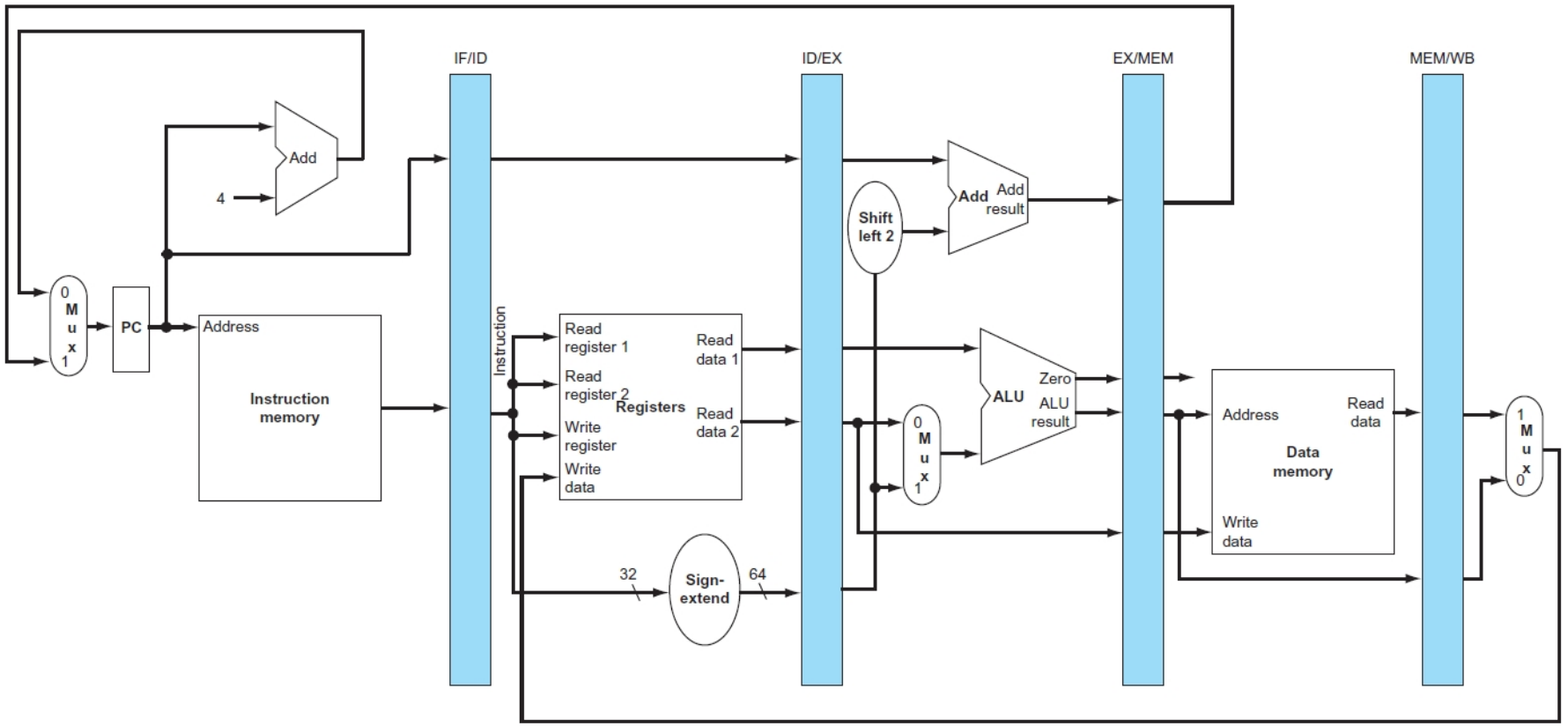
01076114
องค์ประกอบและสถาปัตยกรรมคอมพิวเตอร์
Computer Organization and Architecture

The Processor #2

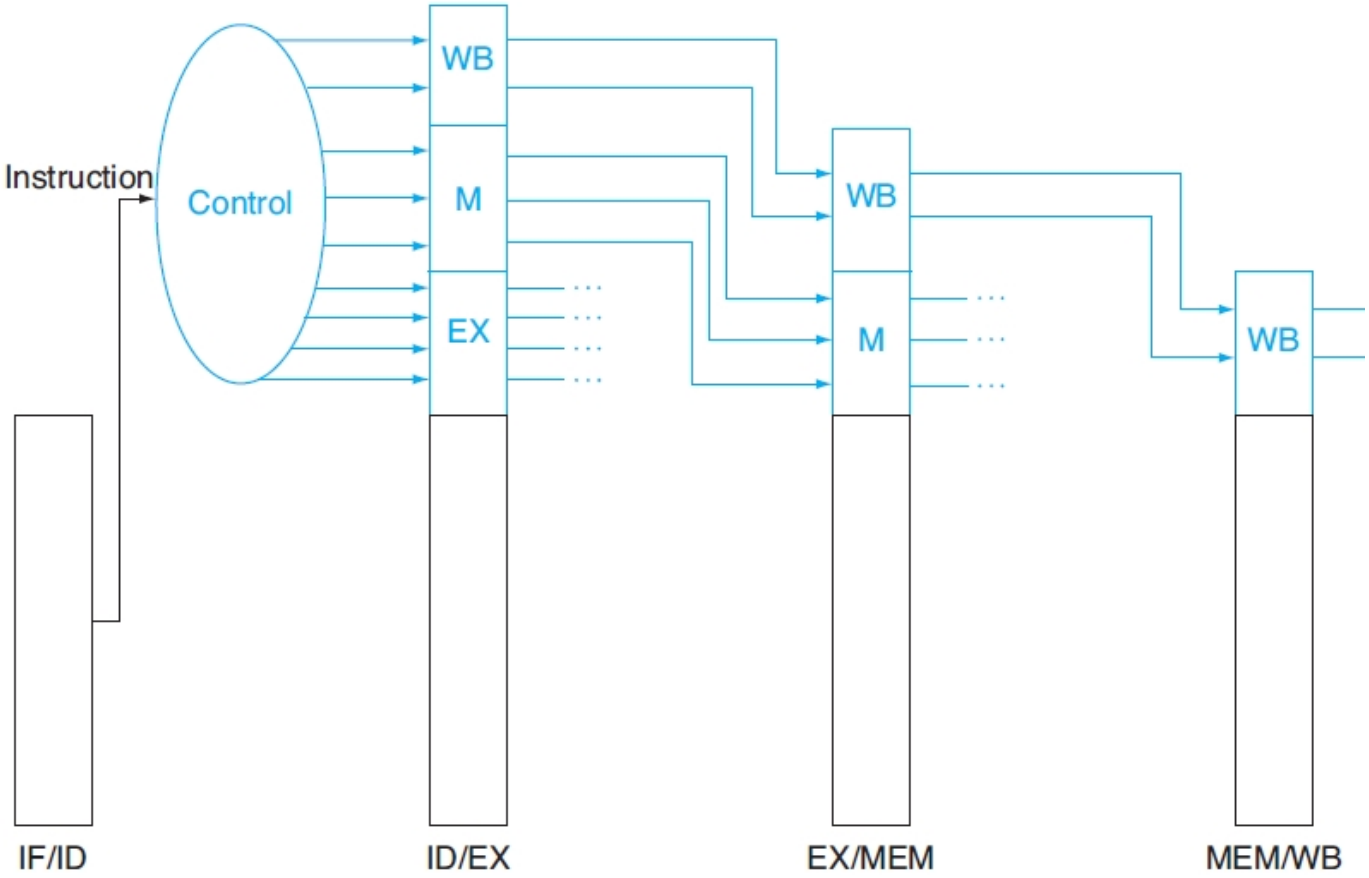
Pipeline Datapath

- การทำงานทั้ง 5 ส่วนใน pipeline จะแยกออกจากกันเด็ดขาด โดยมี pipeline register ทำหน้าที่เป็นตัวเชื่อม

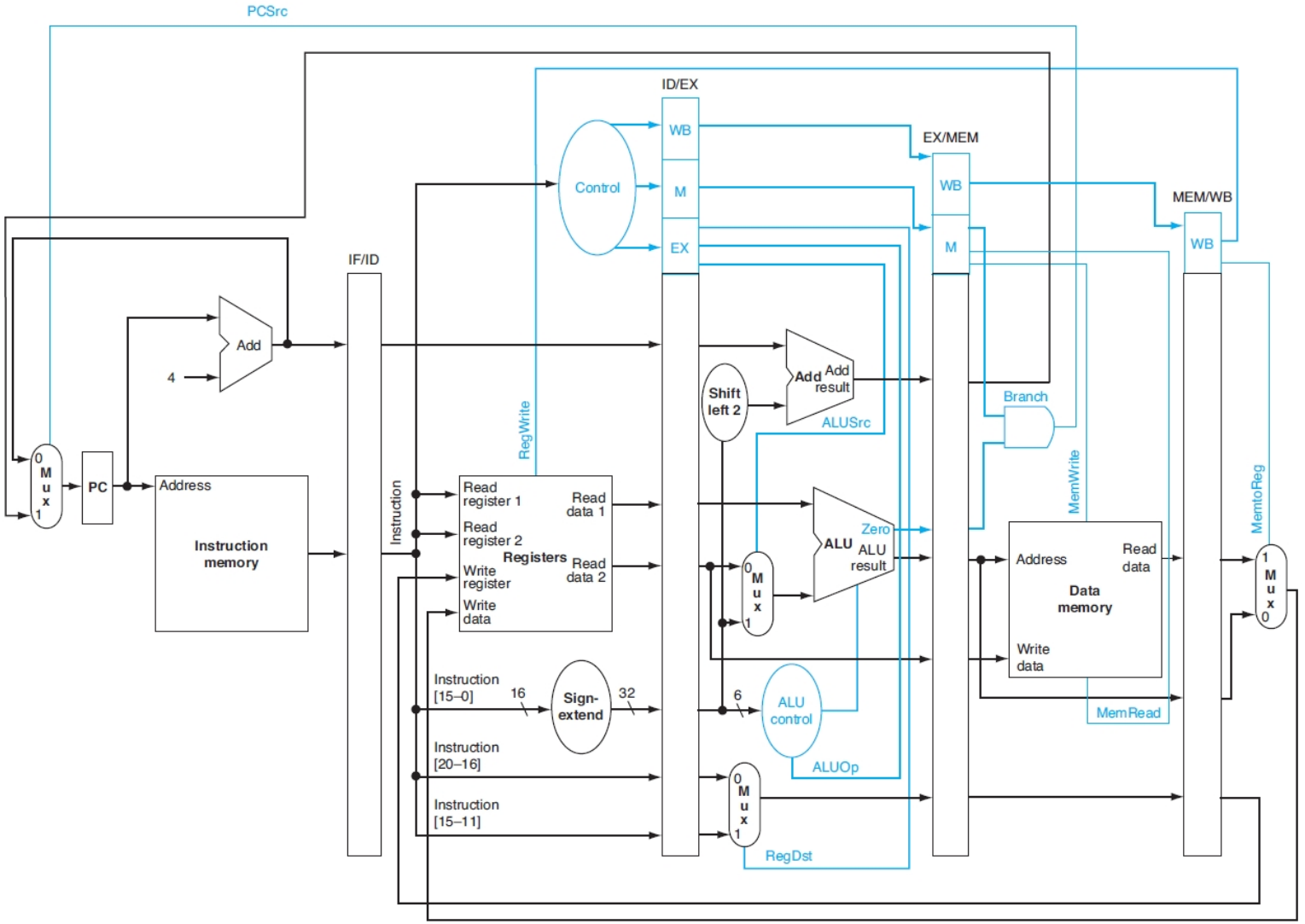


Pipeline Datapath

- ข้อมูลที่เก็บใน pipeline register จะต้องเก็บข้อมูลของสัญญาณ Control เอาไว้ด้วย โดยจะส่งต่อไปในแต่ละ stage ตามรูป



Pipeline Datapath



Hazards

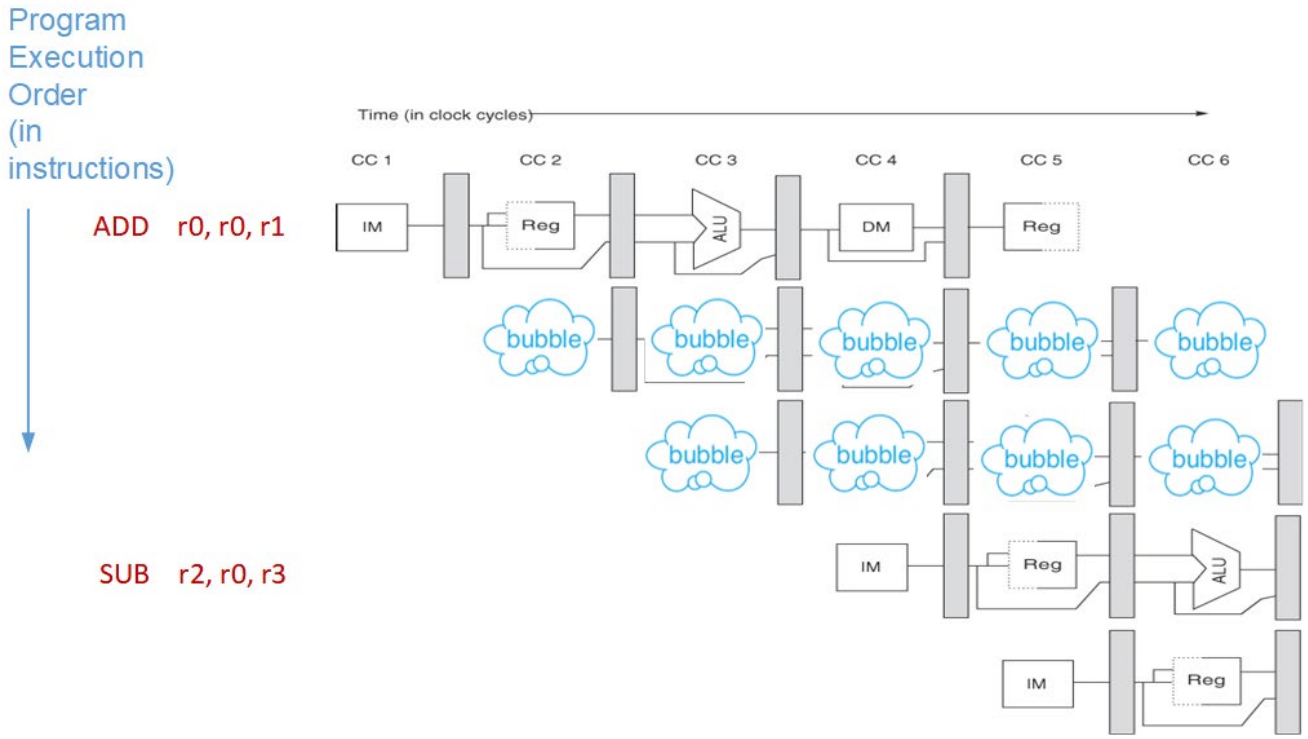
- Hazards คือ สิ่งที่เป็นอุปสรรคต่อการทำงานของ pipeline แบ่งเป็น
  - Structural Hazard เกิดจากคำสั่งตั้งแต่ 2 คำสั่งขึ้นไปใน pipeline แย่งกันใช้ทรัพยากรตัวเดียวกัน ทำให้ต้องเกิดการคอยกัน
  - Data Hazard คำสั่งที่อยู่ลำดับหลังของ pipeline ไม่สามารถทำงานต่อได้ เนื่องจากต้องรอค่าในรีจิสเตอร์ จากคำสั่งก่อนหน้า ซึ่งยังทำไม่เสร็จ เช่น มีคำสั่ง ADD r1, r1, r2 และ ADD r1, r1, r3 อยู่ติดกัน
  - Control Hazard เกิดจากคำสั่ง Branch เนื่องจาก pipeline ไม่ทราบว่าจะ Fetch คำสั่งอะไรต่อไป เพราะไม่ทราบว่าโปรแกรมจะ Branch ไปทางไหน

