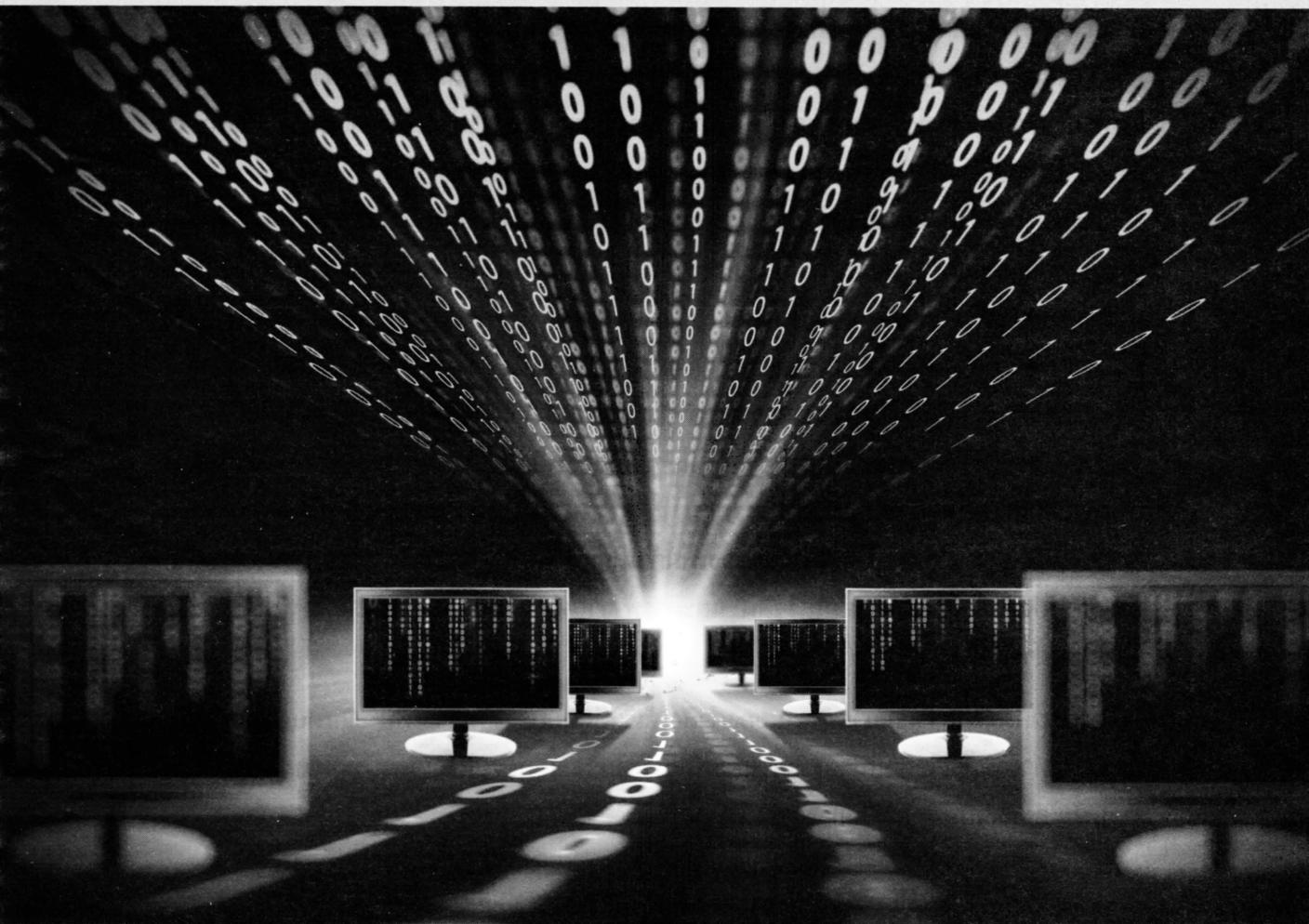


# 6

## การทำงานแบบไปoline

เบื้องต้นการเขียนโปรแกรม



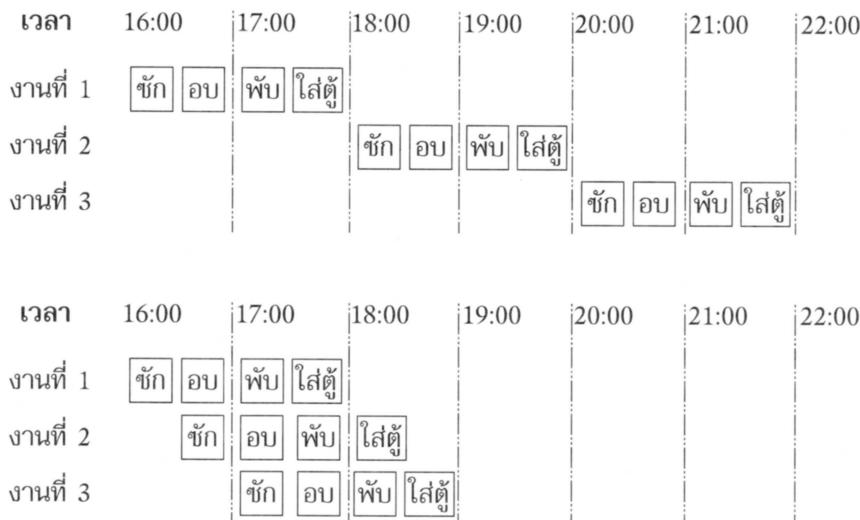
ในปัจจุบัน สถาปัตยกรรมคอมพิวเตอร์อาศัยการทำงานแบบไปปีลน์เพื่อให้การทำงานเร็วขึ้น การทำงานแบบไปปีลน์เป็นการสนับสนุนการทำงานแบบขนาดใหญ่ในรูปแบบหนึ่ง และพัฒนามาจากแนวคิดของการทำงานแบบหลายไชเกิล ในบทนี้จะกล่าวถึงลักษณะการทำงานแบบไปปีลน์ ประโยชน์ของการทำงานแบบไปปีลน์ การประยุกต์การออกแบบ Data Path และ Control Path กับไปปีลน์ ประเด็นการจัดการเรื่อง Exception และขัดจังหวะ (Interrupt) ในกรณีการใช้ไปปีลน์ความสัมพันธ์กับประสิทธิภาพ ตามลำดับ

## ■ 6.1 แนวคิดการทำงานแบบไปปีลน์

จากตัวอย่างในบทที่แล้วที่ได้กล่าวถึงการทำงานแบบไชเกิลเดียวและหลายไชเกิลนั้น จุดประสงค์หลักของการออกแบบก็คือ การเพิ่มประสิทธิภาพ และเป้าหมายที่สำคัญก็คือการลดค่า CPI ให้เหลือน้อยที่สุด จากการที่ใช้หลายไชเกิลอาจจะทำให้การออกแบบ Data Path ที่ได้นั้นทำให้บางคำสั่งมีค่า CPI เป็น 1 และบางคำสั่งต้องการหลายไชเกิลในการทำงานอยู่ ดังนั้น แนวคิดการทำให้ CPI เป็น 1 จึงยังเป็นเป้าหมายในการออกแบบ การใช้รูปแบบหลายไชเกิลยังมีข้อเสียในการใช้ทรัพยากรอยู่ เพราะว่าระหว่างการคำนวนคำสั่งหนึ่งๆ หน่วยคำนวนอื่นอาจจะว่าง ไม่ได้ถูกใช้งาน ถ้าในแต่ละไชเกิล หน่วยคำนวนทุกหน่วยถูกใช้งานอย่างเต็มที่ การลด CPI ให้เหลือ 1 ก็จะเป็นไปได้มากขึ้น แนวคิดของการทำงานระหว่างคำสั่งจึงเกิดขึ้น โดยให้คำสั่งที่สองเริ่มทำงานเมื่อหน่วยคำนวนพร้อม ไม่ต้องรอให้คำสั่งแรกทำงานเสร็จลื้น เรียกว่าเป็นการทำงานทับซ้อน (Overlap) การทำงานของคำสั่ง ทั้งนี้ก็เพื่อที่จะเริ่มการทำงานของคำสั่งใหม่ให้เร็วที่สุด โดยอาจจะไม่ต้องให้คำสั่งก่อนหน้าเสร็จสมบูรณ์ โดยมองที่ทรัพยากรที่ต้องใช้ในการทำงานของคำสั่งนั้นๆ เป็นหลัก หากหน่วยต่างๆ ที่ต้องใช้ในการประมวลผลคำสั่งนั้นพร้อม ก็จะให้เริ่มทำงานได้ รูปแบบนี้จึงเป็นรูปแบบหนึ่ง ที่ทำให้การใช้ทรัพยากร่วมกันมีประสิทธิภาพอีกด้วย แต่อย่างไรก็ดี แม้จะเป็นการเพิ่มประสิทธิภาพโดยการใช้ทรัพยากร่วมกัน แต่ฮาร์ดแวร์ที่ใช้ในการทำงานก็จะซับซ้อนมากขึ้น Data Path และ Control Path ก็จะต้องเพิ่มตัวเลือก (MUX) และบัฟเฟอร์ในการเก็บผลลัพธ์ ขั้วรวมของคำสั่งที่ยังทำงานไม่เสร็จลื้น และการทำงานดังกล่าวอาจจะมีข้อเสียอื่นๆ ที่ต้องระวังดังจะได้กล่าวต่อไป

แนวคิดของการเริ่มการทำงานของคำสั่งให้เร็วที่สุดนั้นจะเป็นการเพิ่ม Throughput หรืออัตราการทำงานสำเร็จต่อหน่วยเวลา และเป็นการลด Latency หรือเวลาประมวลผลของคำสั่งนั้นๆ นั่นเอง จึงเป็นผลให้เป็นการลด CPI

แนวคิดการทำงานของไปปีลайнจากหนังสือของ Hennessy เปรียบเทียบได้กับขั้นตอนของ การซักผ้า ดังแสดงในรูปที่ 6.1 ที่ประกอบด้วยขั้นตอนการซัก การอบแห้ง การพับผ้า และการเก็บใส่ตู้ ขั้นตอนทั้งสี่เป็นขั้นตอนที่ต้องมี รูปด้านบนเป็นวิธีการแบบหลายช่วงเวลา ช่วงเวลาละ 30 นาที โดยงานแต่ละขั้นใช้เวลา 30 นาที แต่รูปด้านล่างจะเป็นการหับข้อนการทำงานระหว่างขั้นตอนในแบบไปปีลайн จะเห็นว่าระหว่างการอบเสื้อผ้าชุดแรก เราจะสามารถนำผ้าจากชุดที่ 2 มาซักได้เลย เนื่องจากเครื่องซัก瓜่งแล้ว เมื่อไปปีลайнเต็มนั้น หมายความว่า หน่วยต่างๆ หรือทรัพยากรถูกใช้งาน ทั้งหมดในเวลาหนึ่ง ซึ่งจากตัวอย่างนี้จะเห็นว่าสามารถซักเสื้อผ้าได้ทั้งหมด 4 ชุดพร้อมกันนั่นเอง จะเห็นว่าในรูปแบบหับข้อนนี้จะทำให้งานเสร็จเร็วกว่า

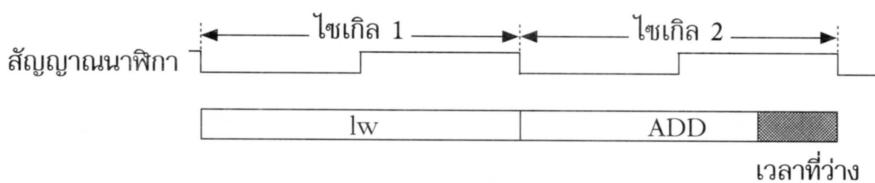


รูปที่ 6.1 การซักผ้าแบบหลายชุดเกล (รูปบน) และแบบไปปีลайн (รูปล่าง)

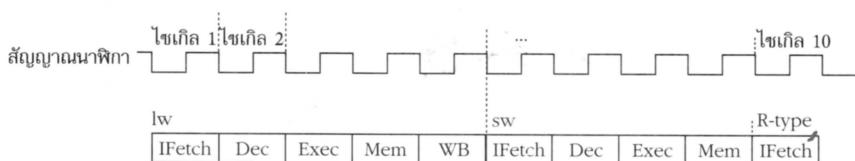
มุ่งมองอีกอย่างหนึ่ง การสร้างซอฟต์แวร์แบบ Waterfall อาจจะประกอบด้วยขั้นตอนต่างๆ ดังแสดงในรูปที่ 6.2 ได้แก่ การวิเคราะห์ความต้องการ (Requirement Analysis) การออกแบบ (Design) การพัฒนา (Implementation) และการทดสอบ (Testing) โดยบริษัทอาจจะมีแต่ละแผนกทำหน้าที่ในแต่ละขั้นตอน ในการทำงานแต่ละโปรเจกต์ก็จะทำได้ในแบบไปปีลайн เช่น ก็จะสามารถเริ่มงานวิเคราะห์ความต้องการของโปรเจกต์ที่สองได้เลย เพื่อไม่ให้เสียเวลาและใช้ทรัพยากรให้คุ้มค่า

รูปที่ 6.2 ตัวอย่างการทำงานแบบไปป์ไลน์

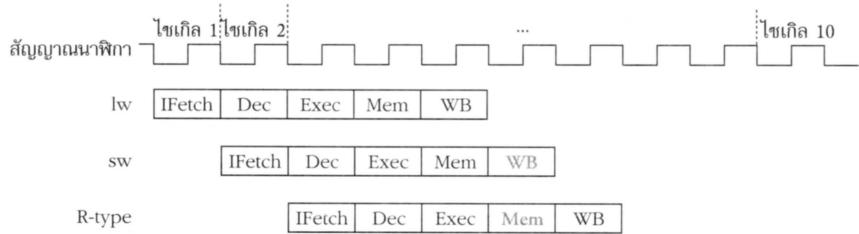
เมื่อนำแนวคิดมาใช้กับขั้นตอนการทำงานของคำสั่ง พิจารณาจากรูปที่ 6.3 ถึงรูปที่ 6.5 จากรูปที่ 6.3 ที่เป็นรูปแบบไข่เกลเดียวย ซึ่งจะเห็นว่า Cycle Time จะขึ้นอยู่กับขั้นตอนของคำสั่งที่ทำงานนานที่สุด รูปที่ 6.4 เป็นการทำงานแบบหลายไข่เกล จะเห็นว่าจำนวนไข่เกลที่ใช้ต่อคำสั่งไม่เท่ากัน ขึ้นอยู่กับลักษณะของคำสั่ง และรูปที่ 6.5 ไข่เกลเป็นการทำงานแบบไปปีลайн ซึ่ง 1 รอบจะทำเพียงขั้นตอนเดียวเท่านั้น ในเวลาหนึ่งๆ จะเห็นว่าจะมีคำสั่งที่ทำงานอยู่ในขั้นตอนต่างๆ กัน ดังนั้น การออกแบบความยาวของรอบเวลาหนึ่งจะขึ้นอยู่กับขั้นตอนที่ใช้เวลามากที่สุด และจำนวนคำสั่งที่ทำได้พร้อมกันมากที่สุดในเวลาหนึ่งจะขึ้นอยู่กับจำนวนขั้นตอนที่ใช้ห้องแม่ข่ายไปปีลайн



