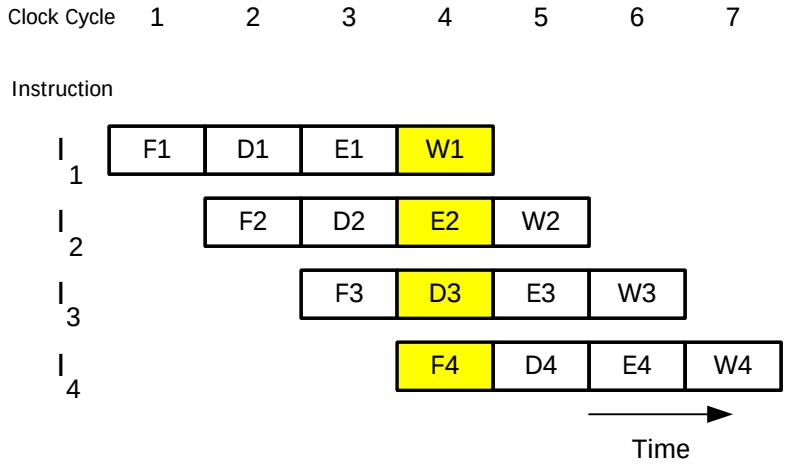
บทที่ 7
โพรเซสเซอร์แบบไปป์ลายน์และซุปเปอร์สเกลาร์

7.1 หลักการทำงานของไปป์ลายน์

ไปป์ลายน์ (Pipelining) เป็นวิธีการเพิ่มความสามารถในการทำงานของซีพียูโดยจะแบ่งซีพียูออกเป็นสเตทการทำงานย่อยๆ และซึ่งสามารถทำงานพร้อมกันได้ โดยสเตทหนึ่งจะส่งผลการทำงานของมันให้กับสเตทต่อไปและรับข้อมูลอื่นเข้ามาประมวลผลต่อไป ดังนั้นแต่ละสเตทจึงสามารถทำงานพร้อมกันได้ แต่ทำอยู่บนคนละคำสั่งกันดังนั้นเมื่อมองจากภาพรวมแล้วจะได้ผลลัพธ์การทำงาน 1 คำสั่งต่อ 1 ไซเคิล

ตัวอย่างเช่น ทำการแบ่งซีพียูออกเป็น 4 สเตท ดังนี้

1. fetch เป็นการอ่านข้อมูลจากหน่วยความจำเข้ามาในซีพียู
2. decode เป็นการถอดรหัสคำสั่งและการเตรียม operand ให้กับการทำงานของซีพียู
3. execute เป็นทำการประมวลผลคำสั่ง เช่นการบวก, ลบ หรือเลื่อนข้อมูล(Shift) เป็นต้น
4. writeback เป็นการเขียนผลลัพธ์จากการทำงานลงสู่รีจิสเตอร์และหน่วยความจำ

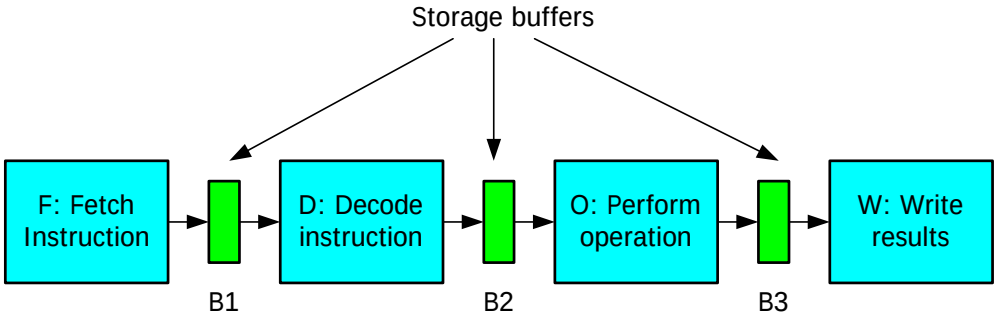


รูปที่ 7.1 การทำงานของคำสั่งซึ่งแบ่งการทำงานเป็น 4 ขั้นตอน

ในขณะที่ซีพียูทำการเฟตช์คำสั่งที่ 4 อยู่นั้น สเตท decode จะทำการถอดรหัสคำสั่งของคำสั่งที่ 3 ส่วนสเตท execute จะทำการประมวลผลคำสั่งที่ 2 และสเตท writeback จะทำการเขียนผลลัพธ์การทำงานของคำสั่งที่ 1 ซึ่งการทำงานทั้งหมดนั้นเกิดขึ้นพร้อมกันในเวลาเดียวกัน ดังนั้นหากสามารถทำให้ทุกคำสั่งสามารถทำงานได้โดยอิสระและไม่มีการรอคอยกันแบบนี้แล้ว ซีพียูจะสามารถทำคำสั่งให้เสร็จได้ภายใน 1 สัญญาณนาฬิกา

ในการสร้างออกแบบโครงสร้างของซีพียูนั้นจะต้องมีฮาร์ดแวร์ 4 ส่วนแยกจากกันและแต่ละส่วนจะต้องทำงานขนานกันได้ โดยแต่ละหน่วยจะทำงานอยู่บนข้อมูลคนละชุดกันและจะส่งผลลัพธ์ให้กับหน่วยต่อไปโดยผ่าน storage buffer ซึ่งในกรณีของการสร้างไปป์ลายน์แบบ 4 stage นั้นจะมี storage buffer อยู่ 3 ตัวด้วยกันดังรูป ในขณะที่อยู่ในไซเคิลที่ 4 นั้นข้อมูลที่อยู่ในบัฟเฟอร์จะเป็นดังนี้

- บัฟเฟอร์ B1 จะเก็บคำสั่ง I3 ซึ่งถูกเฟตช์เข้ามาในไซเคิลที่ 3และกำลังถูก decode โดยหน่วย instruction-decoding unit
- บัฟเฟอร์ B2 จะเก็บทั้ง source operand ของคำสั่ง I2 และรายละเอียดการทำงานของคำสั่ง I2 นั้น โดยข้อมูลที่เก็บอยู่นั้นจะถูกสร้างโดยวงจร decoder ในไซเคิลที่ 3
- บัฟเฟอร์ B3 จะใช้ในการเก็บผลลัพธ์ที่ได้จาก operation unit ของคำสั่ง I1

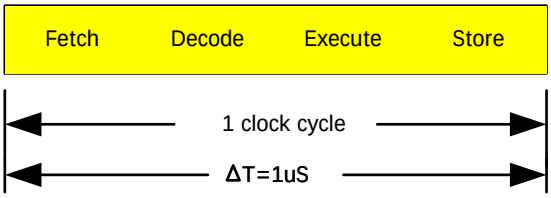


รูปที่ 7.2 โครงสร้างทางฮาร์ดแวร์ของซีพียูที่ใช้เทคนิคไปป์ลายน์

7.2 ประสิทธิภาพที่ปรับปรุงขึ้นจากการใช้เทคนิคไปป์ลายน์

ในการเปรียบเทียบประสิทธิภาพของโพรเซสเซอร์แบบไปป์ลายน์กับโพรเซสเซอร์ที่ไม่มีการทำงานแบบไปป์ลายน์ จะทำการพิจารณาได้สองด้านคือ ประการแรกดูจากประสิทธิภาพที่เพิ่มขึ้นของแต่ละคำสั่งเอง และประการที่สองคือพิจารณาจากประสิทธิภาพโดยรวมของระบบ ในรูปที่ 7.3 แสดงให้เห็นว่าโพรเซสเซอร์แบบไม่มีไปป์ลายน์นั้นจะต้องใช้ขั้นตอนในการทำงาน 4 ขั้นตอนด้วยกันคือการเฟตช์(Fetch) การถอดรหัส(Decode) การประมวลผล(Execute) และการเขียนผลลัพธ์(Store) ซึ่งเป็นเวลาในการทำงานของโพรเซสเซอร์ธรรมดานี้หากโพรเซสเซอร์ใช้เวลาในการทำงานทั้ง 4 ขั้นตอนได้เสร็จภายใน 1 คล๊อกไซเคิลแล้ว การทำคำสั่ง 1 คำสั่งให้เสร็จสิ้นจะใช้เวลาในการทำงานเท่ากับ 1 ไมโครวินาทีเมื่อโพรเซสเซอร์ทำงานที่ความถี่ 1 MHz ดังนั้นเมื่อเวลาผ่านไป 5 วินาทีจะทำให้โพรเซสเซอร์สามารถทำงานได้ 5 ล้านคำสั่ง

$$5\,\text{seconds} \times \left(\frac{1 \times 10^{6}\,\text{cycles}}{1\,\text{second}} \right) \times \left(\frac{1\,\text{instruction}}{1\,\text{cycle}} \right) = 5 \times 10^{6}\,\text{instructions}$$

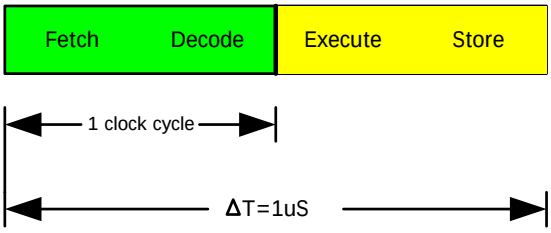


รูปที่ 7.3 การทำงานของโพรเซสเซอร์ที่ไม่มีการทำงานแบบไปป์ลายน์

หากแบ่งโพรเซสเซอร์ในรูปที่ 7.3 ให้มีการทำงานแบบไปป์ลายน์ 2 สเตทโดยสเตทแรกจะทำหน้าที่เฟตช์และถอดรหัส ส่วนในสเตทที่สองให้ทำการประมวลผลและเขียนผลลัพธ์จะทำให้เวลาที่ต้องใช้ในการทำงานของแต่ละขั้นตอนลดลงเหลือแค่ ½ ทำให้สามารถเพิ่มความเร็วของสัญญาณนาฬิกาได้ 2 เท่าเป็น 2 MHz แต่หากพิจารณาให้ละเอียดจะเห็นว่าแม้ความเร็วในการทำงานของโพรเซสเซอร์เพิ่มขึ้น 2 เท่าแต่เวลาในแต่ละคล๊อกไซเคิลนั้นสามารถทำคำสั่งได้เพียง ½ ของงานเดิมเท่านั้น นั่นคือการจะทำคำสั่งให้ครบในแต่ละคำสั่งจะต้องใช้เวลาในการทำงานถึง 2 คล๊อกไซเคิลซึ่งหมายถึงว่าเวลาที่ใช้ในการทำงานของแต่ละคำสั่งให้เสร็จสิ้นจะต้องใช้เวลา 1 ไมโคร วินาที(ที่ความถี่ 2 MHz) ซึ่งจะเห็นว่าใช้เวลาในการทำงานเท่ากันกับโพรเซสเซอร์แบบเดิมในรูปที่ 7.3