Structural Hazard

- เกิดจากคำสั่งตั้งแต่ 2 คำสั่งขึ้นไปใน pipeline แย่งกันใช้ทรัพยากรตัวเดียวกัน ทำให้ต้องเกิดการคอยกัน
- เช่น หากไม่มีการแยกเป็น I-Cache กับ D-Cache อาจจะมีบางครั้งที่ Stage 4 (MEM) อาจจะ Access memory พร้อมกับ Stage 1 (IF) ทำให้เกิดการรอกัน
- ทำให้คำสั่งที่รอใน pipeline ทั้งหมด จะต้อง delayed ไปจนกว่า ทรัพยากรที่รอจะว่างและสามารถเข้าใช้ได้ (เกิด pipeline stall หรือ pipeline bubble)
- Structural Hazard สามารถแก้ไขได้ไม่ยาก โดยการเพิ่มทรัพยากรเข้าไป เช่น มีปัญหาในการแย่งใช้หน่วยความจำแคช ก็แยกจาก 1 แคช เป็น 2 แคช คือ I-Cache กับ D-Cache หรือ เกิดจากรีจิสเตอร์ไม่พอ ก็เพิ่มจำนวนรีจิสเตอร์ เป็นต้น

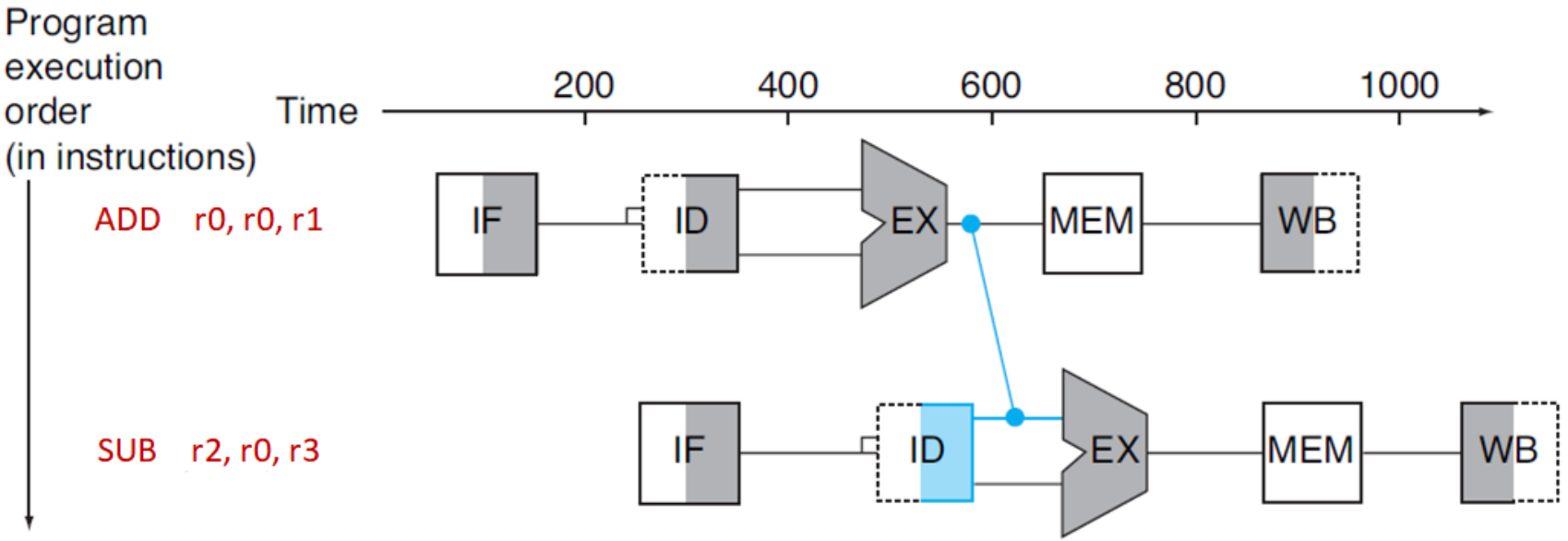
Data Hazard

- เมื่อคำสั่งหนึ่งใน pipeline สร้างผลลัพธ์ขึ้น แล้วคำสั่งที่ตามมา จำเป็นต้องใช้ผลลัพธ์นั้น เช่น
  - add r0, r0, r1
  - sub r2, r0, r3
- ทำให้คำสั่งถัดมาต้องรอคำสั่งก่อนหน้า



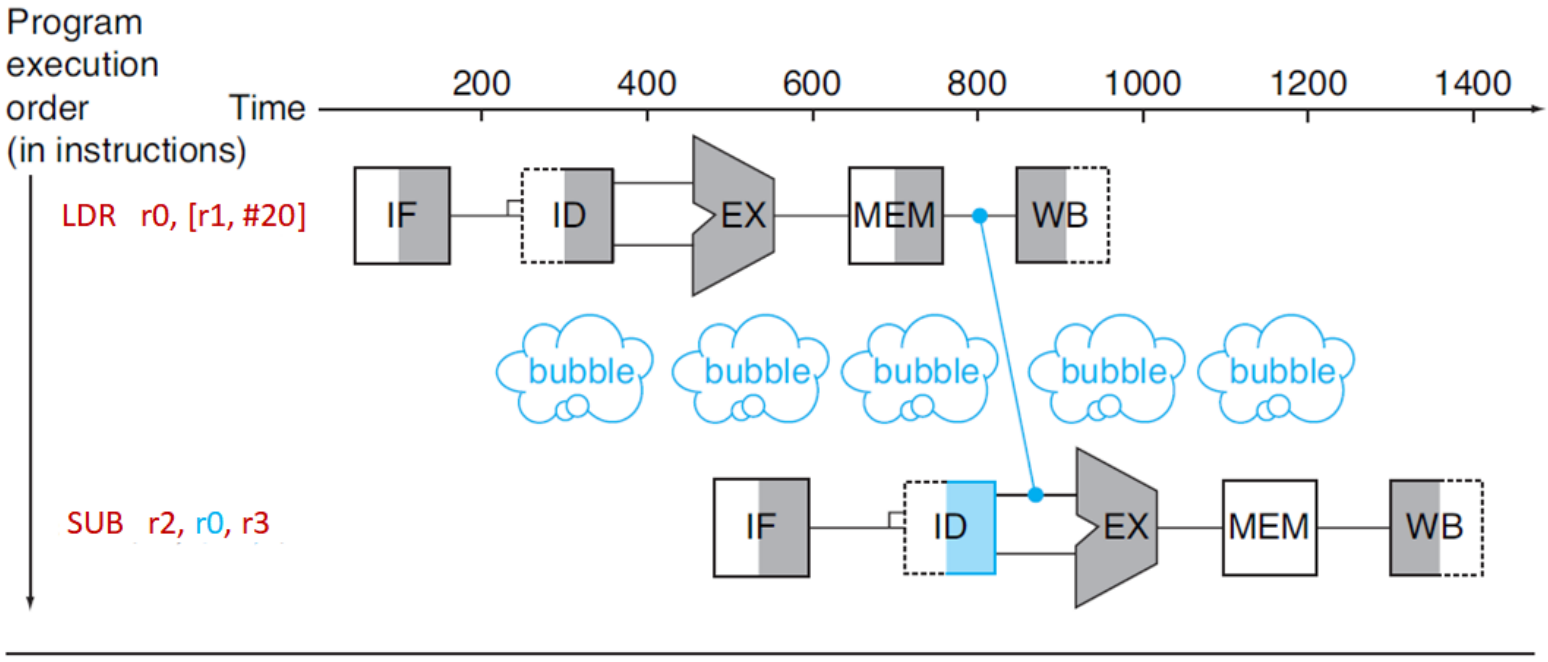
Data Hazard

- การแก้ไข Data Hazard สามารถทำได้ โดยใช้ Forwarding หรือ Bypassing
- Forwarding เป็นวิธีการส่งค่าผลลัพธ์จาก ALU ไปยัง Pipeline ถัดไปโดยตรง ก่อนที่จะ write ลง register จากรูปจะเห็นว่ามีการส่งค่าของรีจิสเตอร์ r0 ไปให้กับคำสั่งถัดไปจากผลลัพธ์ของ ALU

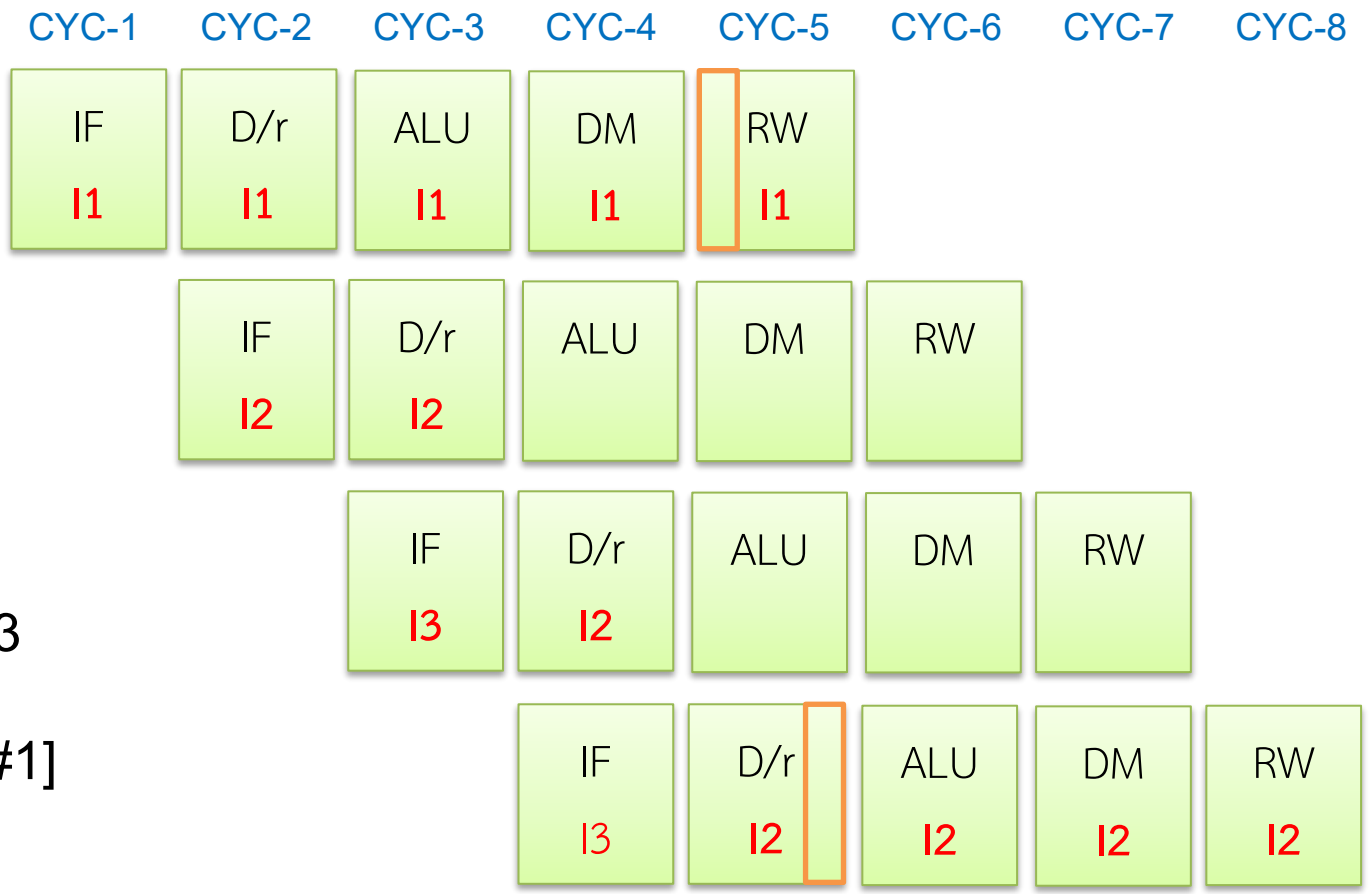


Data Hazard

- การแก้ไขโดยใช้ Forwarding จะสามารถทำได้ กรณีที่ผลลัพธ์เกิดขึ้นก่อน การนำไปใช้เท่านั้น หมายความว่า คำสั่งที่สามารถใช้ Forwarding ต้องเป็นคำสั่งประเภท R-Type เท่านั้น เพราะคำสั่งที่ต้องใช้ข้อมูลในหน่วยความจำ ถ้าดูจาก pipeline จะอยู่ในลำดับเดียวกัน ก็ยังเกิด stall อยู่ดี



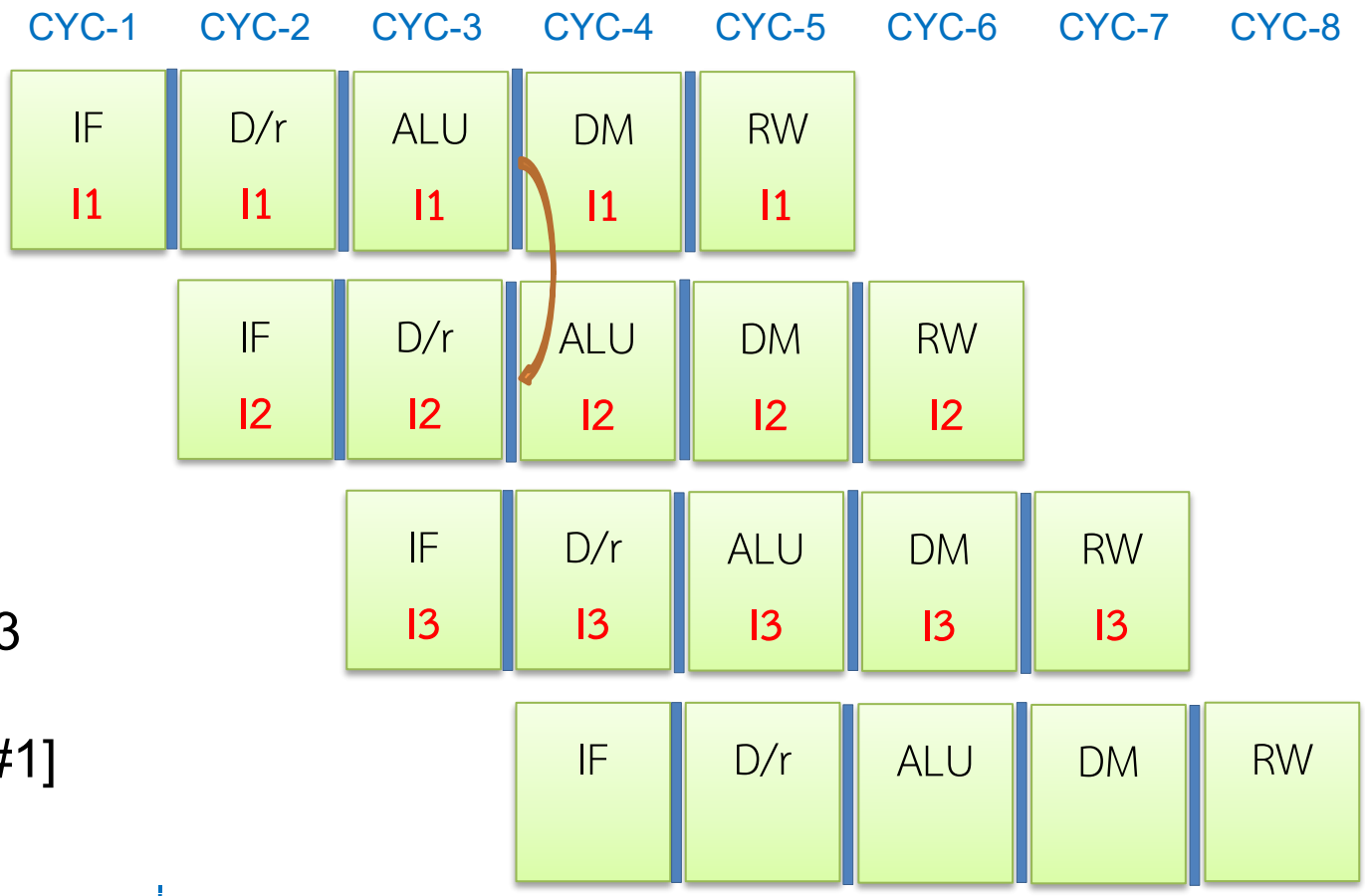
Example : No forwarding



ADD r1, r2, r3

LDR r4, [r1,#1]