รูปที่ 6.3 การทำงานแบบไฮเกิลเดีย



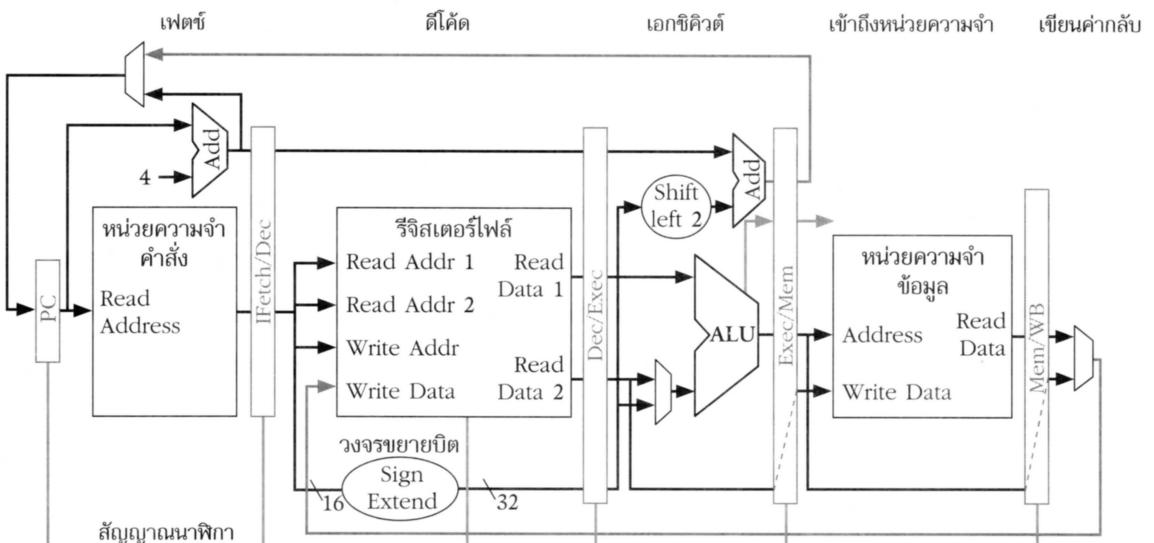
รูปที่ 6.4 การทำงานแบบหลอยใช้เกล



รูปที่ 6.5 ไขเกิลการทำงานแบบไปป์ไลน์

จะเห็นว่าในการทำงานแบบหลายไขเกิล เมื่อรูปแบบคำสั่งต่างกัน จะใช้จำนวนไขเกิลต่อคำสั่งไม่เท่ากัน ดังนั้น จึงจำเป็นต้องมีการปรับปรุง Data Path ของเดิมให้รองรับการทำงานลักษณะไปป์ไลน์นี้

พิจารณา Data Path ของ MIPS รูปแบบตามที่อ่อนหน้านี้ จะเห็นว่าหากแบ่งการทำงานเป็นขั้นตอน จะเทียบได้กับการแบ่ง Data Path ดังแสดงในรูปที่ 6.6



รูปที่ 6.6 การแบ่งขั้นตอนต่างๆ ของ MIPS

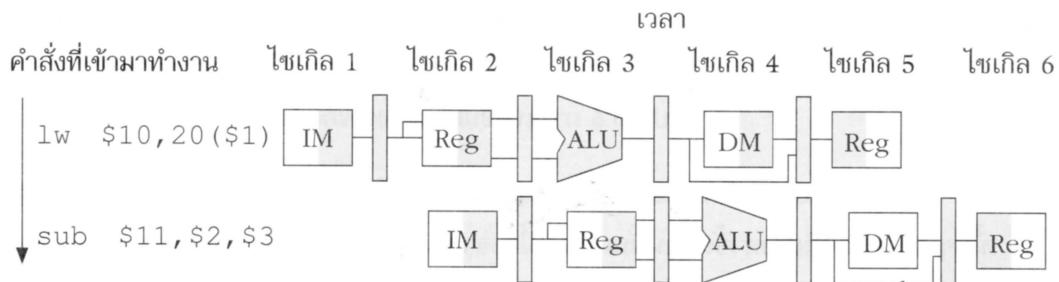
ในรูปที่ 6.6 ແບບที่ค้นระหว่างขั้นตอนจะหมายถึงบัฟเฟอร์เก็บผลลัพธ์การทำงานของขั้นตอนนั้นๆ ของคำสั่งที่เพิ่งทำขั้นตอนนั้นเสร็จ เนื่องจากในเวลาหนึ่งๆ แต่ละขั้นตอนจะทำงานในคำสั่งที่ต่างกัน ดังนั้น การป้อนอินพุตเข้าทำงานแต่ละขั้นตอนจึงจะต้องระมัดระวังมากขึ้น ในรูปนี้จะแสดงให้

เพื่อนการใช้หน่วยต่างๆ ที่สำคัญในแต่ละขั้นตอนด้วย เช่น ขั้นการเฟช (Fetch) ต้องใช้หน่วยความจำคำสั่งในการอ่านคำสั่งเข้ามาทำงาน ขั้นดีโคด (Decode) จะต้องใช้รีจิสเตอร์ไฟล์ในการถอดรหัสคำสั่ง ขั้นເອັກຝຶກົວຕົວ (Execute) ต้องใช้ ALU ในการคำนวณ ขั้นเข้าถึงหน่วยความจำ (Memory Access) เป็นขั้นของการใช้หน่วยความจำข้อมูลเพื่ออ่านหรือเขียน และขั้นสุดท้ายเป็นการเขียนค่ากลับ (Write Back) ซึ่งใช้รีจิสเตอร์ไฟล์เพื่อเขียนค่าเข่นกัน หน่วยทรัพยากรที่ต้องใช้ในขั้นเหล่านี้จะต้องนำถูกคำนึงถึงเมื่อทำงานคำสั่งในขั้นตอนต่างๆ รูปที่ 6.7 เป็นสัญลักษณ์ของแต่ละหน่วยโดยย่อที่ใช้สำหรับแต่ละขั้นตอน



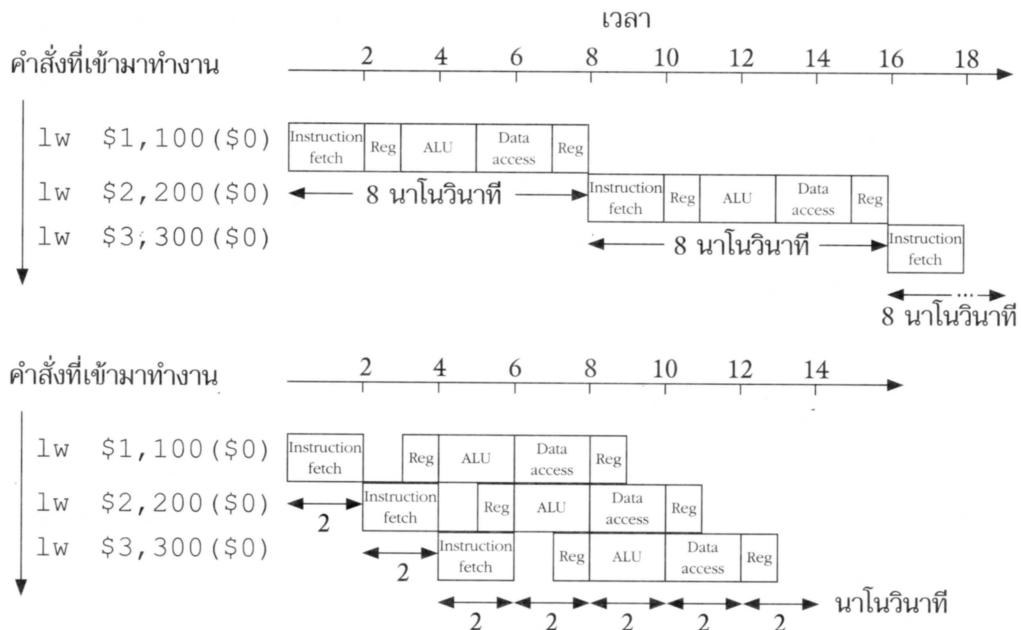
รูปที่ 6.7 สัญลักษณ์ของการใช้ขั้นตอนต่างๆ

รูปที่ 6.8 เป็นตัวอย่างการทำงานคำสั่ง LW และ SUB เมื่อพิจารณาการใช้หน่วยต่างๆ ในแต่ละขาเกิล จะเห็นว่าแบบสีแสดงเวลาที่ใช้ในแต่ละขาเกิลนั้นที่ได้ใช้งานหน่วยนั้นๆ จริง จากรูปนี้จะพอมองเห็นว่าขั้นตอนการทำงานทั้ง 5 ขั้นตอนนั้นเป็นขั้นตอนที่ต้องสัมพันธ์กับการออกแบบ Data Path แม้ว่าบางคำสั่งจะไม่ได้ใช้งานขั้นตอน การทำงานทั้งหมดก็ยังต้องเป็น 5 ขั้นตอน ในรูปจะเห็นว่าคำสั่ง SUB เป็นการลบระหว่างค่าในรีจิสเตอร์ซึ่งไม่ต้องการใช้หน่วยความจำข้อมูล แต่ขั้นตอนนี้ก็ไม่สามารถหลีกเลี่ยงไปได้ ทำให้จำนวนขาเกิลที่ต้องใช้สำหรับคำสั่ง SUB เป็น 5 ขาเกิล เช่น ในเวลาขาเกิลที่ 4 คำสั่ง SUB จะอยู่ในขั้นตอนເອັກຝຶກົວຕົວซึ่งใช้ ALU อยู่ และคำสั่ง LW จะอยู่ในขั้นตอนที่ใช้หน่วยความจำข้อมูลอยู่



รูปที่ 6.8 การทำงานคำสั่ง LW, SUB โดยใช้สัญลักษณ์ของการใช้ขั้นตอนต่างๆ

รูปที่ 6.9 แสดงการเปรียบเทียบเวลาการทำงานที่ใช้สำหรับการทำงาน 2 รูปแบบ ในโ dik นี้ ประกอบด้วยคำสั่ง LW สามคำสั่งที่ติดกัน



รูปที่ 6.9 เปรียบเทียบเวลาการทำงานระหว่างหลายไชเกิลและไปป์ไลน์

ตัวอย่างเช่น ในการทำงานนี้สมมติว่าเวลาของแต่ละหน่วยตามขั้นตอนเป็นดังนี้

ขั้นตอน	เวลาที่ใช้ (นาโนวินาที)
เฟตซ์คำสั่งจากหน่วยความจำคำสั่ง	2
อ่านค่าโอเพอแรนด์	1
เอิกซ์คิวต์โดยใช้ ALU	2
เข้าถึงหน่วยความจำข้อมูล	2

ในรูปที่ 6.9 จะเห็นว่าคำสั่ง LW จะใช้เวลาทั้งหมด 8 นาโนวินาทีสำหรับการทำงาน คำสั่ง LW สามคำสั่งจะใช้เวลาทั้งหมด 24 นาโนวินาที จากตารางจะพบว่าการออกแบบไปป์ไลน์นั้น เวลาของแต่ละไชเกิลจะขึ้นอยู่กับขั้นตอนที่ใช้เวลามากที่สุด ในที่นี้ได้แก่ 2 นาโนวินาที เนื่องจาก

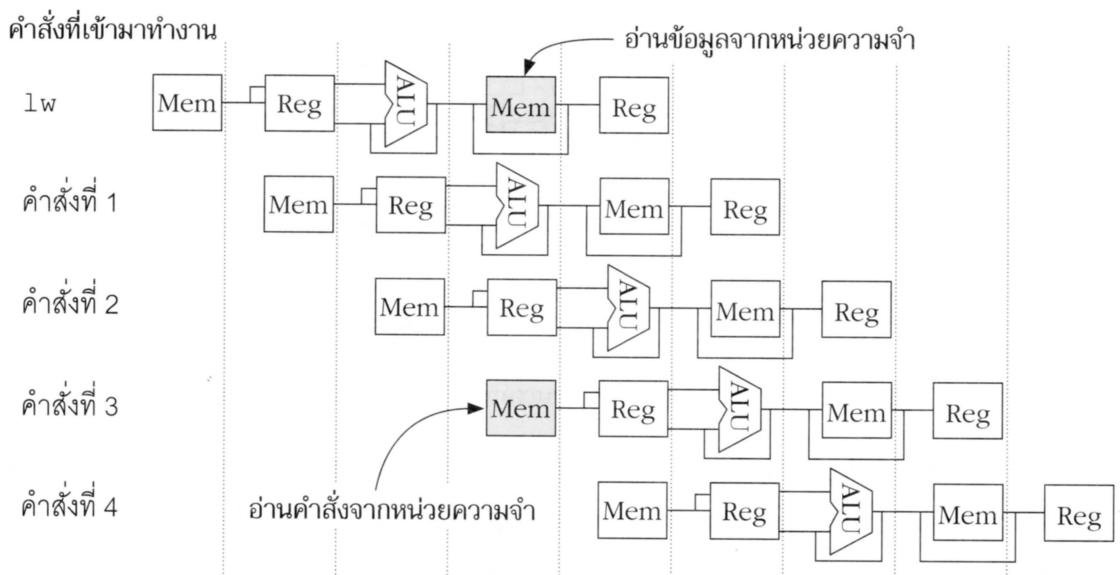
1 ไซเกิลจะต้องทำหลายขั้นตอน ดังนั้น คำสั่ง LW หนึ่งคำสั่งจะใช้เวลา 10 นาโนวินาที แต่เนื่องจากการทำงานสามารถทับช้อนขั้นตอนได้ การทำงานทั้งหมดสามคำสั่งนี้จะใช้เวลา 14 นาโนวินาที ดังแสดงในรูปที่ 6.9 ดังนั้น Speedup จะเท่ากับ  $24/14 = 1.7$  เท่า การทำงานโดยการทับช้อนการทำงานของคำสั่งต่างๆ นี้ บางครั้งจะเรียกว่าเป็นการทำงานแบบบานานะดับคำสั่ง (Instruction-Level Parallelism / ILP) เนื่องจากเป็นรูปแบบการทำงานแบบบานานะ แต่ละคำสั่งสามารถทำงานพร้อมกันได้ แต่อยู่ในขั้นตอนต่างกัน จริงๆ และการทำงานแบบบานานะมีหลายรูปแบบ ไม่ว่าจะเป็นรูปแบบมัลติทาสก์ รูปแบบมัลติโพรเซสเซอร์ ลักษณะของไปปีลайнก็จัดว่าเป็นรูปแบบการทำงานแบบบานานได้แบบหนึ่ง และการทำงานแบบไปปีลайнได้ถูกใช้อย่างแพร่หลายในสถาปัตยกรรมคอมพิวเตอร์ปัจจุบันแล้ว

## ■ 6.2 Hazard แบบต่างๆ

แม้ว่าการทำงานแบบไปปีลайнจะเอื้อ กับการทำงานแบบบานาน แต่เนื่องจากมีการใช้ทรัพยากร่วมกัน ทำให้อาจจะมีปัญหาต่างๆ เกิดขึ้น ซึ่งอาจจะเนื่องมาจากการตัวโคดโปรแกรมเอง หรืออาจจะมาจากการ hardware ก็ได้ สิ่งเหล่านี้รวมเรียกว่า Hazard ได้มีการแบ่งประเภทของ Hazard ในไปปีลайнเป็น 3 ประเภท ได้แก่ Structural Hazard, Data Hazard และ Control Hazard ดังรายละเอียดต่อไปนี้