อย่างไรก็ตามถึงแม้ว่าเวลาที่ต้องใช้ในการทำคำสั่ง 1 คำสั่งจะใช้เวลาเท่ากันแต่เนื่องจากโพรเซสเซอร์แบบไปป์ลายน์ในรูปที่ 7.4 นั้นสามารถทำงานได้สองคำสั่งพร้อมๆ กันแต่ทำคำสั่งละ ½ ขั้นตอนการทำงาน ทำให้ค่า IPC(Instruction Per Clock) rating ซึ่งเป็นค่าที่แสดงถึงจำนวนคำสั่งที่ทำได้ใน 1 คล๊อกไซเคิลจะมีค่าเท่ากับ 1 ดังนั้นหากเวลาผ่านไป 5 วินาที โพรเซสเซอร์จะทำคำสั่งได้จำนวน 10 ล้านคำสั่ง

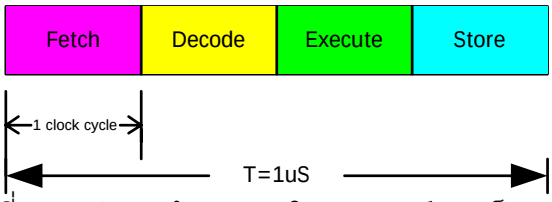
$$5\ \text{seconds} \times \left(\frac{2 \times 10^{6}\ \text{cycles}}{1\ \text{second}}\right) \times \left(\frac{2\ \text{instructio ns} \times 0.5\ \text{operation}}{1\ \text{cycle}}\right) = 10 \times 10^{6}\ \text{instructio ns}$$



รูปที่ 7.4 แบ่งการทำงานของโพรเซสเซอร์ออกเป็น 2 สเตท

ในทำนองเดียวกันหากแบ่งโพรเซสเซอร์ในรูปที่ 7.3 ใหม่ให้มีการทำงานแบบไปป์ลายน์จำนวน 4 สเตทดังรูปที่ 7.5 จะทำให้สามารถเพิ่มความถี่สัญญาณนาฬิกาได้เป็น 4 เท่าเป็น 4 MHz ดังนั้นเมื่อเวลาผ่านไป 5 วินาที โพรเซสเซอร์จะทำคำสั่งได้จำนวน 20 ล้านคำสั่ง

$$5 \text{ seconds} \times \left(\frac{4 \times 10^6 \text{ cycles}}{1 \text{ second}} \right) \times \left(\frac{4 \text{ instructio ns} \times 0.25 \text{ operation}}{1 \text{ cycle}} \right) = 20 \times 10^6 \text{ instructio ns}$$



รูปที่ 7.5 แบ่งการทำงานของโพรเซสเซอร์ออกเป็น 4 สเตท

ตัวอย่างเช่น ด้วยเทคโนโลยีขนาด 0.18 ไมครอนเท่ากันในโพรเซสเซอร์ Intel Pentium III ซึ่งมีจำนวนไปป์ลายน์ 12 สเตทสามารถทำงานที่ความถี่สูงสุด 1 GHz ในขณะที่โพรเซสเซอร์ Intel Pentium 4 ซึ่งมีจำนวนไปป์ลายน์สเตทเท่ากับ 20 สามารถทำงานได้ที่ความถี่ 2.0 GHz ที่เทคโนโลยีการผลิตเดียวกัน

7.3 จำนวนสเตทของไปป์ลายน์

แม้ว่าในทางทฤษฎีแล้วยิ่งมีจำนวนสเตทของไปป์ลายน์มากก็จะทำให้ประสิทธิภาพยิ่งเพิ่มสูงขึ้นตามไปด้วยแต่ในแต่ละสเตทของไปป์ลายน์นั้นจะมีส่วนโอเวอร์เฮดของการเคลื่อนย้ายข้อมูลจากบัฟเฟอร์ของสเตทหนึ่งไปยังบัฟเฟอร์ของอีกสเตทและการจัดเตรียมฟังก์ชั่นการทำงานเพื่อให้ส่งสเตทถัดไป ดังนั้นจำนวนสเตทที่เพิ่มมากขึ้นจะทำให้โพรเซสเซอร์ความต้องการจำนวนเกทและทรานซิสเตอร์ในการออกแบบเพิ่มสูงขึ้นในขณะเดียวกันจำนวนของวงจรลอจิกที่ใช้ในการจัดการแก้ไขปัญหาการขึ้นต่อกันของข้อมูลในหน่วยความจำและรีจิสเตอร์จะเพิ่มความซับซ้อนขึ้นสูงมากเมื่อเทียบกับจำนวนสเตทที่เพิ่มขึ้นเพียงไม่กี่สเตทซึ่งส่งผลให้วงจรควบคุมการทำงานระหว่างสเตทมีความซับซ้อนกว่าสเตทที่ถูกควบคุมได้ นอกจากนี้แล้วการเพิ่มจำนวนสเตททำให้โอกาสที่ไปป์ลายน์จะเกิดการ Stall อันเนื่องมาจากปัญหาของการทำงานแบบไปป์ลายน์ก็จะเพิ่มสูงมากขึ้น ดังนั้นในโพรเซสเซอร์ส่วนใหญ่แล้วจะมีจำนวนสเตทของไปป์ลายน์ไม่เกิน 10 สเตทเท่านั้นดังตัวอย่างในตารางที่ 7.1

ตารางที่ 7.1 จำนวนสเตทของไปป์ลายน์ในซีพียูต่างๆ

ซีพียู	จำนวนสเตทของไปป์ลายน์
80486	5
68040	3
Pentium	5
Alpha 21164	7
PowerPC604	4
Ultra Sparc	9
Mips R1000	5

7.4 ปัญหาที่เกิดขึ้นของการทำงานแบบไปป์ลายน์ (Pipeline Hazards)

ปัญหาที่เกิดขึ้นกับซีพียูไปป์ลายน์โดยทั่วไปมีอยู่ 3 ปัญหาด้วยกันคือปัญหาทางด้านการควบคุม(Control hazards) ปัญหาทางด้านการขึ้นต่อกันของข้อมูล (Data hazards) และปัญหาด้านโครงสร้าง (Structure hazards)

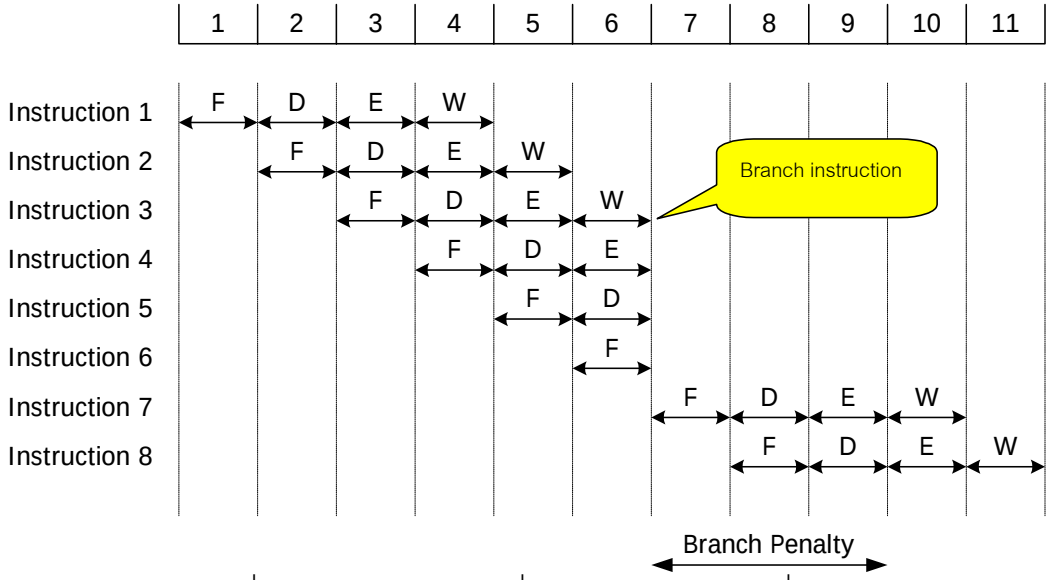
7.4.1 ปัญหาด้านการควบคุม (Control hazards)

ปัญหาของการทำงานแบบไปป์ลายน์คือ หากคำสั่ง I_j ที่กำลังถอดรหัสเป็นคำสั่งกระโดด (Branch instruction) และผลลัพธ์จากการทำคำสั่งทำให้เกิดการกระโดดไปยังแอดเดรสเป้าหมาย แล้วโพรเซสเซอร์จะต้องทิ้งคำสั่งที่ I_{j+1} ไปจนถึงคำสั่งที่ $I_{j+(n-1)}$ (เมื่อ n เท่ากับจำนวนสเตทของไปป์ลายน์) ซึ่งเป็นคำสั่งที่อยู่หลังคำสั่งกระโดดและเริ่มเฟตช์คำสั่งที่ I_k ซึ่งเป็นแอดเดรสเป้าหมายของการกระโดดใหม่ดังรูปที่ 7.5 ซึ่งจะเห็นว่าช่วงเวลาตั้งแต่คล๊อกลูกที่ 7 – 9 นั้นซีพียูไม่มีผลลัพธ์จากการทำงานเกิดขึ้นเลย ซึ่งเวลาที่เสียไปซึ่งเป็นผลจากการกระโดดนั้นจะเรียกว่าข้อเสียจากคำสั่งกระโดด (Branch penalty)

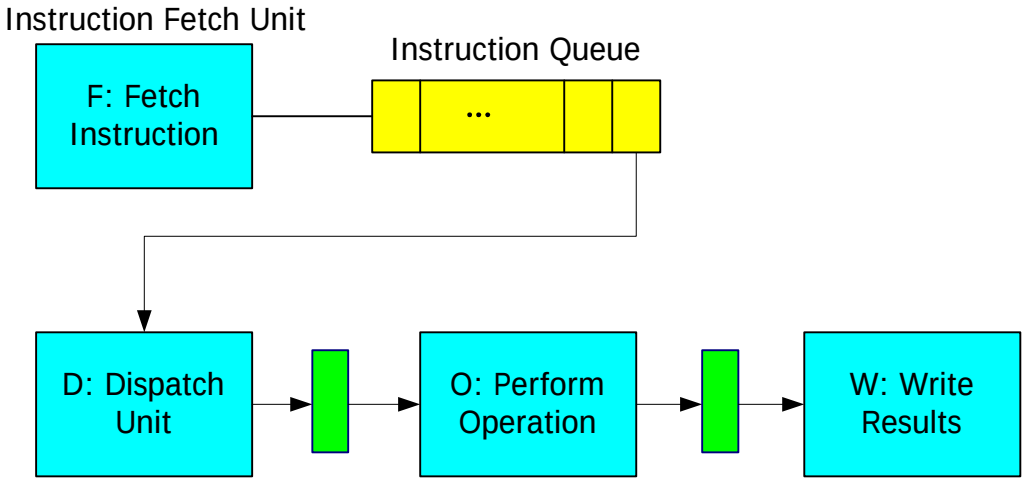
จากรูปที่ 7.6 จะเห็นว่าในขณะที่ทำการเขียนผลลัพธ์ของคำสั่งที่ 3 อยู่นั้นซีพียูจะต้องทำการทิ้งข้อมูลของคำสั่งที่ 4 – 6 ที่ได้ทำงานไปก่อนแล้วทิ้งทั้งหมดแล้วเริ่มทำคำสั่งที่ 7 ซึ่งเป็นคำสั่งที่อยู่ในแอดเดรสเป้าหมายของการกระโดดใหม่