Example : Forwarding



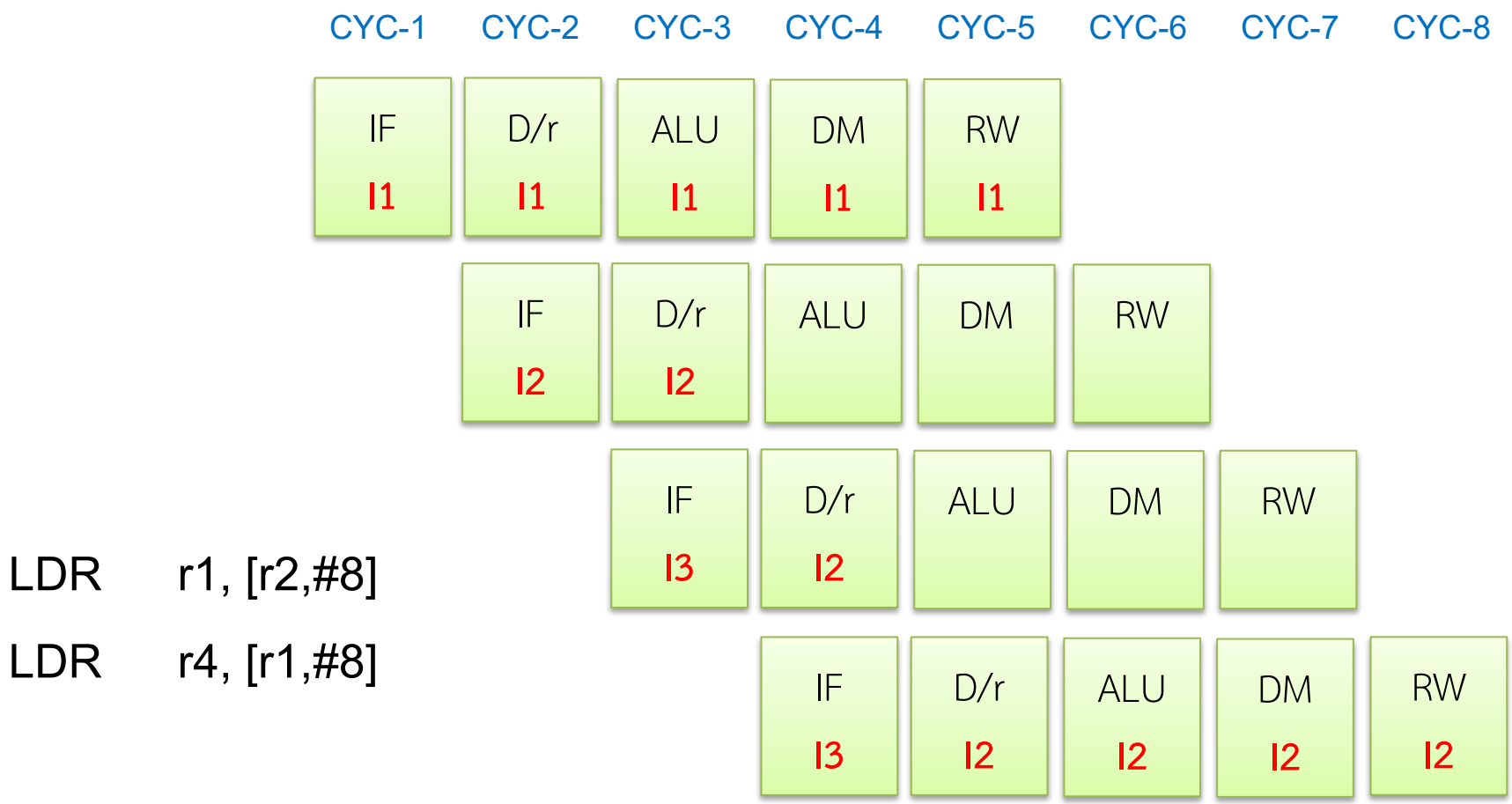
ADD r1, r2, r3

LDR r4, [r1,#1]

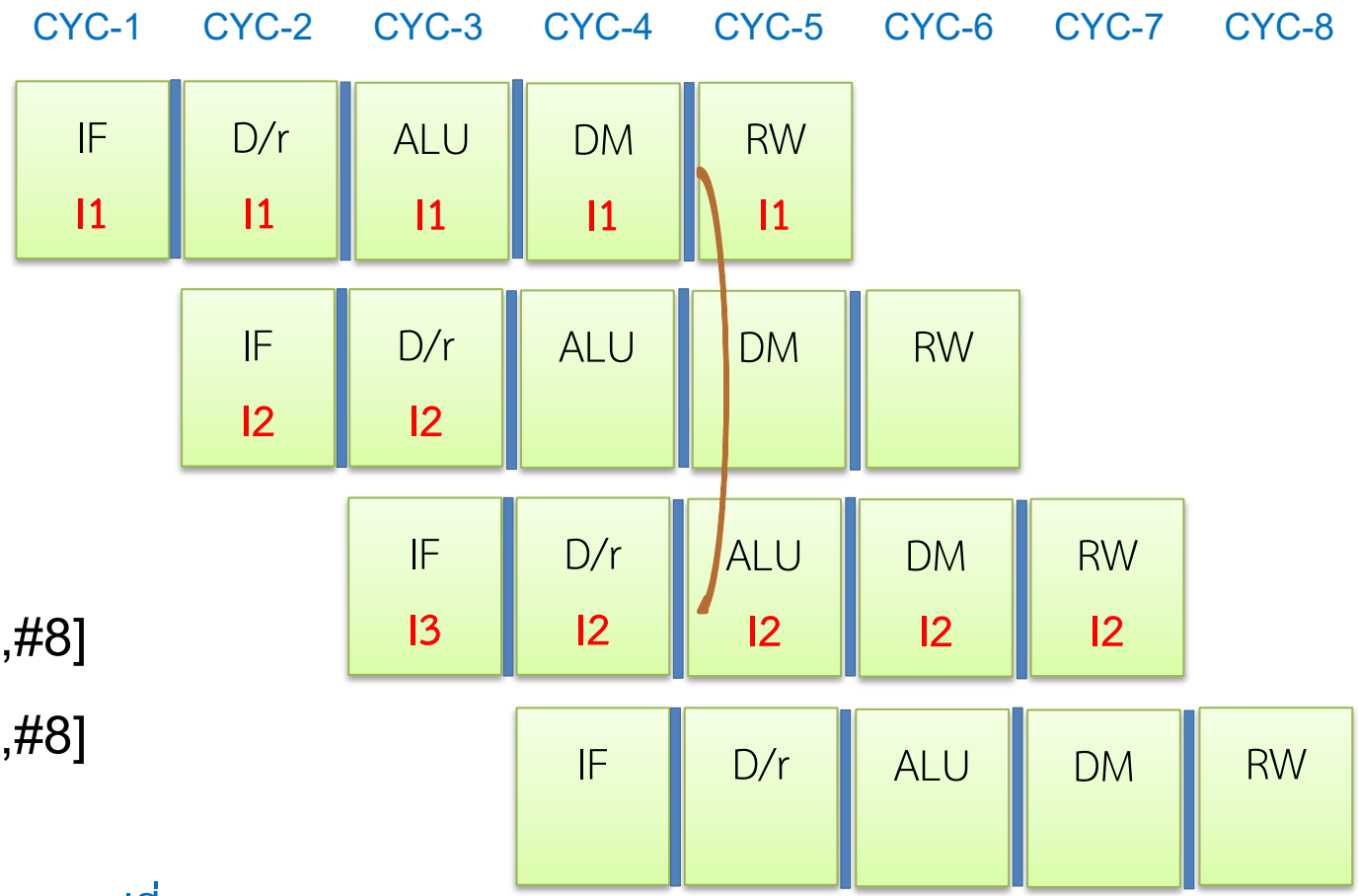
Point of Production อยู่ที่ EX/MEM

ALU จะต้องมี MUX สำหรับเลือก Input ว่าจะมาจาก EX/MEM หรือ ID/EX

Example 2 : No forwarding



Example 2 : Forwarding

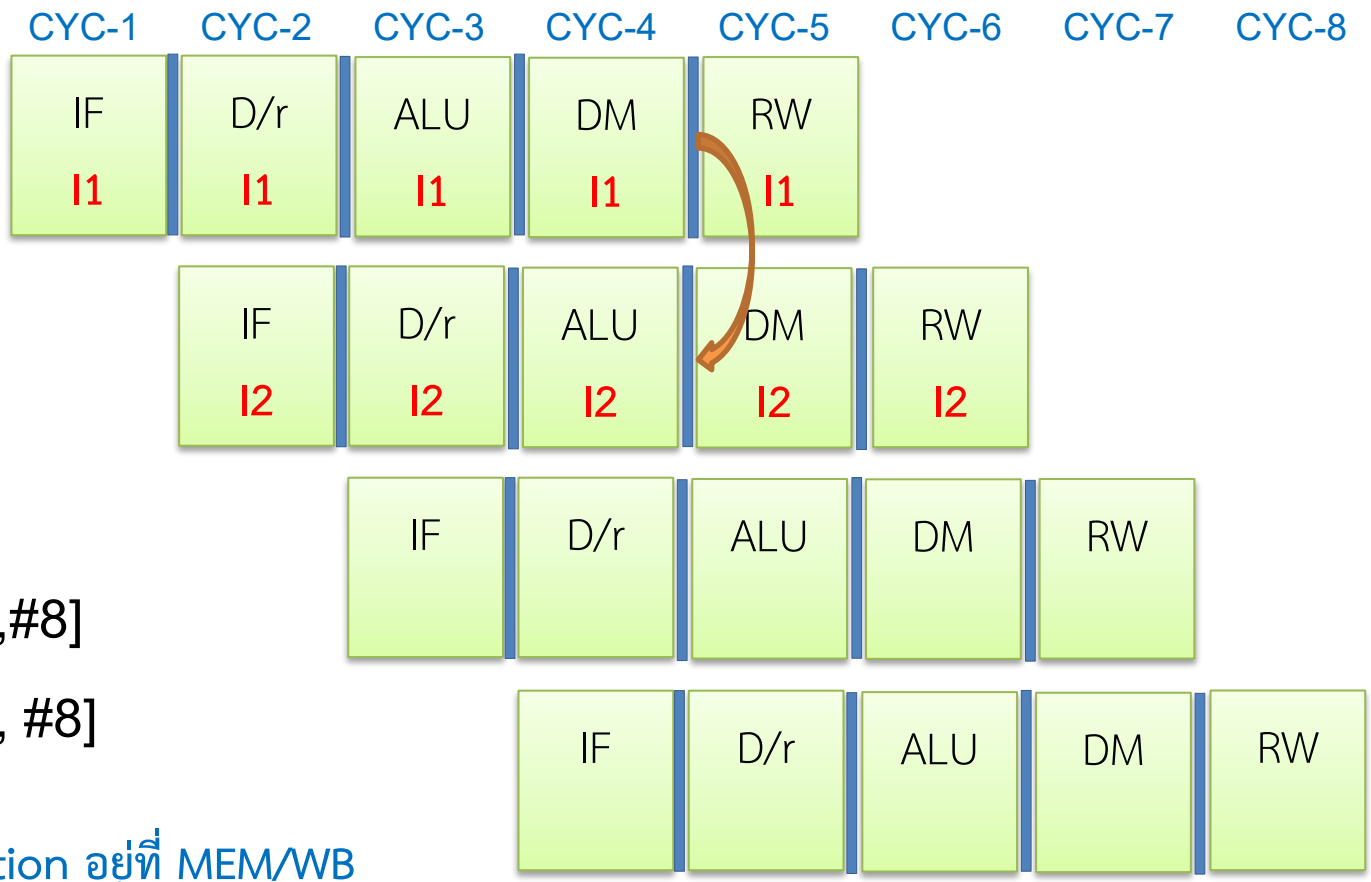



```
LDR     r1, [r2,#8]
LDR     r4, [r1,#8]
```

Point of Production อยู่ที่ MEM/WB

ALU จะต้องมี MUX สำหรับเลือก Input ว่าจะมาจาก MEM/WB หรือ ID/EX หรือ EX/MEM

Example 3 : Forwarding




```
LDR     r1, [r2,#8]
STR     r1, [r3, #8]
```

Point of Production อยู่ที่ MEM/WB

ALU จะต้องมี MUX สำหรับเลือก Input ว่าจะมาจาก MEM/WB หรือ EX/MEM

ถ้าเปลี่ยนคำสั่งที่ 2 เป็น STR r2, [r1, #8]

Data Hazard

- อีกวิธีการหนึ่งที่สามารถแก้ไข Data Hazard ได้ คือ การจัดลำดับ code ลองพิจารณาโปรแกรมดังนี้

```
a = b + e;
c = b + f;
```

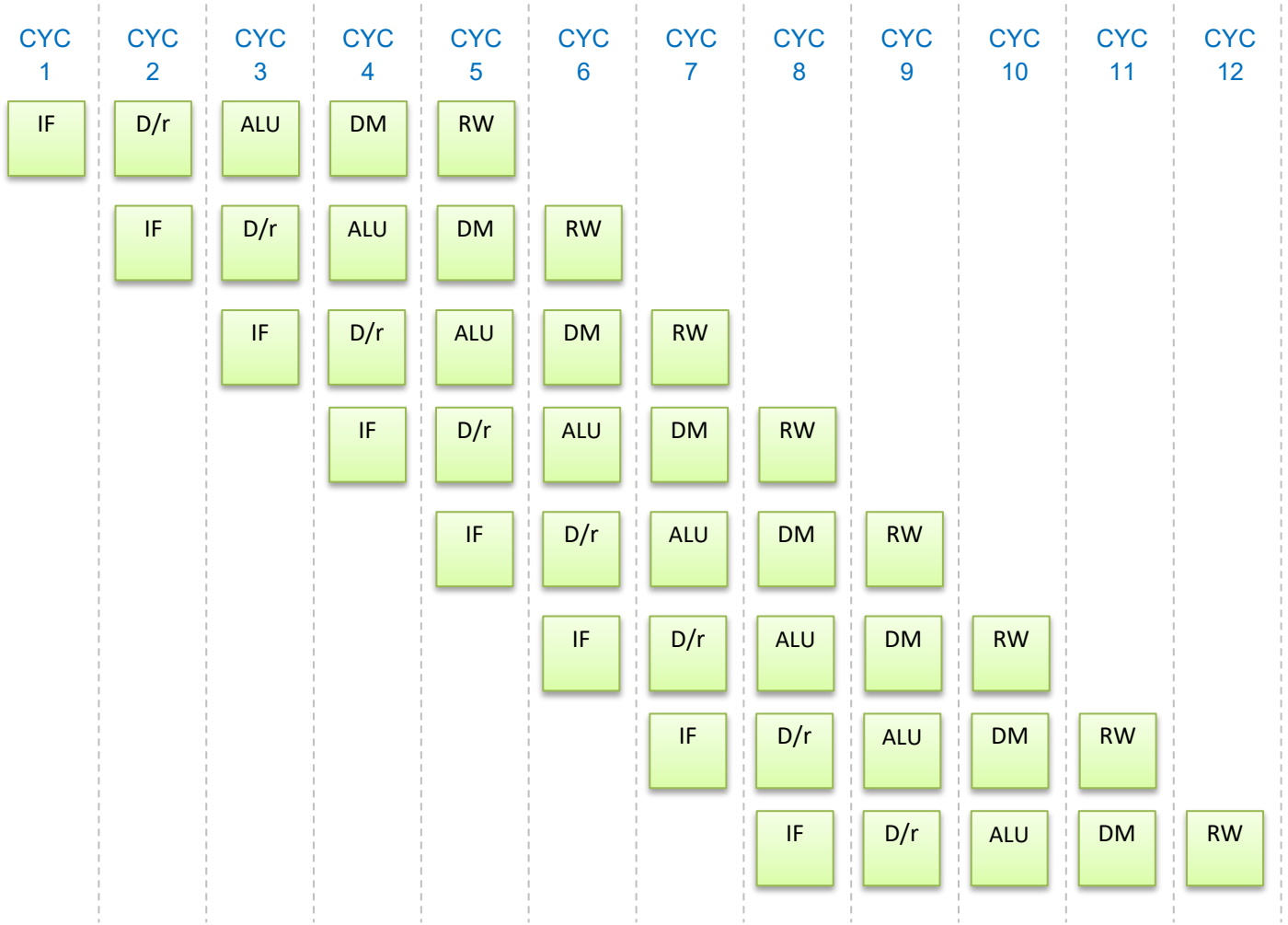
- โปรแกรมข้างต้นเมื่อแปลงเป็นภาษา assembly จะได้

```
LDR   r1, [r0,#0]      @ b
LDR   r2, [r0,#4]      @ e
ADD   r3, r1, r2       @ add
STR   r3, [r0, #12]    @ store a
LDR   r4, [r0, #8]     @ f
ADD   r5, r1, r4
STR   r5, [r0, #16]    @ c
```

- เกิด pipeline stall หรือไม่? ที่บรรทัดใด?