### 6.2.1 Structural Hazard

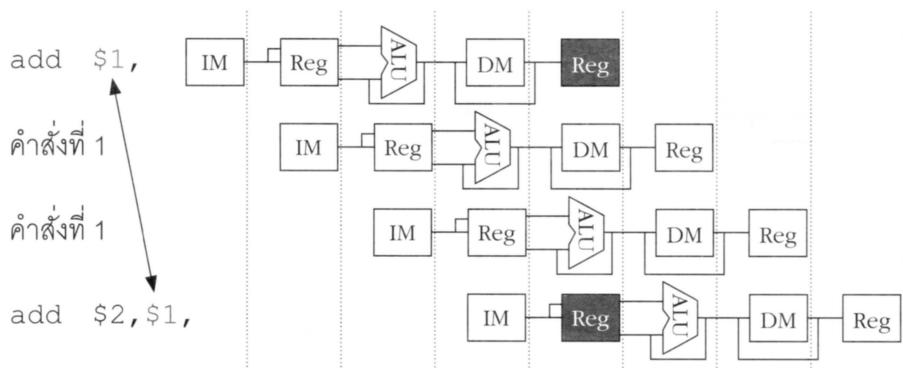
Hazard ที่เกิดจากปัญหาทางด้าน hardware และโครงสร้าง เนื่องจากในเวลาหนึ่งมีสองคำสั่งจะต้องการใช้หน่วยเดียวกัน ทั้งนี้ Hazard แบบนี้จะขึ้นอยู่กับการออกแบบ hardware หากมีจำนวนหน่วยคำนวนต่างๆ จำกัด ก็จะทำให้มีปัญหานี้เกิดขึ้นได้ ดังแสดงในรูปที่ 6.10



รูปที่ 6.10 สถานการณ์การเกิด Structure Hazard

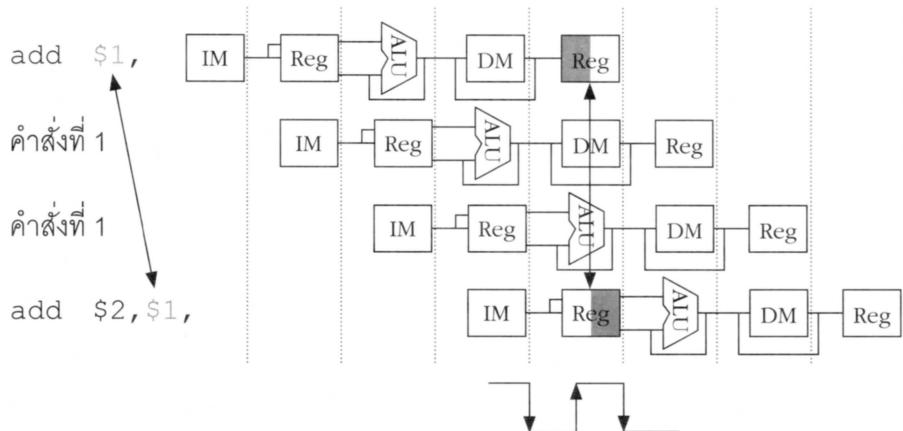
จากรูปที่ 6.10 จะเห็นว่าในไบเกิลที่ 4 จะมีโอกาสการเกิด Structure Hazard เนื่องจาก การใช้หน่วยความจำ ในคำสั่งแรกจะใช้หน่วยความจำ เพราะต้องการอ่านค่าข้อมูลที่หน่วยความจำ คำสั่งที่สามต้องการอ่านคำสั่งจากในหน่วยความจำ ถ้าฮาร์ดแวร์นี้มีหน่วยความจำข้อมูลที่พิพอร์ต อ่านข้อมูลพอร์ตเดียว จะทำให้หลายหน่วยไม่สามารถอ่านค่าจากหน่วยความจำข้อมูลได้พร้อมกัน ทำให้คำสั่งที่สามต้องหยุดรอ (Stall) หนึ่งไบเกิลเพื่อให้พิพอร์ตสำหรับอ่านว่างจากการใช้งานก่อน จะทำการอ่านจากพิพอร์ตได้ หรืออาจจะแก้ปัญหาโดยการแยกหน่วยความจำข้อมูลและหน่วยความจำคำสั่งออกจากกัน ในคำสั่งแรกจะเป็นการอ่านหน่วยความจำข้อมูล และคำสั่งหลังจะเป็นการอ่านหน่วยความจำคำสั่ง ซึ่งจะทำได้พร้อมกันเพื่อลดข้อขัดแย้งในการขอใช้พิพอร์ตในเวลาเดียวกัน

รูปที่ 6.11 แสดงตัวอย่างของปัญหาการเข้าถึงรีจิสเตอร์ไฟล์ ในไบเกิลที่ 5 เมื่อคำสั่ง Add ตัวแรกเข้าสู่ขั้นตอนการเขียนผลลัพธ์ และคำสั่ง Add ตัวหลังกำลังอ่านໂໂປແຣນດ์จากรีจิสเตอร์ไฟล์ ตัวอย่างนี้เป็นปัญหาที่เกิดขึ้นเมื่อคำสั่งแรกเป็นคำสั่ง Add ระหว่างรีจิสเตอร์อยู่ในขั้นตอนการเขียนผลลัพธ์ลงสู่รีจิสเตอร์ และคำสั่งที่สี่เป็นคำสั่ง Add เช่นกัน แต่ต้องการต่อตัวกับโอເປົອແຣນດ์ ซึ่งทำให้ทั้ง 2 คำสั่งต้องใช้รีจิสเตอร์ไฟล์พร้อมกัน ทำให้เกิดการใช้รีจิสเตอร์ไฟล์



รูปที่ 6.11 สถานการณ์การเกิด Structure Hazard ของการใช้รีจิสเตอร์ไฟล์

วิธีการแก้ปัญหาเป็นดังรูปที่ 6.12 ซึ่งจะแบ่งการทำงานในการเขียนอ่านรีจิสเตอร์ไฟล์โดยให้การเขียนทำงานในครึ่งไฟเกลแรก และการอ่านทำงานในครึ่งไฟเกลหลังดังแบบสีในรูปของคำสั่งทั้งสอง โดยใช้การทริกเกอร์ของสัญญาณนาฬิกาที่ขาขึ้นและขาลงที่ต่างกัน



รูปที่ 6.12 การแก้ปัญหาสถานการณ์การเกิด Structure Hazard ของการใช้รีจิสเตอร์ไฟล์

## 6.2.2 Data Hazard

Data Hazard เป็น Hazard ที่เกิดจากความสัมพันธ์ของข้อมูลในโปรแกรมหนึ่งๆ อันนี้สืบเนื่องมาจากการตัวต่อตัวของโปรแกรมเอง ลักษณะความสัมพันธ์ของการเขียนและอ่านข้อมูลในโปรแกรมนั้นมีหลายรูปแบบ

พิจารณาโค้ดต่อไปนี้

```
temp = v[k];
v[k] = v[k+2];
v[k+2] = temp;
```

จะเป็นการสลับที่ค่าในอาร์เรย์  $v[k]$ ,  $v[k+2]$  จะเห็นว่าถ้าแปลงเป็น MIPS พิจารณาโค้ดต่อไปนี้

```
#reg $t1 เก็บแอดเดรส  $v[k]$ 
lw $t0,0($t1) #reg $t0 (temp) =  $v[k]$ 
lw $t2,4($t1) #reg $t2 (temp) =  $v[k+2]$ 
sw $t2,0($t1) # $v[k]$  = reg $t2
sw $t0, 4($t1) # $v[k+2]$  = reg $t0 (temp)
```

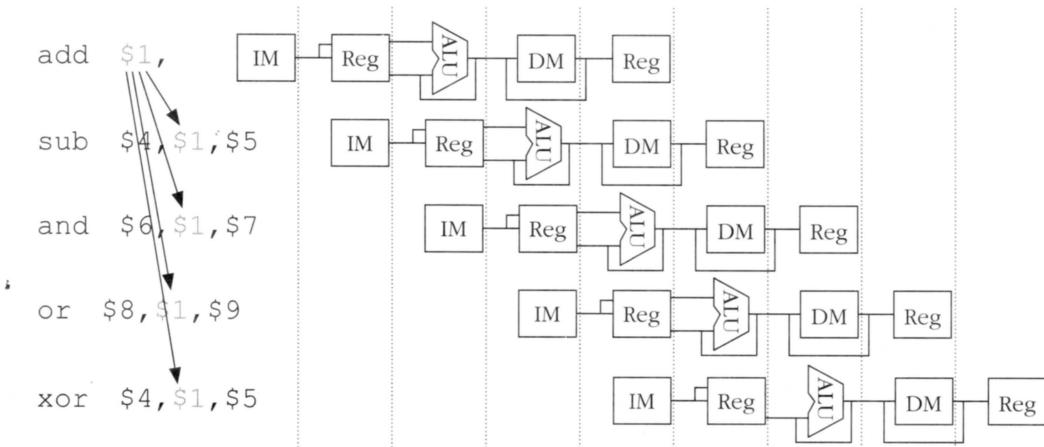
ซึ่งจะเห็นความสัมพันธ์ของคำสั่ง  $Lw$   $$t2$  และ  $Sw$   $$t2$  ถ้าสามารถเขียนโค้ดได้แบบข้างล่าง โดยการสลับที่คำสั่ง  $Sw$  ก็จะช่อน Stall จากคำสั่ง  $Lw$  อันที่สองได้ การสลับที่แบบนี้เป็นหน้าที่ของตัวแปลภาษาในการช่วยจัดลำดับคำสั่งเพื่อลดจำนวน Stall ที่เกิดจากความสัมพันธ์ของข้อมูลในรีจิสเตอร์

```
#reg $t1 เก็บแอดเดรส  $v[k]$ 
lw $t0,0($t1) #reg $t0 (temp) =  $v[k]$ 
lw $t2,4($t1) #reg $t2 (temp) =  $v[k+2]$ 
sw $t0,4($t1) # $v[k+2]$  = reg $t0 (temp)
sw $t2,0($t1) # $v[k]$  = reg $t2
```

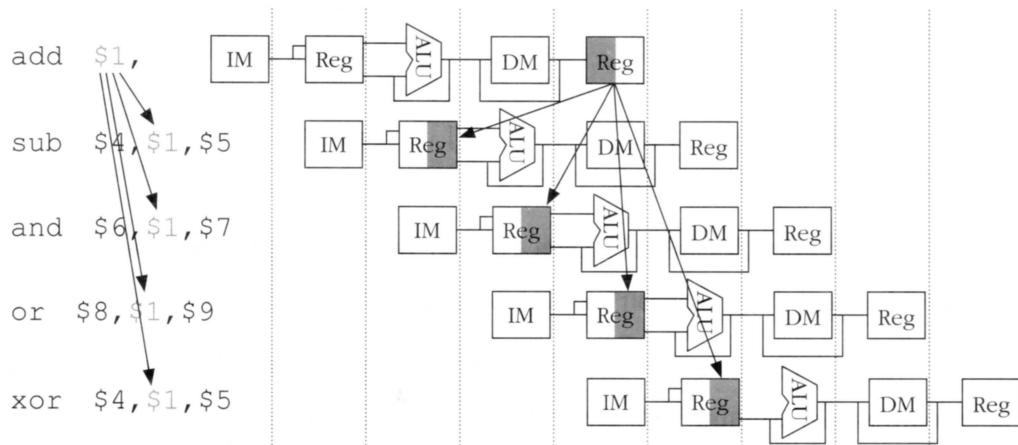
ในหัวข้อต่อไปนี้จะกล่าวถึง Data Hazard ที่สามารถเกิดขึ้นได้จากไปป์ไลน์และคำสั่งแบบ MIPS ในแต่ละรูปแบบ สำหรับการแก้ปัญหาในส่วนของการแปลงโค้ดเหล่านี้จะกล่าวถึงในบทต่อไป

รูปที่ 6.13 แสดงความเป็นไปได้ของการใช้ข้อมูลที่ได้จากคำสั่งแรก คำสั่งแรกนี้จะเขียนข้อมูลใหม่ลงไปในรีจิสเตอร์  $\$1$  และคำสั่งถัดๆ ไปจะใช้ข้อมูลใหม่ใน  $\$1$  นี้ รูปที่ 6.14 แสดงความสัมพันธ์ในเชิงเวลาว่าในการอ่านข้อมูลใน  $\$1$  ของแต่ละคำสั่งถัดมาต้องการใช้ในไบเกลิลได้บ้างโดยพิจารณาจากลูกศร นอกจากนี้ จากรูปนี้ข้างต้นจะเห็นว่าเรามุมดิว่า การอ่านรีจิสเตอร์ทำงานในครึ่งไบเกลล์ และการเขียนรีจิสเตอร์เกิดขึ้นในครึ่งไบเกลล์แรก จากรูปจะพูดว่าคำสั่งที่ 2 และคำสั่งที่ 3 มีปัญหาเนื่องจากลูกศรนั้นย้อนกลับไปข้างหน้า ซึ่งหมายถึงว่า ข้อมูลไม่สามารถส่งจาก

ไซเกิลที่ 5 ไปยังไซเกิลที่ 2 (ตรงที่โอเปอเรนด์ของคำสั่ง Sub) และไซเกิลที่ 5 ไปยังไซเกิลที่ 3 (โอเปอเรนด์ของคำสั่ง Or) ได้ จึงต้องมีวิธีการแก้ปัญหาเพื่อให้คำสั่ง Sub และคำสั่ง Or ได้ข้อมูลที่ถูกต้องไปใช้ในการทำงาน ปัญหานี้เรียกว่า Read Before Write สังเกตว่าอันนี้เป็นปัญหาระหว่างคำสั่งประเภท R-Type ด้วยกัน

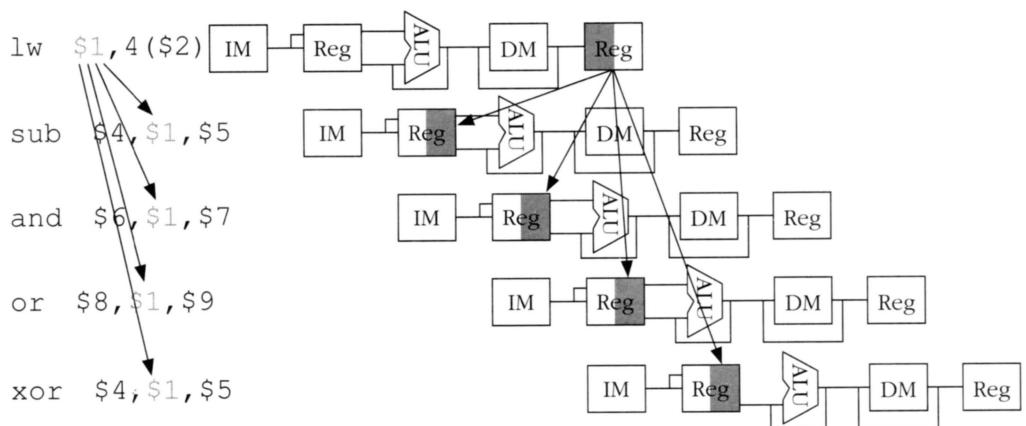


รูปที่ 6.13 ตัวอย่าง Data Hazard



รูปที่ 6.14 ความสัมพันธ์ของข้อมูลในรีจิสสเตอร์ \$1

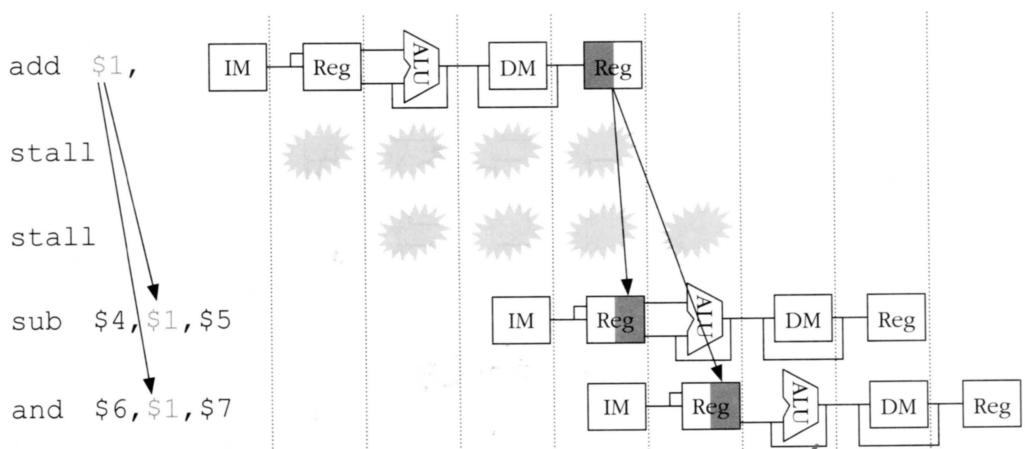
รูปที่ 6.15 แสดงปัญหาแบบเดียวกัน แต่เป็นปัญหาที่เกิดขึ้นระหว่างคำสั่ง Lw และคำสั่ง อื่นที่ใช้โอเปอเรนด์ที่ได้จากคำสั่ง Lw