ในการแก้ไขปัญหาของคำสั่งกระโดดนั้นสามารถทำได้หลายวิธีซึ่งปกติคำสั่งกระโดดแบ่งเป็น 2 ประเภทคือคำสั่งกระโดดแบบไม่มีเงื่อนไขและคำสั่งกระโดดแบบมีเงื่อนไข ซึ่งการแก้ปัญหา ข้อเสียของไปป์ลายน์ที่เกิดจากคำสั่กระโดดแบบไม่มีเงื่อนไขนั้นสามารถทำได้โดยการใช้ตัวจัดการคิวของคำสั่ง (Instruction queue) เข้ามาช่วยในการเก็บคำสั่งที่โพรเซสเซอร์จะต้องทำงานไว้ในคิวและมีส่วน Dispatch unit ไว้ดึงคำสั่งออกไปทำงาน เมื่อหน่วยของการเฟตช์คำสั่ง

เจอคำสั่งกระโดดแบบไม่มีเงื่อนไขแล้วมันจะเฟตช์คำสั่งที่อยู่ในแอดเดรสที่ระบุในแอดเดรสเป้าหมายของ



รูปที่ 7.6 ไซเคิลการทำงานที่ว่างงานอันเกิดจากคำสั่งกระโดด



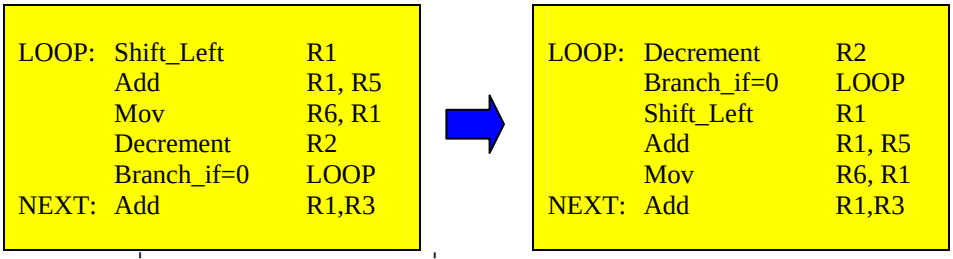
รูปที่ 7.7 การใช้ตัวจัดการคิวของคำสั่งในการแก้ปัญหาที่เกิดจากคำสั่งกระโดดแบบไม่มีเงื่อนไข

คำสั่งกระโดดเข้ามาเป็นคำสั่งถัดไปดังรูปที่ 7.7 ส่วนการแก้ไขปัญหาเนื่องจากคำสั่งกระโดดนั้นทำได้ยากกว่ามากเนื่องจากโพรเซสเซอร์จะไม่สามารถรู้ล่วงหน้าได้เลยว่าการกระโดดจะเกิดขึ้นหรือไม่จนกว่าจะทำคำสั่งกระโดดนั้นเสร็จแล้ว แต่ก็มีการพยายามนำเสนอวิธีในการแก้ไขปัญหาจากการทำคำสั่งกระโดดแบบมีเงื่อนไขอยู่หลายวิธีดังนี้

- Delayed Branch
- Multiple Streams
- Branch Prediction

7.4.1.1 เทคนิค Delayed Branch

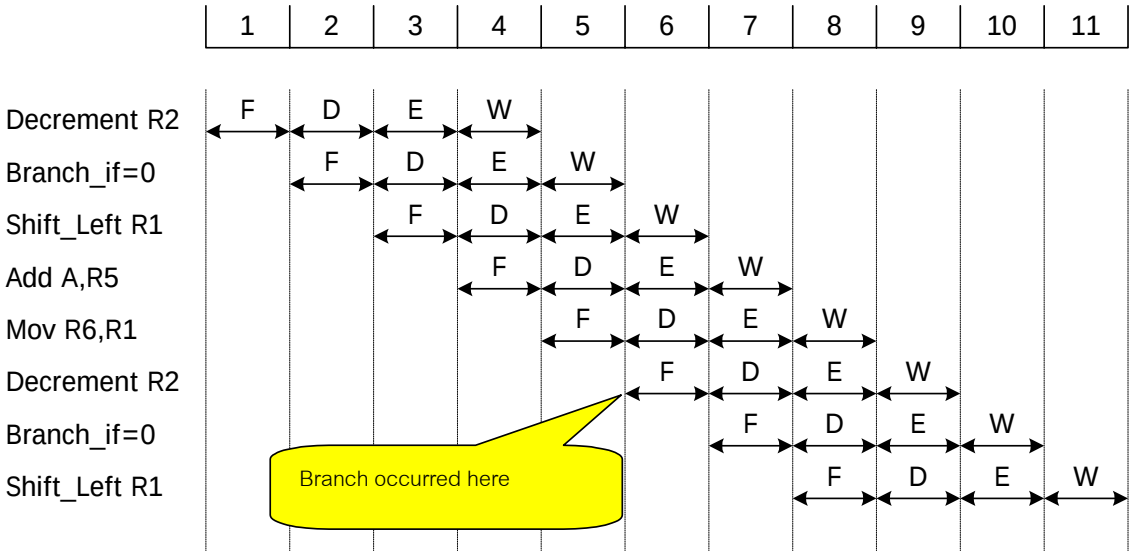
จากปัญหาของไปป์ลายน์ในรูปที่ 7.6 จะเห็นว่าโพรเซสเซอร์เริ่มทำคำสั่งที่ 4 – 7 ก่อนที่มันจะรู้ว่าคำสั่งที่ 3 นั้นเกิดการกระโดดหรือไม่ ดังนั้นเมื่อทำคำสั่งที่ 3 เสร็จแล้วและเกิดการกระโดดขึ้น โพรเซสเซอร์จะต้องทิ้งคำสั่งที่ 4 – 7 ไปเสีย ในการแก้ไขปัญหานี้จะกำหนดให้ตำแหน่งที่เก็บคำสั่งซึ่งตามหลังคำสั่งกระโดดนี้ว่าเป็น Branch delay slot และกำหนดให้โพรเซสเซอร์ให้ทำคำสั่งที่อยู่ในตำแหน่ง Branch delay slot นี้ก่อนที่จะมีการกระโดดจริงๆ เกิดขึ้น ซึ่งจะเห็นว่าการกระโดดนั้นถูกยืดเวลาออกไป ดังนั้นในการใช้งานจะต้องมีการจัดเรียงคำสั่งของโปรแกรมใหม่แล้วจัดคำสั่งที่เหมาะสมใส่ในส่วนของ Branch delay slot นี้ หากไม่สามารถหาคำสั่งมาใส่ได้แล้วสล๊อตนี้จะต้องถูกเติมด้วยคำสั่ง NOP (No-Operation) รูปที่ 7.8 แสดงถึงตัวอย่างของโปรแกรมที่ใช้งานกับเทคนิค Delayed Branch นี้



รูปที่ 7.8 การแก้ไขโปรแกรมเพื่อรองรับการทำงานของ Delayed branch

จากรูปที่ 7.8 จะเห็นว่าคำสั่งของโปรแกรมทางด้านซ้ายมือคำสั่งที่ 2-4 นั้นสามารถนำมาใส่ใน Branch delay slot ได้โดยไม่ทำให้การทำงานของโปรแกรมเสียไป ดังนั้นโปรแกรมที่แก้ไขแล้วทางด้านขวามือจะยังคงทำงานถูกต้องเหมือนเดิมทุกประการซึ่งในการทำงานจะมีการทำคำสั่งที่อยู่ใน Branch delay slot ก่อนที่จะมีการกระโดดจริงเกิดขึ้น ซึ่งการทำงานของโปรแกรมในไปป์ลายน์จะแสดงในรูปที่ 7.9

เทคนิค Delayed branch แม้ว่าจะสามารถนำมาใช้แก้ไขปัญหาได้แต่ในทางปฏิบัตินั้นประสิทธิภาพจะขึ้นอยู่กับความสามารถของคอมไพเลอร์ที่จะต้องหาคำสั่งที่เหมาะสมสำหรับใส่ใน Branch delay slot ให้ได้ ซึ่งจากการวิจัยพบว่าเทคนิคในการคอมไพล์สมัยใหม่สามารถหาคำสั่งใส่ในสล๊อตได้ประมาณ 85 เปอร์เซนต์เมื่อขนาดของ Branch delay slot เท่ากับ 1 สล๊อต แต่ถ้าเพิ่มขนาดของ Branch delay slot ให้ใหญ่ขึ้นแล้วโอกาสที่จะหาคำสั่งใส่ในสล๊อตได้ครบจะลดลงมาก ทำให้ประสิทธิภาพที่ได้ต่ำ ดังนั้นหากโพรเซสเซอร์มีจำนวนสเตทของไปป์ลายน์มากๆ แล้วโอกาสที่จะหาคำสั่งเติมลงในสล๊อตได้ครบจะพบได้น้อย



รูปที่ 7.9 การทำงานของเทคนิค Delayed branch

7.4.1.2 เทคนิค Multiple Streams

ไปป์ลายน์ได้รับผลกระทบจากคำสั่งกระโดดเนื่องจากมันจะต้องเลือกว่าจะเฟตช์คำสั่งถัดไปคำสั่งใด เพื่อกันการผิดพลาดของการเลือกคำสั่งเทคนิค Multiple Stream นี้จะทำการสร้างส่วนเริ่มของไปป์ลายน์ขึ้นมาอีกชุดหนึ่งและอนุญาตให้ทั้งสองส่วนนี้ทำการเฟตช์คำสั่งทั้งสองคำสั่งขึ้นมาพร้อมกันหรือมีสตรีมของคำสั่งจำนวนสองสตรีมนั่นเอง เทคนิคมีปัญหาในการสร้างอยู่สองประการด้วยกันคือ เมื่อมีไปป์ลายน์หลายสตรีมจะมีปัญหาการแย่งกันเข้าถึงหน่วยความจำและรีจิสเตอร์ของแต่ละสตรีม และอีกประการหนึ่งคืออาจเจอคำสั่งกระโดดตัวอื่นอีกในแต่ละสตรีมก่อนที่จะทำคำสั่งกระโดดเสร็จ นั่นหมายถึงจะต้องมีสตรีมเพิ่มเติมเพื่อจัดการคำสั่งกระโดดตัวใหม่ซึ่งทำให้จัดการได้ยาก