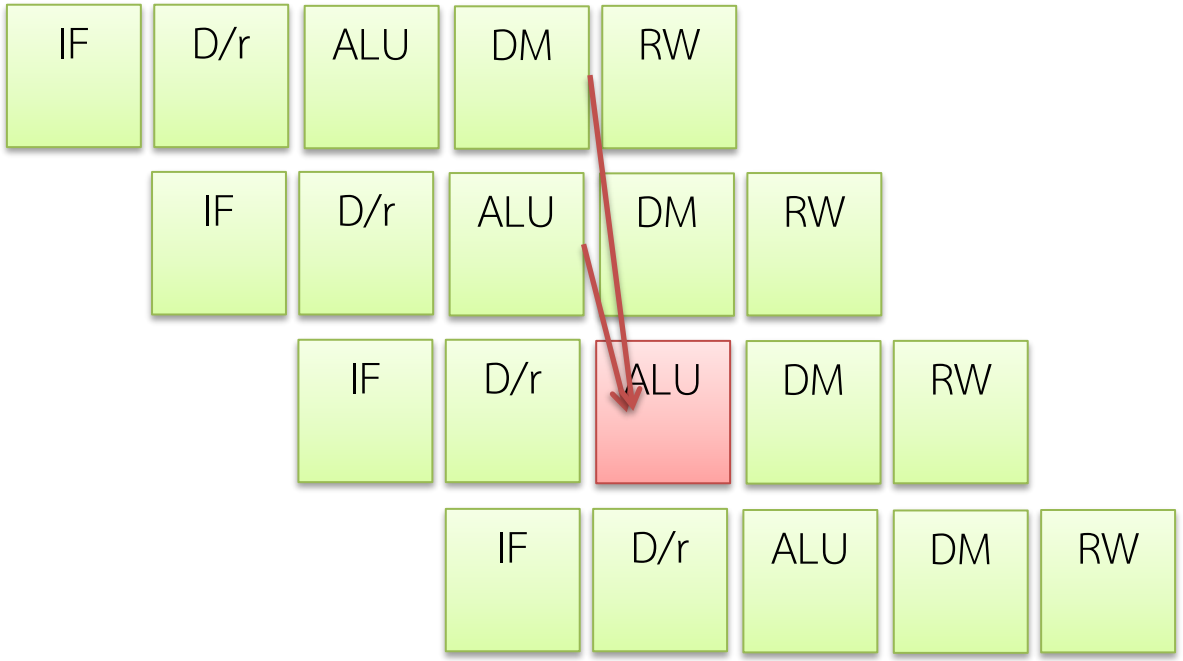
Data Hazard

- ใช้โปรแกรมหน้าที่แล้วเขียนแบบ No forwarding



Data Hazard

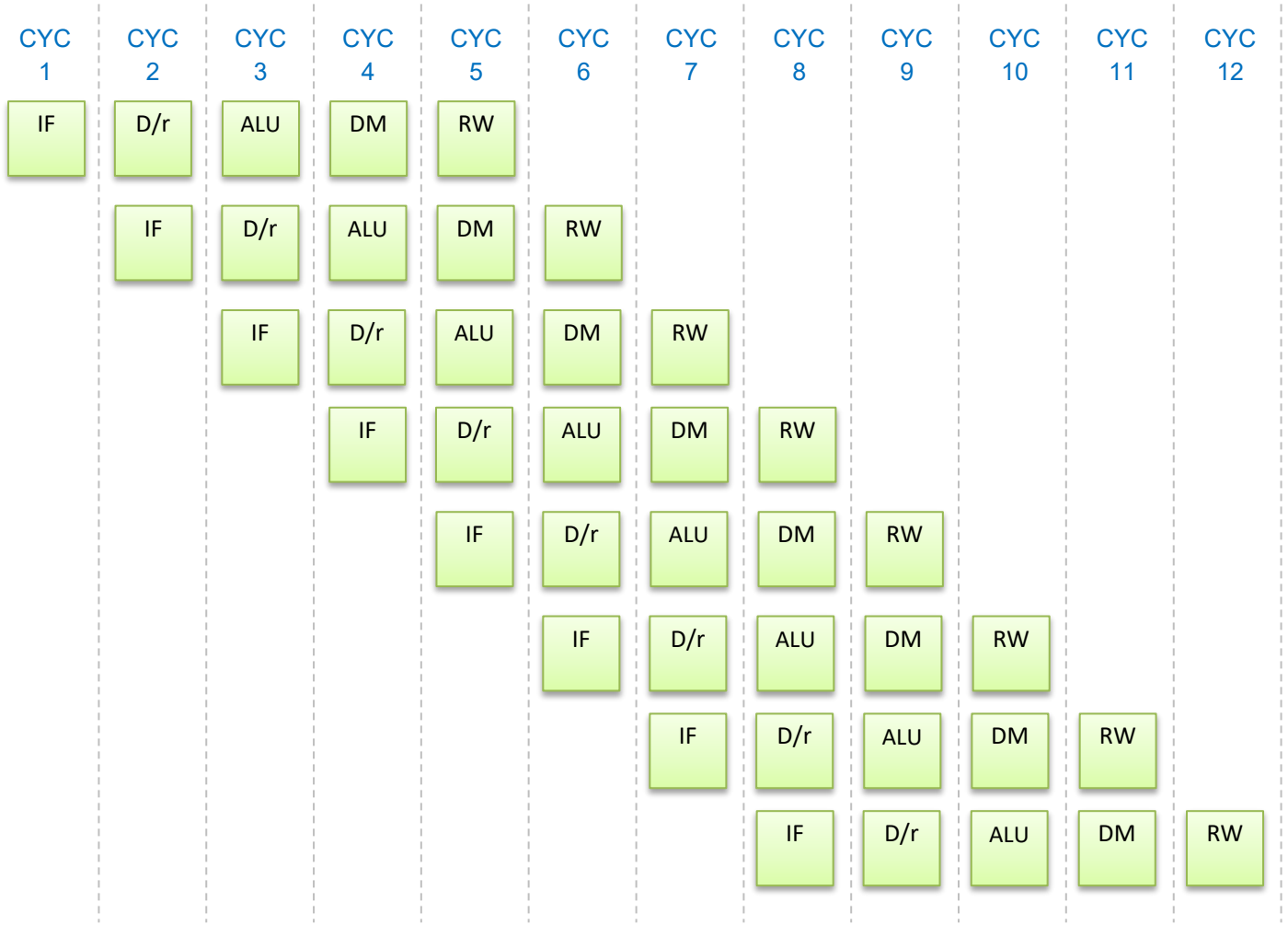
- `LDR r1, [r0,#0]`
- `LDR r2, [r0,#4]`
- `ADD r3, r1, r2`
- `STR r3, [r0, #12]`



- จะเห็นว่าที่คำสั่ง ADD ของทั้ง 2 ที่เกิด Data Hazard โดย ADD แรก จะรอ r1,r2 และ ADD หลังจะรอ r4
- ที่ ALU จะต้องใช้ r1 ซึ่งจะมาจาก MEM/WB และ r2 มาจาก EX/MEM

Data Hazard

- ใช้โปรแกรมหน้าที่แล้วเขียนแบบ Forwarding



Data Hazard

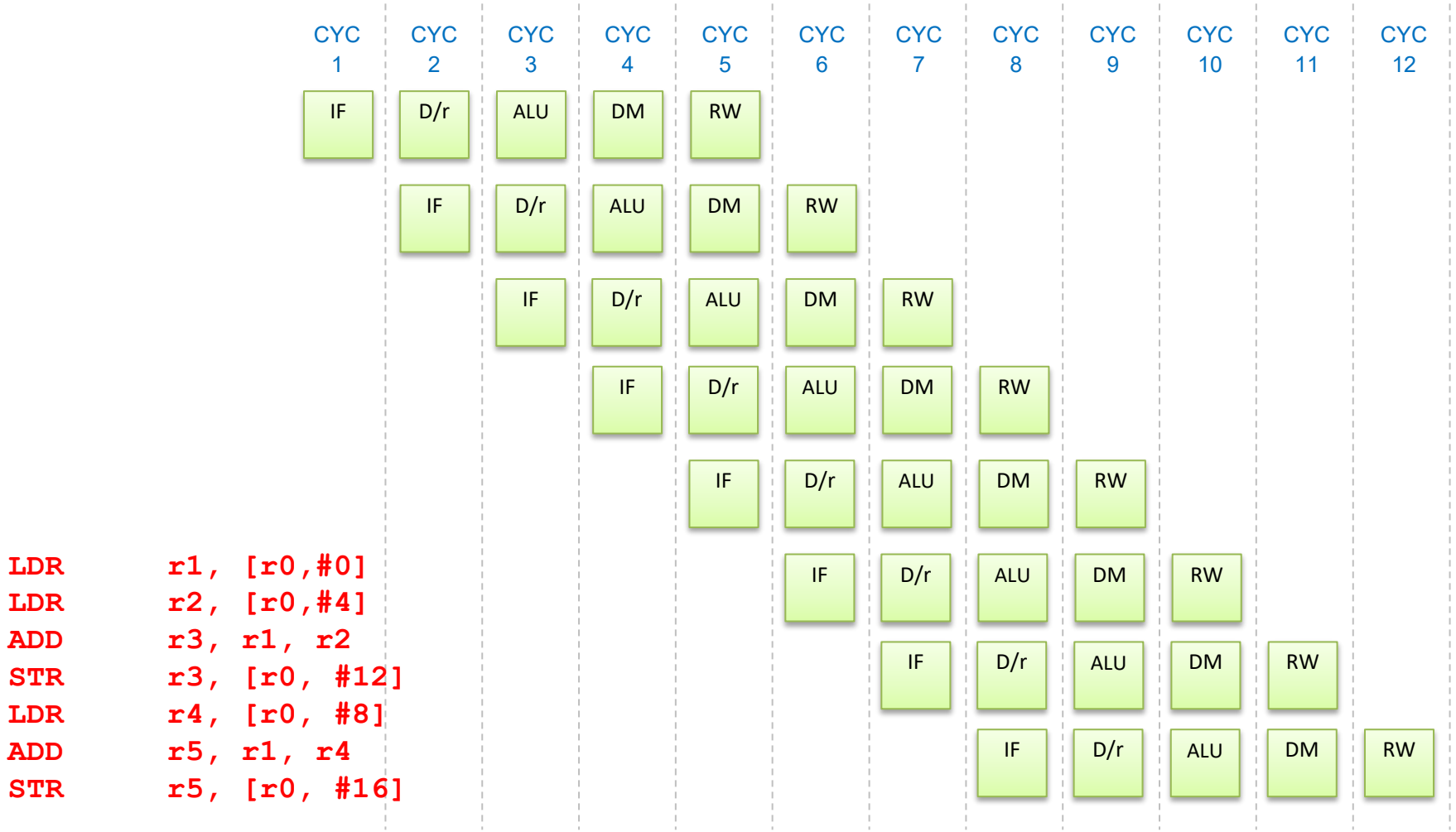
- แต่หากเรา reorder โปรแกรมใหม่ดังนี้

```
LDR   r1, [r0,#0]      @ b
LDR   r2, [r0,#4]      @ e
LDR   r4, [r0, #8]     @ f
ADD   r3, r1, r2       @ add
STR   r3, [r0, #12]    @ store a
ADD   r5, r1, r4
STR   r5, [r0, #16]    @ c
```

- จะเห็นว่า จะไม่เกิด pipeline stall อีก ทำให้โปรแกรมนี้จะทำงานได้เร็วกว่าเวอร์ชันก่อนหน้า 2 clock cycle
- **หากไม่ใช่ forwarding จะเกิด stall หรือไม่?**