รูปที่ 6.15 ความสัมพันธ์ของคำสั่ง LW และการใช้ผลลัพธ์ของคำสั่ง LW โดยคำสั่งที่เป็นประเภท R-Type

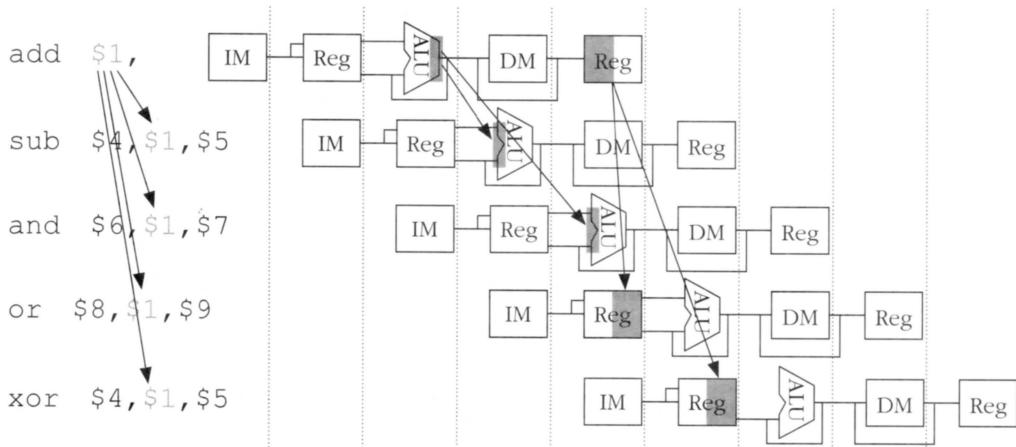
ในรูปที่ 6.16 แสดงการแก้ปัญหาของรูปที่ 6.14 โดยการใส่ Stall หรือ No Operation เข้าไปเพื่อให้คำสั่ง Sub และ And รอให้ผลการทำงานจากคำสั่ง Add ก่อน เพื่อจะได้ค่าถูกต้องไปใช้งาน วิธีการดังกล่าวอาจจะเป็นการเพิ่มไก่ลอกของการทำงานทั้งหมดได้ เพราะเมื่อกับกันว่าต้องใส่คำสั่ง Nop ไประหว่างคำสั่งทั้งสองนั้นเอง

```
add $1,$2,$3
nop
nop
sub $4,$1,$5
```



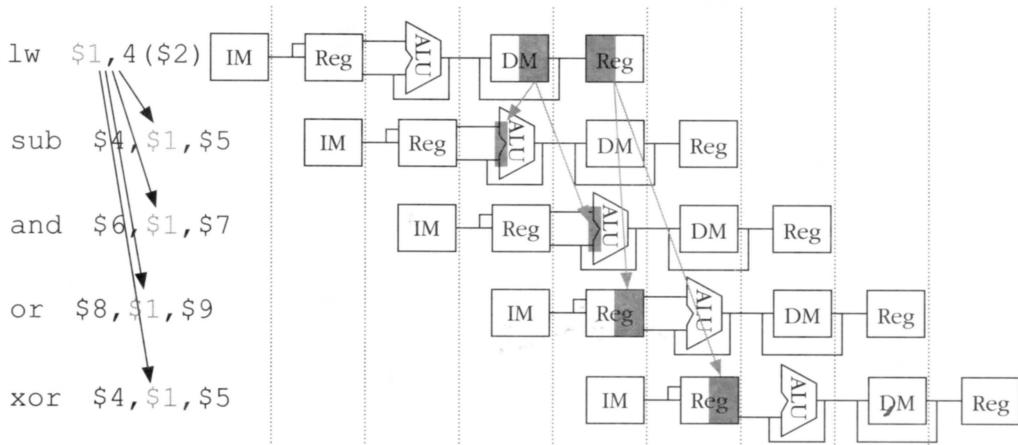
รูปที่ 6.16 การแก้ปัญหาคำสั่ง Add และการใช้ผลลัพธ์ของคำสั่ง Add ด้วยคำสั่ง And โดยการใส่ Stall

อีกครึ่งหนึ่งเป็นการปรับปรุง Data Path ใหม่ให้สามารถส่งผลลัพธ์ของการทำงานไปให้ได้ทันที เมื่อผลลัพธ์พร้อม โดยไม่ต้องรอให้ผลลัพธ์เขียนลงรีจิสเตอร์ไฟล์ก่อนแล้วค่อยอ่านจากรีจิสเตอร์ไฟล์ วิธีนี้เรียกว่า Forwarding ดังแสดงในรูปที่ 6.17



รูปที่ 6.17 การแก้ปัญหาคำสั่ง Add และการใช้ผลของคำสั่ง Add ด้วยการ Forward

ในการ Forward นี้จะเป็นการส่งผลลัพธ์ของคำสั่ง Add ที่เกิดขึ้นในเมื่อสิ้นไฟเกลิที่ 3 และส่งไปให้กับคำสั่ง Sub เมื่อต้องการใช้คำนวนในต้นไฟเกลิถัดไป ให้ไปใช้สำหรับโอเปอเรนเดอร์ตัวแรก และส่งไปให้คำสั่ง And ในต้นไฟเกลิที่ 5 จะเห็นว่าการแก้ปัญหาทำให้มีต้องใช้ Stall ในการทำงานทั้งหมดเลย แต่จะต้องเพิ่มฮาร์ดแวร์ช่วยในการ Forward จากเอกสารพุทธของ ALU ไปยังอินพุตของ ALU ทั้ง 2 ขา ดังแสดงในรูปที่ 6.18

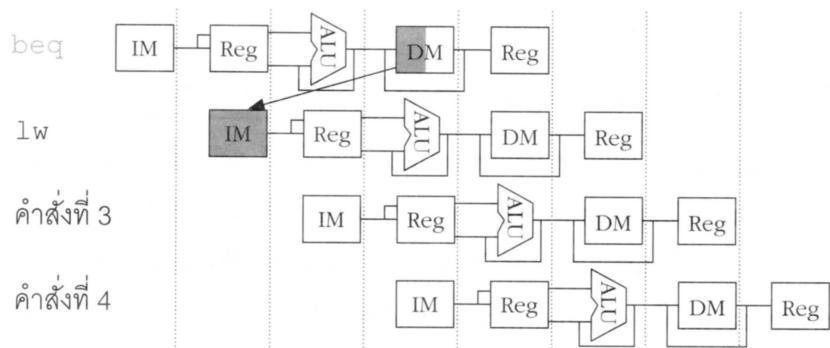


รูปที่ 6.18 การแก้ปัญหาคำสั่ง Lw และการใช้ผลลัพธ์ของคำสั่ง Lw ด้วยการ Forward

รูปที่ 6.18 เป็นการแก้ปัญหาการคำสั่ง Lw แต่จะเห็นว่าการโหลดนั้นจะได้ข้อมูลหลังไปเกิลที่ 4 ซึ่งเป็นขั้นตอนการเข้าถึงหน่วยความจำข้อมูล ดังนั้น การ Forward ผลลัพธ์ไปยังคำสั่ง And สามารถทำได้ แต่ก็ยังไม่สามารถ Forward ไปยังคำสั่งที่ตามมาติดกันซึ่งได้แก่คำสั่ง Sub ได้ ดังนั้นการทำงานคำสั่ง Sub ยังต้องการ Stall อยู่หนึ่งไซเคิล จึงกล่าวได้ว่าสำหรับคำสั่งโหลดนั้นและคำสั่งที่ใช้ผลจากการโหลดนั้นควรจะต้องอยู่ห่างกันอย่างน้อยหนึ่งไซเคิล

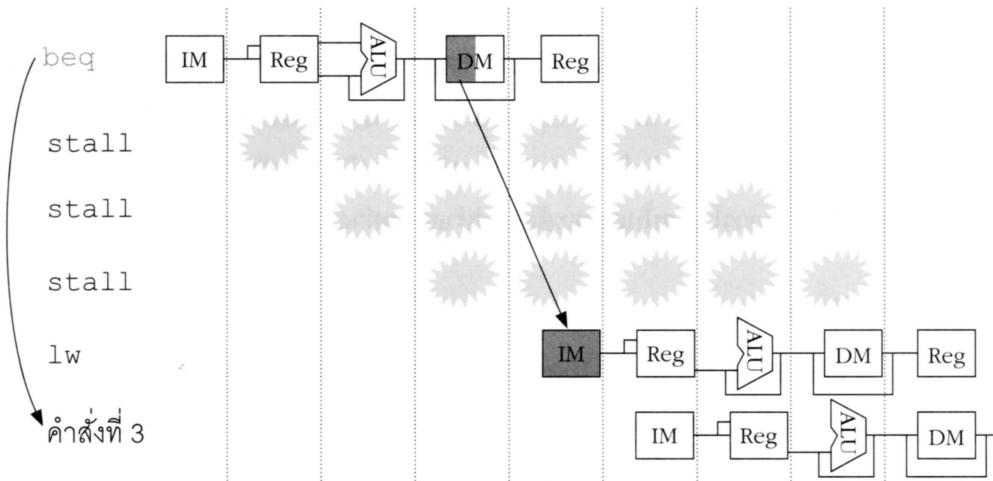
### 6.2.3 Branch Hazard

สำหรับ Branch Hazard เกิดจากคำสั่งประเภท Branch ซึ่งเป็นคำสั่งที่ต้องการการเปรียบเทียบ และเมื่อการเปรียบเทียบเป็นจริง จะต้องมีการเปลี่ยนลำดับคำสั่ง คำสั่งแบบนี้จะทำให้เกิด Hazard ที่เกี่ยวกับการควบคุมลำดับการทำงาน (Control Hazard) ดังตัวอย่างในรูปที่ 6.19



รูปที่ 6.19 ตัวอย่าง Control Hazard

รูปที่ 6.19 แสดงให้เห็นว่าคำสั่ง Beq จะรู้ผลว่าจะกระโดดหรือไม่ และถ้ากระโดดจะไปทำงานคำสั่ง ณ ตำแหน่งนั้นในต้นไปเกิลที่ 4 ดังนั้นหากมีการกระโดดจะต้องไป ณ ตำแหน่งที่กำหนด เพื่ออ่านคำสั่งที่ต้องการ เข่น คำสั่ง Lw ขึ้นมา ซึ่งทำให้การทำงานคำสั่ง Lw สามารถทำงานได้ทันที ดังแสดงในรูปที่ 6.19 แต่เราต้องทราบผลการกระโดดก่อนที่จะอ่านคำสั่งปลายทางขึ้นมา ดังลูกศรย้อนกลับในเข็มเวลา ทำให้ต้องแก้ปัญหาการรอผลนี้ด้วยการใส่ Stall ดังแสดงในรูปที่ 6.20 เพื่อให้ทราบผลคำสั่ง Branch ก่อนไปอ่านคำสั่งถัดไปได้ ลองสังเกตจำนวนไซเคิลของ การ Stall ที่ต้องการในกรณีนี้ว่าเท่ากับ 3 ไซเคิล แต่ในกรณีของ Lw เมื่อมีการ Forward แล้ว ต้องการ Stall จำนวน 1 ไซเคิล จะเห็นว่าจำนวนไซเคิลของการ Stall ที่ต้องการจะไม่เท่ากัน ทั้งนี้ขึ้นอยู่กับหลายๆ กรณี เช่น ลักษณะของคำสั่ง การทำงานของคำสั่ง ขั้นตอนของการทำงานของชีพชุด รูปแบบของชาร์ดแวร์ เป็นต้น



รูปที่ 6.20 ตัวอย่าง Control Hazard และการ Stall

### ■ 6.3 การออกแบบ Data Path ของไปป์ไลน์

ในรูปที่ 6.21 เมื่อจัดการทำงานเป็นแบบไปป์ไลน์ แต่ละขั้นตอนจะทำงานในคำสั่งที่ต่างกัน ดังนั้น จึงจำเป็นต้องมีการบัฟเฟอร์ระหว่างขั้นตอน เพราะแต่ละขั้นตอนการทำงานจะไม่เข้าแก่กัน อินพุตของแต่ละขั้นตอนจะมาจากบัฟเฟอร์ เมื่อทำงานเสร็จขั้นตอนนั้นๆ ก็จะเขียนผลลัพธ์ไปยัง บัฟเฟอร์ของขั้นถัดไป เพื่อขั้นถัดไปจะนำผลลัพธ์ไปใช้ พร้อมจะทำงานในขั้นต่อไป บัฟเฟอร์ดังกล่าว ระหว่างขั้นตอนมีชื่อเรียกดังนี้

IF/ID เป็นบัฟเฟอร์ที่เก็บข้อมูลออกของขั้นตอนเฟตช์ เพื่อเป็นข้อมูลเข้าของขั้นตอนดีโค้ด

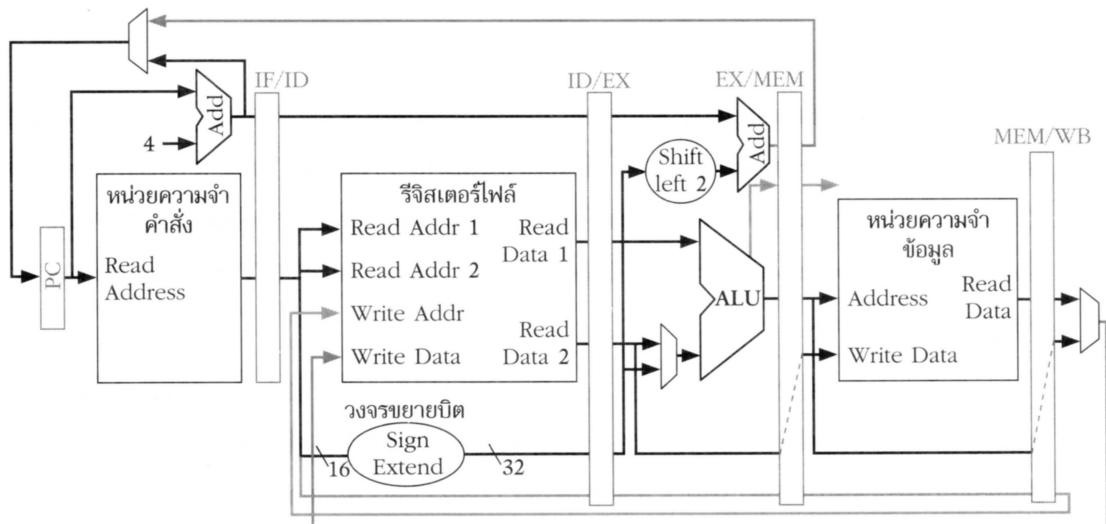
ID/EXE เป็นบัฟเฟอร์ที่เก็บข้อมูลออกของขั้นตอนดีโค้ด เพื่อเป็นข้อมูลเข้าของขั้นตอน เอ็กซ์คิวต์