7.4.1.3 เทคนิค Branch Prediction

เทคนิคนี้จะใช้วิธีการทำนายการเกิดของคำสั่งกระโดดนี้ซึ่งมีอยู่หลายวิธีด้วยกันในการทำนาย อันได้แก่

- ทำนายว่าไม่เกิดการกระโดดขึ้นเลย
- ทำนายว่าจะเกิดการกระโดดขึ้นเสมอทุกครั้ง
- ทำนายจากการดูอ๊อปโค๊ดของแต่ละคำสั่ง
- ใช้ Taken/Not Taken Switch
- ใช้ Branch History Table

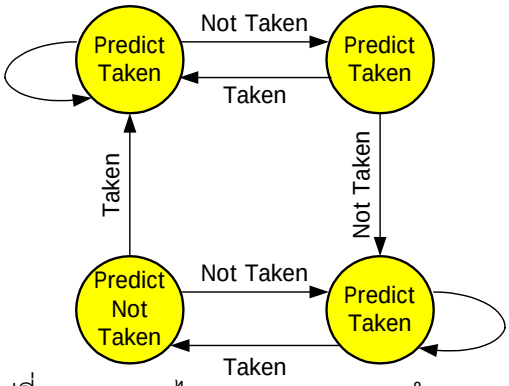
ในวิธีแรกที่ทำนายว่าไม่เกิดการกระโดดขึ้นเลยนั้นตัวซีพียูจะเฟตช์คำสั่งถัดจากคำสั่งกระโดดเข้ามาเสมอ ซีพียูที่ใช้วิธีนี้ได้แก่ 68020 และ VAX 11/780 ส่วนวิธีที่สองนั้นจะตรงข้ามคือ

ซีพียูจะคิดว่าการกระโดดจะเกิดขึ้นเสมอ ดังนั้นเมื่อเจอคำสั่งกระโดดแบบมีเงื่อนไขแล้วซีพียูจะทำการเฟตช์คำสั่งถัดไปจากแอดเดรสเป้าหมายของคำสั่งกระโดดทันทีซึ่งจากการศึกษาวิจัยพบว่าคำสั่งกระโดดแบบมีเงื่อนไขนั้นโอกาสที่จะกระโดดสูงกว่า 50 เปอร์เซ็นต์ ซึ่งหมายความว่าการเฟตช์คำสั่งในแอดเดรสเป้าหมายของการกระโดดล่วงหน้าจะให้ประสิทธิภาพที่สูงกว่าการเฟตช์คำสั่งถัดจากคำสั่งกระโดด แต่อย่างไรก็ตามในซีพียูที่มีการทำงานแบบเพจจิ้ง (Paging) แล้วการใช้เทคนิคนี้จะทำให้โอการที่จะเกิด Page fault สูงขึ้นซึ่งจะต้องมีกลไกในการจัดการกับปัญหานี้ด้วย ส่วนวิธีที่สามนั้นจะดูจากอ๊อปโค๊ดของคำสั่งโดยที่ซีพียูจะมีรายการของคำสั่งอยู่ว่าคำสั่งใดน่าจะเกิดการกระโดดขึ้นเสมอและคำสั่งใดไม่น่าจะเกิดการกระโดดซึ่งจาการวิจัยพบว่าอัตราความถูกต้องของการทำนายสูงกว่า 75 เปอร์เซ็นต์ ทั้งสามวิธีนี้จะทำงานแบบสเตติก นั่นคือไม่มีการเก็บประวัติการทำงานของตัวโปรแกรมไว้ในการตัดสินใจการกระโดดซีพียูแต่อย่างใด

วิธีที่ 4 และ 5 นั้นจะเป็นการทำงานแบบไดนามิกนั่นคือมีการเก็บประวัติการทำงานของตัวโปรแกรมก่อนหน้านี้เอาไว้เพื่อใช้ในการตัดสินใจของคำสั่งกระโดดที่จะเกิดขึ้นในโอกาสต่อไปโดยที่ในเทคนิคการใช้ Taken/Not Taken switch นั้นจะมีการบันทึกว่าการกระโดดในครั้งปัจจุบันเกิดขึ้นหรือไม่ และค่านี้จะถูกใช้ในการตัดสินใจการกระโดดครั้งต่อไป ซึ่งบิทที่ใช้ในการเก็บประวัติว่ามีการกระโดดหรือไม่นี้จะเก็บอยู่ในตัวซีพียูเอง โดยอาจมีตารางขนาดเล็กเอาไว้เก็บคำสั่งกระโดดที่เพิ่งทำเสร็จไปโดยในแต่ละช่องของตารางเก็บค่าเพียง 1 บิทเดียวต่อคำสั่ง 1 คำสั่ง

การเก็บค่าเพียงบิทเดียวของวิธีที่ 4 นั้นจะสามารถเก็บข้อมูลได้ว่าการทำงานครั้งสุดท้ายของคำสั่งกระโดดคำสั่งนั้นๆ เกิดการกระโดดหรือไม่ซึ่งมีข้อเสียคือจะได้ผลดีเฉพาะในการทำคำสั่งลูปซึ่งมีการวนคำสั่งบ่อยๆ เท่านั้นแต่จะมีข้อผิดพลาดจากการทำนายเกิดขึ้นสองครั้งในการใช้ลูปคือในตอนเข้าลูปครั้งแรกและการออกจากลูปซึ่งปัญหานี้แก้ได้โดยใช้การเก็บบิทจำนวนสองบิทดังรูปที่ 7.9 โดยที่กระบวนการในการตัดสินใจจะแสดงดังตัวไฟไนท์สเตทแมชชีนซึ่งมีจำนวนสเตทเท่ากับสี่สเตท หากทิศทางของการกระโดดของคำสั่งเกิดในทิศทางเดียวกันสองครั้งการทำนายจะถือว่าครั้งต่อไปจะเกิดการในทิศทางนั้น หากการทำนายเกิดการผิดพลาดมันจะทำนายเหมือนเดิมอีกครั้งหนึ่งจนหากผิดอีกมันจะทำนายว่าการกระโดดจะไปในทางตรงกันข้าม ดังนั้นจะเห็นว่าในการเปลี่ยนทิศทางของการทำนายจะเกิดขึ้นเมื่อมีการทำนายผิดพลาดติดต่อกันสองครั้งเท่านั้น ดังนั้นในการทำลูปจะทำให้เกิดข้อผิดพลาดในการทำนายแค่ครั้งเดียวเท่านั้น

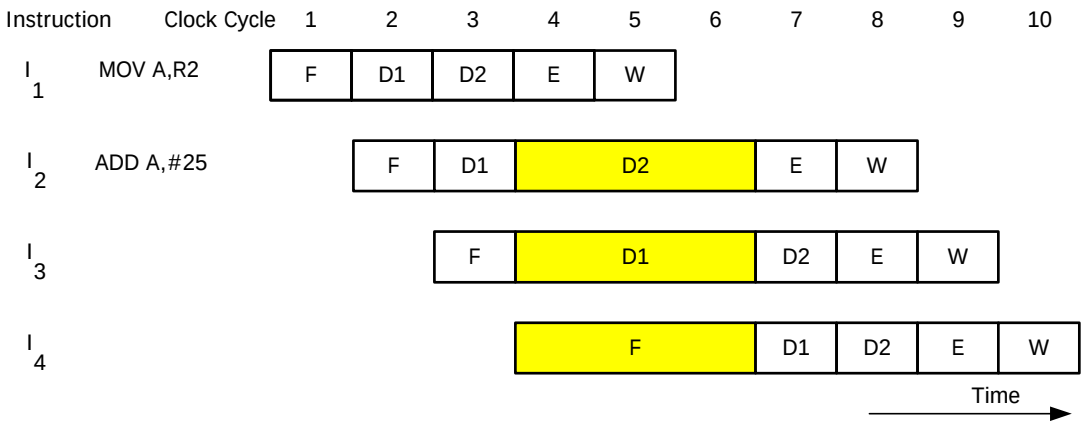
การใช้ Branch history table เป็นการใช้หน่วยความจำแคชขนาดเล็กควบคู่กับสเตทการเฟตช์ของตัวไปปลายน์ โดยแต่ละเอนทรีในตารางจะประกอบด้วย History bits ซึ่งเก็บสเตทการใช้งานของคำสั่งบรานช์นั้นๆ



รูปที่ 7.10 สเตทไดอะแกรมของวงจรทำนายการบรานช์

7.4.2 **ปัญหาด้านการขึ้นต่อกันของข้อมูล** (Data Hazards)

ปัญหา Data hazard เกิดจากคำสั่ง I_j ต้องใช้ผลลัพธ์ที่ได้จากการทำคำสั่ง I_{j-1} ซึ่งเป็นคำสั่งที่อยู่ก่อนหน้าคำสั่ง I_j นี้ทำให้คำสั่ง I_j ต้องรอให้การทำงานของคำสั่ง I_{j-1} ทำการเขียนผลลัพธ์จนเสร็จสิ้นเสียก่อนจึงจะเริ่มเฟตช์โอเปอแรนด์ที่ต้องการในการทำคำสั่งนั้นไปใช้งานได้ดังรูปที่ 7.11 ซึ่งจะเห็นว่าคำสั่ง I_2 ซึ่งก็คือคำสั่ง ADD A,#25 นั้นต้องการใช้ผลลัพธ์ของคำสั่ง I_1 ซึ่งเป็นคำสั่งก่อนหน้านี้ซึ่งก็คือคำสั่ง MOV A,R2 โดยในการทำงานนั้นคำสั่ง ADD A,#25 จะต้องคอยให้การทำงานของ MOV A,R2 นั้นทำงานเสร็จสิ้นเสียก่อนแล้วจึงจะให้คำสั่ง ADD A,#25 ทำการอ่านค่าของแอคคิวมูเลเตอร์ขึ้นมาได้ ในการรอคอยนี้ทำให้บางสเตทของไปป์ลายน์เกิดการว่างงานขึ้นซึ่งจะทำให้ประสิทธิภาพของไปป์ลายน์ลดลง