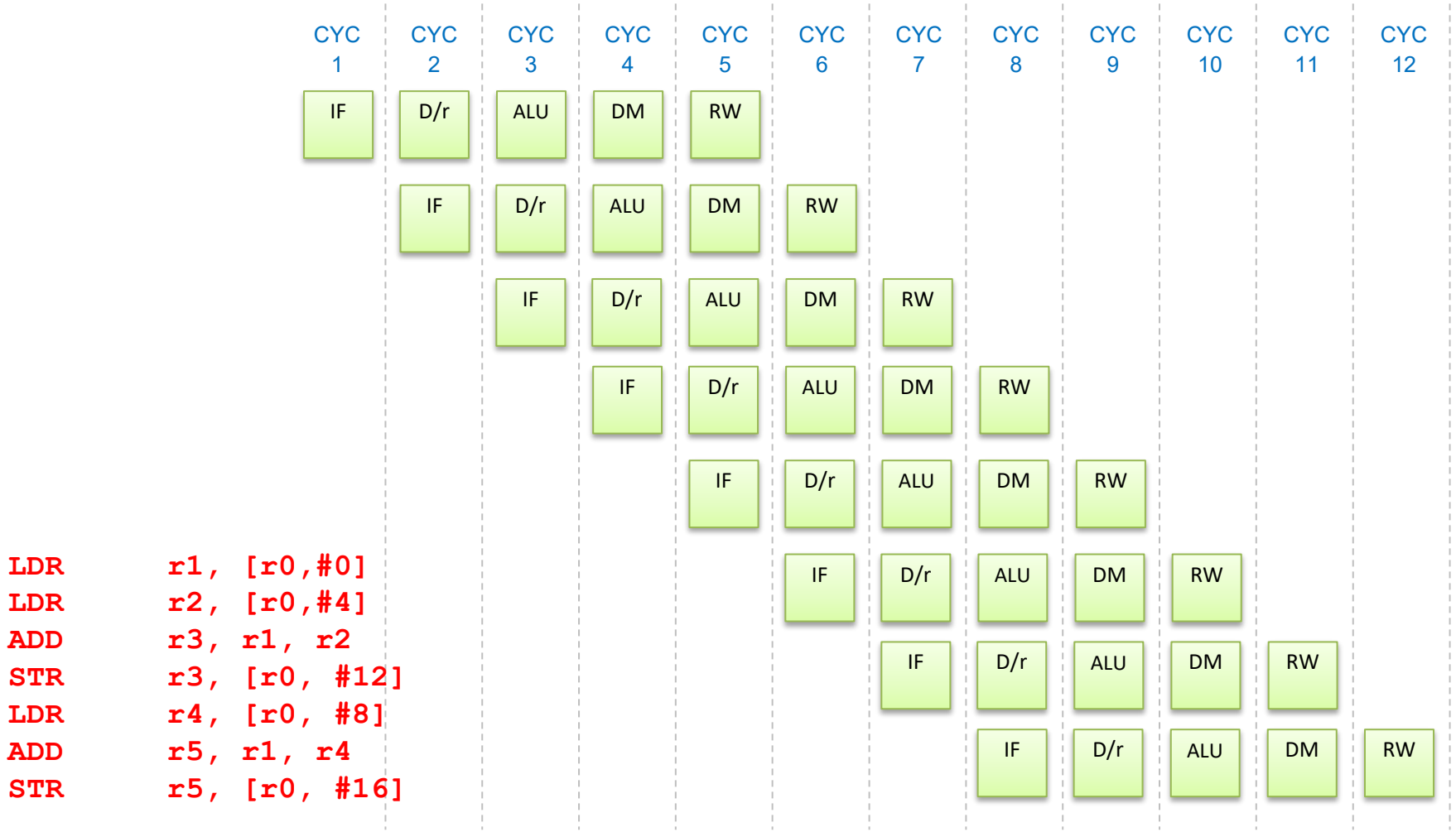
Data Hazard

- ใช้โปรแกรมหน้าที่แล้วเขียนแบบ No forwarding



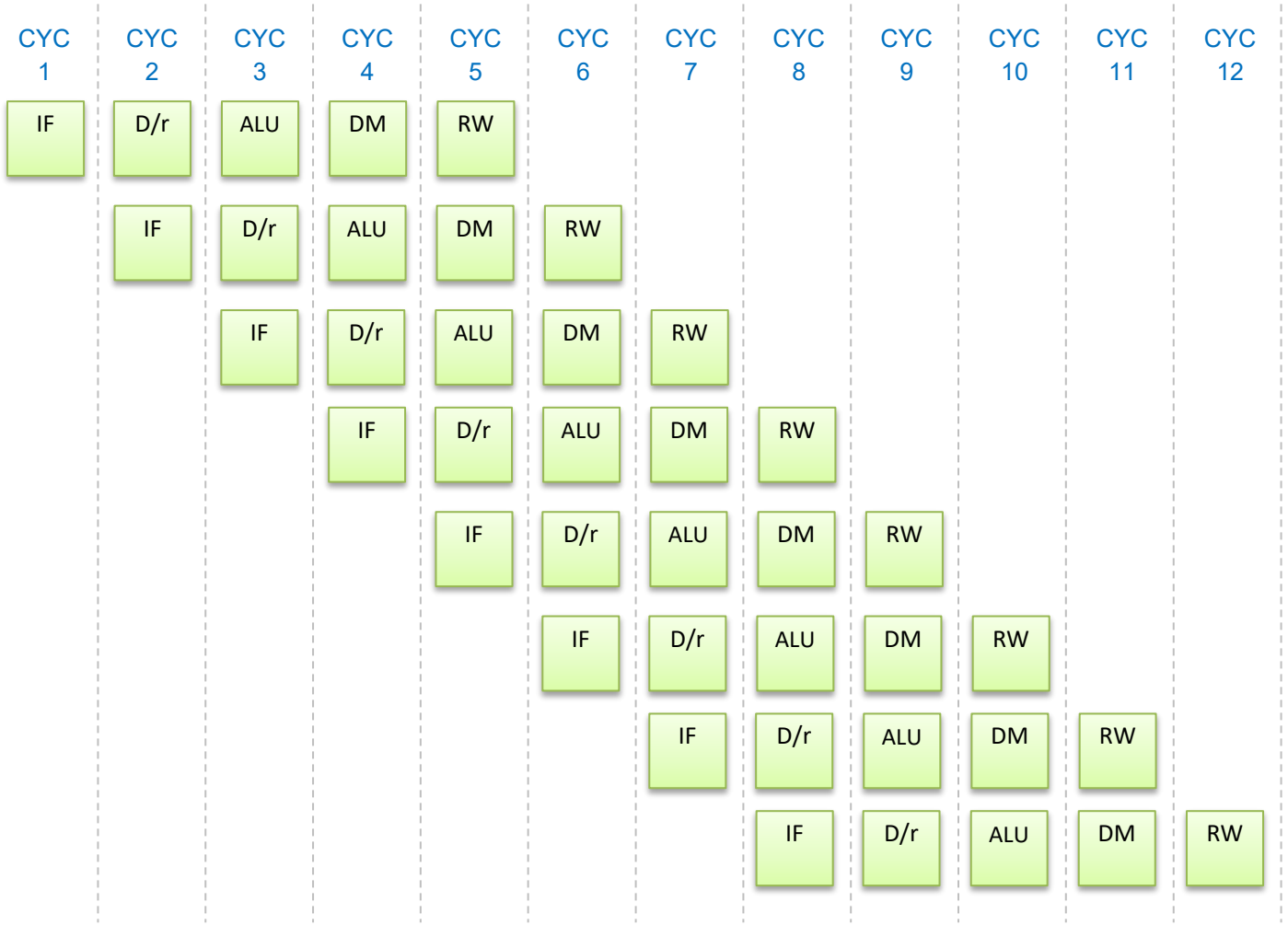
Data Hazard

- ใช้โปรแกรมหน้าที่แล้วเขียนแบบ Forwarding



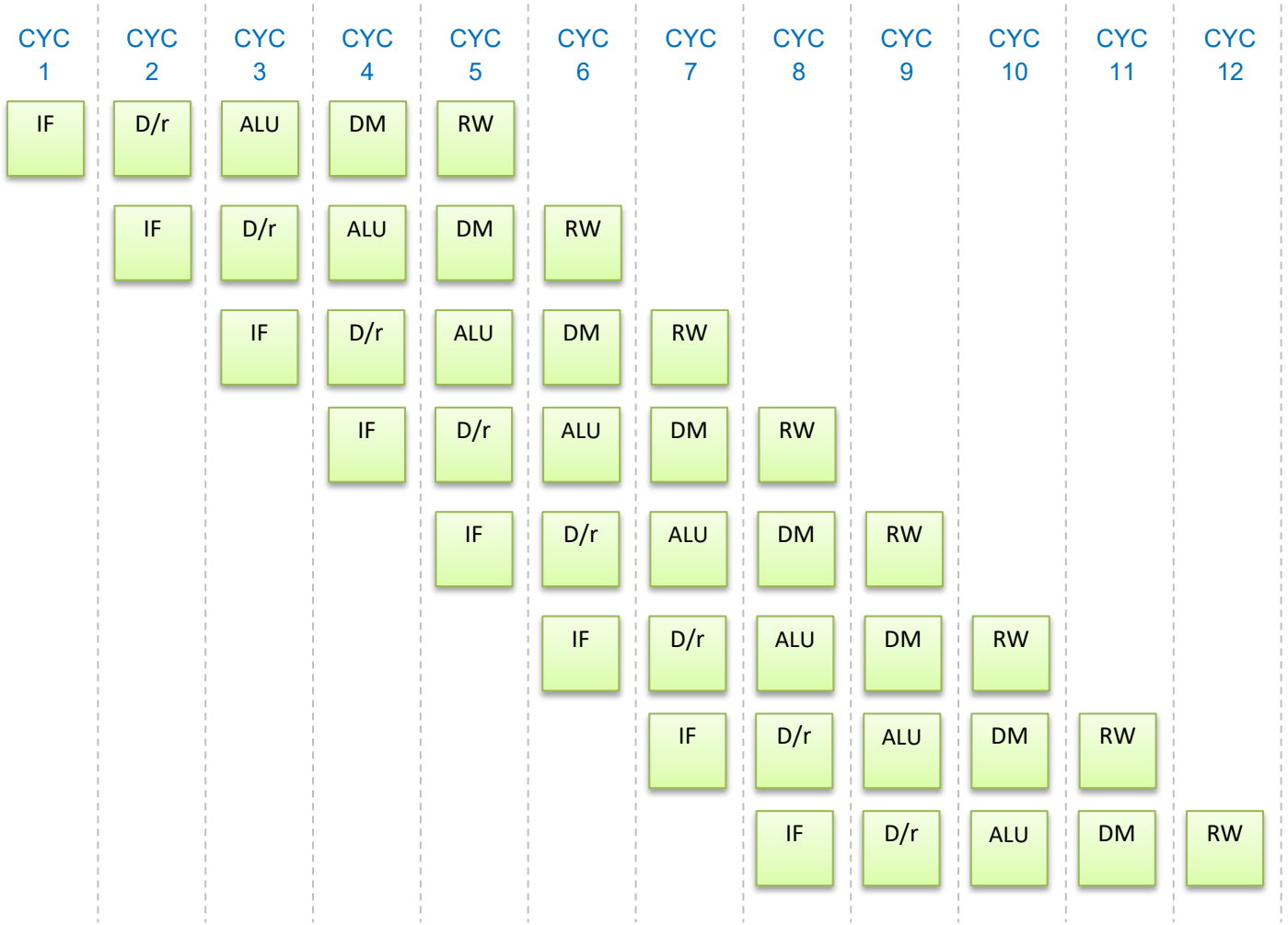
Data Hazard - Homework

- กำหนดคำสั่ง ADD r3,r1,r2 ADD r5,r3,r4 ADD r9, r3, r8 ให้แสดงการ execute ใน pipeline โดยไม่มี forwarding ให้คำนวณค่า CPI ด้วย



Data Hazard - Homework

- กำหนดคำสั่ง ADD r3,r1,r2 ADD r5,r3,r4 ADD r9, r3, r8 ให้แสดงการ execute ใน pipeline โดยมี forwarding ให้คำนวณค่า CPI



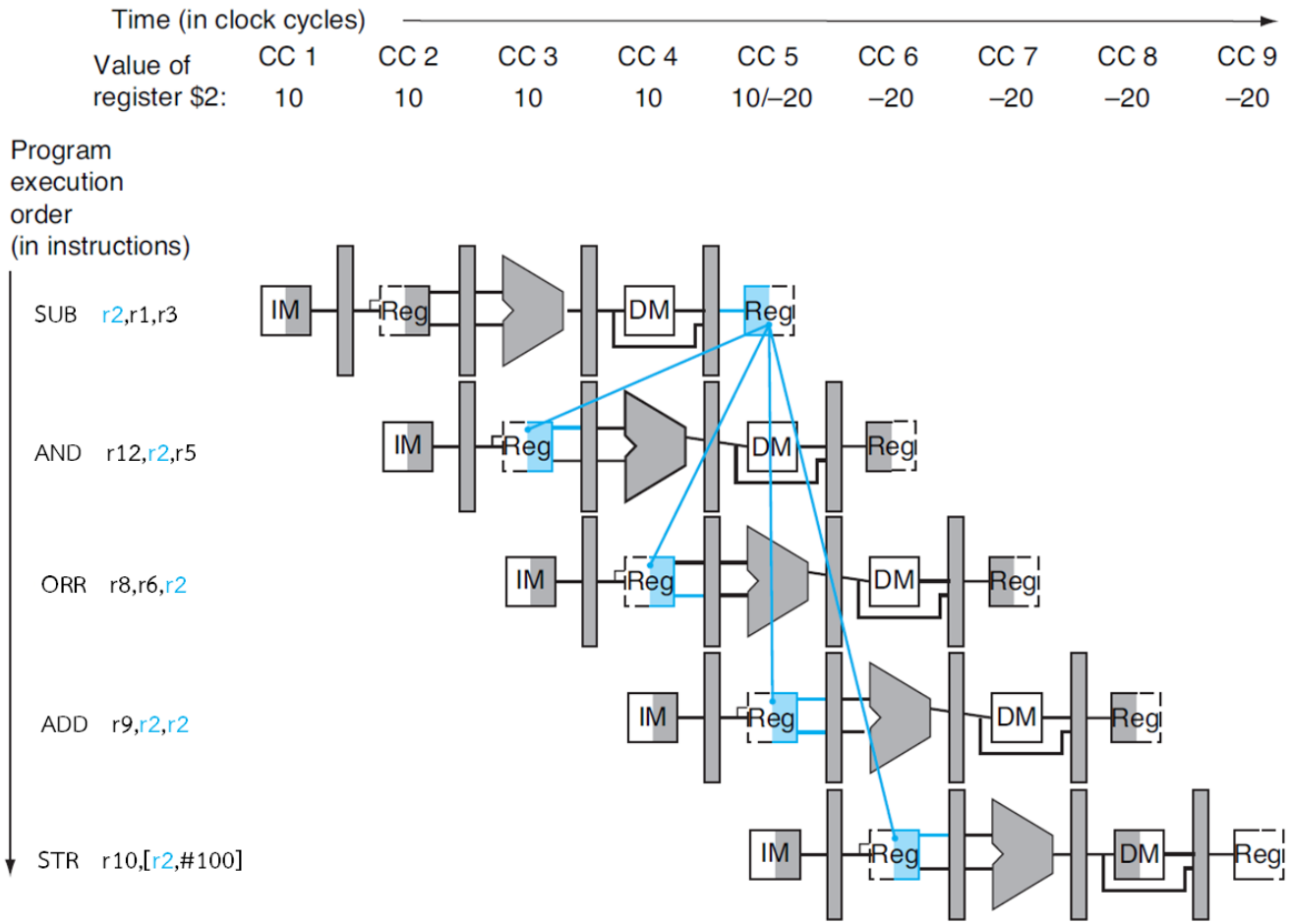
Data Hazard Detection

- ต่อไปจะแสดงการตรวจสอบว่าเกิด Hazard ขึ้น จาก code จะเห็นว่า 4 คำสั่งหลัง มีการใช้รีจิสเตอร์ r2 ที่อยู่ในคำสั่งแรก สมมติว่ารีจิสเตอร์ r2 เดิมมีค่า =10 และเมื่อได้รับค่าจาก r1-r3 แล้ว =-20 ดังนั้น r2 ที่ใช้ในคำสั่งต่อมา ต้องเป็น -20

```
SUB     r2, r1,r3       @ register r2 written by SUB
AND     r12,r2,r5       @ 1st operand(r2) depends on SUB
ORR     r8,r6,r2        @ 2nd operand(r2) depends on SUB
ADD     r9,r2,r2        @ 1st(r2) & 2nd(r2) depend on SUB
STR     r10,[r2,#100]   @ Base (r2) depends on SUB
```

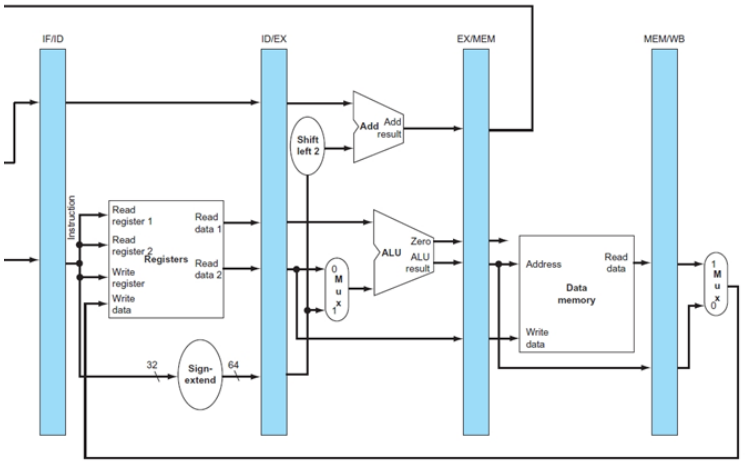
Data Hazard Detection

- แสดงการ execute คำสั่ง จะเห็นว่า r2 จะมีการเปลี่ยนค่าที่ cc 5 โดยเส้นสีฟ้า แสดงการอ้างอิง r2 จะเห็นว่า 2 คำสั่งหลัง ไม่มีปัญหา Data Hazard



Data Hazard Detection

- Point of production ของ r2 จะอยู่ที่ EX stage และเก็บไว้ที่ EX/MEM ณ จุดสิ้นสุดของ cc 3
- คำสั่ง AND ต้องการข้อมูล r2 ที่ EX stage ใน cc 4 ซึ่งปกติจะใช้ข้อมูลจาก ID/EX
- แต่ก่อนอื่นต้องตรวจสอบให้ได้ก่อนว่า เกิด Hazard ขึ้น โดยตรวจสอบ dependent ของรีจิสเตอร์ ซึ่งก็คือการที่รีจิสเตอร์ของคำสั่งก่อนหน้า ถูกใช้ในคำสั่งถัดไป
- เงื่อนไขของการเกิด Hazard จะมี 4 กรณี ดังนี้



1. EX/MEM.registerRd = ID/EX.registerRn
2. EX/MEM.registerRd = ID/EX.registerRm
3. MEM/WB.registerRd = ID/EX.registerRn
4. MEM/WB.registerRd = ID/EX.registerRm

- จะเห็นว่าที่ cc3/cc4 ID/EX.registerRn (คำสั่ง AND) = r2 และ EX/MEM.registerRd (คำสั่ง SUB) = r2 เช่นเดียวกัน จึงเกิด Data Hazard ตามเงื่อนไขที่ 1

Data Hazard Detection

- สำหรับคู่ SUB – ORR จะใช้ข้อมูลใน cc5 โดยเป็น Input ที่ 2 ของ ORR (ID/EX.register2) โดยข้อมูล r2 ที่ถูกต้อง (-20) จะส่งไปที่ MEM/WB.registerRd ดังนั้นคู่นี้จะเข้ากรณีที่ 4
- สำหรับอีก 2 คำสั่งที่เหลือจะไม่เกิด Hazard เพราะสามารถนำข้อมูลในรีจิสเตอร์มาใช้ได้ตามปกติ
- อย่างไรก็ตามการเปรียบเทียบเฉพาะค่าในรีจิสเตอร์ของ pipeline register แบบนี้ยังมีจุดอ่อน เพราะแม้คำสั่งไม่ได้มีการ Write register เงื่อนไขข้างต้นก็จะตรวจจับว่าเกิด Hazard ดังนั้นเพื่อความถูกต้องจึงต้องเพิ่ม RegWrite เข้ามาเป็นเงื่อนไขด้วย