EXE/MEM เป็นบัฟเฟอร์ที่เก็บข้อมูลออกของขั้นตอนเอ็กซ์คิวต์ เพื่อเป็นข้อมูลเข้าของขั้น ตอนการเข้าถึงหน่วยความจำ

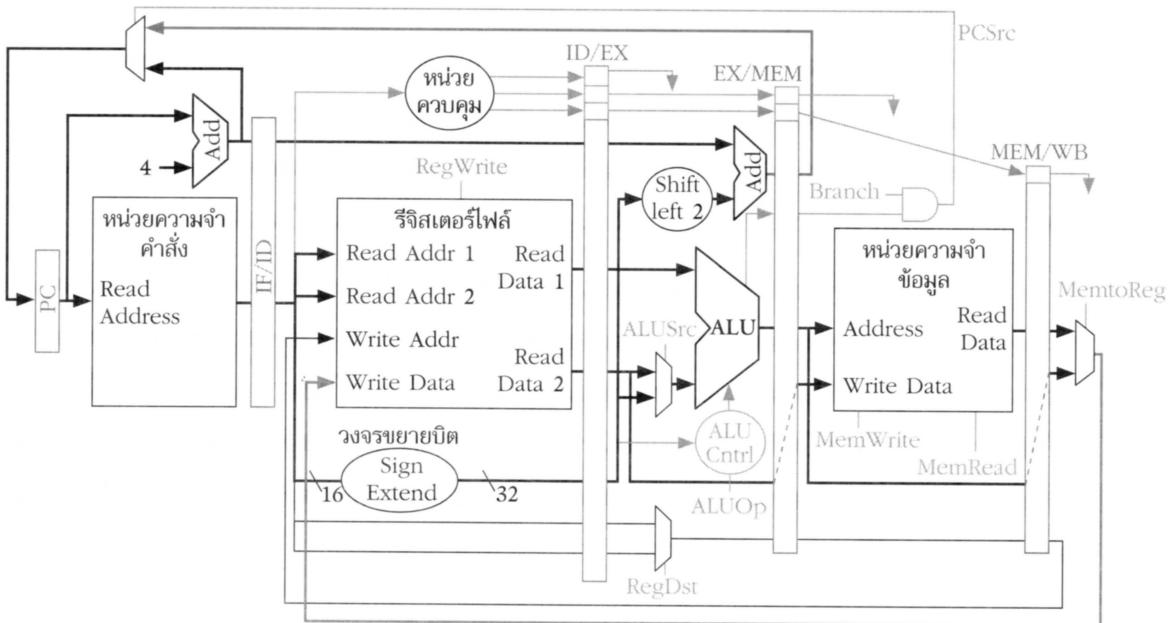
MEM/WB เป็นบัฟเฟอร์ที่เก็บข้อมูลออกของขั้นตอนการเข้าถึงหน่วยความจำ เพื่อเป็นข้อมูล เข้าของขั้นตอนการเขียนค่ากลับ

โดยบัฟเฟอร์แต่ละชุดจะมีจำนวนรีจิสเตอร์ชั้วคราวภายในเมื่อถูกกันกับใน Data Path เดิม แบบไม่เป็นไลน์ ดังนั้นจะเห็นว่าถ้าการออกแบบเพิ่มจำนวนขั้นตอนมากขึ้น อาจจะเพื่อเพิ่มโอกาสการทำงานแบบขนาน แต่จะทำให้เพิ่มจำนวนบัฟเฟอร์เหล่านี้มากขึ้น และจะทำให้สัญญาณควบคุม ยุ่งยากมากขึ้น เพราะต้องเลือกจากบัฟเฟอร์ที่เหมาะสม และการพิจารณาการเกิด Hazard ก็ยุ่งยากมากขึ้นด้วย เพราะต้องพิจารณา Hazard ระหว่างทุกขั้นตอนที่เป็นไปได้

เมื่อเกิดบัฟเฟอร์ดังกล่าวขึ้น จึงจำเป็นต้องมีการแก้ไขเส้นทางเดินของข้อมูลเข้าของแต่ละหน่วยให้อ่านมาจากบัฟเฟอร์ และให้คัดลอกข้อมูลของบัฟเฟอร์เดิมไปยังบัฟเฟอร์ของขั้นตอนถัดไป ในบางกรณี ข้อมูลในบัฟเฟอร์อาจจะถูกเปลี่ยนแปลง บางกรณีอาจจะไม่มีการเปลี่ยนแปลงก็ได้ เมื่อมีบัฟเฟอร์ดังกล่าวก็จำเป็นต้องมีการเปลี่ยนแปลงสัญญาณควบคุมด้วย ดังแสดงในรูปที่ 6.22 ซึ่งแสดงภาพโดยรวมของอินพุตและเอาต์พุตของสัญญาณของหน่วยควบคุมนี้ โดยจะต้องไปควบคุมการเขียนอ่านของบัฟเฟอร์เหล่านี้ด้วย



รูปที่ 6.21 โครงสร้าง Data Path เดิมและเมื่อใส่บัฟเฟอร์ระหว่างขั้นตอน



รูปที่ 6.22 เพิ่มตัวควบคุมในไปปีไลน์

รูปที่ 6.23 แสดงสัญญาณควบคุมที่เกี่ยวข้องในขั้นตอนต่างๆ ที่สำคัญที่เปลี่ยนแปลงจากบทที่แล้ว ว่าในแต่ละขั้นตอนนั้นต้องการให้สัญญาณใดเป็น 1 (enable) และ 0 (disable) สำหรับ X หมายถึง Don't Care จะเห็นว่าการให้สัญญาณควบคุมต้องดูรายละเอียดในแต่ละคำสั่ง ในที่นี้จะจัดกลุ่มประเภทคำสั่งเป็นประเภท Load, Store, Branch และ R-Type เพื่อให้ง่ายต่อการออกแบบวงจรควบคุม

ในตัวอย่างจะเห็นว่ากรณีของ R-Type ต้องระบุ ALUop เป็น 10 ในขั้นเงื่อนไขซีควิตร์ต้องกำหนดให้มีสัญญาณเอาต์พุตไปยังรีจิสเตรอร์ (RegDst) เป็น 1 และขั้นเขียนค่ากลับต้องการสัญญาณการเขียนรีจิสเตรอร์ (RegWrite) เป็น 1 สำหรับคำสั่ง Load (Lw) ต้องการการอ่านข้อมูลในขั้นเข้าถึงหน่วยความจำ โดยให้สัญญาณการอ่านข้อมูลจากหน่วยความจำ (MemRead) เป็น 1 และต้องการให้มีการเขียนไปยังรีจิสเตรอร์ผลลัพธ์จากหน่วยความจำข้อมูลที่อ่านได้ ระบุโดยสัญญาณ RegWrite และ MemtoReg สำหรับคำสั่ง Branch (ตัวอย่าง เช่น คำสั่ง Beq) ต้องระบุลักษณะของ ALUop เป็น 01 และระบุสัญญาณ Branch เป็น 1 ในขั้นการเข้าถึงหน่วยความจำ

คำสั่ง	ขั้นເອົກຊື່ຄົວຕົວ				ขັ້ນເຂົາເສີ່ງພ່າຍຄວາມຈຳ			ขັ້ນເບີຍນັ້ນຂອ່ມູລກສັນ	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem Read	Mem-Write	Reg-Write	MemtoReg
R	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

รูปที่ 6.23 สัญญาณควบคุมในไปป์ไลน์

## ■ 6.4 การเปลี่ยนแปลง Data Path เพื่อแก้ปัญหา Hazard

ในส่วนต่อไปนี้จะกล่าวถึงวิธีการแก้ปัญหาของ Hazard ที่ได้กล่าวไปแล้วข้างต้น และผลที่เกิดขึ้นทางด้านฮาร์ดแวร์ในเชิงสถาปัตยกรรม

สำหรับวิธีการแก้ปัญหา Data Hazard วิธีหนึ่งคือการใช้ Forwarding ซึ่งหมายถึงการส่งข้อมูลที่ได้จากเอกสาร์พุตของขั้นตอนนั้นๆ ไปยังอินพุตของ ALU ในขั้นตอนการເອົກຊື່ຄົວຕົວเพื่อมาคำนวนในไบเกิลตัดไป โดยถ้าต้องการ Forward ข้อมูลจากเอกสาร์พุตของ ALU ข้อมูลที่ต้องการจะส่งนั้นอยู่ในบັບເພື່ອຮົບອອກ EX/MEM ดังนั้นในการแก้ปัญหานี้จะต้องเพิ่มสัญญาณควบคุมและเปลี่ยนแปลง Data Path ดังนี้

- เพิ่มตัวเลือก (MUX) สำหรับอินพุตของ ALU ให้มาจากเอกสาร์พุตของ ALU ที่มาจากการขั้นตอน EX/MEM
- เพิ่มสัญญาณที่ให้ข้อมูลที่จะเขียน (ระบุโดย Rd) ไปยังบັບເພື່ອຮົບອອກ EX/MEM หรือบັບເພື່ອຮົບของ MEM/WB ไปยังอินพุตของ ALU ทั้งสองขาที่ระบุโดย Rs และ Rt
- เพิ่มสัญญาณควบคุมสำหรับตัว MUX เหล่านี้

นอกจากนี้ หน่วยอื่นๆ ยังต้องการการ Forward ในลักษณะเดียวกัน เช่น ถ้าเป็นจากขั้นตอนการເຂົາເສີ່ງພ່າຍຄວາມຈຳໄປยังขั้นตอนการເອົກຊື່ຄົວຕົວ ก็ต้องสามารถ Forward จากเอกสาร์พุตของหน่วยความจำข้อมูลไปยังอินพุตทั้งสองของ ALU ได้ กรณีการใช้การ Forward นี้จะทำให้ CPI ของคำสั่งสามารถลดลงเหลือ 1 ได้ แม้ว่าจะมีความสัมพันธ์ของข้อมูลระหว่างคำสั่งก็ตาม

ในโค้ดข้างล่างเป็นการเพิ่มสัญญาณควบคุม Forward และแสดงเงื่อนไขการให้ค่าสัญญาณ Forward ตามกรณีต่างๆ ในที่นี้มี 2 กรณี คือ การ Forward จาก ALU และการ Forward จากหน่วยความจำข้อมูล โดยสัญญาณ ForwardA หมายถึง มีการ Forward ไปยังโอเปอเรนเดอร์ขางของ ALU และ ForwardB หมายถึง มีการ Forward ไปยังโอเปอเรนเดอร์ขางของ ALU

#### กรณีที่ 1 : การ Forward จาก ALU

```

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
ForwardB = 10

```

ในกรณีนี้จะดูจากสัญญาณ RegWrite ของ EX/MEM ซึ่งจะบอกว่าต้องการการเขียนไปยังรีจิสเตอร์ สัญญาณนี้จะให้โดยหน่วยควบคุมกรณีที่เมื่อคำสั่งนั้นต้องการการเขียนรีจิสเตอร์ เบ่นคำสั่ง ALU R-Type โดยระบุว่ารีจิสเตอร์ที่ต้องการจะเขียนไปตรงกับโอเปอเรนเดอร์ขางของ ALU (Rs หรือ Rt) ถ้าเป็น Rs ก็ให้ ForwardA = 10 คือไปยังโอเปอเรนเดอร์ขางของ ALU และกรณีหลังคือ Forward ไปยังโอเปอเรนเดอร์ขางของ ALU (ForwardB)

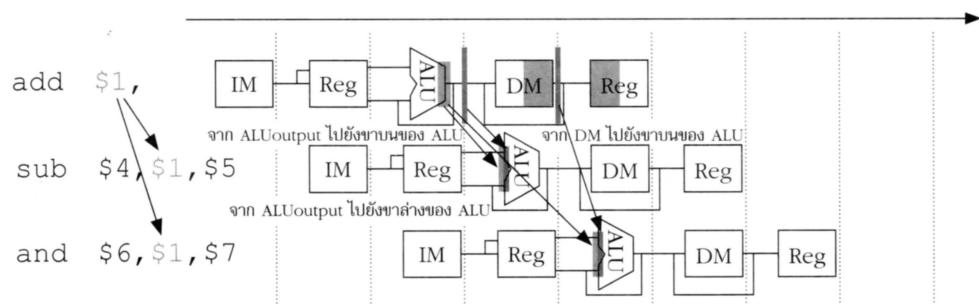
#### กรณีที่ 2 : การ Forward จากหน่วยความจำข้อมูล

```

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
ForwardB = 01

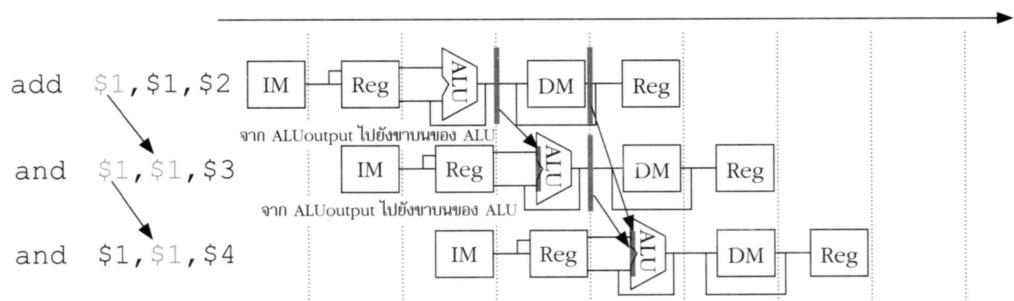
```

ในการเน้นพิจารณาจากสัญญาณ RegWrite ของ MEM/WB (กรณีคำสั่งของ Load) ซึ่งจะบอกว่าต้องการการเขียนไปยังรีจิสเตอร์ สัญญาณนี้จะให้โดยหน่วยควบคุมเมื่อคำสั่งนั้นต้องการการเขียนรีจิสเตอร์สำหรับคำสั่ง Lw โดยระบุว่ารีจิสเตอร์ที่ต้องการจะเขียนไปตรงกับโอเปอเรนด์ขาเดียวของ ALU (Rs หรือ Rt) ถ้าเป็น Rs ก็ให้สัญญาณ ForwardA = 01 คือ Forward ไปยังโอเปอเรนด์ขาล่างของ ALU และกรณีหลังคือให้ Forward ไปยังโอเปอเรนด์ขาล่างของ ALU (ForwardB)



