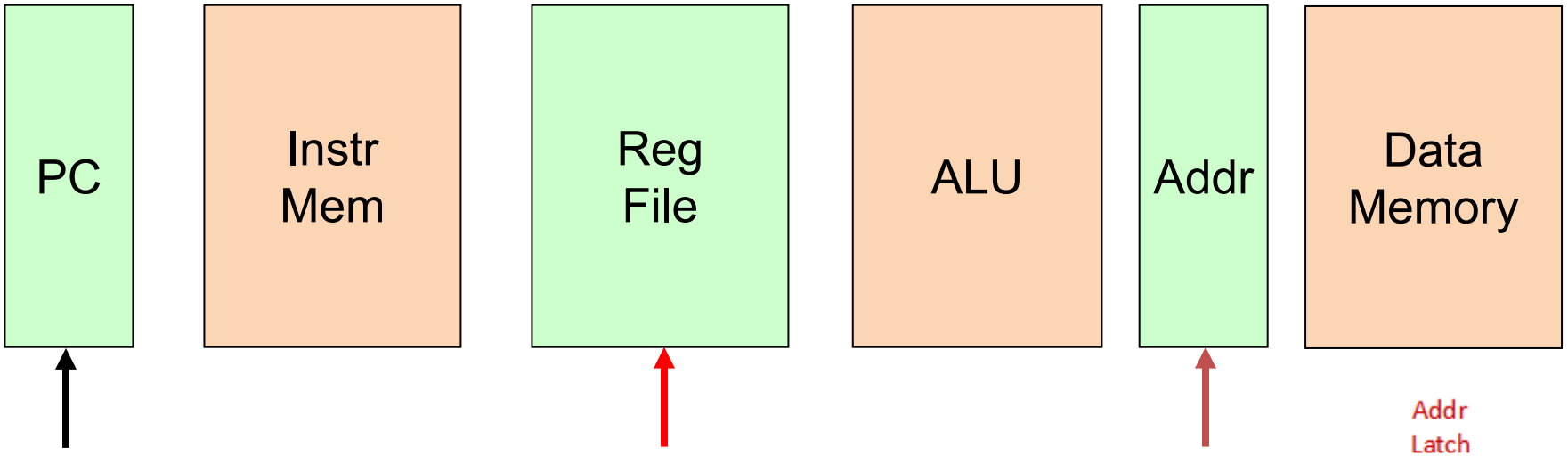
	I-Mem	Add	Mux	ALU	Regs	D-Mem	Sign-Extend	Shift-Left-2
a.	200ps	70ps	20ps	90ps	90ps	250ps	15ps	10ps
b.	750ps	200ps	50ps	250ps	300ps	500ps	100ps	0ps

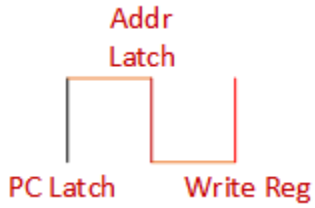
- กำหนดค่าการ Latency ที่ใช้ในแต่ละ Block คำสั่งข้างต้น (ใช้ a) จะใช้เวลาเท่าไรในการทำงาน
- ควรใช้ค่า clock เท่าไร

Basic Pipelining

- 1-stage design
- 5-state design
- 5-state pipeline
- Hazards

Latch and Clocks in a Single-Cycle Design

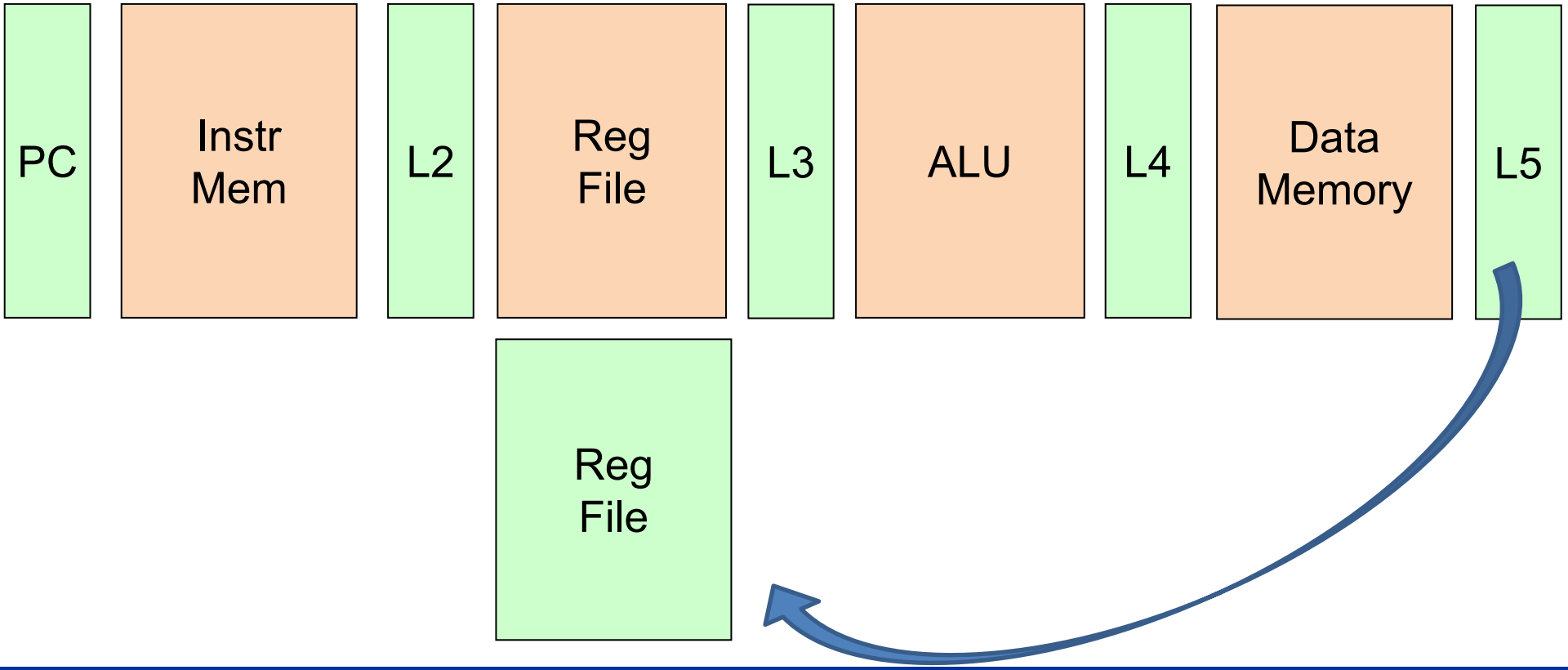




- การทำงานทั้งหมดในคำสั่งเกิดขึ้นใน 1 clock cycle
- กล่องสีเขียว คือ ส่วนที่มี Latch
- ที่ขอบขาขึ้นของ clock จะ Latch ข้อมูลตำแหน่ง (PC) และจะค้างข้อมูลและส่งมาที่ Instruction Memory ↑
- ที่ขอบขาขึ้น ผลลัพธ์ของคำสั่งก่อนหน้าจะถูกบันทึกเช่นกัน ↑
- ที่ขอบขาลงข้อมูล Address ของคำสั่ง Load/Store จะ Latch ดังนั้นจะสามารถเข้าถึงหน่วยความจำข้อมูล (Data Memory) ได้ในครึ่งหลังของ clock ↑

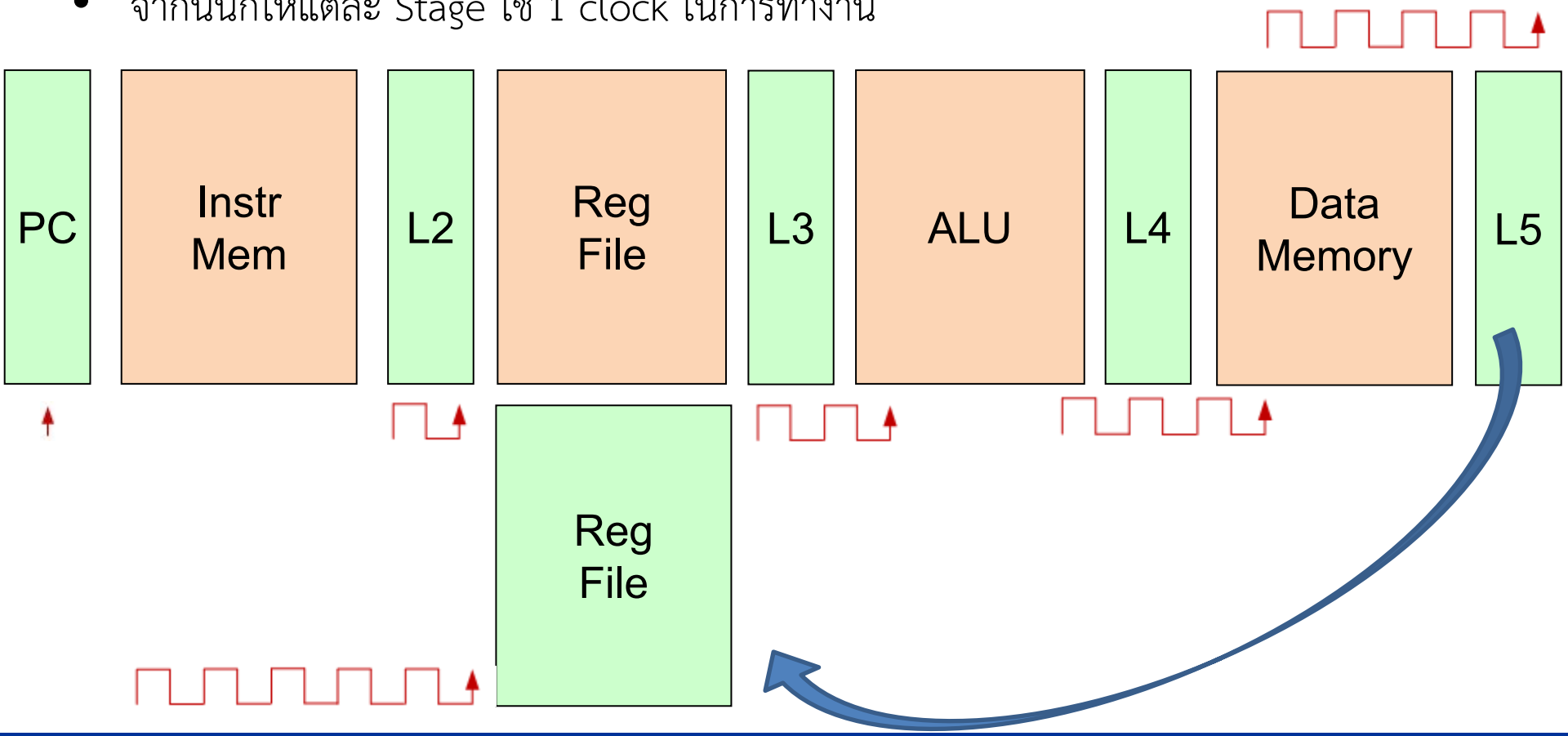
Multi-Stage Circuit

- โครงสร้างในแบบ Single Stage ไม่เป็นที่นิยมในปัจจุบัน เนื่องจากประสิทธิภาพต่ำ โดยในขณะที่ทำอย่างใดอย่างหนึ่ง หน่วยอื่นๆ ใน CPU จะอยู่เฉยๆ
- จึงเป็นที่มาของ Multi-Stage Circuit