Data Hazard Detection

- สามารถเขียนเป็นเงื่อนไขได้ดังนี้

Ex hazard

If (Ex/MEM.regWrite and EX/MEM.registerRd = ID/EX.registerRn)

    ForwardA = 10

If (Ex/MEM.regWrite and EX/MEM.registerRd = ID/EX.registerRm)

    ForwardB = 10

Mem hazard

If (MEM/WB.regWrite

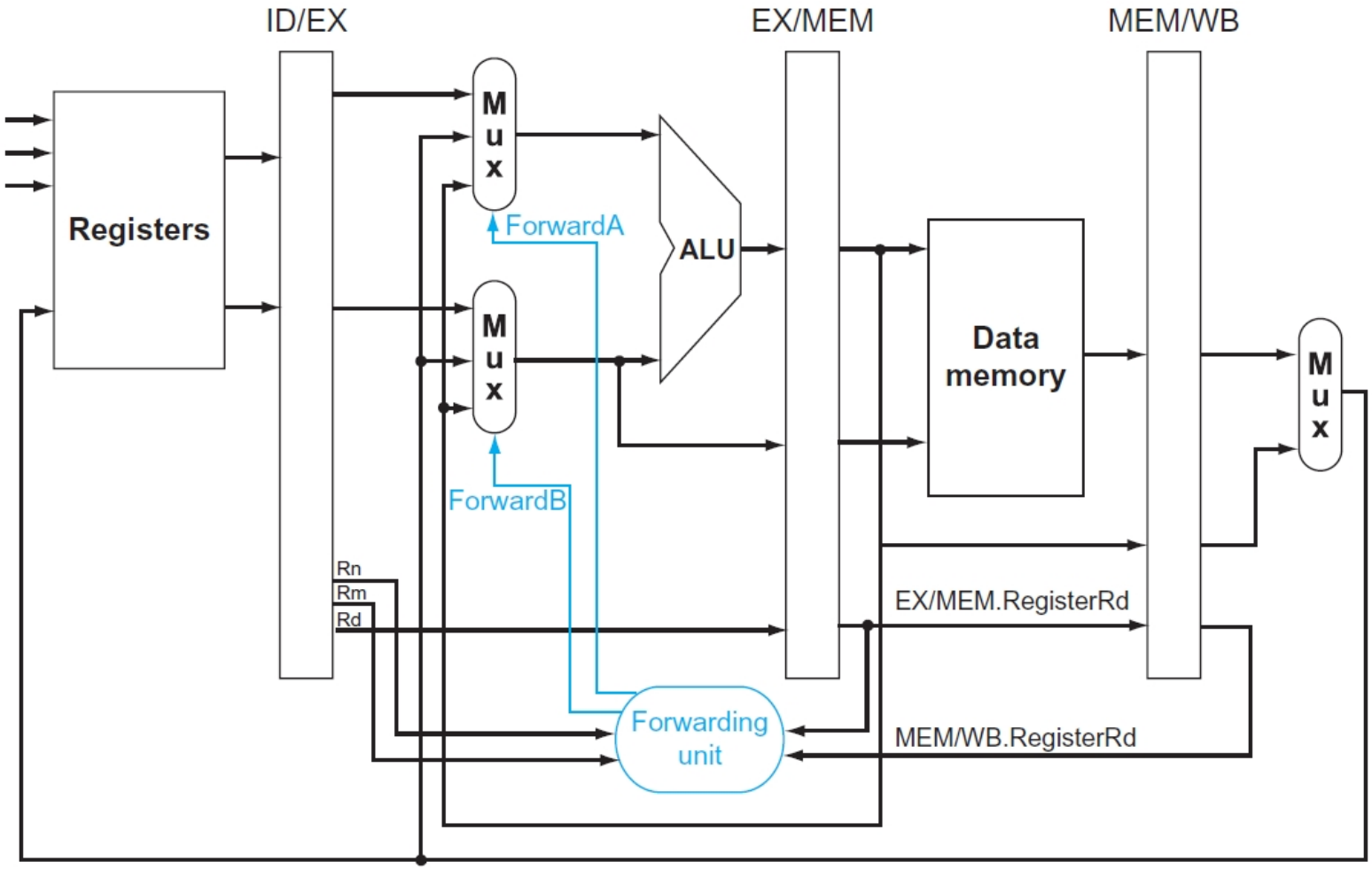
and MEM/WB.registerRd = ID/EX.registerRn) ForwardA = 01

If (MEM/WB.regWrite

and MEM/WB.registerRd = ID/EX.registerRm) ForwardB = 01

Data Hazard Detection

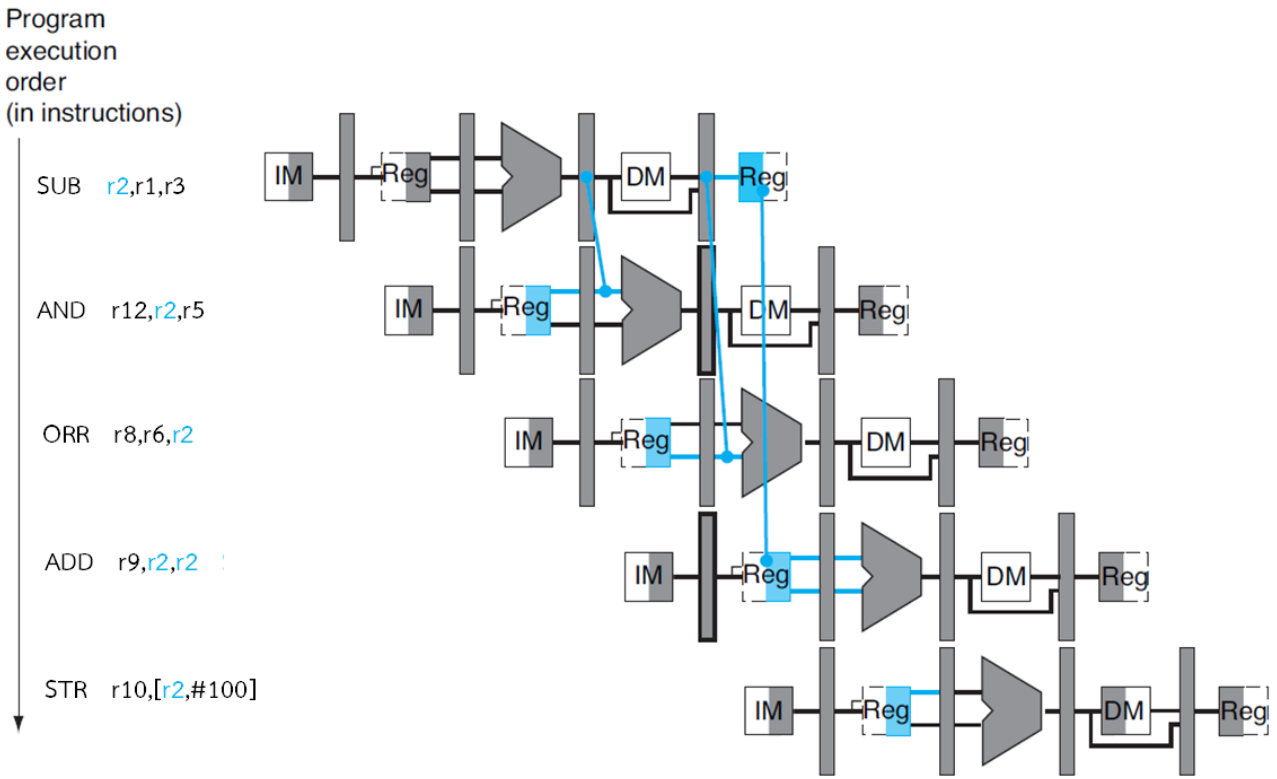
- แสดงการทำงานของ forwarding Unit ตามเงื่อนไขที่กำหนด



Data Hazard Detection

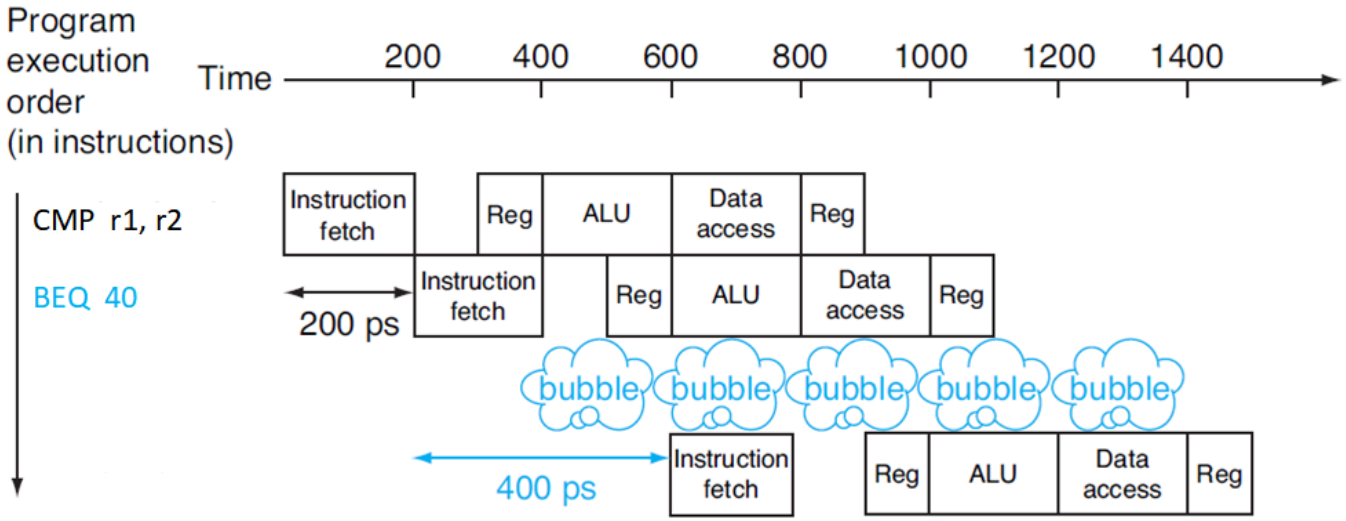
- แสดงผลการทำงานของ forwarding ตามเงื่อนไข

Time (in clock cycles)	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2:	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM:	X	X	X	–20	X	X	X	X	X
Value of MEM/WB:	X	X	X	X	–20	X	X	X	X



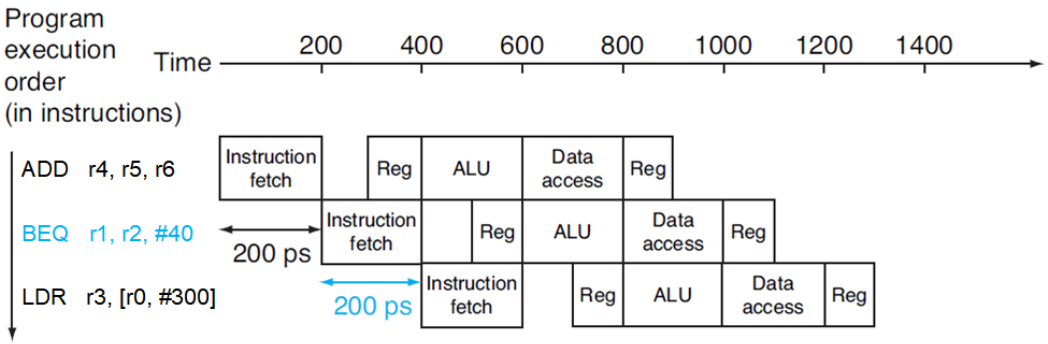
Control Hazard

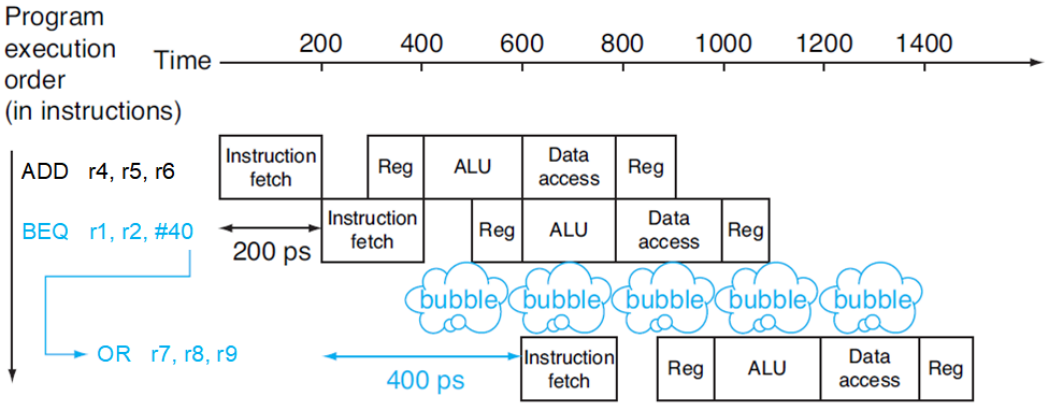
- ที่ผ่านมาได้กล่าวถึง กรณีที่ pipeline ทำงานตามลำดับ เนื่องจากความจริงที่ว่า คำสั่งจะต้อง Fetch ทุก cc เพื่อเติมคำสั่งใน pipeline ดังนั้นเมื่อมีการ Branch เกิดขึ้น ก็จะเป็นปัญหากับ pipeline ปัญหานี้เรียกว่า Control Hazard หรือ Branch Hazard
- สำหรับวิธีรับมือ อาจทำได้หลายวิธี (ค่าเฉลี่ยของโปรแกรม ทุก 6 คำสั่งจะมี Branch)
- วิธีที่ 0 : เมื่อ Fetch คำสั่ง Branch ให้หยุดรอ 1 cc เพื่อให้ทราบว่าจะต้อง Branch ไปทางไหน เมื่อทราบแล้วจึง Fetch ต่อไป (ตามรูป)



Control Hazard

- วิธีที่ 0 ถือได้ว่าเป็นวิธีที่ไม่มีประสิทธิภาพ เนื่องจากจะต้องหยุดทำงานทุกๆ Branch
- วิธีที่ 1 : สมมติว่าไม่ Branch คือ ถ้าไม่ Branch ก็ทำงานต่อ ถ้า Branch ก็ flush pipeline





Control Hazard

- วิธีที่ 2 : Smart Branch
  - Branch Delay Slots จะเป็นการ Optimize โดย Compiler หลักการคือ หลังคำสั่ง Branch ให้นำคำสั่งอื่นมาทำไป หรือ แทรก nop เข้าไป เพื่อให้รู้ Address ที่จะกระโดดไปได้ทัน

```
// R1 = a, R2 = b, R3 = c
// R4 = d, R5 = f
a = b + c;
if (d == 0) {f = f + 1;}
f = f + 2;
```



```
     add R1,R2,R3
     cmp R4,#0
     bne L1
     nop  //delay slot
     add R5, R5, #1
L1:  add R5, R5, #2
```

Can we do better?

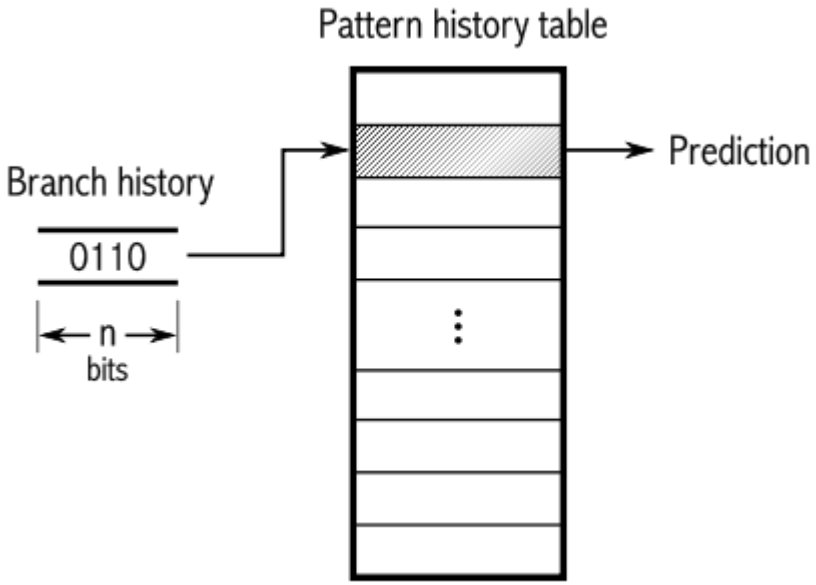
```
     cmp R4, #0
     bne L1
     add R1,R2,R3 // delay slot
     add  R5, R5, #1
L1:  add  R5, R5, #2
```

Fill the delay slot with a useful and valid instruction

Other suggestion using condition code in ARM?