รูปที่ 6.24 การ Forward จากบัฟเฟอร์ EX/MEM และจากบัฟเฟอร์ MEM/WB

รูปที่ 6.24 แสดงการ Forward จากบัฟเฟอร์ทั้งสองไปยังโอเปอเรนด์ของ ALU โดยล่งผลลัพธ์ไปยังคำสั่ง Sub ที่ข้างบน และส่งผลลัพธ์ไปคำสั่ง And ที่ข้างล่างของ ALU แต่ยังมีปัญหาอีกกรณี ดังแสดงในรูปที่ 6.25 ซึ่งเป็นกรณีที่เอาต์พุตกับอินพุตของ ALU เป็นรีจิสเตอร์ตัวเดียวกัน ในที่นี้คำสั่ง Add ที่สองซึ่งอ่านและเขียนโอเปอเรนด์ \$1 ตัวเดียวกัน



รูปที่ 6.25 การ Forward จากบัฟเฟอร์ EX/MEM และจากบัฟเฟอร์ MEM/WB  
กรณีเอาต์พุตเหมือนกับอินพุต

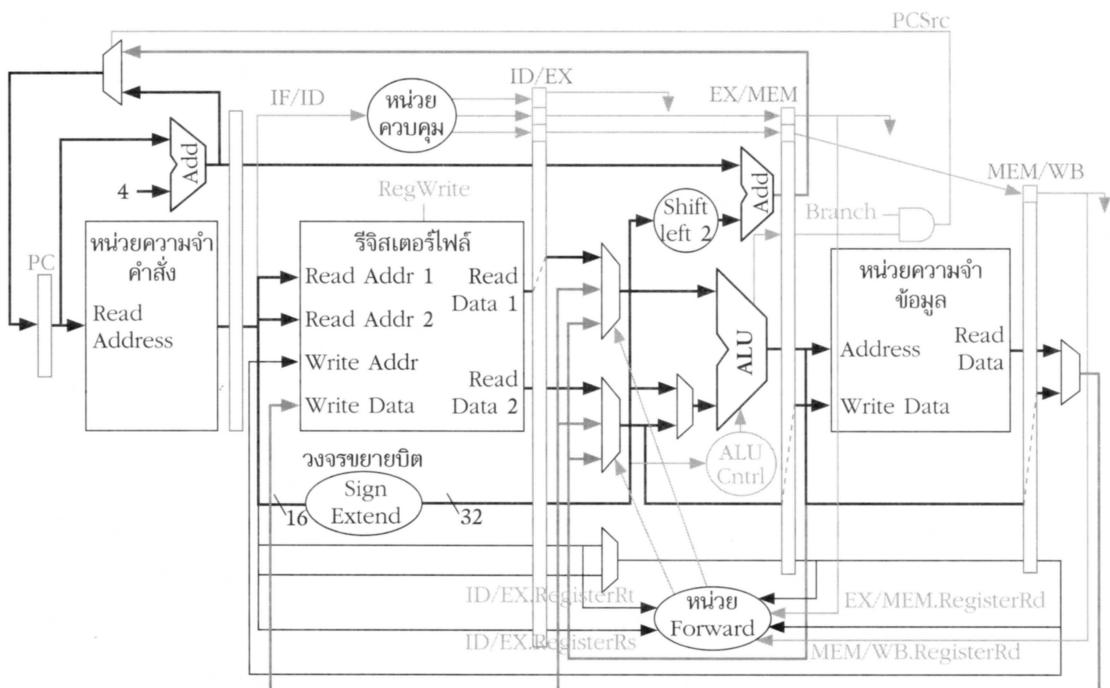
จึงต้องปรับเงื่อนไขดังต่อไปนี้สำหรับกรณีที่ 2 ในขั้นตอนของการเข้าถึงหน่วยความจำ

```

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRs)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
ForwardB = 01

```

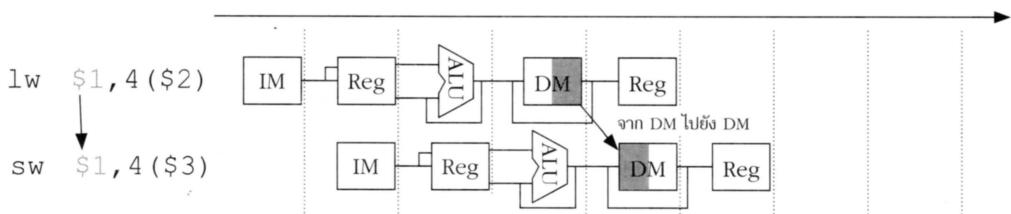
รูปที่ 6.26 แสดง Data Path เมื่อเพิ่มสัญญาณ Forward สำหรับทั้ง 2 กรณีดังกล่าวเข้าไป  
หน่วยควบคุมการ Forward เป็นตัวคำนวนเงื่อนไขดังกล่าว และให้สัญญาณควบคุม Forward A  
และ Forward B โดยอ่านอินพุตจากบัสเฟอร์ ID/EX และ EX/MEM และคำนวนตามเงื่อนไข<sup>1</sup>  
การ Forward และส่งເອົາດີປຸດໄປควบคุม MUX ที่เพิ่มขึ้นมา 2 ตัว การให้สัญญาณควบคุมสรุป  
ได้ดังตารางต่อไปนี้ ซึ่งสัญญาณควบคุมเป็นເອົາດີປຸດสำหรับหน่วย Forward ของ Source แสดง  
แหล่งข้อมูลที่จะต้อง Forward ไป



รูปที่ 6.26 Data Path ของการ Forward จากบัฟเฟอร์ EX/MEM, MEM/WB

สัญญาณควบคุม สำหรับMUX	Source	อธิบาย
Forward A = 00	ID/EX	โ้อเปอแรนด์ตัวแรกของ ALU มาจากรีจิสเตอร์ไฟล์
Forward A = 10	EX/MEM	โ้อเปอแรนด์ตัวแรกของ ALU ถูก Forward มาจากผลการคำนวณจาก ALU ในไชเกิลก่อนหน้า
Forward A = 01	MEM/WB	โ้อเปอแรนด์ตัวแรกของ ALU ถูก Forward มาจากผลจากหน่วยความจำข้อมูล หรือจาก ALU ในไชเกิลก่อนหน้า
Forward B = 00	ID/EX	โ้อเปอแรนด์ตัวที่สองของ ALU มาจากรีจิสเตอร์ไฟล์
Forward B = 10	EX/MEM	โ้อเปอแรนด์ตัวที่สองของ ALU ถูก Forward มาจากผลการคำนวณจาก ALU ในไชเกิลก่อนหน้า
Forward B = 01	MEM/WB	โ้อเปอแรนด์ตัวที่สองของ ALU ถูก Forward มาจากผลจากหน่วยความจำข้อมูล หรือจาก ALU ในไชเกิลก่อนหน้า

สำหรับกรณีการคัดลอกค่าจากหน่วยความจำโดยมีคำสั่ง Load และต่อด้วยคำสั่ง Store ก็สามารถใช้การ Forward ช่วยทำให้ลด Stall ได้ โดยต้อง Forward จากบันพเฟอร์ MEM/WB ไปยังอินพุตของหน่วยความจำข้อมูล

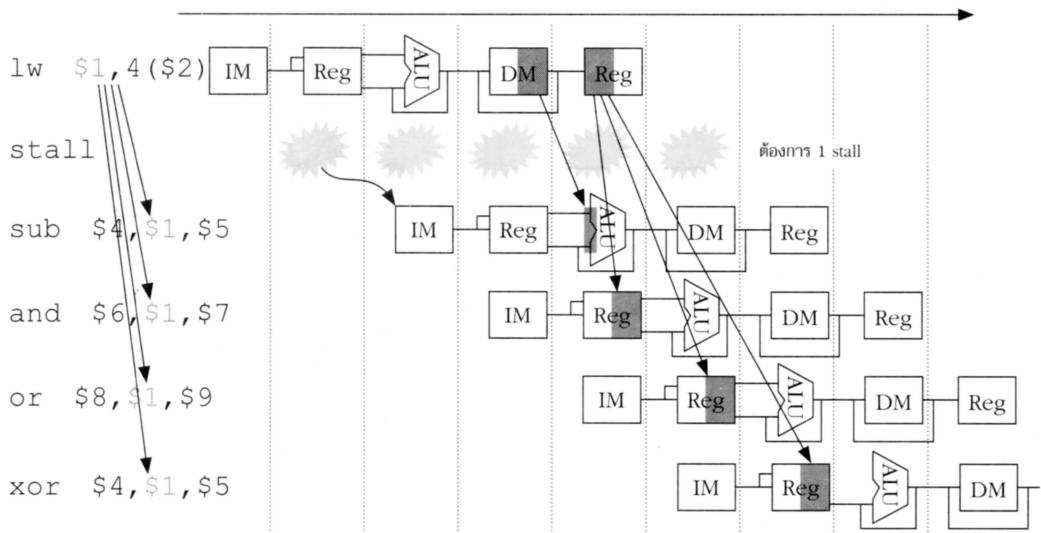


รูปที่ 6.27 การคัดลอกค่าระหว่าง Load กับ Store

สำหรับกรณีของคำสั่ง Load แม้ว่าจะทำการ Forward แล้วก็ยังต้องการ Stall อีก 1 ไซคล ดังแสดงในรูปที่ 6.28 ดังนั้น ใน Data Path ก็ยังต้องเพิ่มเงื่อนไขการ Stall ไปปีเลนเข้าไป ใน การตรวจสอบกรณีต้องตรวจสอบในขั้นต่อไป โดยอ่านจากบันพเฟอร์ ID/EX ซึ่งดูจากลัญญาณ MemRead จะเกิดขึ้นในกรณีของคำสั่ง Load และตรวจสอบโอเปอเรนเดอร์ทั้งสองของคำสั่ง (โดย ดูจากลัญญาณ Rt และ Rs) ของคำสั่งปัจจุบัน และคำสั่งก่อนหน้า (ได้แก่คำสั่ง Load) และ เพราะคำสั่ง Load เขียนไปยังโอเปอเรนเดอร์ Rt จึงต้องตรวจสอบกับบันพเฟอร์ ID/EX.RegisterRt

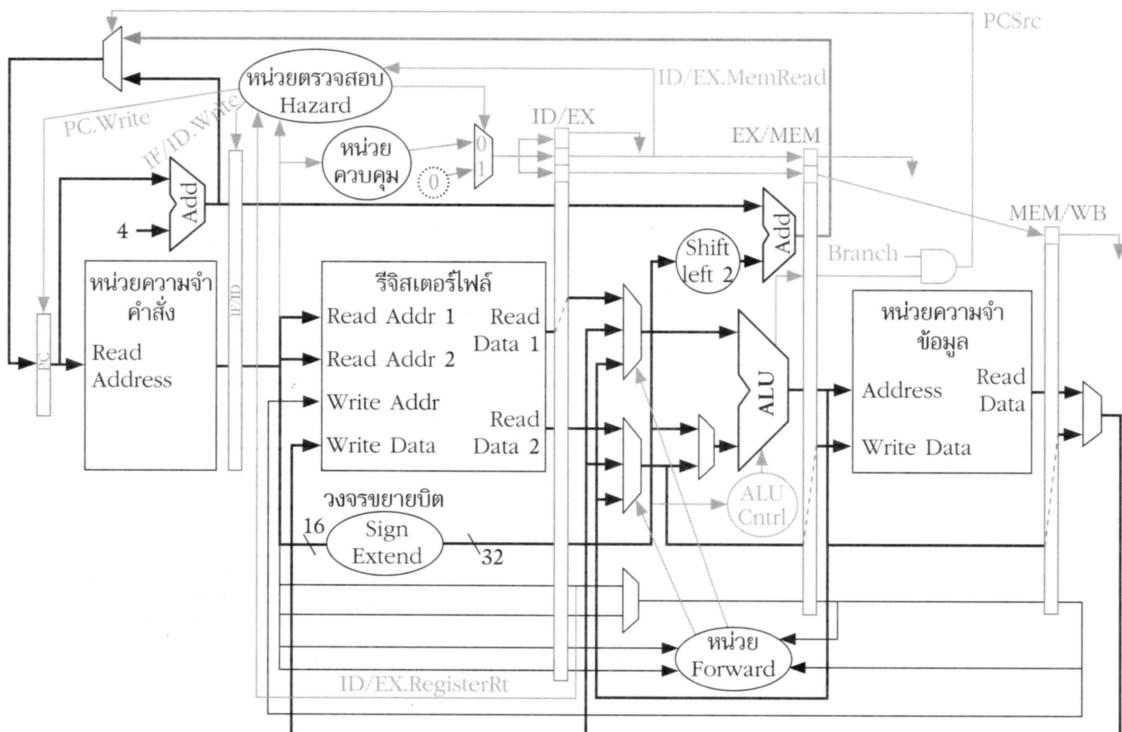
```
if (ID/EX.MemRead
    and ((ID/EX.RegisterRt = IF/ID.RegisterRs)
        or (ID/EX.RegisterRt = IF/ID.RegisterRt)))
```

ให้ Stall ไปปีลайн



รูปที่ 6.28 การ Forward ของคำสั่ง Load ที่ต้องการ Stall

ในการ Stall ไปปีลайнนั้น เพื่อป้องกันไม่ให้คำสั่งที่เกิดที่หลังจากขั้นเฟตช์ทำงาน ต้องล็อกค่า PC เอาไว้ และเก็บค่าต่างๆ ในบัฟเฟอร์ IF/ID ไม่ให้เปลี่ยนแปลง หน่วยตรวจสอบ Hazard จะทำหน้าที่นี้โดยการใช้สัญญาณ PC.Write และ IF/ID.Write ไปควบคุมการเขียนค่าในรีจิสเตอร์ PC และการเขียนไปในบัฟเฟอร์ IF/ID จากนั้นจะทำการใส่ Stall หรือ Nop ลงในระหว่างคำสั่ง Lw และคำสั่งที่ใช้ผลของ Lw หรือเป็นการใส่ Nop ในการนี้จะทำการเขตค่าบิตควบคุมต่างๆ ของ EX, MEM, WB ให้เป็น 0 และหน่วยตรวจสอบ Hazard จะควบคุมการเลือกใน MUX ให้เลือกระหว่างค่าสัญญาณควบคุมค่าเดิมและค่า 0 จากนั้นจะให้คำสั่ง Lw ทำงานตามปกติใน 1 ไบต์ เพื่อให้สามารถ Forward ได้ในไบต์ต่อไป รูปที่ 6.29 แสดง Data Path เมื่อมีหน่วยตรวจสอบ Hazard แล้ว หน่วยตรวจสอบ Hazard จะส่งสัญญาณไปล็อกค่า (โดยการเลือกค่า 0 สำหรับ MUX ที่วงกลมไว้ในรูป) ในบัฟเฟอร์ต่างๆ ไว้ โดยอ่านค่าอินพุตสัญญาณจากบัฟเฟอร์ ID/EX.MemRead, ID/EX.RegisterRt, ID/EX.RegisterRs และ IF/ID.RegisterRt



รูปที่ 6.29 การเพิ่มหน่วยตรวจสอบ Hazard และการ Forward ของคำสั่ง Load โดยต้องการ Stall ใน 1 ไซเกล