Multi-Stage Circuit

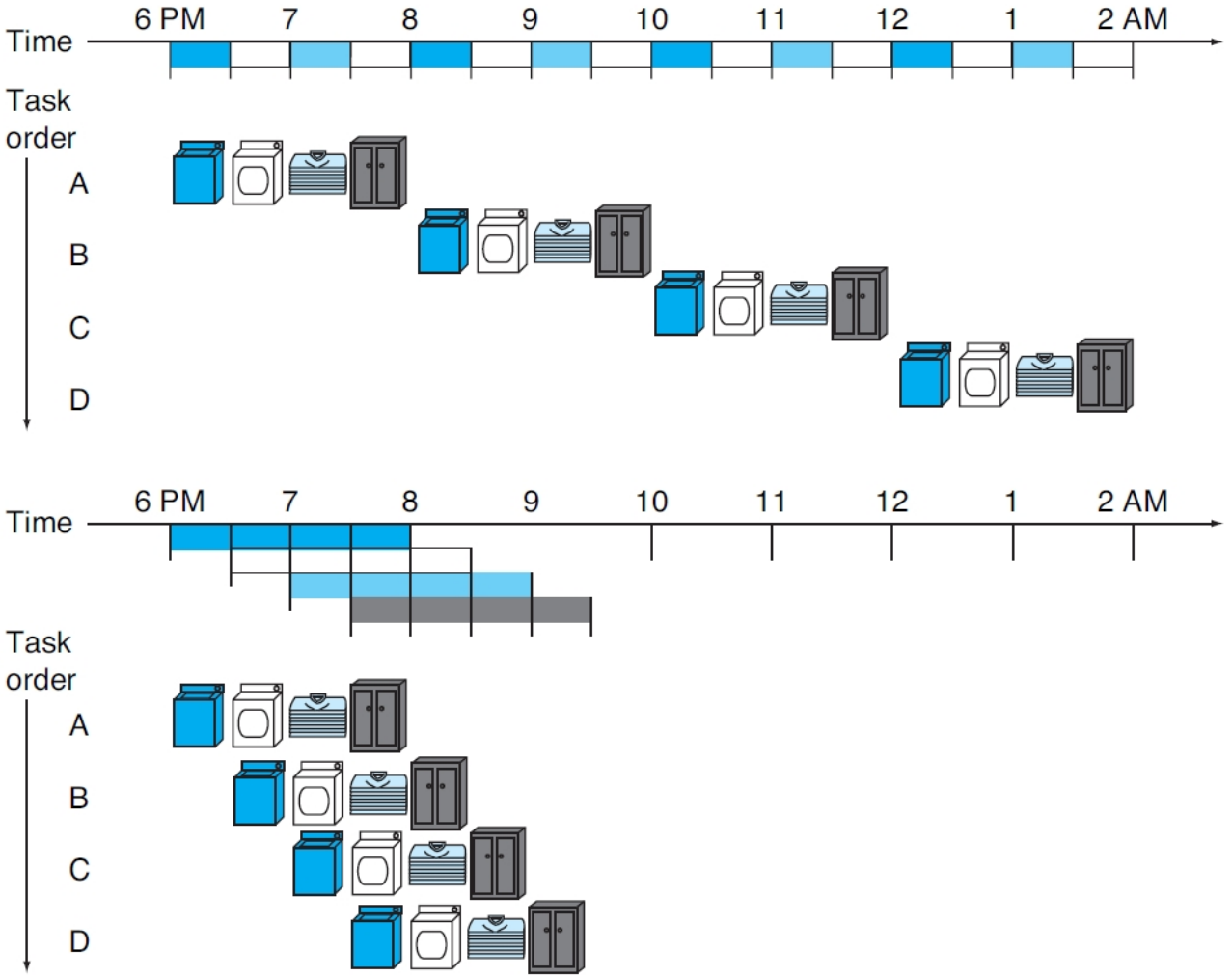
- Multi-Stage Circuit ใช้หลักการว่า แทนที่จะทำงานทั้งคำสั่งในคราวเดียว (single stage) ก็ทำการแบ่งการทำงานออกเป็น Stage ย่อยๆ หลาย Stage โดยแต่ละ Stage คั่นด้วย Latch
- จากนั้นก็ให้แต่ละ Stage ใช้ 1 clock ในการทำงาน



Performance Improvement?

- การแบ่งงานเป็นส่วนๆ นี้เรียกว่า pipeline
  - การทำเช่นนี้ จะทำให้งานเสร็จช้าลงหรือไม่?
  - การทำเช่นนี้ จะทำให้หลายงานเสร็จเร็วขึ้นหรือไม่
  - ควรจะตั้งสมมติฐานอย่างไร เพื่อที่จะตอบคำถามข้างต้น
  - แล้ว 10 stage pipeline จะดีกว่า 5 stage pipeline หรือเปล่า

Laundry Pipeline

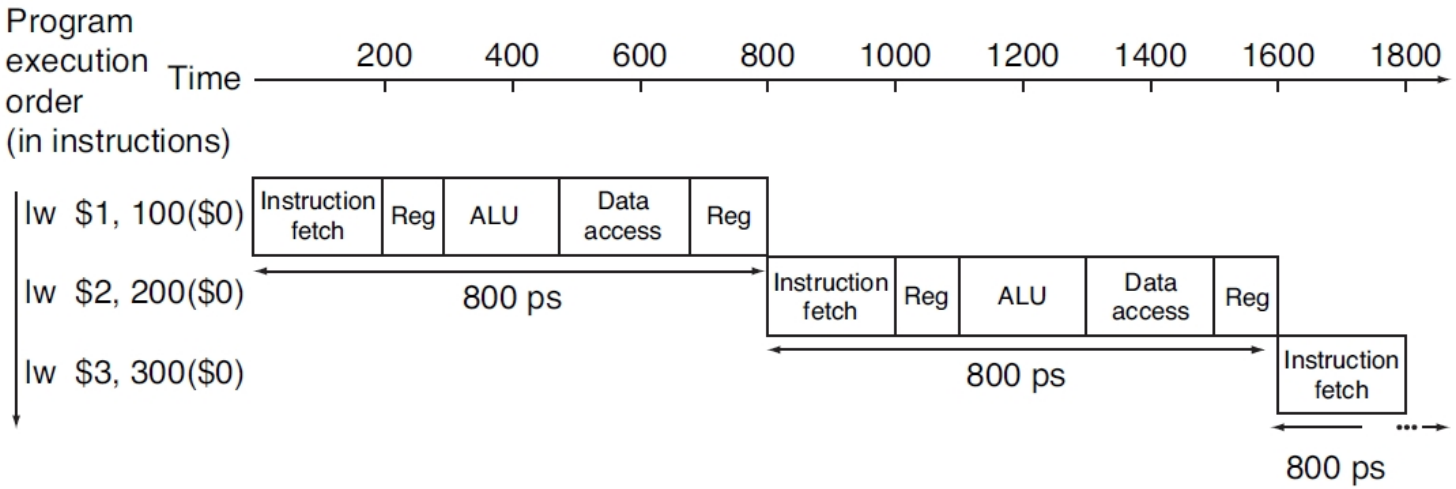


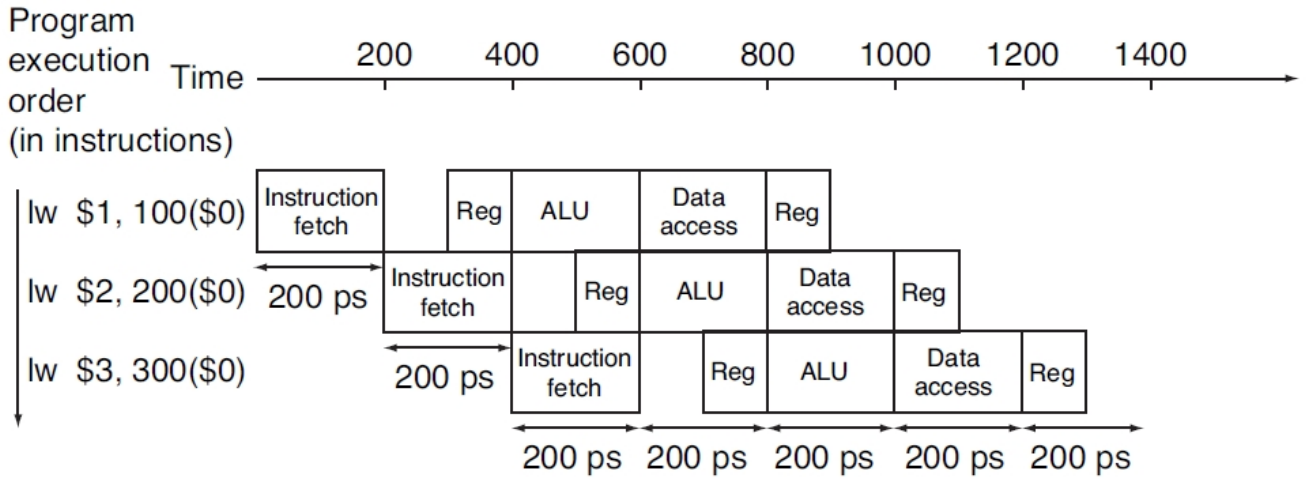
Instruction Time

- สมมติว่าแต่ละขั้นตอนของ CPU ทำงานใช้เวลาตามรูป
- แต่ละขั้นตอนจะใช้เวลาไม่เท่ากัน
  - ส่วนที่เกี่ยวกับหน่วยความจำ กำหนดให้ใช้ 200 ps
  - ส่วนที่เกี่ยวกับ Register File กำหนดให้ใช้ 100 ps
  - ส่วนที่เกี่ยวกับ ALU กำหนดให้ใช้ 200 ps

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add`, `sub`, `AND`, `OR`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

Instruction Pipelining





Performance Improvement

- ถ้า Pipeline แต่ละ stage มีเวลาเท่ากัน สามารถใช้สูตร

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

- สมมติว่าคำสั่งใช้เวลา 800 ps โดยมี 5 stage ดังนั้นเวลาที่ใช้เมื่อทำ pineline แล้วจะประมาณ 160 ps
- แต่จากรูปในสไลด์ก่อนหน้า จะเห็นว่าจาก 3 คำสั่งที่ใช้ 2,400 ps (non pipeline) เมื่อทำ pipeline แล้วจะเหลือ 1400 ps ซึ่งเร็วขึ้นประมาณ 1.7 เท่า
- แต่หากทำงานจำนวนคำสั่งมากๆ แล้ว ค่าสัดส่วนนี้จะเพิ่มขึ้นเป็นประมาณ 4.0
- อาจจะสงสัยว่าในเมื่อเดิมก็สามารถทำเสร็จได้ใน 1 clock cycle อยู่แล้ว จะเร็วขึ้นได้อย่างไร -> เมื่องานลดลงก็สามารถจะเพิ่ม clock cycle ได้