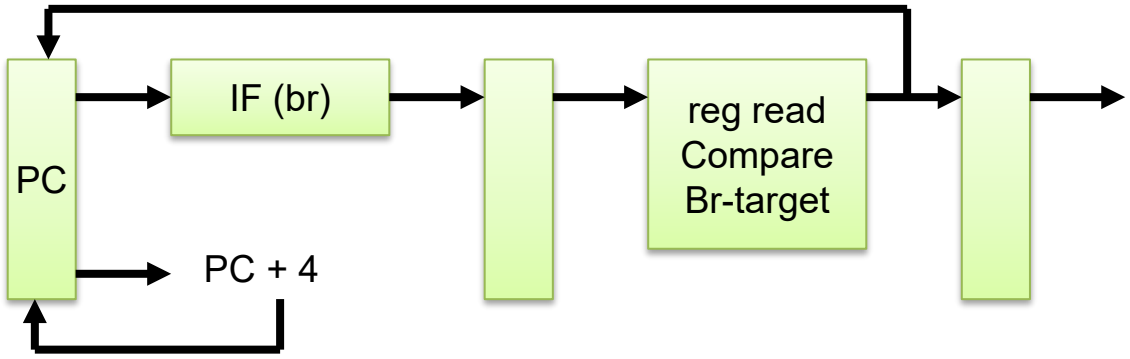
Control Hazard

- วิธีที่ 2 : Smart Branch
  - Dynamic Branch Prediction หลักการ คือ จำการ Branch ครั้งสุดท้ายเอาไว้ โดยใส่ใน Branch Prediction Buffer หรือ Branch History Table แล้ว Branch ไปตาม Address ที่เก็บในตาราง หากการ Branch จริง ไม่ตรงกับในตาราง เช่น for (i=0;i<10;i++) ครั้งแรกก็จะผิด->เก็บ Address ใน Table จากนั้นครั้งที่ 1-9 จะถูก และครั้งที่ 10 จะผิดอีกครั้ง->ลบตาราง

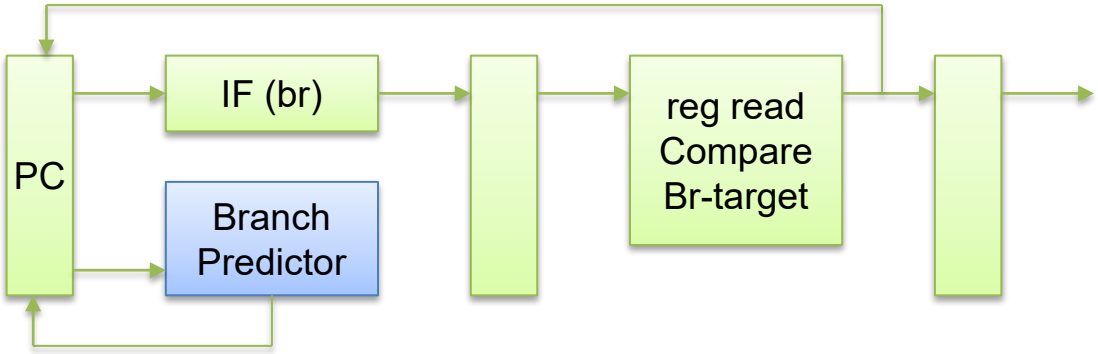


Branch Prediction

- ใน Branch ปกติ หลังจาก PC Latch แล้ว ข้อมูลจะเข้าสู่ Mini ALU เพื่อ PC=PC+4 แล้วมารอที่ Mux โดยข้อมูลจาก Branch Target จะมารอที่ Mux เพื่อรอเลือกว่าจะ Branch หรือไม่

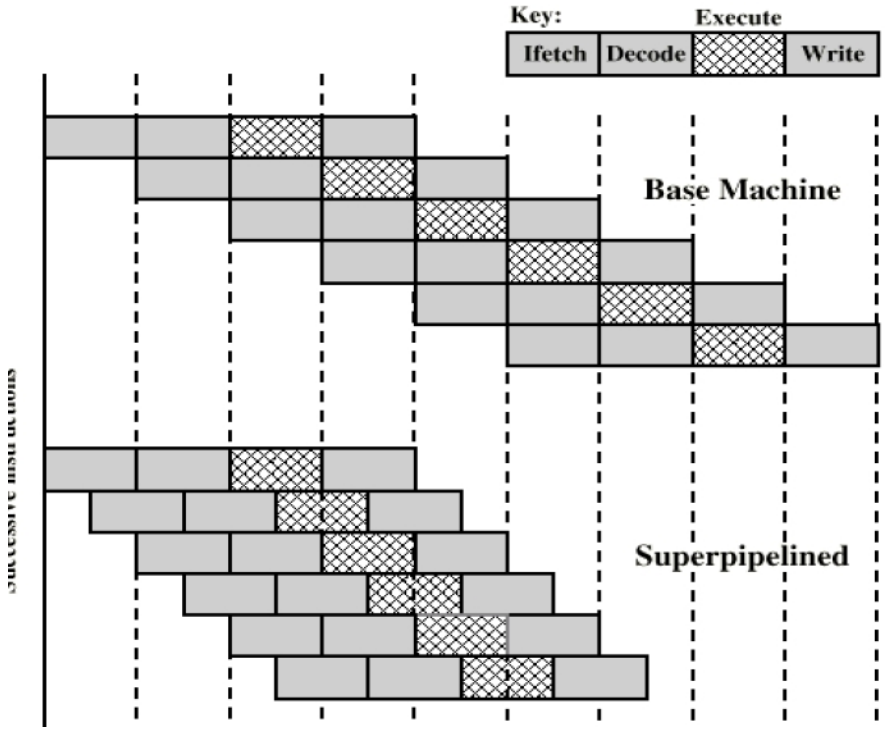


- แต่ใน Branch Prediction Model จะมีการสร้าง Hardware ขึ้นมา เพื่อทำหน้าที่ป้อน Next-PC ให้กับ PC (ตามรูป)



Instruction Level Parallelism

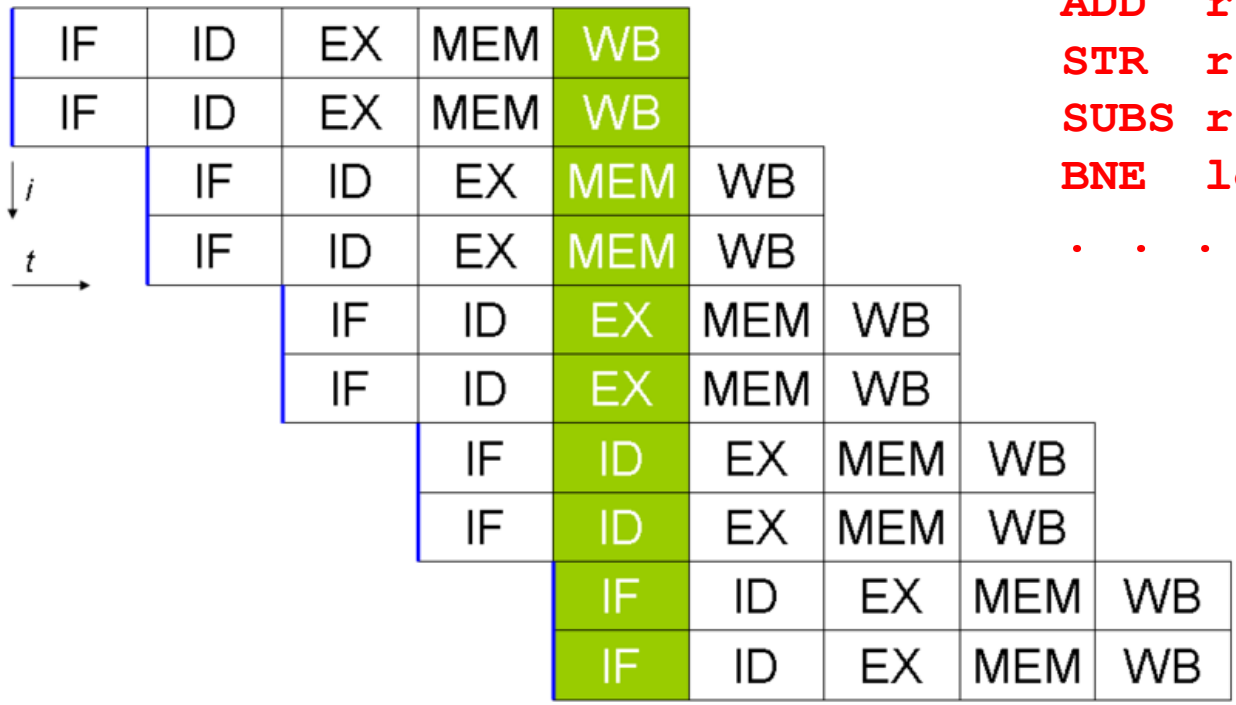
- ในการเพิ่มประสิทธิภาพในการทำงานแบบ pipeline มีแนวทาง 2 วิธี ได้แก่
  - เพิ่มความลึกของ pipeline ให้มีจำนวน pipeline มากขึ้น หรือเหลื่อมเวลาใน pipeline เรียกว่า super pipeline



Instruction Level Parallelism

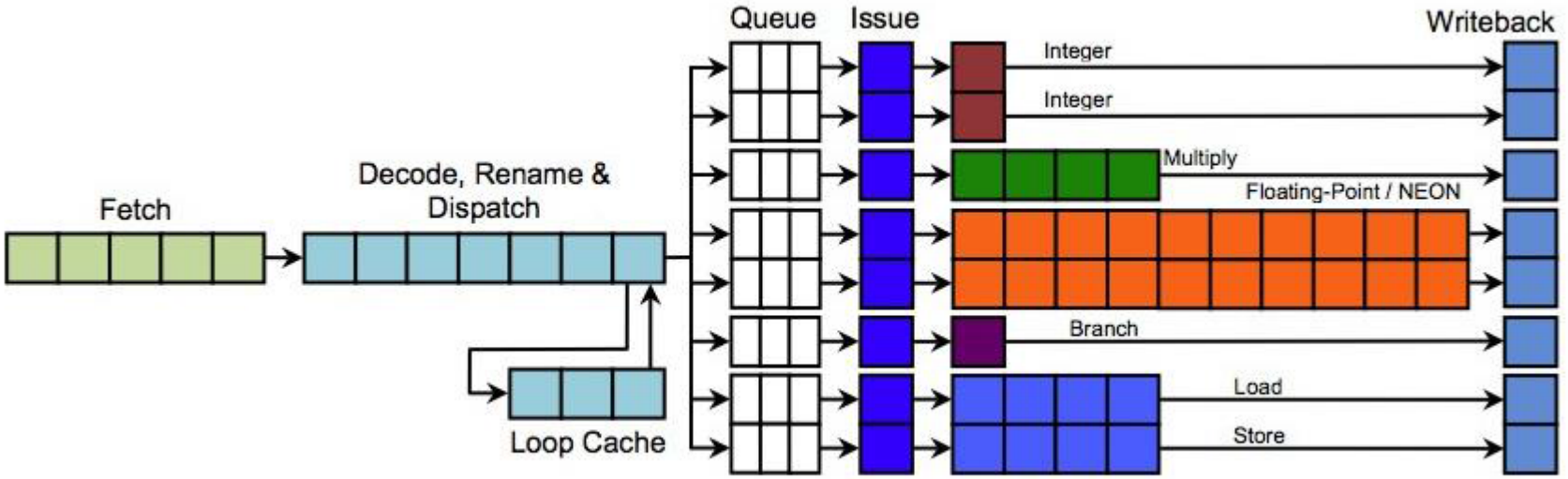
- เพิ่มประสิทธิภาพใน pipeline
  - อีกวิธีหนึ่ง คือ เพิ่มจำนวนหน่วยที่ทำงาน เรียกว่า superscalar โดยคำสั่งที่ทำงานใน pipeline มักเป็นคำสั่งที่ทำงานคนละแบบ เช่น ALU กับ Load/Store ซึ่งต้องพึ่งการทำงานของ Compiler อย่างมาก

```
Loop:  LDR  r0,[r1,#0]
       ADD  r1,r1,#2
       STR  r0,[r1,#0]
       SUBS r1,r1,#4
       BNE  loop
       .  .  .
```



Instruction Level Parallelism

- หรืออาจจะอยู่ในรูปแบบนี้ก็ได้
  - อาจมีการใช้ Loop Unrolling เช่น มี 4 Loop ก็สร้างการทำงานขึ้นมา 4 ชุด ให้แต่ละชุดทำงานกับแต่ละ Loop
  - ในการทำ Loop Unrolling ต้องมีการทำ register renaming เพื่อให้แต่ละคำสั่งไม่ขึ้นต่อกัน



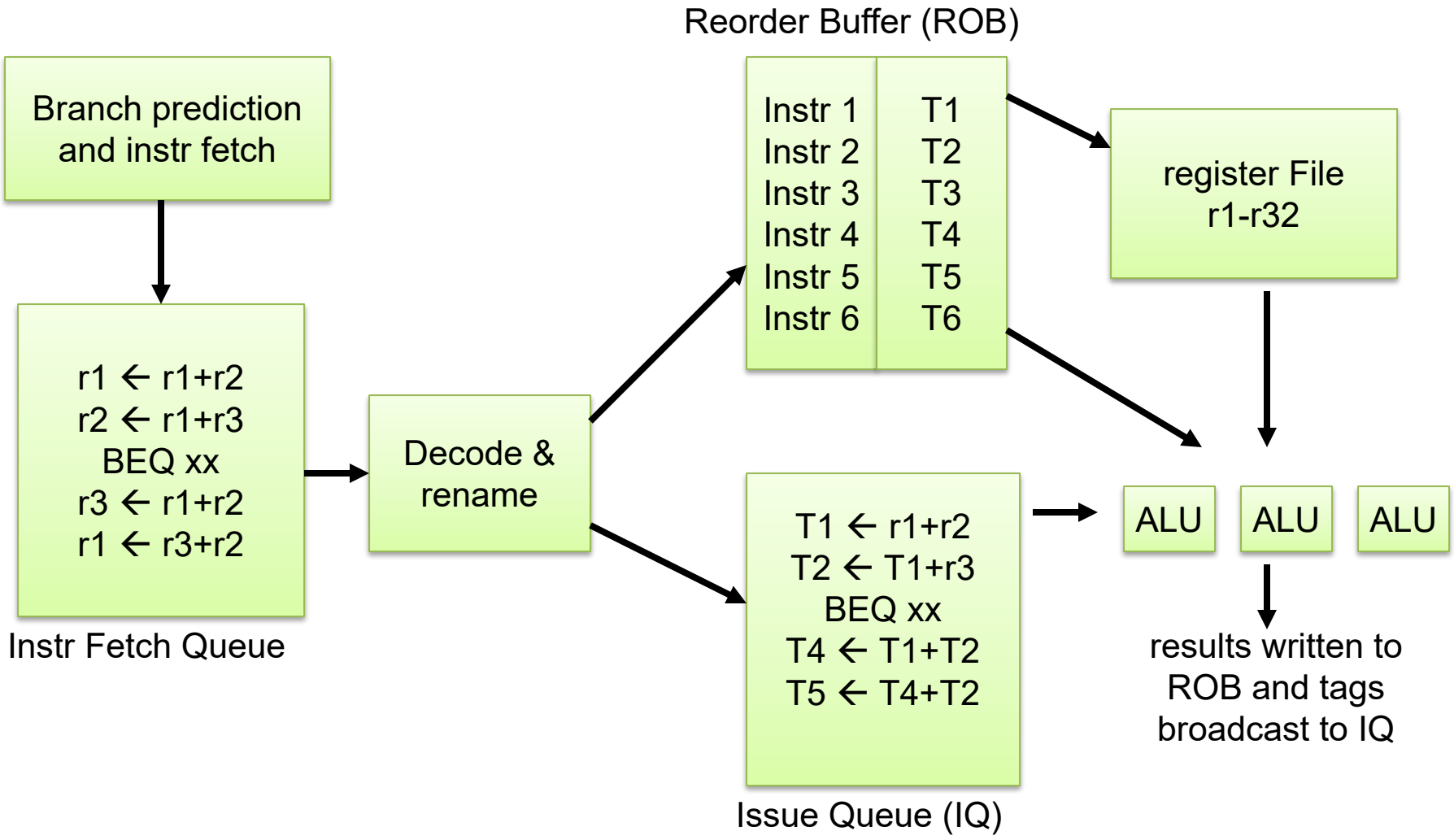
Loop Unrolling

- ตัวอย่างของการทำ Loop Unrolling

ALU	Data Transfer
ADD r1,r1,#-16	LDR r0a,[r1,#0]
	LDR r0b,[r1,#12]
ADD r0a,r0a,#2	LDR r0c,[r1,#8]
ADD r0b,r0b,#2	LDR r0d,[r1,#4]
ADD r0c,r0c,#2	STR r0a,[r1,#16]
ADD r0d,r0d,#2	STR r0b,[r1,#12]
CMP r1, r3	STR r0c,[r1,#8]
BNE loop	STR r0d,[r1,#4]

```
Loop:  LDR  r0,[r1,#0]
       ADD  r0,r0,#2
       STR  r0,[r1,#0]
       CMP  r0,r3
       BNE  loop
       .  .  .
```