## ■ 6.5 Data Path เมื่อพิจารณาจาก Control Hazard

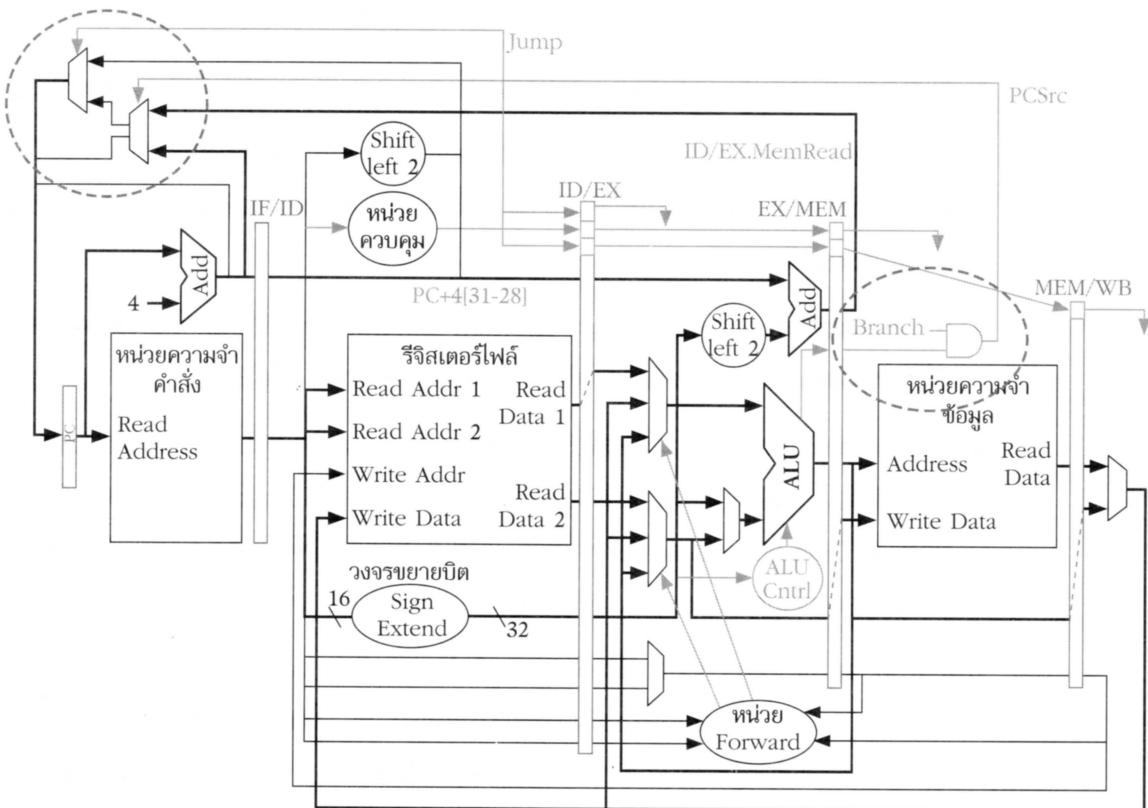
การเกิด Control Hazard มีสาเหตุมาจากการเปลี่ยนแปลงค่าใน PC เนื่องจากคำสั่งที่จะถูกทำงานตัดไปอาจจะไม่ได้มาจากแอดเดรสตัดไป ( $PC + 4$ ) กรณีต่างๆ ที่อาจทำให้เกิดการเปลี่ยนแปลงลำดับของการทำงานคำสั่ง ได้แก่

- คำสั่งกระโดดแบบมีเงื่อนไข (Branch)
- คำสั่งกระโดดแบบไม่มีเงื่อนไข (Jump)
- การเกิด Exception

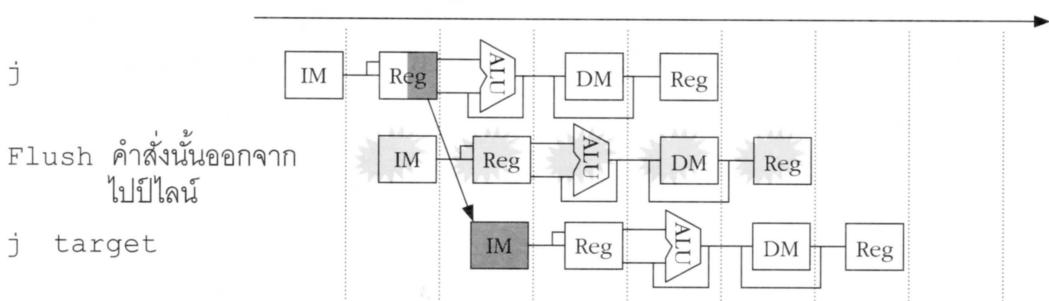
### วิธีการแก้ไข ได้แก่

- การใส่ Stall หรือหยุดการทำงานของไปป์ไลน์ในรอบนั้นซึ่งจะมีผลต่อ CPI รวมของระบบ
- การเปลี่ยน Data Path เพื่อให้การตัดสินใจเปลี่ยนค่าในรีจิสเตอร์ PC อยู่ในขั้นตอนที่เร็วที่สุดในไปป์ไลน์เท่าที่จะทำได้ เพื่อลดจำนวนไขเกิลที่ต้อง Stall สำหรับคำสั่งประเภทนี้
- การหน่วงเวลาการกระโดด (Delay Branch) และให้ทำการเปลี่ยนค่าในรีจิสเตอร์ PC เมื่อรู้ตำแหน่งแน่นอนแล้วซึ่งต้องให้ตัวแปลภาษาช่วยออดไมโครโค้ด
- การนำหายาแนวที่จะกระโดดไป และใช้การคาดคะเนว่าจะนำหายังไง เพื่อจะทำให้ไขเกิลที่ต้อง Stall น้อยที่สุด แต่จริงๆ แล้วการเกิด Control Hazard จะมีโอกาสเกิดได้ น้อยกว่า Data Hazard แต่ว่าจำนวน Stall ไขเกิลที่จะต้องใช้เมื่อเกิด Control Hazard มักจะมีมากกว่ากรณีของ Data Hazard

รูปที่ 6.30 แสดง Data Path เมื่อพิจารณาคำสั่ง Branch และ Jump แล้ว จะพบว่าต้องเปลี่ยนแปลง Data Path ตรงที่เป็นอินพุตของรีจิสเตอร์ PC โดยให้มาจากค่าที่อ่านมาจากคำสั่ง Jump หรือ Branch (ดูที่ MUX ทั้ง 2 ตัว) สำหรับคำสั่ง Jump โอเปอเรนด์ที่ใช้จะเป็นค่าแอดเดรสจริงโดยนำมายกต่อกับค่าใน PC ให้ครบขนาดของ Word สำหรับคำสั่ง Branch ค่าแอดเดรสจะเป็นค่าสัมพัทธ์ที่มาจากการอีเพอแรนด์นำมาย้ายบิต (Sign Extension) จากนั้นจะถูกนำมายกับกับค่าใน PC โดย ALU นอกจากนี้ จะมีการให้สัญญาณควบคุม Branch ซึ่งจะเป็น 1 เมื่อคำสั่งนั้นเป็นคำสั่ง Branch และให้สัญญาณ Jump เป็น 1 เมื่อเป็นคำสั่ง Jump เพื่อควบคุมการเปลี่ยนค่าในรีจิสเตอร์ PC



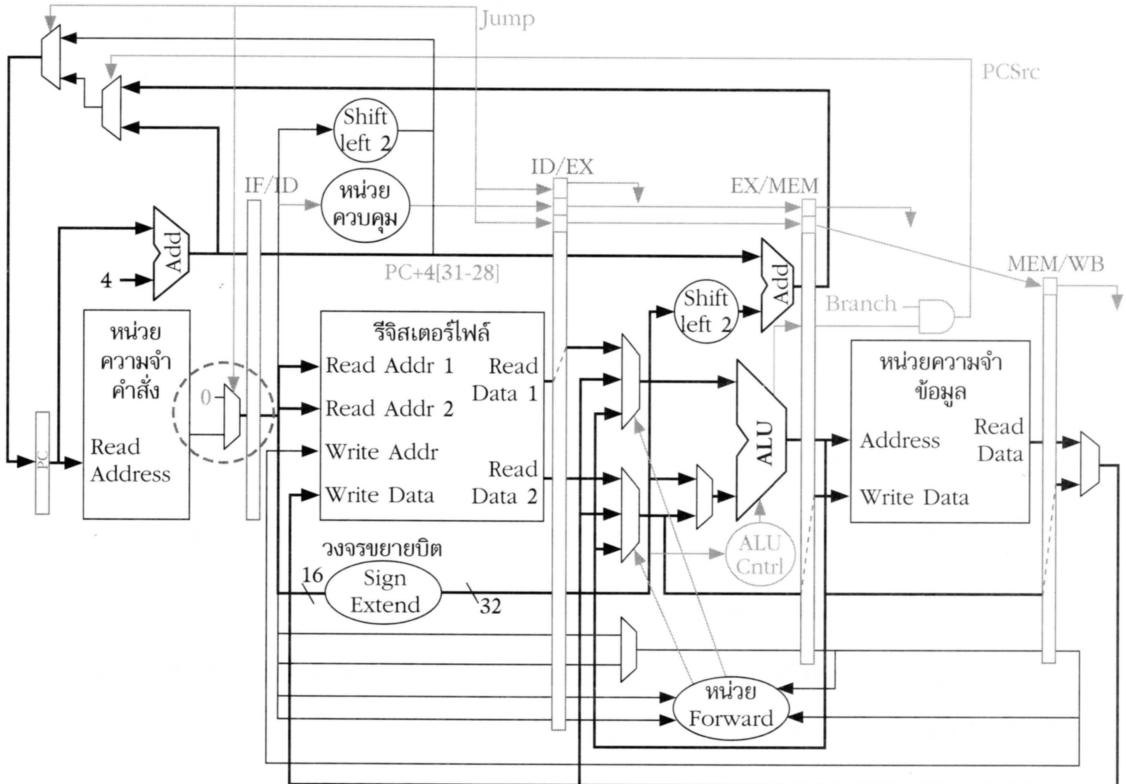
รูปที่ 6.30 Data Path เมื่อพิจารณาคำสั่ง Branch/Jump



รูปที่ 6.31 กรณีคำสั่ง Jump ต้องถูก Flush คำสั่งในไปป์ไลน์

รูปที่ 6.31 แสดงกรณีคำสั่ง Jump ซึ่งจะรู้ตำแหน่งที่จะกระโดดไปในขั้นตอนต่อไป ดังนั้นคำสั่งที่จะกระโดดไปจะต้องอยู่ตั้งไปจากคำสั่ง Jump อย่างน้อย 1 คำสั่ง ดังนั้น คำสั่งที่อยู่ระหว่าง 2 คำสั่งนั้นจะต้องถูกเอาออกจากไปป์ (Flush) ซึ่งจะต้องเคลียร์ค่าในบัฟเฟอร์ IF/ID โดยการเพิ่ม

MUX และเลือกระหว่างค่า 0 และค่าสัญญาณเดิมดังแสดงในรูปที่ 6.32



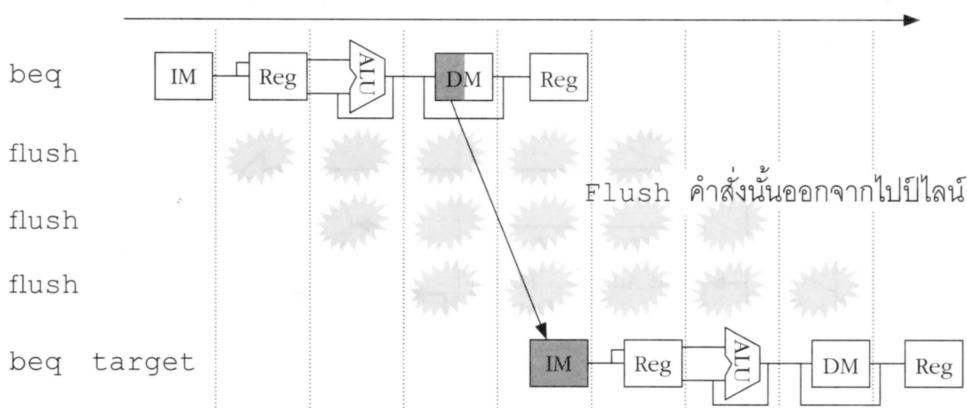
รูปที่ 6.32 การแก้ไข Data Path กรณีคำสั่ง Jump โดยต้องเอาคำสั่งในไปป์ไลน์ออกไป

การ Stall สามารถอิมพลีเมนต์ในชาร์ดแวร์ได้ 2 รูปแบบได้แก่

- การใส่คำสั่ง Nop ระหว่างคำสั่งทั้งสองเข่นเดียวกับกรณีของคำสั่ง Load เป็นการอนุญาตให้คำสั่งที่เกิดก่อนหน้านี้สามารถทำเสร็จได้โดยไม่ลงไปในไปป์ไลน์ตามขั้นตอนปกติ คำสั่งที่เกิดขึ้นที่หลังจากคำสั่ง Jump จะหยุดการทำงานไว้ และคำสั่ง Nop จะทำการเคลียร์ค่าในบิตควบคุมต่างๆ อย่างเหมาะสม
- ทำการ Flush คำสั่งในไปป์ไลน์โดยการแทนที่ด้วย Nop และเคลียร์ค่าในบิตควบคุมของคำสั่งในไปป์ไลน์เหล่านี้

สำหรับคำสั่ง Branch ตามรูปแบบการทำงานของ MIPS จะต้องการ Stall ไม่น้อยกว่า 3 ไซเคิล เนื่องจากว่าจะกระโดดหรือไม่ และถ้าจะกระโดดจะไปที่ตำแหน่งใดในขั้นตอนการเข้าถึง

หน่วยความจำข้อมูลภายในครึ่งไฟเกลลาร์ ดังแสดงในรูปที่ 6.33 ดังนั้น จะต้องหาทางแก้โดยอาจ  
จะต้องปรับปรุงขั้นตอนต่างๆ และฮาร์ดแวร์ที่เกี่ยวข้อง เพื่อให้สามารถรู้ตำแหน่งที่จะกระโดดไป  
ทำงานได้เร็วที่สุด



รูปที่ 6.33 กรณีการเกิดคำสั่ง Branch ต้อง Flush คำสั่งในไปปีลайнจำนวน 3 คำสั่ง

สรุปได้ว่าทางเลือกต่างๆ ในการแก้กรณีนี้มี 2 วิธี ได้แก่

วิธีที่ 1 ทำการเพิ่มเกต AND และเพิ่ม MUX ในขั้นตอนเอ็กซ์คิวต์ ให้การตัดสินใจว่าจะกระโดดหรือไม่อยู่ในขั้นตอนเอ็กซ์คิวต์ จะทำให้ลดจำนวน Stall เหลือ 2 ไฟเกลล์

วิธีที่ 2 ทำการคำนวนตำแหน่งที่จะไปในขั้นตอนต่อไป ดังนั้น การคำนวนตำแหน่งจะเกิดขึ้นพร้อมกับการตัดคำสั่งและการอ่านเรจิสเตอร์ไฟล์ แต่การเปรียบเทียบจะทำได้หลังจากการอ่านค่าในเรจิสเตอร์ไฟล์แล้ว และเพิ่ม MUX เพิ่มตัวเปรียบเทียบ (Comparator) เพิ่มเกต AND ไปควบคุมการเปลี่ยนแปลงค่าในเรจิสเตอร์ PC อันนี้จะมีผลกับการ Forward ด้วย ซึ่งจะทำให้ต้องเพิ่มฮาร์ดแวร์สำหรับการ Forward จากขั้นตอนต่อไป วิธีการทั้งหมดนี้จะลดจำนวนไฟเกลล์ในการ Stall เหลือเพียง 1 ไฟเกลล์

สำหรับกรณีไปปีลайнรูปแบบอื่นๆ จะต้องการการจัดการที่ต่างกันไป ถ้าเป็นไปปีลайнที่ลีกมากกว่านี้ จะต้องการฮาร์ดแวร์สำหรับช่วยการ Forward มากขึ้น และอาจจะทำให้จำนวน Stall ที่ต้องใช้ในกรณีของคำสั่ง Branch มีมากขึ้นได้