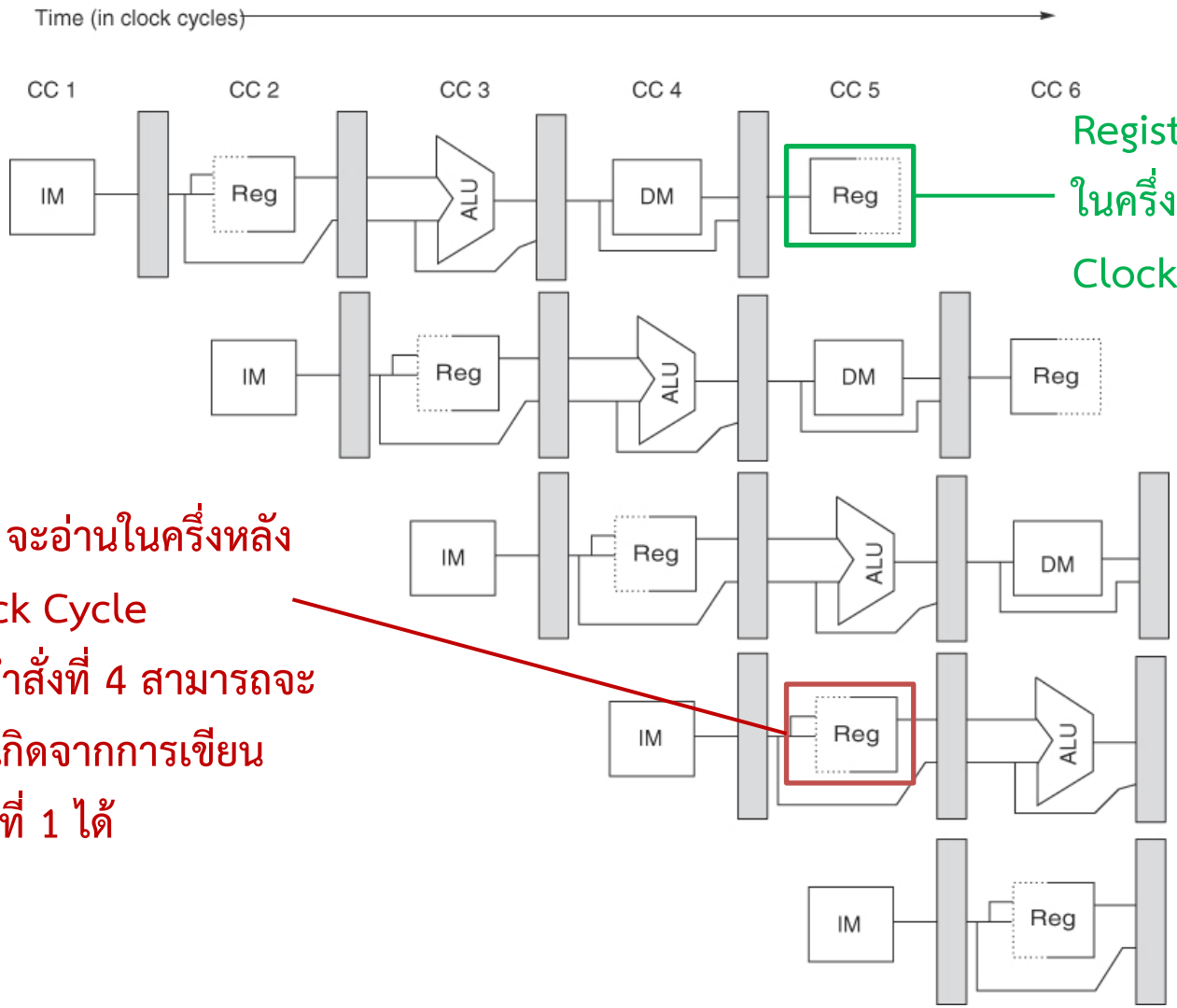
Performance Improvement?

- การแบ่งงานเป็นส่วนๆ นี้เรียกว่า pipeline
  - การทำเช่นนี้ จะทำให้งานเสร็จช้าลงหรือไม่? **บางทีอาจจะช้าลงเล็กน้อย**
  - การทำเช่นนี้ จะทำให้หลายงานเสร็จเร็วขึ้นหรือไม่ Yes
  - ควรจะตั้งสมมติฐานอย่างไร เพื่อที่จะตอบคำถามข้างต้น
    - **แต่ละคำสั่งที่ทำงานต่อกัน จะต้องไม่ขึ้นต่อกัน เช่น คำสั่งที่ 2 ต้องการผลจากคำสั่งแรก ก็จะต้องรอ (เรียกว่า stall)**
    - **ไม่มี Overhead จาก Latch**
  - แล้ว 10 stage pipeline จะดีกว่า 5 stage pipeline หรือเปล่า
    - **ไม่แน่ การมี stage มาก ทำให้โอกาสเกิด stall มากขึ้น และ Latch Overhead จะมีมากขึ้น**

Quantitative Effects

- ผลจากการใช้ pipeline
  - เวลาที่ใช้ต่อคำสั่งจะเพิ่มขึ้น
  - แต่ละคำสั่งจะต้องใช้จำนวน clock cycle มากขึ้นในการทำงาน
  - แต่โดยเฉลี่ยแล้ว CPI จะเท่าเดิม
  - สามารถเพิ่ม clock speed ได้
  - เวลารวมที่ใช้ในการทำงานจะลดลง ทำให้เวลาเฉลี่ยต่อคำสั่งจะลดลง
  - Ideal Condition : จำนวน pipeline stage = จำนวนเท่าของ clock speed ที่เพิ่มขึ้น

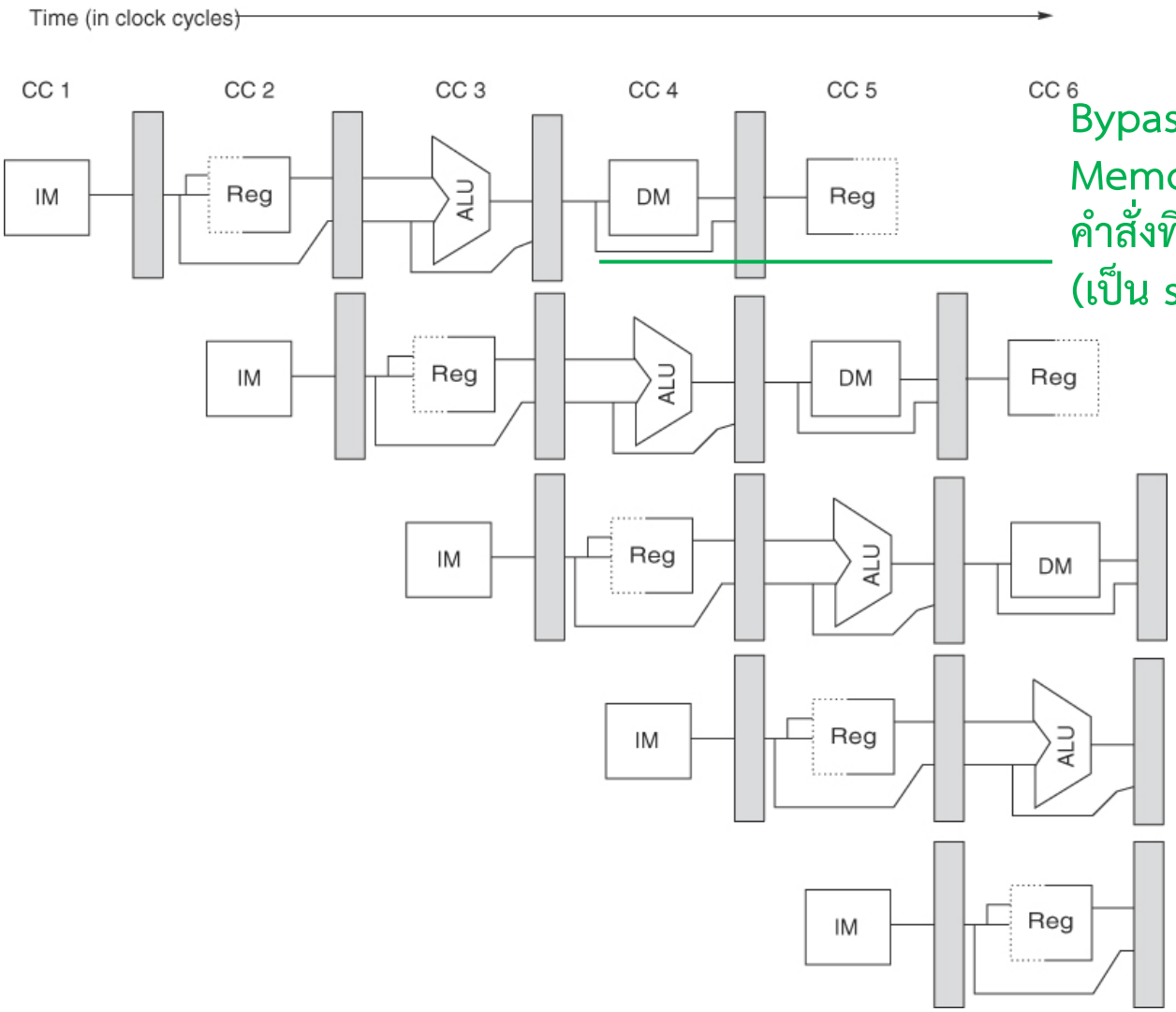
5 Stage-Pipeline



Register จะเขียนในครึ่งแรกของ Clock Cycle

Register จะอ่านในครึ่งหลังของ Clock Cycle ใน cc5 คำสั่งที่ 4 สามารถจะอ่านผลที่เกิดจากการเขียนของคำสั่งที่ 1 ได้