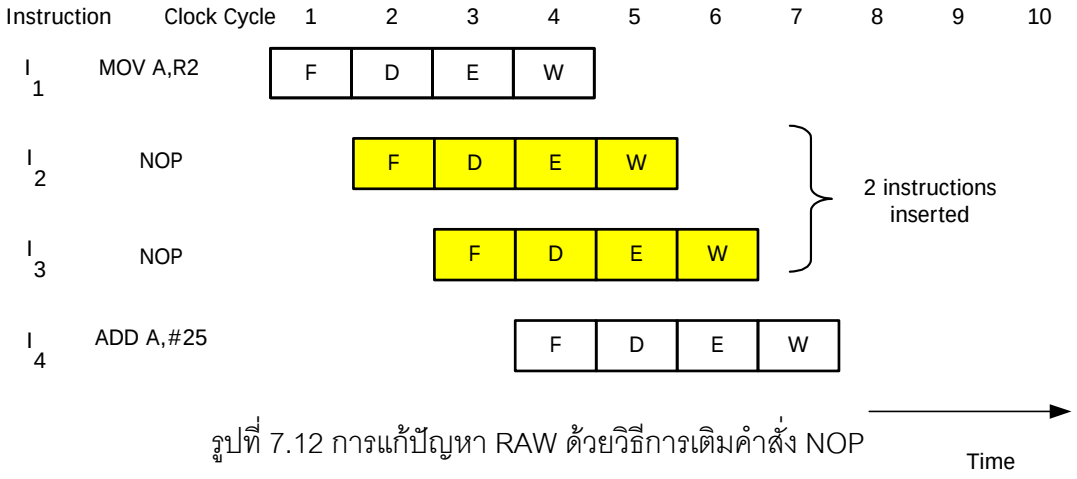


รูปที่ 7.11 การรอคอยกันของแต่ละสเตทในไปป์ลายน์เนื่องจากการขึ้นต่อกันของข้อมูล

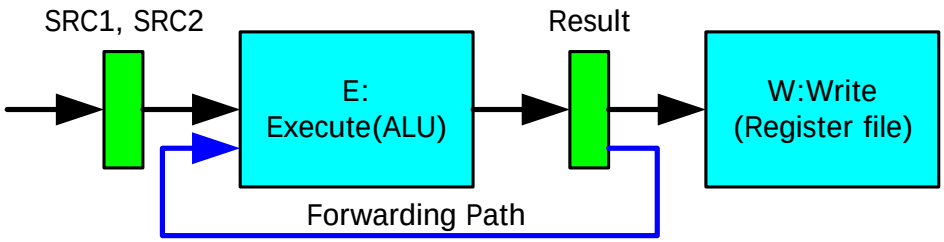
ปัญหา Data Hazards นี้แบ่งย่อยออกเป็น 3 รูปแบบได้แก่

1. RAW (Read After Write)
2. WAW (Write After Write)
3. WAR (Write After Read)

ปัญหา RAW เกิดจากคำสั่งที่ I_{j+1}พยายามที่จะอ่านข้อมูลก่อนที่คำสั่งที่ I_j จะเขียนข้อมูลเสร็จดังนั้นคำสั่งที่ I_{j+1} จะได้ข้อมูลค่าเดิมก่อนที่คำสั่งที่ I_j จะเขียนข้อมูลใหม่ลงไปทำให้การทำงานของโปรแกรมมีข้อผิดพลาดเกิดขึ้น ปัญหา RAW นี้สามารถแก้ไขได้สองวิธีคือ การเติมคำสั่ง NOP โดยอัตโนมัติโดยซีพียูระหว่างคำสั่งที่มีการขึ้นต่อกันของข้อมูลดังรูปที่ 7.12 ส่วนอีกวิธีคือการใช้วิธีการทำการฟอร์เวิร์ดโอเปอแรนด์ (Operand Forwarding) ดังรูปที่ 7.13 ปัญหาที่สองของ Data Hazards คือปัญหา WAW ซึ่งเกิดจากคำสั่งที่ I_{j+1} พยายามที่จะเขียนข้อมูลลงไปในโอเปอแรนด์ก่อนที่คำสั่งที่ I_j จะทำการเขียนข้อมูลเสร็จ ดังนั้นเมื่อคำสั่งที่ I_j ได้ทำงานเสร็จสิ้นแล้วมันก็จะเขียนข้อมูลลงไปทับค่าซึ่งคำสั่งที่ I_{j+1} ได้เขียนลงไปก่อนหน้านี้แล้วทำให้ผลลัพธ์ของคำสั่งที่ I_{j+1} หายไป



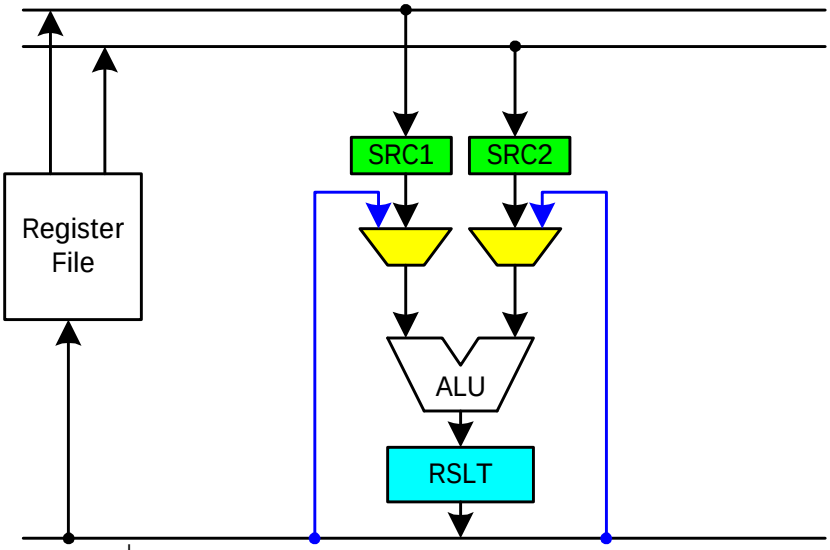
รูปที่ 7.12 การแก้ปัญหา RAW ด้วยวิธีการเติมคำสั่ง NOP



รูปที่ 7.13 การแก้ปัญหา RAW ด้วยวิธีการฟอร์เวิร์ดโอเปอแรนด์

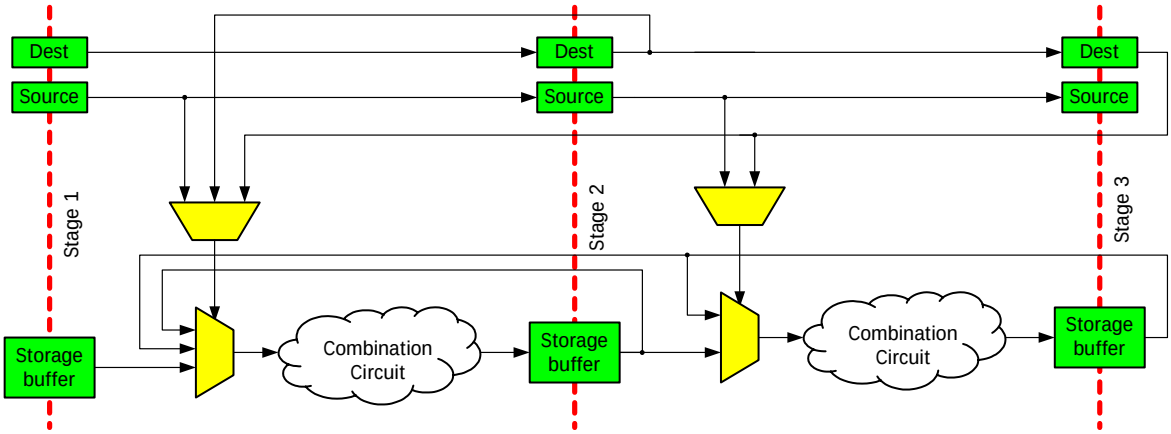
ซึ่งทำให้เกิดข้อผิดพลาดของโปรแกรมขึ้น การแก้ปัญหา WAW นี้จะต้องอาศัยการควบคุมไปป์ลายน์ซึ่งซับซ้อนมากซึ่งซีพียูที่มีปัญหานี้เกิดขึ้นมักเป็นซีพียูที่มีการทำงานเป็นแบบไม่เรียงลำดับ (Out-of-order execution) ปัญหาที่สามของ Data Hazards คือปัญหา WAR ซึ่งเกิดจากคำสั่งที่ I_{j+1} พยายามที่จะเขียนข้อมูลลงไปในโอเปอแรนด์ก่อนที่คำสั่งที่ I_j จะอ่านตัวโอเปอแรนด์นั้นขึ้นมา

ทำให้การทำงานของคำสั่งที่ I_j ได้รับค่าโอเปอแรนด์ที่ผิดพลาดไป การแก้ปัญหา WAR ต้องการกลไกควบคุมไปป์ลายน์ที่ซับซ้อนมากเช่นเดียวกับปัญหา WAW



รูปที่ 7.14 เส้นทางเดินของข้อมูลในการฟอร์เวิร์ดโอเปอแรนด์

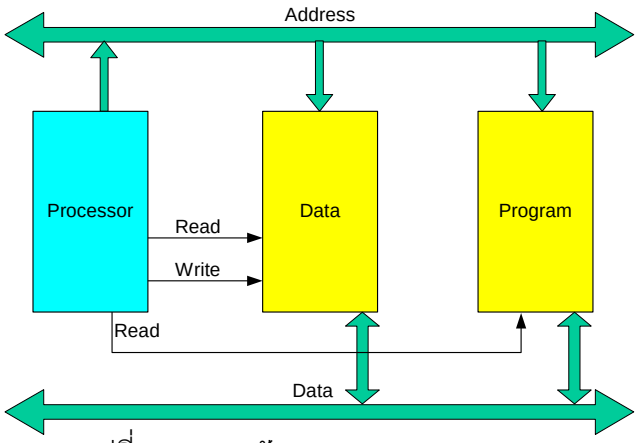
ในการทำการฟอร์เวิร์ดโอเปอแรนด์เพื่อแก้ไขปัญหา RAW นั้นสิ่งที่จะต้องคำนึงถึงคือสิ่งความซับซ้อนของวงจรซึ่งทำการฟอร์เวิร์ดข้อมูลจากสเตทหนึ่งไปยังอีกสเตทหนึ่ง หากวงจรมีจำนวนสเตทมาก จะต้องมีวงจรคอยสนับสนุนการตรวจจับการขึ้นต่อกันของข้อมูลในแต่ละสเตทที่ซับซ้อนมากตามไปด้วย ดังจะเห็นได้จากรูปที่ 7.15 ซึ่งแสดงการฟอร์เวิร์ดข้อมูลของไปป์ลายน์จำนวน 3 สเตท จะเห็นได้ว่าวงจรการฟอร์เวิร์ดในสเตทที่ 2 จะมีความซับซ้อนมากกว่าวงจรฟอร์เวิร์ดใน สเตทที่ 1 เนื่องจากผลลัพธ์จากการทำงานของสเตทที่ 3 จะมีผลต่อวงจรทั้งสเตทที่ 1 และสเตทที่ 2 แต่ผลลัพธ์จากสเตทที่ 2 จะมีผลต่อวงจรในสเตทที่ 1 เท่านั้น