OOO – Out of Order Execution



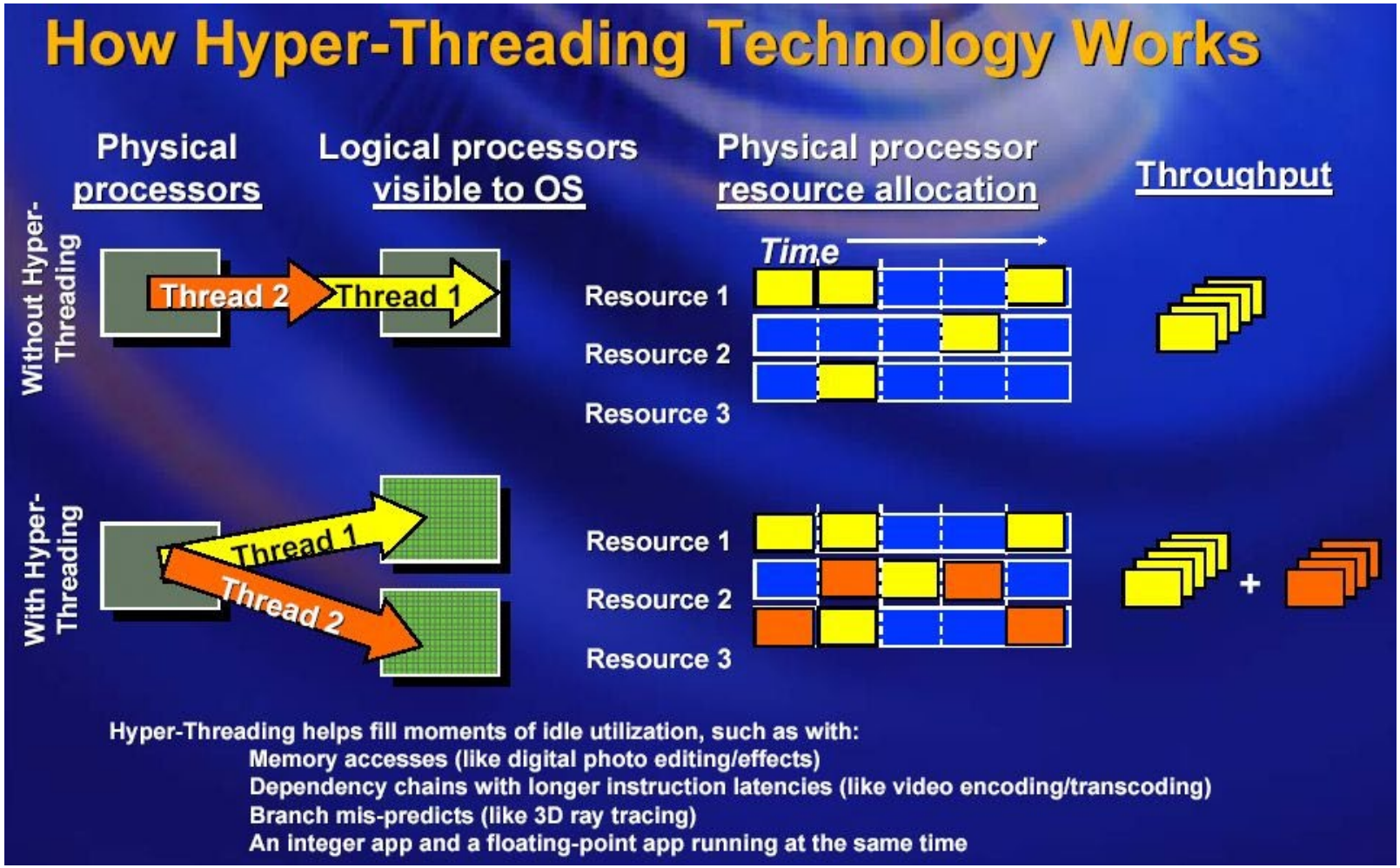
OOO – Out of Order Execution

- Issue Queue (IQ) ทำหน้าที่วิเคราะห์ Dependent ของคำสั่ง การทำงานของ Issue Queue จะใช้ระบบ Windowing คือ มีกรอบคำสั่งที่จะวิเคราะห์เช่น ในอีก 20 คำสั่งถัดไปจาก PC เป็นต้น
- Reorder Buffer (ROB) ทำหน้าที่เก็บลำดับคำสั่งการทำงานเอาไว้ และเก็บผลการทำงานที่ได้จาก ALU เอาไว้ เมื่อคำสั่งบนสุดของ Buffer ทำงานเสร็จ ก็จะนำผลการทำงานจริงไปใส่ใน register จากนั้นจึงนำคำสั่งออกจาก ROB และเลื่อน Window คำสั่ง
- ดังนั้นจะเห็นว่า การ Execution จะไม่เป็นไปตามลำดับ แต่ผลที่เกิดขึ้นในรีจิสเตอร์ จะยังคงเรียงตามลำดับ ซึ่งทำให้โปรแกรมยังคงทำงานได้ถูกต้อง

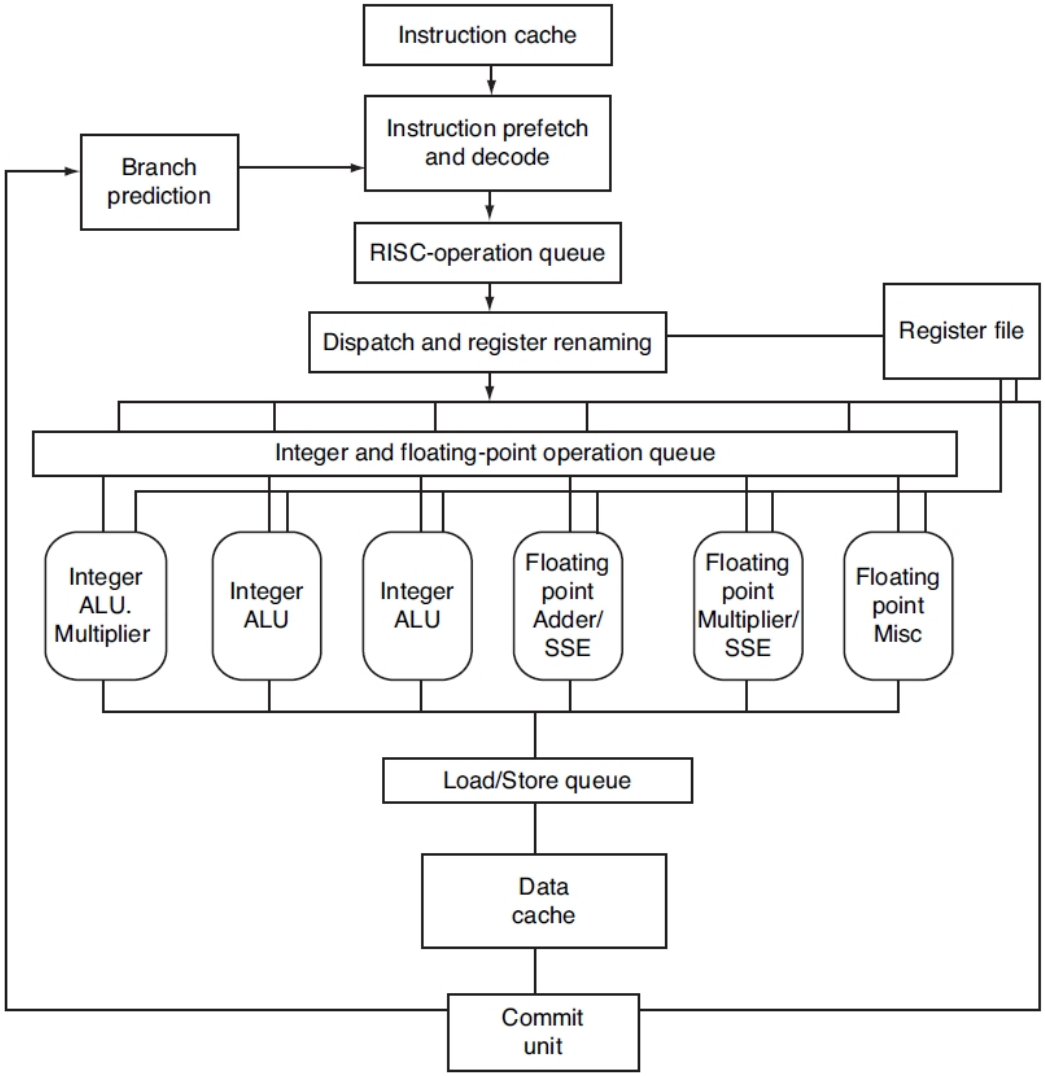
OOO Example

Completion times		with in-order	with ooo
ADD	r1, r2, r3	5	5
ADD	r4, r1, r2	6	6
LDR	r5, r4, #8	7	7
ADD	r7, r6, r5	9	9
ADD	r8, r7, r5	10	10
LDR	r9, r4, #16	11	7
ADD	r10, r6, r9	13	9
ADD	r11, r10, r9	14	10

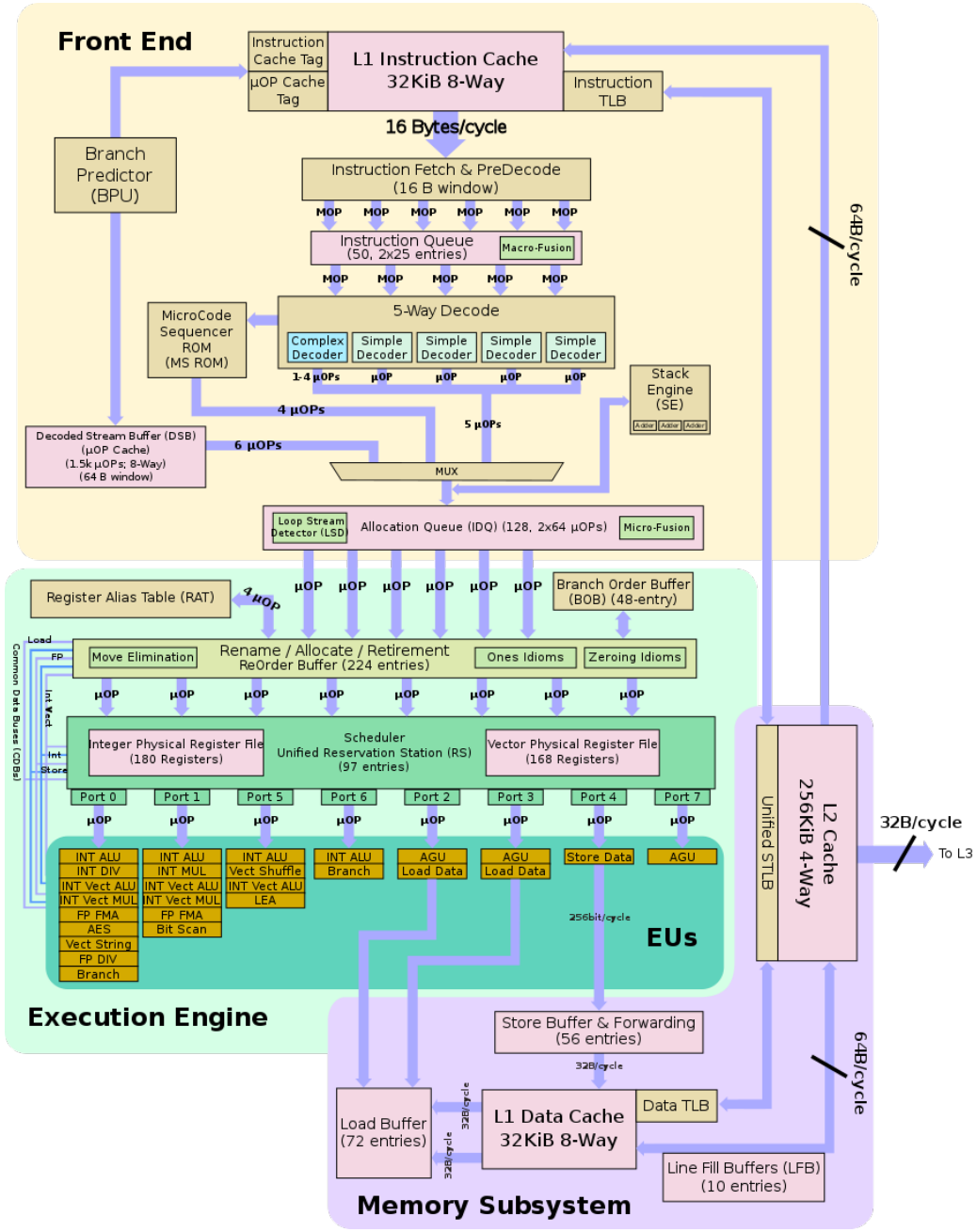
Hyper threading



AMD Opteron X4 Microarchitecture



Intel Core i


Front End
Instruction Cache Tag
µOP Cache Tag
L1 Instruction Cache 32KiB 8-Way
Instruction TLB
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
MOP
Instruction Queue (50, 2x25 entries)
Macro-Fusion
64B/cycle
MicroCode Sequencer ROM (MS ROM)
5-Way Decode
Complex Decoder
Simple Decoder
1-4 µOPs
µOP
Stack Engine (SE)
4 µOPs
5 µOPs
Decoded Stream Buffer (DSB) (µOP Cache) (1.5k µOPs; 8-Way) (64 B window)
6 µOPs
MUX
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (128, 2x64 µOPs)
Micro-Fusion
Register Alias Table (RAT)
4 µOP
Branch Order Buffer (BOB) (48-entry)
Load
FP
Move Elimination
Rename / Allocate / Retirement ReOrder Buffer (224 entries)
Ones Idioms
Zeroing Idioms
Common Data Buses (CDBs)
Int Vect
Int
Store
Integer Physical Register File (180 Registers)
Scheduler Unified Reservation Station (RS) (97 entries)
Vector Physical Register File (168 Registers)
Port 0
Port 1
Port 5
Port 6
Port 2
Port 3
Port 4
Port 7
INT ALU
INT DIV
INT Vect ALU
INT Vect MUL
FP FMA
AES
Vect String
FP DIV
Branch
INT MUL
Bit Scan
Vect Shuffle
LEA
AGU
Load Data
Store Data
256bit/cycle
EUs
Execution Engine
Unified STLB
L2 Cache 256KiB 4-Way
32B/cycle
To L3
Store Buffer & Forwarding (56 entries)
32B/cycle
Load Buffer (72 entries)
L1 Data Cache 32KiB 8-Way
Data TLB
Line Fill Buffers (LFB) (10 entries)
64B/cycle
Memory Subsystem

For your attention