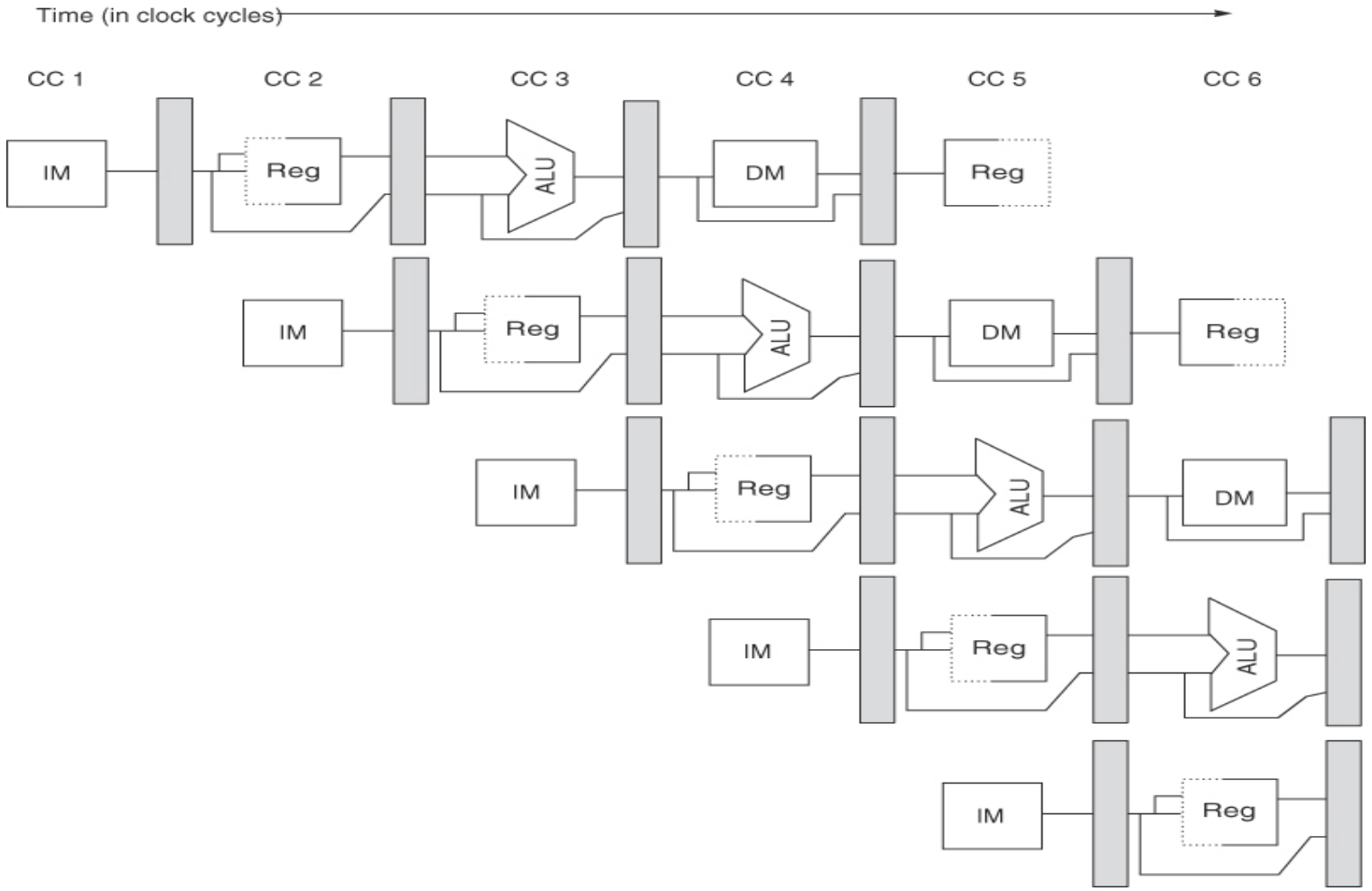
5 Stage-Pipeline



Bypass Data Memory สำหรับคำสั่งที่ไม่อ่าน mem (เป็น stage ว่าง)

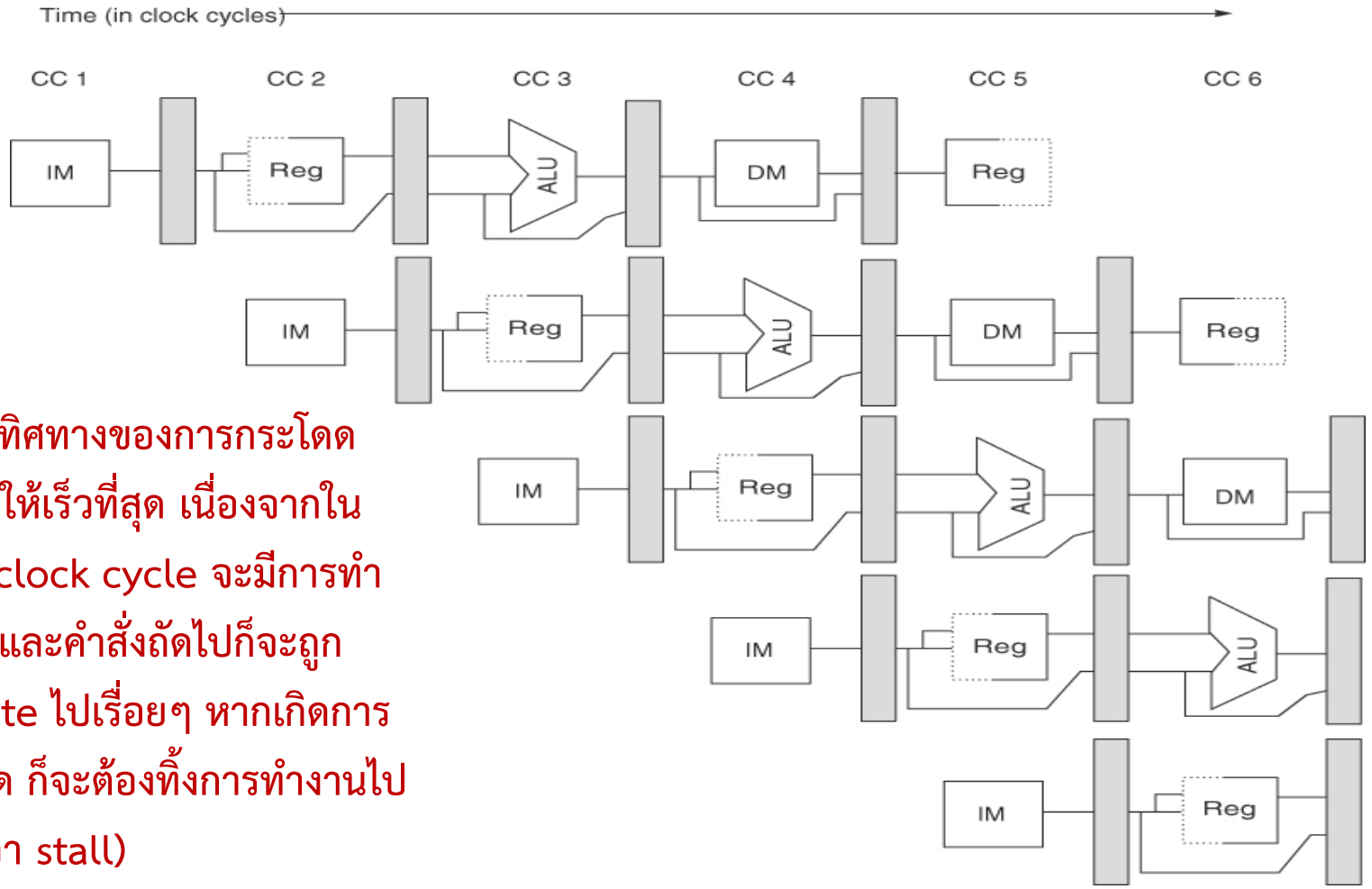
5 Stage-Pipeline

Stage 1 : ใช้ PC เพื่อ access I-cache และเพิ่มค่า PC ขึ้น 4



5 Stage-Pipeline

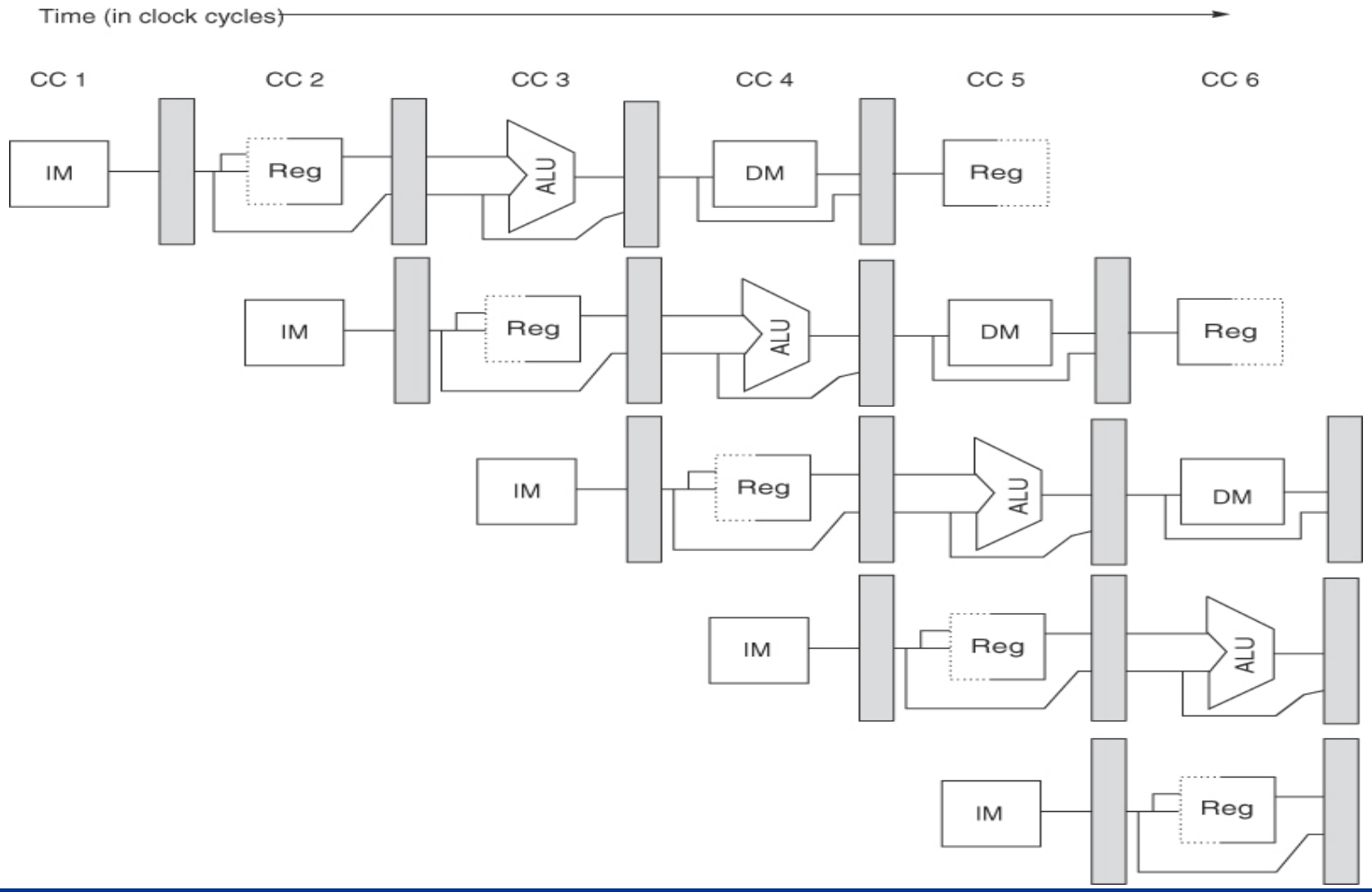
Stage 2 : อ่านรีจิสเตอร์ หรือเปรียบเทียบรีจิสเตอร์ เพื่อคำนวณการกระโดด (Branch)



การหาทิศทางของการกระโดด ต้องทำให้เร็วที่สุด เนื่องจากในแต่ละ clock cycle จะมีการทำ PC+4 และคำสั่งถัดไปก็จะถูก execute ไปเรื่อยๆ หากเกิดการกระโดด ก็จะต้องทิ้งการทำงานไป (เรียกว่า stall)