รูปที่ 7.15 การฟอร์เวิร์ดข้อมูลของสเตทการทำงาน 3 สเตท

7.4.3 **ปัญหาด้านโครงสร้าง** (Structural hazards)

แม้ว่าเทคนิคในทางหลักการแล้วไปป์ลายน์จะช่วยเพิ่มความเร็วในการทำงานโดยการทำหลายๆ คำสั่งพร้อมๆ กัน แต่บนงานที่แตกต่างกัน แต่ในการนำไปสร้างวงจรใช้งานจริงกลับประสบปัญหาหลายๆ ด้าน อันเนื่องมาจาก อุปกรณ์หลายๆ ตัวไม่สนับสนุนการทำงานแบบขนาน ยกตัวอย่างเช่นหน่วยความจำ ในการทำงานแบบไปป์ลายน์นั้นผู้ออกแบบต้องการให้สเตทเฟตช์สามารถอ่านหน่วยความจำในขณะที่สเตทการถอดรหัสอาจจำเป็นต้องอ่านค่าจากหน่วยความจำซึ่งอีกคำสั่งในไปป์ลายน์ต้องการและใจขณะเดียวกันสเตทการเขียนผลลัพธ์ก็ต้องการเขียนผลลัพธ์การทำงานลงสู่หน่วยความจำเช่นกัน จะเห็นได้ว่ามีสเตทจำนวน สามสเตทด้วยกันต้องการเข้าถึงหน่วยความจำพร้อมๆ กันซึ่งปกติแล้วหน่วยความจำทั่วๆ ไปไม่สนับสนุนให้อุปกรณ์หลายๆ ตัวเข้าถึงข้อมูลได้พร้อมๆ กัน การแก้ปัญหาคือการใช้สถาปัตยกรรมแบบ Harvard ซึ่งแยกหน่วยความจำของโปรแกรมและข้อมูลออกจากกันอย่างเด็ดขาดส่งผลให้สามารถเฟตช์โปรแกรมจากหน่วยความจำโปรแกรมได้พร้อมๆ กับการเขียนข้อมูลผลลัพธ์จากการทำโปรแกรมลงสู่หน่วยความจำข้อมูล

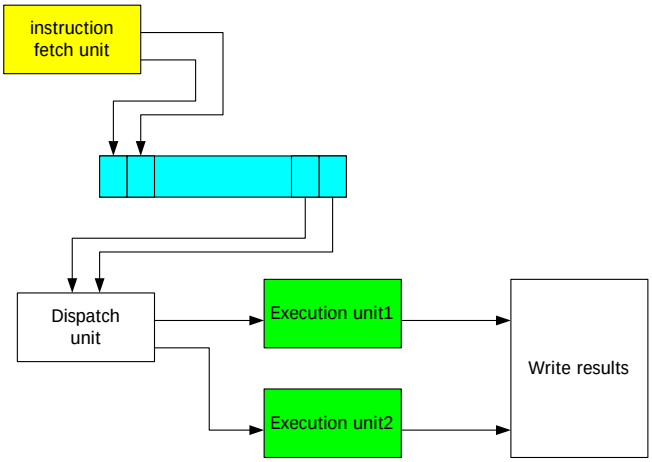


รูปที่ 7.16 สถาปัตยกรรมแบบ Harvard

7.5 **สถาปัตยกรรมแบบซุปเปอร์สเกลาร์** (Superscalar)

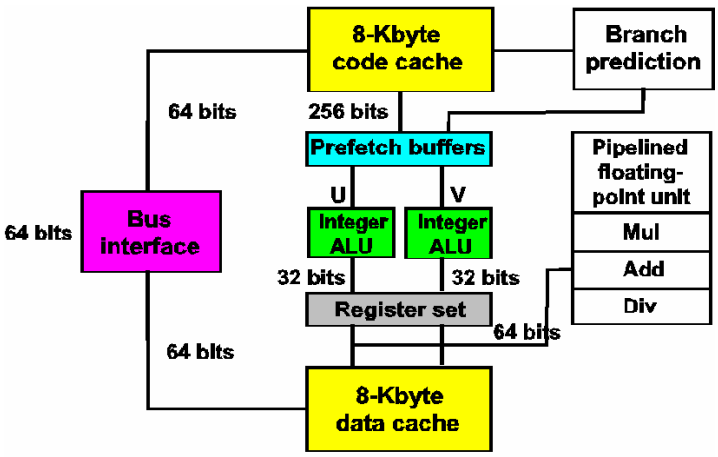
แม้ว่าการนำเทคนิคไปป์ลายน์มาใช้จะช่วยเพิ่มสมรรถนะของโพรเซสเซอร์ได้ แต่อย่างไรก็ตามเทคนิคไปป์ลายน์ไม่สามารถทำให้ซีพียูสามารถทำงานได้เกิน 1 คำสั่งต่อ 1 สัญญาณนาฬิกาได้ เทคนิค ซุปเปอร์สเกลาร์สามารถข้ามข้อจำกัดตรงนี้ได้โดยการเพิ่มหน่วย Execution unit ของโพรเซสเซอร์ให้มากกว่า 1 ส่วน ส่งผลให้สามารถเอกซีคิวต์คำสั่งมากกว่า 1 คำสั่งใน 1 สัญญาณนาฬิกาได้ รูปที่ 7.17 แสดงให้เห็นสถาปัตยกรรมแบบซุปเปอร์สเกลาร์ที่มีหน่วย Execution Unit จำนวน 2 ตัว ส่งผลให้สามารถเอกซีคิวต์คำสั่งได้พร้อมกัน 2 คำสั่งต่อ 1 สัญญาณ นาฬิกา อย่างไรก็ตาม เทคนิคซุปเปอร์สเกลาร์ก็ประสบปัญหาการขึ้นต่อกันของข้อมูลเช่นเดียวกันกับเทคนิคไปป์

ลายน์ ดังนั้นวงจร Dispatch unit จะต้องตรวจสอบการขึ้นต่อกันของคำสั่งก่อนส่งคำสั่งให้กับ Execution unit โดยหากพบว่าคำสั่งทั้งสองไม่ขึ้นต่อกันก็จะส่งคำสั่งให้กับ Execution unit ทั้งสองพร้อมกัน แต่หากมีการขึ้นต่อกันของคำสั่ง ตัว Dispatch unit จะส่งคำสั่งให้ Execution unit เพียงแค่ตัวเดียวเท่านั้น และจะส่งคำสั่ง No operation ให้ Execution unit อีกตัวทำงาน



รูปที่ 7.17 สถาปัตยกรรมแบบ Superscalar

ซีพียูของอินเทลตัวแรกที่ใช้เทคนิคไปป์ลายน์มาช่วยเพิ่มประสิทธิภาพการทำงานคือซีพียู 80486 ส่วนซีพียูของอินเทลตัวแรกที่ใช้เทคนิคซุปเปอร์สเกลาร์คือ Pentium รูปที่ 7.18 แสดงสถาปัตยกรรมภายในของโพรเซสเซอร์ Pentium ซึ่งเป็นซีพียูซุปเปอร์สเกลาร์แบบ 2 ทาง นั่นหมายถึง Pentium สามารถทำคำสั่งพร้อมกันได้สูงสุด 2 คำสั่ง โดยจะเฟตช์คำสั่งจากหน่วยความจำ แคชมาเก็บไว้ใน Prefetch buffer ก่อน แล้วหากพบว่าสามารถทำงานพร้อมกันได้ก็จะส่งคำสั่งให้ ALU ผ่านทาง U pipe และ V pipe พร้อมๆ กัน



รูปที่ 7.18 สถาปัตยกรรมของโพรเซสเซอร์ Pentium

7.5 โพรเซสเซอร์แบบมีการทำงานไม่เรียงลำดับ

โดยปกติ ไมโครโพรเซสเซอร์โดยทั่วไปจะเอกซีคิวต์คำสั่งตามลำดับที่ได้จัดเรียงไว้ในโปรแกรม ยกตัวอย่างเช่น คำสั่งที่ 1 จะถูกเอกซีคิวต์ก่อนคำสั่งที่ 2 ส่วนคำสั่งที่ 2 จะถูกเอกซีคิวต์ก่อนคำสั่งที่ 3 เป็นต้น อย่างไรก็ตาม จากงานวิจัยทางด้านสถาปัตยกรรมไมโครโพรเซสเซอร์หลายๆ ชิ้นพบว่าหากเปลี่ยนลำดับการเอกซีคิวคำสั่งเสียใหม่ โดยไม่ได้ทำให้ผลลัพธ์การทำงานเปลี่ยนแปลงไปในบางครั้งจะช่วยให้ประสิทธิภาพในการทำงานดีขึ้น นั่นคือที่มาของการทำงานแบบไม่เรียงลำดับ (Out-of-order execution : OOO)

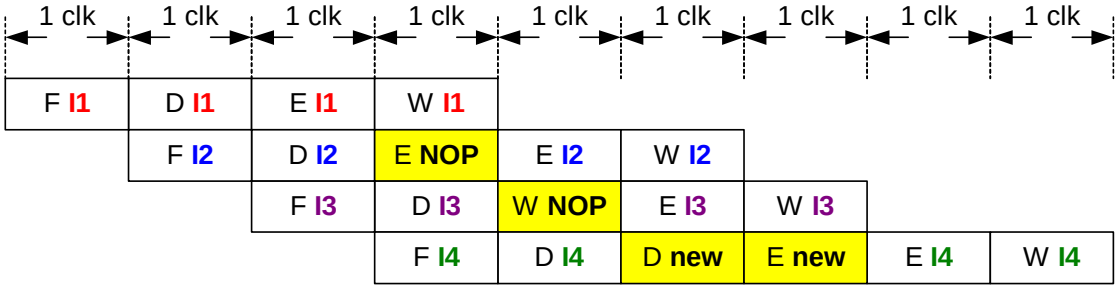
ยกตัวอย่างเช่น หากเราเขียนโปรแกรมในภาษาระดับสูงเพื่อคำนวณหาค่าผลลัพธ์ของการคูณตัวเลข 2 ค่าดังต่อไปนี้

w = x + y * z

a = (b + c) * d

กำหนดให้ตัวแปร a, b, c, d, w, x, y และ z เป็นข้อมูลชนิด floating-point ดังนั้นโปรแกรมทั้งสองบรรทัดที่กล่าวมาจะสามารถแปลได้เป็นคำสั่งภาษาแอสเซมบลี้ได้จำนวน 4 คำสั่ง ดังแสดงในรูปที่ 7.19 โดยกำหนดให้คำสั่ง FMUL คือคำสั่ง Floating-point Multiply และคำสั่ง FADD คือคำสั่ง Floating-point Add หากรันโปรแกรมตามลำดับของคำสั่งบนโพรเซสเซอร์แบบไปป์ลายขนาด 4 สเตทจะได้ผลลัพธ์การทำงานดังรูปที่ 7.19