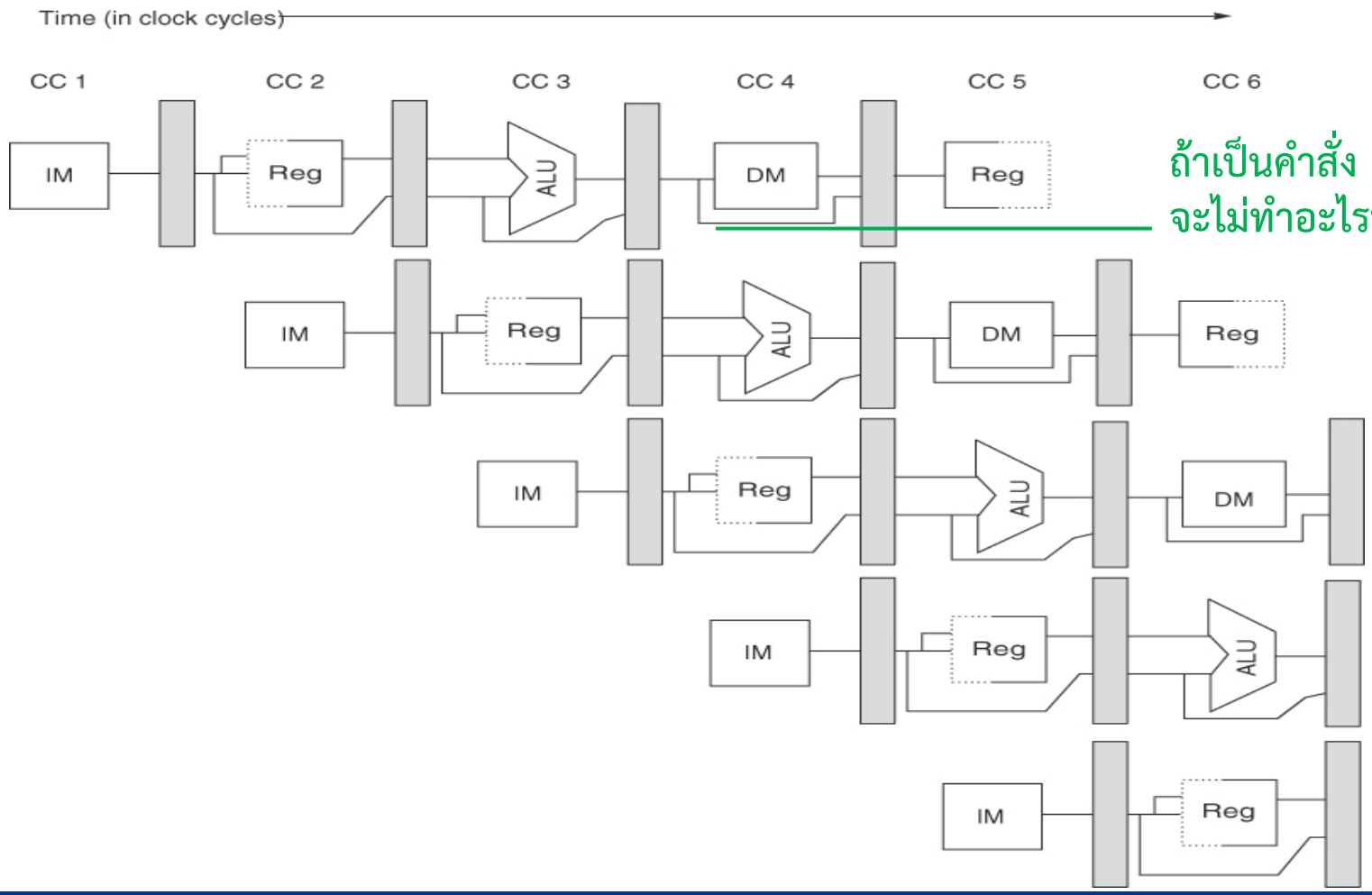
5 Stage-Pipeline

Stage 3 : คำนวณใน ALU หรือหา Address สำหรับ Load/Store



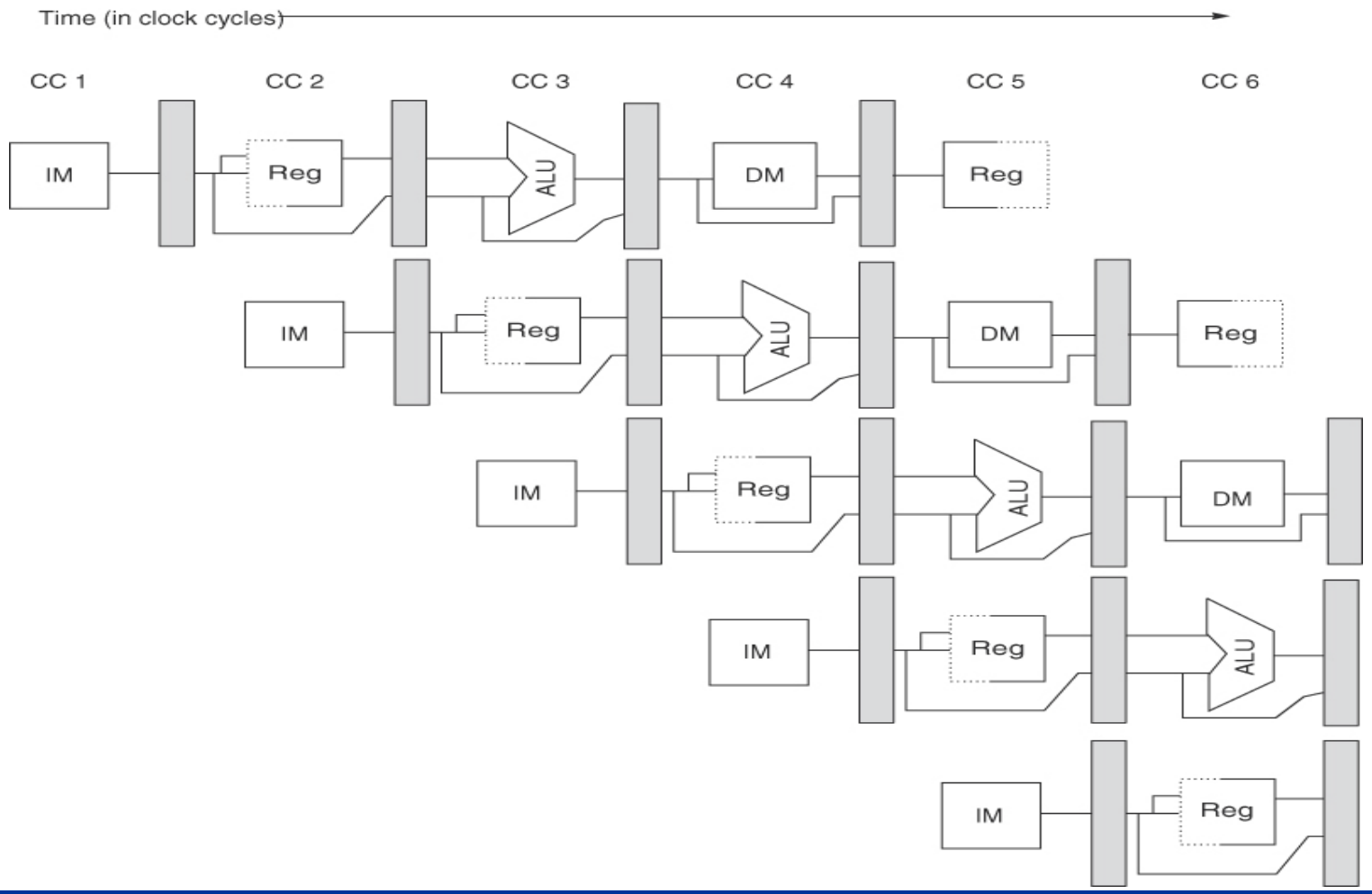
5 Stage-Pipeline

Stage 4 : Memory Access to/from Data Cache คำสั่ง STR เสร็จที่ stage นี้



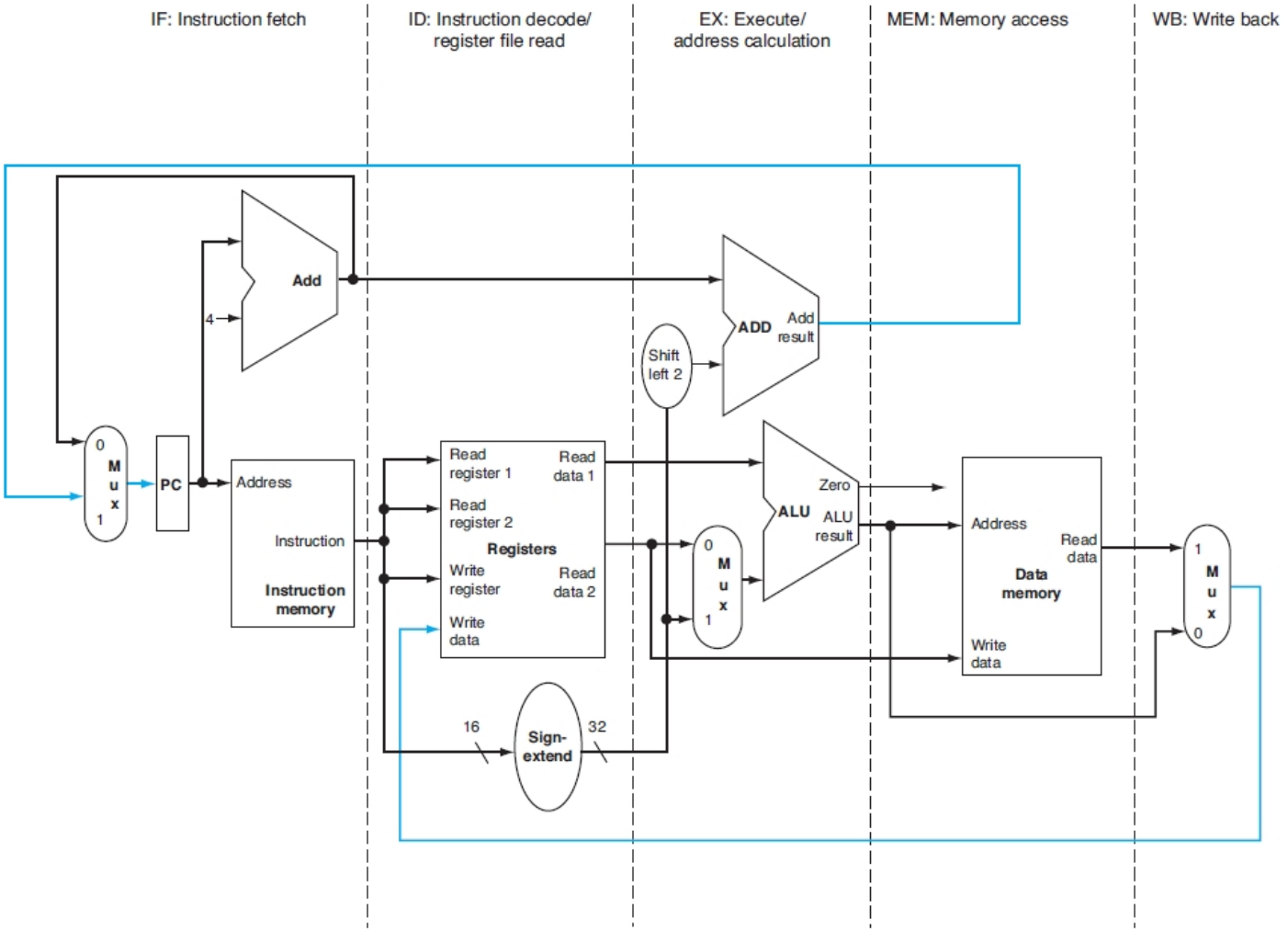
5 Stage-Pipeline

Stage 5 : เขียนผลที่ได้จาก ALU หรือจาก Memory ลงใน register (ขอบขาขึ้น)



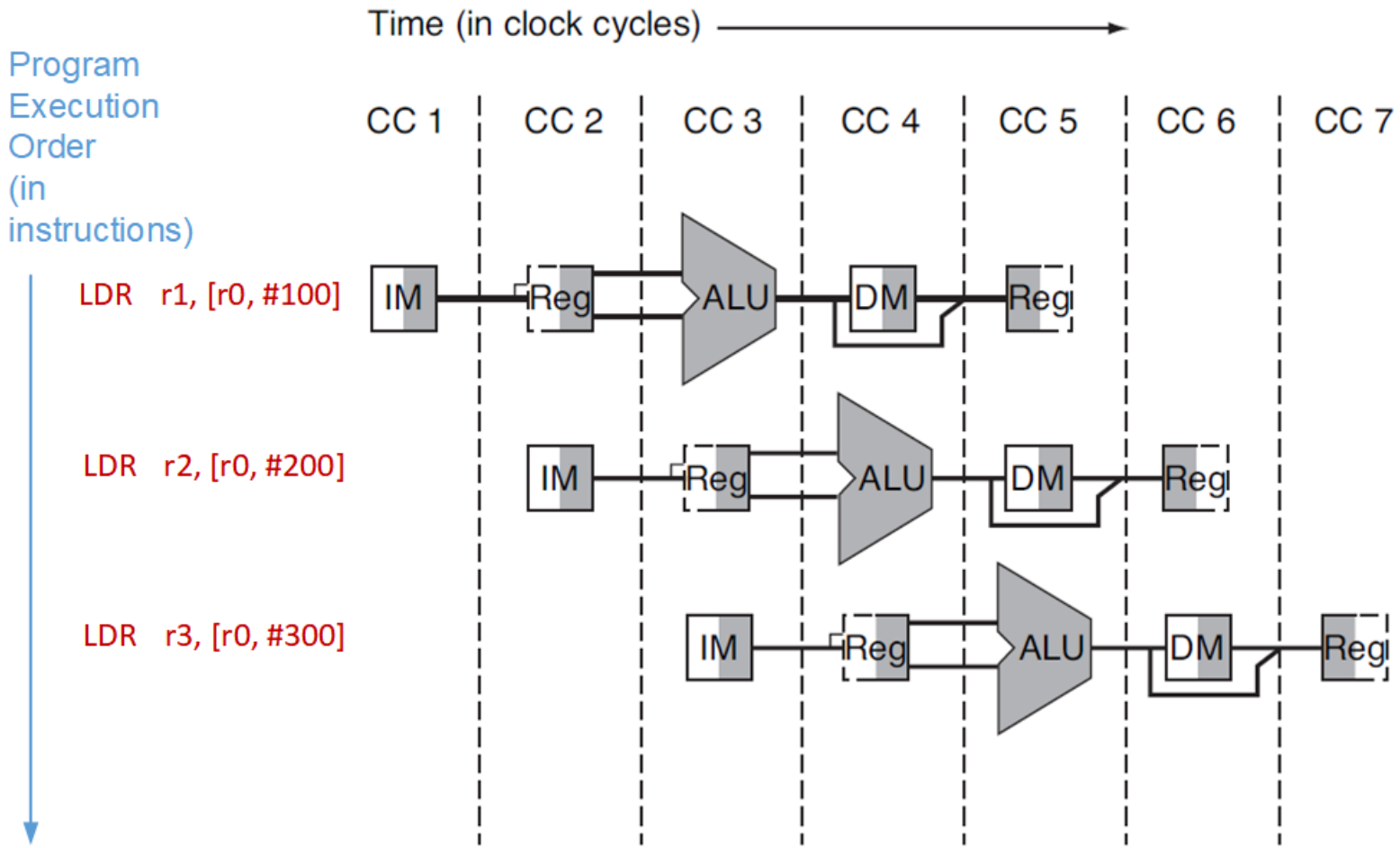
Pipeline Datapath

- Single Instruction datapath



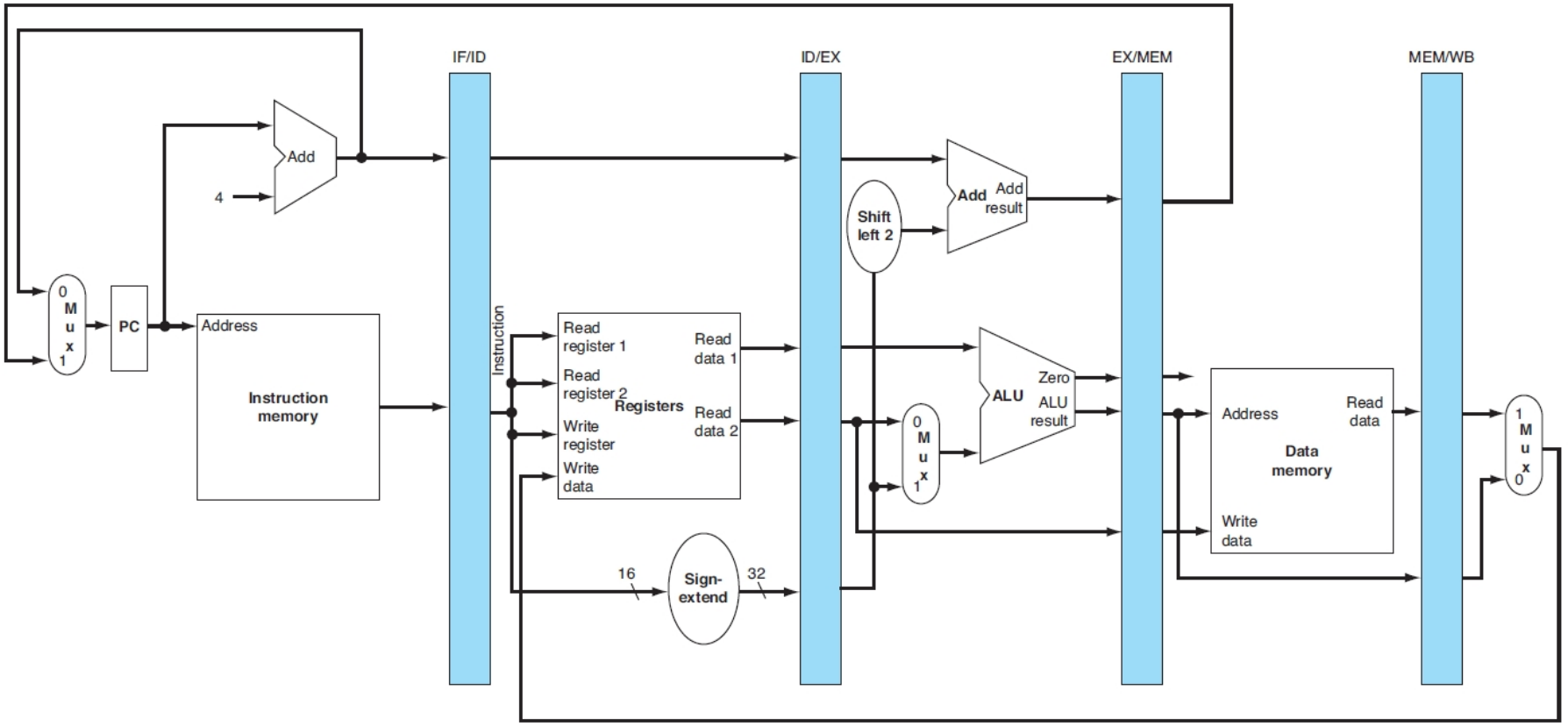
Pipeline Datapath

- Single Instruction datapath



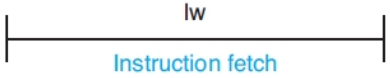
Pipeline Datapath

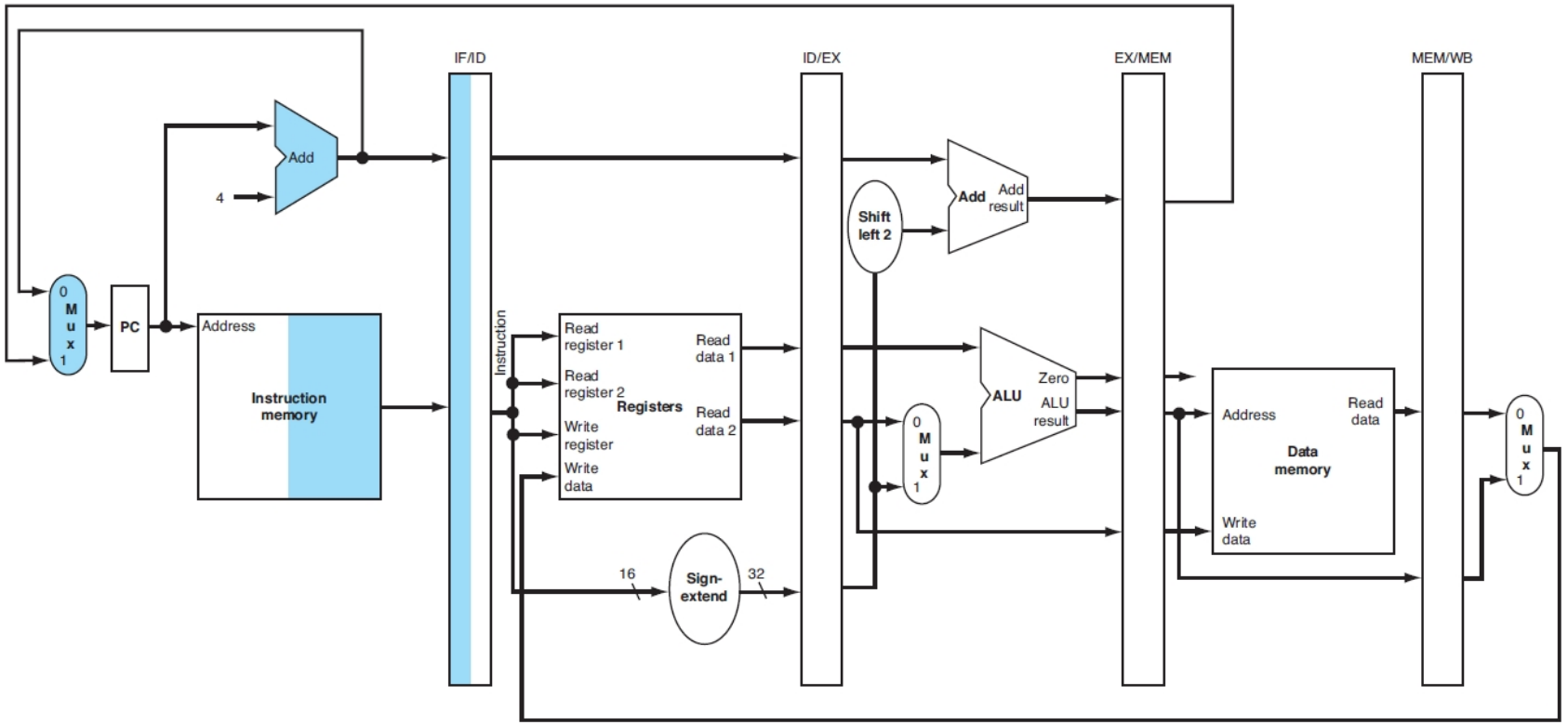
- Pipeline version of datapath



Pipeline Datapath

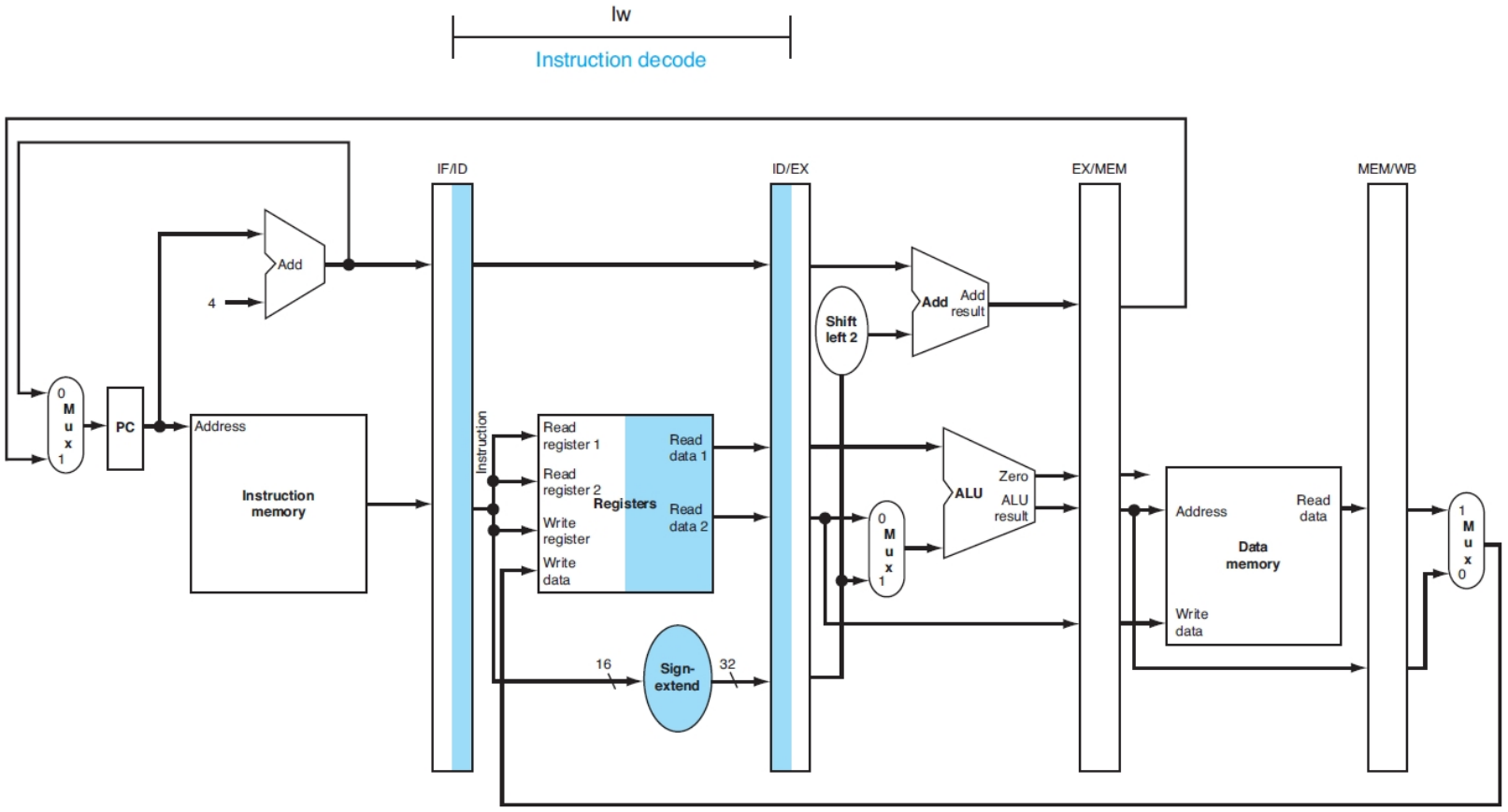
- Instruction fetch : เป็นขั้นตอนที่คำสั่งถูกอ่านจากหน่วยความจำ โดยใช้ตำแหน่งจาก PC และใส่ลงใน