I1 FMUL t1,y,z
I2 FADD w, x, t1
I3 FADD t2,b,c
I4 FMUL a,t2,d



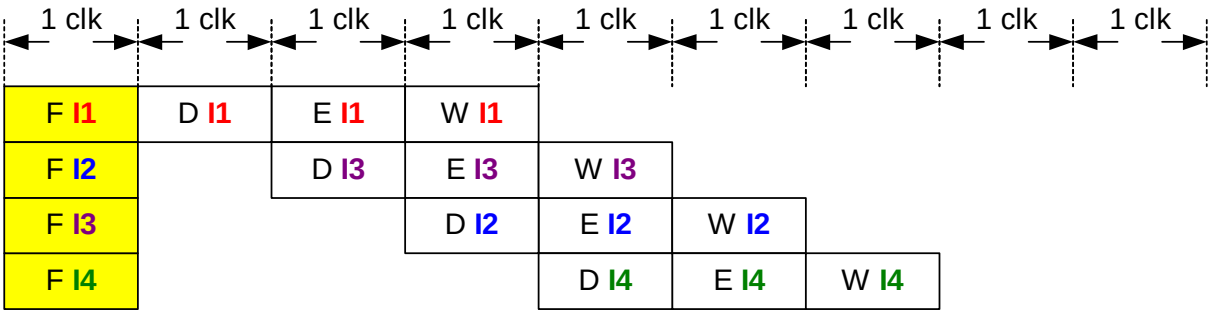
รูปที่ 7.19 การทำงานแบบ In-order execution ของสถาปัตยกรรมแบบไปป์ลายน์

จากรูปที่ 7.19 จะเห็นว่าหากไม่ใช้หลักการ Operand forwarding แล้ว คำสั่งที่ 2 ไม่สามารถที่จะเอกซีคิวต์พร้อมๆ กับคำสั่งที่ 1 ได้เลย ทั้งนี้สืบเนื่องจากคำสั่งที่ 2 ต้องการผลลัพธ์ของคำสั่งที่ 1 ดังนั้นในขณะที่เขียนผลลัพธ์ของคำสั่งที่ 1 สเตทเอกซีคิวต์จำทำคำสั่ง Nop : No Operation ไปก่อน จนกระทั่งคำสั่งที่ 1 เขียนข้อมูลเสร็จเรียบร้อยจึงจะสามารถเอกซีคิวต์คำสั่งที่ 2

ได้ ในกรณีของคำสั่งที่ 4 ก็เช่นเดียวกัน คือต้องรอให้คำสั่งที่ 3 เขียนข้อมูลเสร็จสิ้นก่อนจึงสามารถที่จะเอกซีคิวต์ได้ จะเห็นว่าใช้เวลาไปทั้งหมด 9 คล็อกไซเคิลในการปฏิบัติคำสั่งจำนวน 4 คำสั่ง

หากเราเปลี่ยนสถาปัตยกรรมของโพรเซสเซอร์ใหม่ ให้โพรเซสเซอร์สามารถเฟตช์คำสั่งเข้ามาพร้อมกันได้ครั้งละหลายๆ คำสั่ง โพรเซสเซอร์จะสามารถมองล่วงหน้าได้ว่าคำสั่งไหนสามารถทำงานก่อนคำสั่งอื่นเพื่อให้มีประสิทธิภาพที่สูงขึ้นโดยไม่ได้ทำให้ผลลัพธ์การทำงานเสียไปแล้ว เราจะเห็นว่าคำสั่งที่ 2 และคำสั่งที่ 3 นั้นไม่ขึ้นต่อกัน ดังนั้นไม่ว่าซีพียูจะเอกซีคิวต์คำสั่งที่ 2 ก่อนหรือหลังคำสั่งที่ 3 ก็จะให้ผลลัพธ์การทำงานเหมือนกัน แต่หากเอกซีคิวต์คำสั่งที่ 3 ก่อนจะส่งผลให้สเตท Execute สามารถเอกซีคิวต์คำสั่งที่ 3 พร้อมๆ กับคำสั่งที่ 1กำลังอยู่ในสเตท Writeback ได้ ซึ่งส่งผลให้ไม่ต้องเติมคำสั่ง No Operation เข้าไปในไปป์ลายน์ ส่งผลให้ประสิทธิภาพสูงขึ้น และจากการที่เราสลับการทำงานของคำสั่งที่ 2 และ 3 จะส่งผลให้คำสั่งที่ 4 ไม่ต้องรอให้คำสั่งที่ 2 เขียนผลลัพธ์เสร็จสิ้นก่อน ซึ่งจะเห็นว่าหากการนำเทคนิค Out-of-Order execution เข้ามาใช้จะสามารถทำให้ซีพียูสามารถเอกซีคิวต์คำสั่งทั้ง 4 คำสั่งนี้ได้ในเวลาเพียง 7 คล็อกไซเคิล โดยไม่ต้องใช้เทคนิค Operand forwarding เข้ามาช่วยเลย

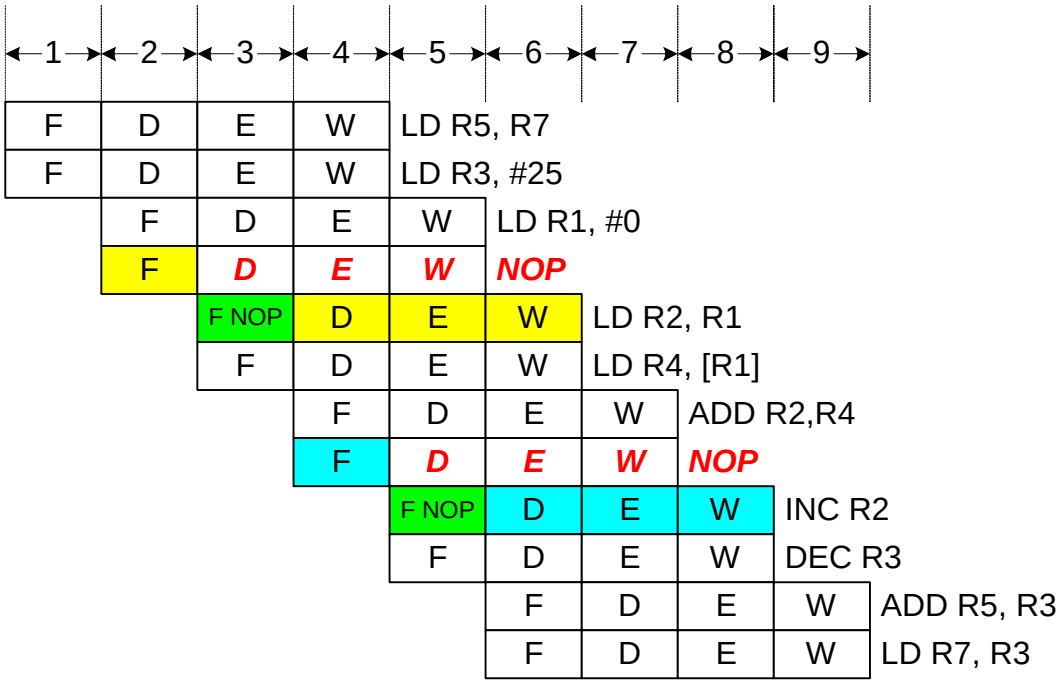
I1 FMUL t1,y,z
I2 FADD w, x, t1
I3 FADD t2,b,c
I4 FMUL a,t2,d



รูปที่ 7.20 การเอกซีคิวต์คำสั่งแบบ Out-of-Order ในซีพียูแบบไปป์ลายน์

อย่างไรก็ตาม เทคนิค Operand forwarding ในซีพียูแบบไปป์ลายน์นี้ หากนำมาใช้ควบคู่กับเทคนิค Superscalar ก็จะช่วยให้ประสิทธิภาพการทำงานสูงขึ้น ยกตัวอย่างดังเช่น ในซีพียูแบบซุปเปอร์สเกลาร์แบบ 2 ทางและใช้ไปป์ลายน์แบบ 4 สเตทและมีการฟอร์เวิร์ดโอเปอแรนด์หากมีการขึ้นต่อกันของข้อมูลในคำสั่ง ในการทำคำสั่งต่อไปนี้ จะได้ผลลัพธ์การทำงานดังแสดงรูปที่ 7.21

คำสั่งที่ 1	LD R3, #25
คำสั่งที่ 2	LD R3, #25
คำสั่งที่ 3	LD R1, #0
คำสั่งที่ 4	LD R2, R1
คำสั่งที่ 5	LD R4, [R1]
คำสั่งที่ 6	ADD R2,R4
คำสั่งที่ 7	INC R2
คำสั่งที่ 8	DEC R3
คำสั่งที่ 9	ADD R5, R3
คำสั่งที่ 10	LD R7, R3



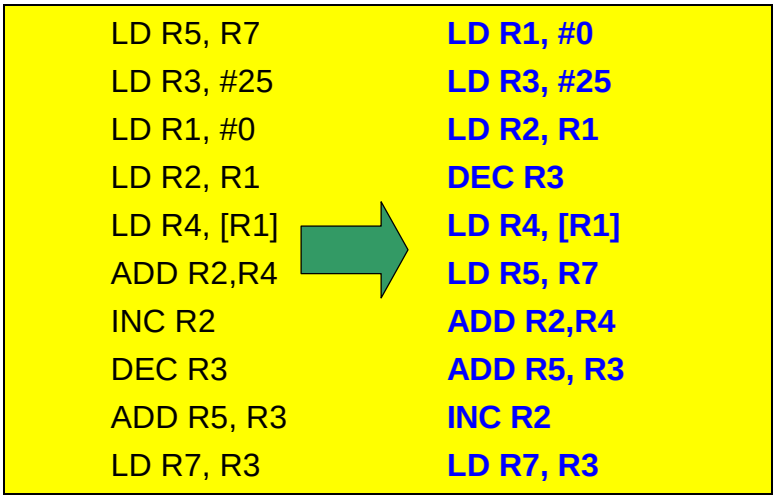
รูปที่ 7.21 การทำงานของซีพียูแบบซุปเปอร์สเกลาร์ 2 ทางแบบ In-order execution