IF/ID pipeline register จากนั้น PC=PC+4 แล้ว write กลับไปที่ PC เพื่อรอ Fetch คำสั่งถัดไปในอีก 1 clock cycle โดยจะเก็บใน IF/ID กรณีที่มีการนำไปใช้อีก เช่น BEQ





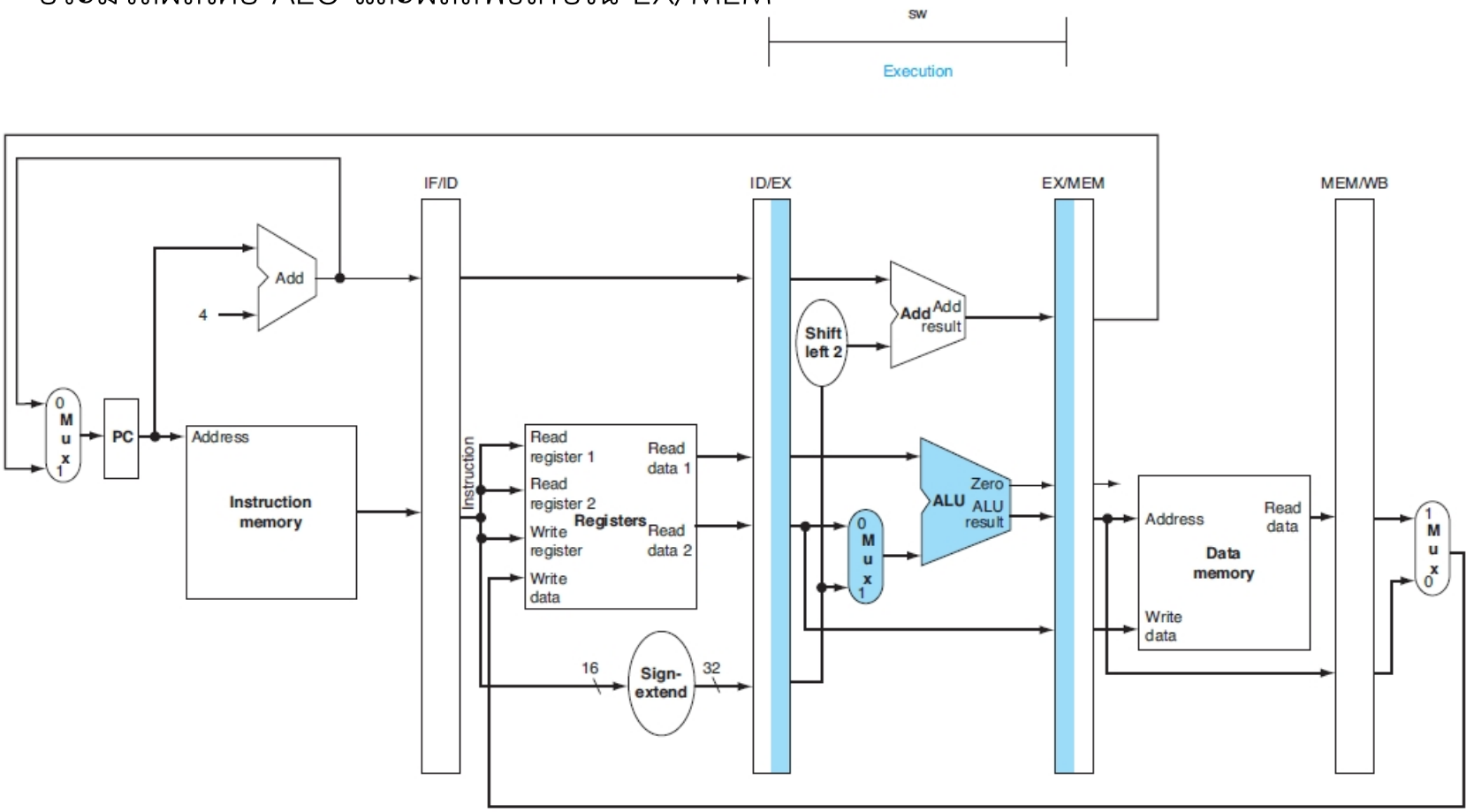
Pipeline Datapath

- Instruction decode and register file read : เป็นขั้นตอนที่ IF/ID pipeline register ส่งข้อมูล Immediate (ที่ทำ sign-extended แล้ว) และข้อมูลใน register จำนวน 2 ตัว (หากมีการสั่งให้ read register) โดยทั้ง 3 ค่าจะเก็บที่ ID/EX pipeline register



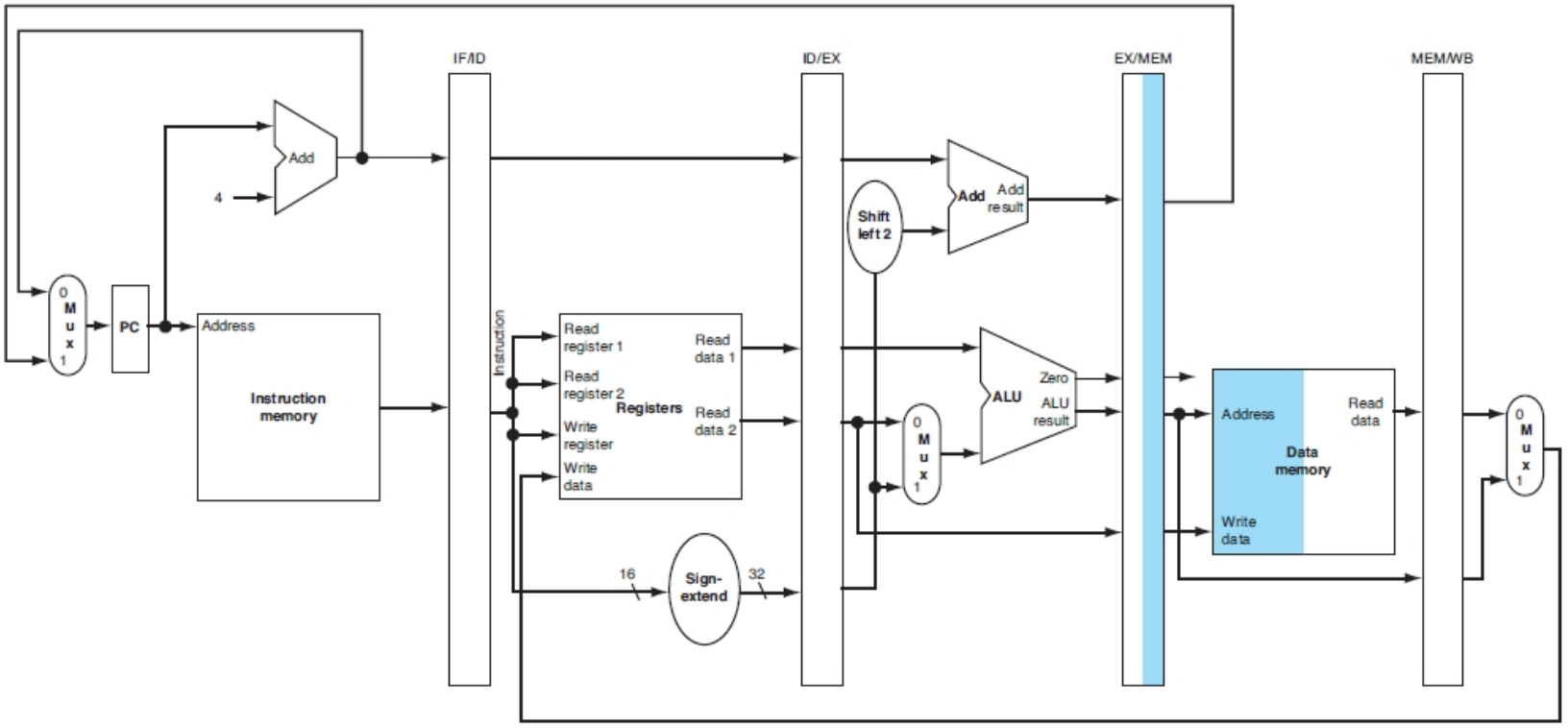
Pipeline Datapath

- Execute or Address calculation : เป็นขั้นตอนที่ข้อมูล Register ที่อ่านจาก Register File และข้อมูล Immediate ที่ทำ Signed-Extended แล้ว ซึ่งเก็บอยู่ใน ID/EX Pipeline Register นำมาประมวลผลโดย ALU และผลลัพธ์เก็บใน EX/MEM



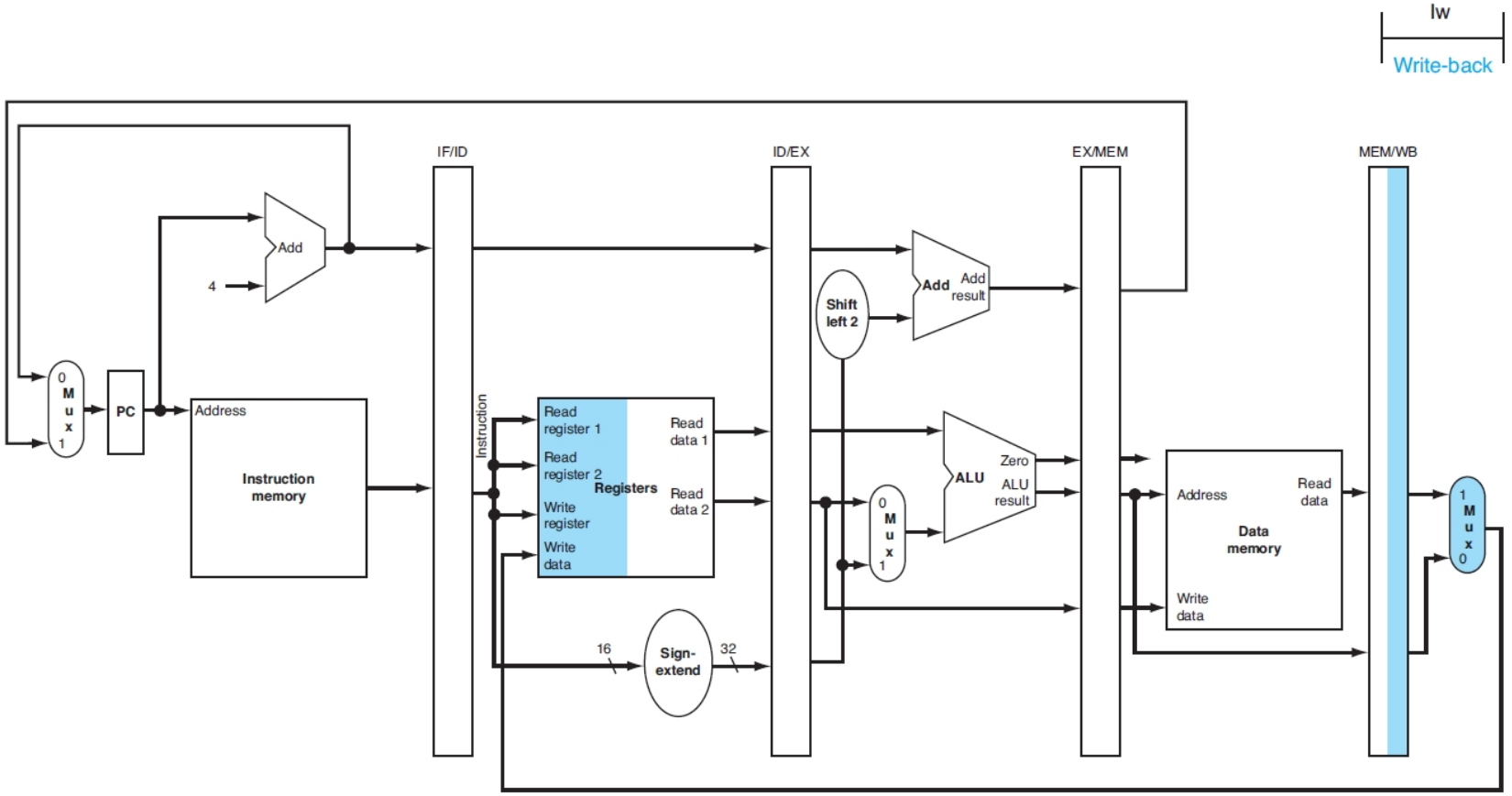
Pipeline Datapath

- Memory Access : เป็นขั้นตอนสำหรับคำสั่งที่เกี่ยวข้องกับหน่วยความจำ เช่น LDR โหลด Address จาก EX/MEM pipeline register และเก็บผลลัพธ์ลงใน MEM/WB pipeline register ในกรณีที่เป็นคำสั่ง R-Type จะ bypass ขั้นตอนนี้ (คำสั่ง STR จะจบที่ขั้นตอนนี้)



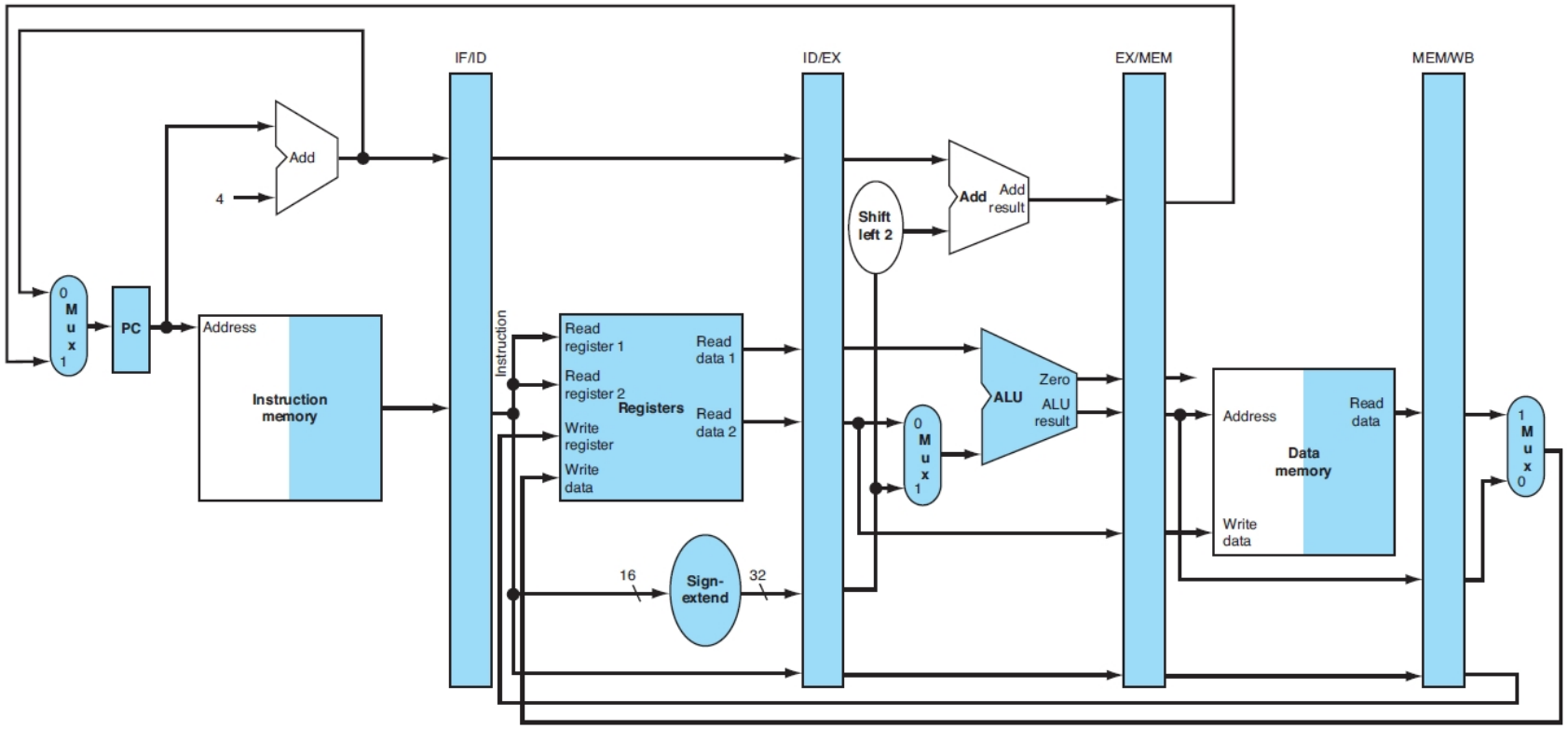
Pipeline Datapath

- Write-back : เป็นขั้นตอนสุดท้ายของ pipeline โดยจะอ่านข้อมูลจาก MEM/WB pipeline register และนำไปเขียนกลับใน register file เช่น LDR



Pipeline Datapath

- สรุปส่วนของ Datapath ทั้งหมด ที่ใช้ในคำสั่ง Load



Conflicts/Problem

- I-Cache และ D-Cache ถูก Access ใน clock cycle เดียวกัน ดังนั้นหากแยก I-Cache และ D-Cache ออกจากกัน ก็จะช่วยให้ทำงานได้ดี
- รีจิสเตอร์ จะมีทั้งอ่าน และ เขียน ใน clock cycle เดียวกัน ดังนั้นจึงต้องแบ่งการอ่านและการเขียนออกไปคนละครึ่ง clock cycle
- หากสามารถคำนวณ Branch Target ได้ใน stage ที่ 2 ส่วนที่เหลือจะทำยังไง

For your attention