จากรูปที่ 7.21 จะเห็นว่าคำสั่งที่ 4 และคำสั่งที่ 3 ไม่สามารถที่จะเอกซีคิวต์ได้พร้อมกันในสถาปัตยกรรมแบบซุปเปอร์สเกลาร์ เนื่องจากคำสั่งที่ 4 ต้องการผลลัพธ์ของคำสั่งที่ 3 ถึงแม้ว่าจะมีเทคนิค Operand forwarding เข้ามาใช้ก็ตาม เนื่องจากการฟอร์เวิร์ดโอเปอแรนด์นั้น เราสามารถฟอร์เวิร์ดข้อมูลจากสเตท Writeback มาให้สเตท Execute ได้ แต่ไม่สามารถฟอร์เวิร์ด

ข้อมูลจากสเตทเอกซีคิวต์ของหน่วย Execution unit หน่วยที่ 1 กลับมายังหน่วย execution unit หน่วยที่ 2 ได้ ดังนั้นคำสั่ง Nop จึงถูกเติมเข้าไปในไปป์ลายน์หน่วยที่ 2 ในคล๊อกที่ 3 และคำสั่งที่ 4 จึงสามารถถูกถอดรหัสได้ในคล๊อกไซเคิลที่ 4 ซึ่ง เทคนิค การฟอร์เวิร์ดโอเปอแรนด์ช่วยให้สามารถเอกซีคิวต์คำสั่งที่ 4 ได้พร้อมๆ กับการเขียนผลลัพธ์ของคำสั่งที่ 3

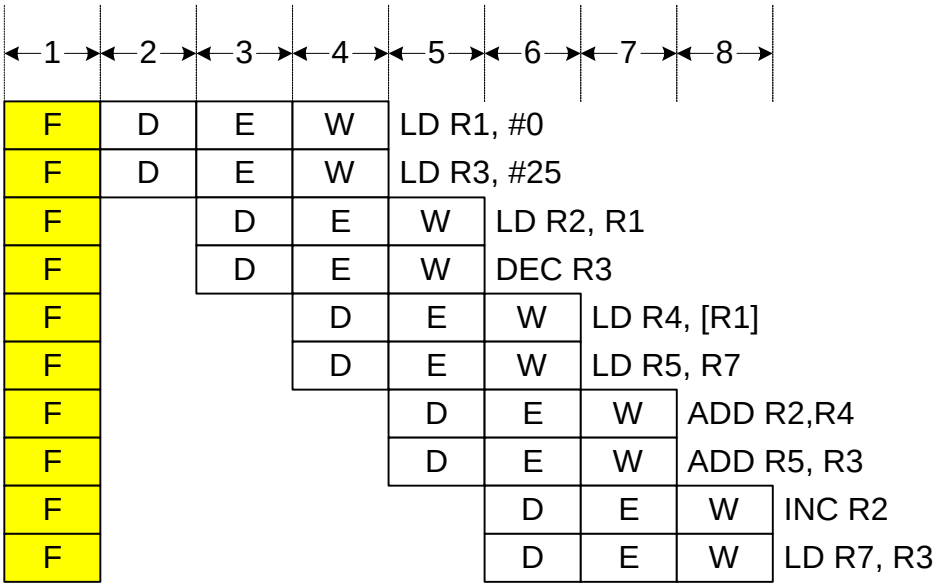
ส่วนคำสั่งที่ 6 และคำสั่งที่ 7 ก็เช่นเดียวกันกับกรณีที่เกิดขึ้นกับคำสั่งที่ 3 และคำสั่งที่ 4 ซึ่งส่งผลให้มีการเติมคำสั่ง Nop เข้าไปในไปป์ลายน์อีก 1 ครั้ง จะเห็นว่า โปรแกรมดังกล่าวใช้เวลาในการปฏิบัติคำสั่งจำนวน 10 คำสั่ง ในเวลา 9 คล๊อกไซเคิล

หากนำโปรแกรมดังที่ได้กล่าวมาแล้วมาจัดเรียงลำดับใหม่ดังรูปที่ 7.22 ก็จะสามารถทำให้โปรแกรมสามารถทำงานได้เร็วขึ้นดังแสดงผลในรูปที่ 7.23



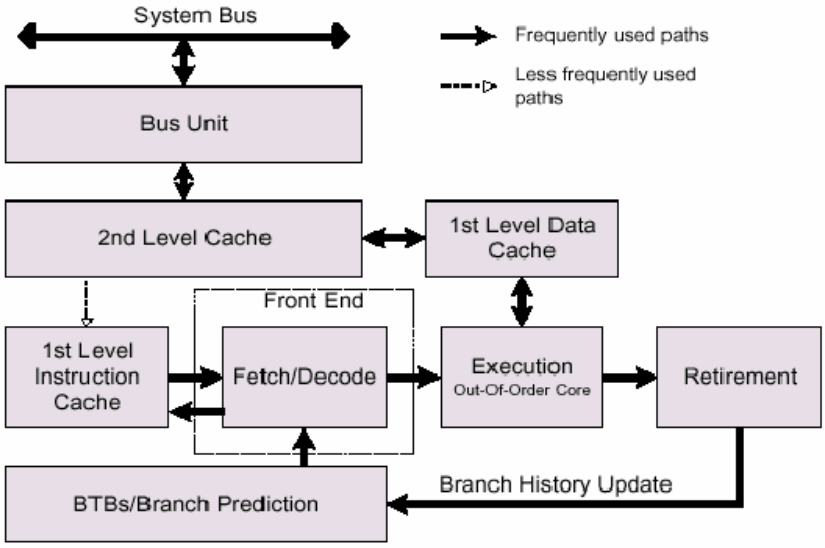
รูปที่ 7.22 การจัดเรียงลำดับการส่งข้อมูลให้ไปป์ลายน์ใหม่โดยไมโครโพรเซสเซอร์

จากรูปที่ 7.23 จะเห็นได้ว่าหากโพรเซสเซอร์จัดเรียงลำดับการทำงานของคำสั่งใหม่ จะทำให้ลดเวลาในการเอกซีคิวต์ลงได้จากเดิมที่ใช้ 9 คล๊อกไซเคิลเหลือ 8 คล๊อกไซเคิล อย่างไรก็ตาม การจัดเรียงคำสั่งใหม่จะต้องการวงจรสนับสนุนที่มีความสลับซับซ้อนสูงมากและนอกจากนี้โพรเซสเซอร์จะต้องมีความสามารถในการเฟตช์คำสั่งเข้ามาพร้อมกันได้หลายๆ คำสั่ง จึงจะสามารถมองไปข้างหน้าได้ว่าควรจะสลับคำสั่งอย่างไรจึงจะให้ผลลัพธ์ออกมาดีที่สุด

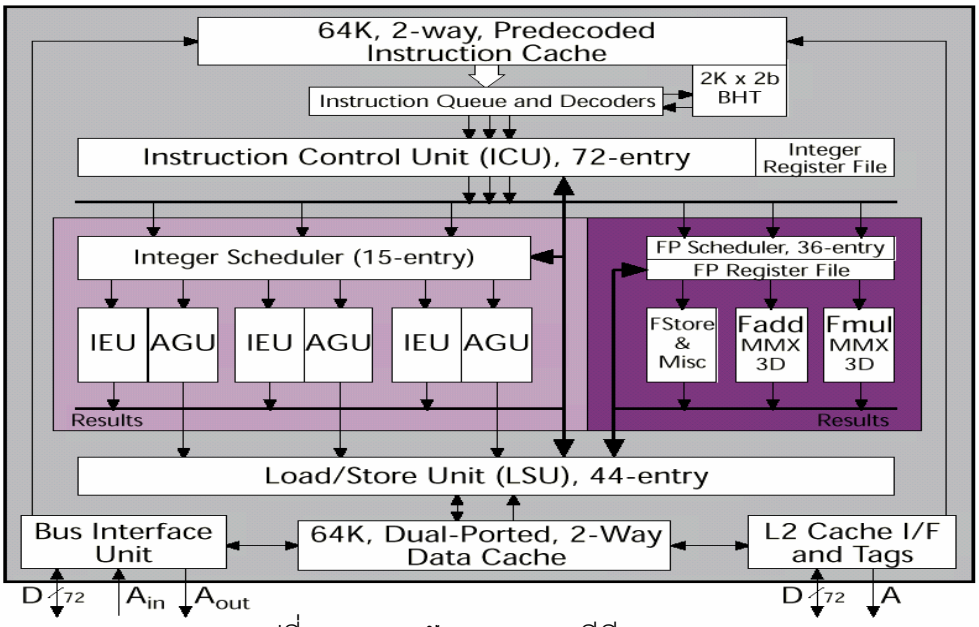


รูปที่ 7.23 การทำงานของซีพียูแบบซุปเปอร์สเกลาร์ 2 ทางแบบ Out-of-Order execution

โพรเซสเซอร์ตัวแรกของอินเทลที่สนับสนุนการทำงานแบบ Out-of-Order execution คือ โพรเซสเซอร์ Pentium Pro ซึ่งแสดงให้เห็นในรูปที่ 7.24 โดยซีพียูรุ่นใหม่ๆ ในปัจจุบันจะสนับสนุนเทคนิค Out-of-Order execution แทบทั้งสิ้น ในรูปที่ 7.25 แสดงสถาปัตยกรรมของซีพียู Athlon ซึ่งประกอบด้วย Reorder buffer ซึ่งสามารถเก็บคำสั่งได้ถึง 72 คำสั่ง และ Reorder buffer จะทำการส่งคำสั่งให้กับ Integer Scheduler และ Floating-point Scheduler ซึ่งมีหน้าที่จัดเรียงลำดับของคำสั่งสำหรับเลขจำนวนเต็มและเลขทศนิยม ตามลำดับ โดยที่ Integer Scheduler จะสามารถบรรจุคำสั่งได้



รูปที่ 7.24 วงจรสนับสนุนการทำงาน OOO ในสถาปัตยกรรมของ P6



รูปที่ 7.25 สถาปัตยกรรมของซีพียู Athlon

จำนวน 15 คำสั่งและ Floating-point scheduler สามารถบรรจุคำสั่งในการจัดการเลขทศนิยมได้ถึง 36 คำสั่งเลยทีเดียว ซีพียูในรุ่นใหม่ๆ จะนำเทคนิคที่ได้กล่าวมาแล้วตั้งแต่เรื่องของหน่วยความจำแคช สถาปัตยกรรมแบบไปป์ลายน์ สถาปัตยกรรมแบบซุปเปอร์สเกลาร์และ OOO มาใช้ช่วยเพิ่มประสิทธิภาพการทำงานให้สูงขึ้น