ในที่นี้จะพิจารณาวิธีที่ 2 ดังนั้น จะแก้ไข Data Path ที่เกี่ยวกับการจัดการ Forward ในกรณีต่างๆ ดังนี้

กรณีที่ 1 จะต้องการการ Forward จากบัฟเฟอร์ MEM/WB ไปยังขั้นต่อไป ไปยังโอเพอแรนด์ของคำสั่ง Branch ดังต่อไปนี้ ซึ่งเป็นการเขียนรีจิสเตอร์ \$1 ก่อนการอ่านค่าโอเพอแรนด์

<b>WB</b>	add3	\$1,
MEM	add2	\$3,
EX	add1	\$4,
<b>ID</b>	beq	<b>\$1, \$2, Loop</b>
IF	next_seq_instr	

กรณีที่ 2 จะต้องการการ Forward จากบัฟเฟอร์ในขั้น EX/MEM ไปยังขั้นต่อไปสำหรับการอ่านโอเพอแรนด์ของคำสั่ง Branch เพื่อการเบรียบเทียบเงื่อนไข

WB	add3	\$3,
<b>MEM</b>	add2	<b>\$1,</b>
EX	add1	\$4,
<b>ID</b>	beq	<b>\$1, \$2, Loop</b>
IF	next_seq_instr	

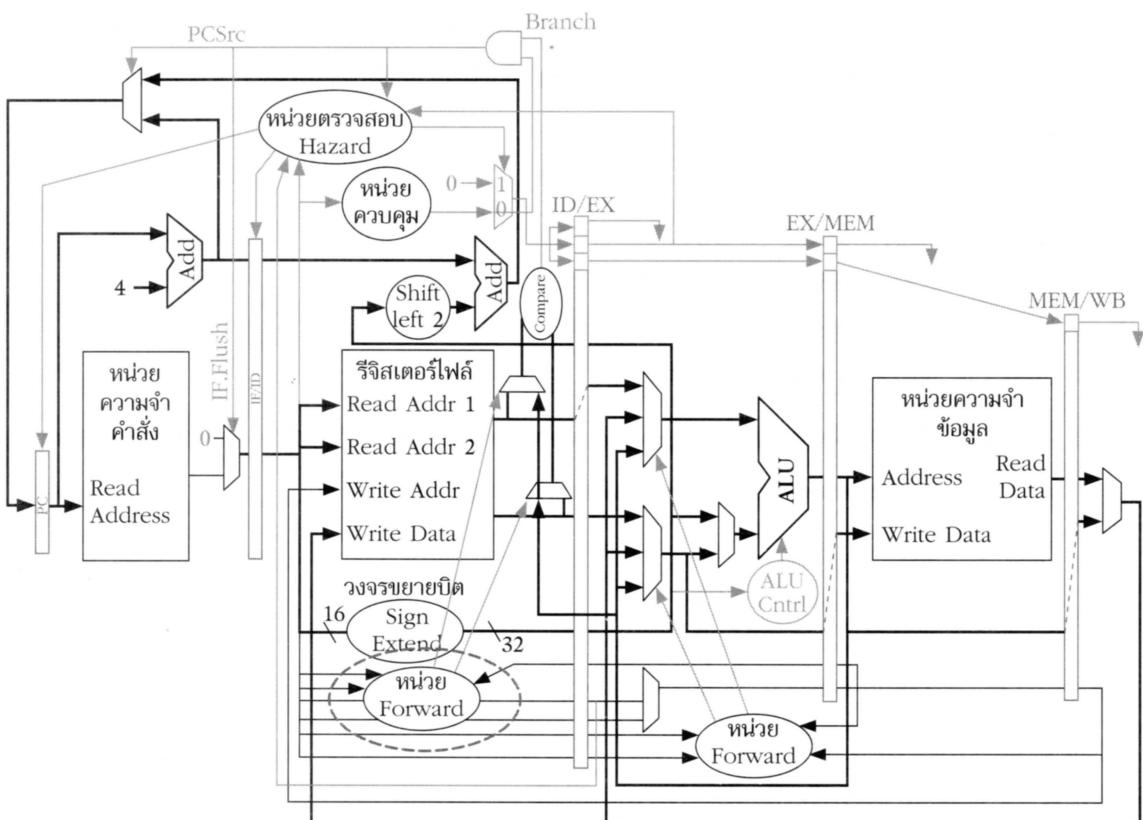
จึงต้องทำการปรับเงื่อนไขการควบคุมดังนี้

```

if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = IF/ID.RegisterRs))
    ForwardC = 1
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = IF/ID.RegisterRt))
    ForwardD = 1

```

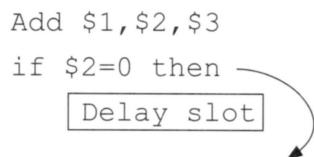
ทำการเพิ่มสัญญาณ ForwardC และ ForwardD เข้าไปเพื่อการตรวจสอบการ Forward เพิ่มสำหรับทั้ง 2 กรณีนี้ ดังแสดงในรูปที่ 6.34 ซึ่งแสดงการปรับปรุงชาร์ดแวร์เมื่อเพิ่มหน่วยควบคุมการ Forward และการปรับปรุง Data Path จะเห็นว่ามีหน่วยควบคุมสัญญาณ Forward ในขั้นตอนต่อไปแล้ว



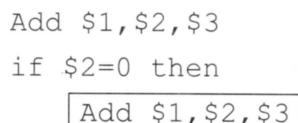
รูปที่ 6.34 การแก้ไข Data Path เพื่อลดการ Stall จากคำสั่ง Branch

เมื่อมีการเปลี่ยนแปลง Data Path ดังกล่าว จะลด Stall เหลือ 1 ไบเกิล ซึ่งจะทำให้กำจัด Stall ได้ด้วยเทคนิค Delay Branch โดยการหาคำสั่งอื่นที่ต้องนำมาทำงานในช่วงเวลาเดียวกัน (Delay Slot) และคำสั่งถัดไปจะเป็นคำสั่งที่เป็นคำสั่งที่จะกระโดดไป อันนี้เป็นหน้าที่ของตัวแปลงภาษาที่ทำการจัดลำดับคำสั่งหลังจากคำสั่ง Branch โดยทั่วไปแล้วโปรเซสเซอร์จะใช้เพียง 1 Delay Slot หรือใช้ 1 ไบเกิลเท่านั้น เพราะจะสามารถข่อน Stall จากคำสั่ง Branch ได้ช้ากว่า ถ้า Delay Slot ใช้มากกว่า 1 ไบเกิล จะทำให้โอกาสที่จะหาคำสั่งหลายๆ คำสั่งมาทำงานใน Slot นี้ยาก กรณีของชาร์ดแวร์เหล่านี้และการใช้เทคนิคการทำนาย Branch (Branch Prediction) ซึ่งมีทั้งในชาร์ดแวร์และซอฟต์แวร์จะเป็นที่นิยมมากกว่า ดังจะได้กล่าวถึงรายละเอียดต่อไป

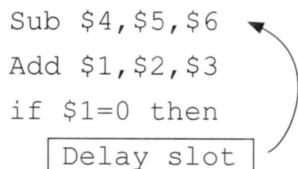
ตัวอย่างการหาคำสั่งมาทำงานใน Delay Slot มี 3 กรณี ได้แก่ กรณีแรก



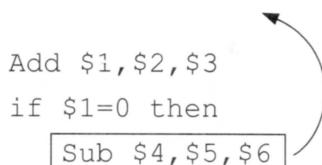
กรณีนี้คำสั่ง Add ไม่มีส่วนเกี่ยวข้องกับคำสั่ง Branch และเป็นคำสั่งที่ต้องทำอยู่แล้ว จึงนำคำสั่ง Add มาไว้ใน Delay Slot ได้เป็น



โดยในรูปแบบนี้จะสามารถย่อน Stall ได้ ซึ่งจะลดจำนวนไชเกิลโดยรวมได้ ส่วนในกรณีที่ 2 เป็นการกระโดดย้อนหลังไปคำสั่ง Sub ในกรณีนี้จะเห็นว่าคำสั่ง Add มีความสัมพันธ์กับคำสั่ง Branch เพราะต้องใช้โอเปอเรนเตอร์ \$1 และจะนำคำสั่ง Sub ไปไว้ใน Delay Slot ได้ เนื่องจากคำสั่ง Sub ไม่เกี่ยวข้องกับคำสั่ง Add แต่วิธีนี้จะช่วย Stall ได้ในกรณีที่ผลของคำสั่ง Branch ต้องกระโดดไปทำงานคำสั่ง Sub กรณีที่ไม่ต้องกระโดดไปทำงาน จะต้องแนใจว่าการทำงานคำสั่ง Sub ไม่มีผลต่อความถูกต้องของโปรแกรม เพียงแต่จะเพิ่มจำนวนคำสั่งทั้งหมดที่ทำงานเท่านั้น



หลังจากย้ายแล้วเป็น



### กรณีที่ 3 เป็นการกระโดดแบบไปข้างหน้า เช่น

```
Add $1,$2,$3
if $1=0 then
    Delay slot
Sub $4,$5,$6
```

สามารถทำการย้ายคำสั่ง Sub มาไว้ที่ Delay Slot ได้เช่นกัน

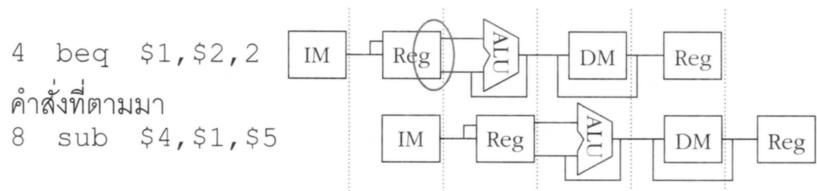
```
Add $1,$2,$3
if $1=0 then
    Sub $4,$5,$6
```

สำหรับกรณีอื่นๆ อาจจะลด Stall ได้โดยการใช้การท่านาย Branch ซึ่งจะเป็นการคาดเดา ตำแหน่งคำสั่งที่จะไปทำงาน (กรณีเงื่อนไขเป็นจริงสำหรับตัวอย่างข้างต้น) และจะไปเพ็ทช์คำสั่งนั้นเข้ามาในไปป์ไลน์ก่อนแม้ว่ายังไม่ทราบผลของการเปรียบเทียบเท่าจะต้องกระโดดไปทำคำสั่งนั้น หรือไม่ ทั้งนี้เพื่อให้เกิด Stall น้อยที่สุดในกรณีที่ต้องการกระโดดไปทำงาน โดยทั่วไปแล้ว ในการท่านายนั้นมีหลายรูปแบบ คือ

1. การท่านายแบบไม่กระโดด (Predict Not Taken) เป็นการทำนายว่าจะไม่กระโดด ซึ่งหมายถึง จะไปเพ็ทช์คำสั่งถัดไปที่ตำแหน่ง PC + 4 มาทำงาน ซึ่งอยู่ในไปป์ไลน์อยู่แล้ว กรณีนี้ถ้ามีการกระโดด จะมี Stall เกิดขึ้น เพราะต้อง Flush ไปป์ไลน์เอกสารคำสั่งถัดไปนั้น ออกจากไปป์ไลน์ แต่ถ้าไม่กระโดดไป จะไม่มี Stall เลย
  - ถ้ามีการกระโดดเกิดขึ้น จะต้องทำการ Flush ไปป์ไลน์โดยเอกสารคำสั่งหลังจากคำสั่ง Branch ออกจากไปป์ไลน์ ซึ่งจำนวน Stall ที่ใช้สำหรับการ Flush จะขึ้นอยู่กับว่า จะได้ผลการตรวจสอบการ Branch ที่ขึ้นตอนใด
  - ถ้าทราบผลการเปรียบเทียบในขั้นตอนการเข้าถึงหน่วยความจำ จะต้อง Flush คำสั่งในขั้นตอนการเพ็ทช์ ขั้นตอนการตีโค้ด ขั้นตอนการอีกซีคิวต์ และจะทำให้เสีย Stall ไป 3 ไซเกล

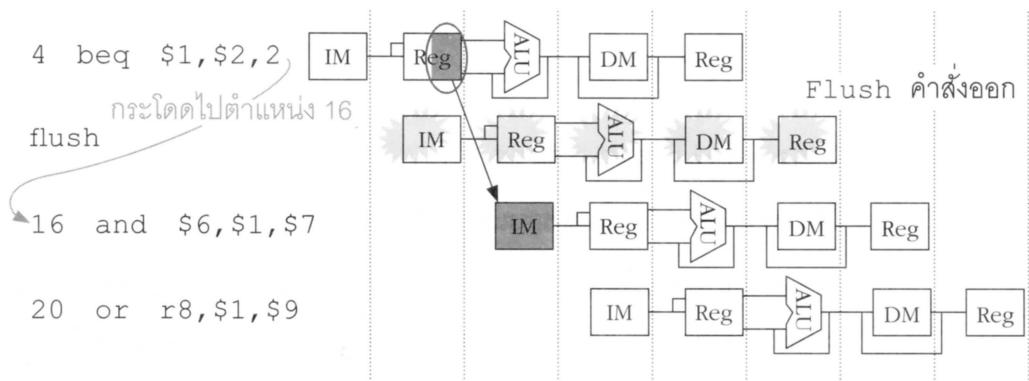
- ถ้าทราบผลการเปรียบเทียบในขั้นตอนการอีกชีคิวต์ จะต้อง Flush คำสั่งในขั้นตอนการเฟตช์ และขั้นการ์ด็อกต์ ซึ่งจะทำให้ไขเกลในการ Stall เป็น 2 ไขเกล
- ถ้าทราบผลการเปรียบเทียบในขั้นตอนการด็อกต์ จะต้อง Flush คำสั่งในขั้นตอนการเฟตช์และจะทำให้ Stall ไป 1 ไขเกล

นอกจากนี้ ในตัวยาาร์ดแวร์เองต้องไม่มีการเปลี่ยนสถานะภายใน ซึ่งควบคุมโดยสัญญาณ MemWrite และสัญญาณ RegWrite ของคำสั่งที่เข้ามาก่อนคำสั่ง Branch และเมื่อ Flush แล้ว จึงสามารถไปเฟตช์คำสั่งที่จะกระโดดไปทำงานได้ (รูปที่ 6.35 และรูปที่ 6.36)



รูปที่ 6.35 ขณะทำงานคำสั่ง Branch และมีคำสั่งถัดไป Sub อยู่ในไปป์ไลน์

ในการ Flush คำสั่งที่อยู่ในขั้นตอนการเฟตช์จะใช้สัญญาณ IF.Flush เพื่อป้อนค่า 0 ไปยังบัฟเฟอร์ IF/ID หรือแปลงคำสั่งในไปป์ไลน์เป็นคำสั่ง Nop



รูปที่ 6.36 ขณะทำงานคำสั่ง Branch คำสั่งถัดไปอยู่ในไปป์ไลน์ต้องถูก Flush ออก และไปทำงานคำสั่งที่จะกระโดดไป

ในการทำนาย Branch แบบไม่กระโดดนั้น จะใช้มากในการกรณีของ Loop แบบที่มีเงื่อนไขอยู่ด้านล่าง (Bottom Loop) เช่น

```

Loop: 1st loop instr
      2nd loop instr
      .
      .
      .
      last loop instr
      bne $1,$2,Loop
      fall out instr
  
```

นั่นคือจะเขียนเงื่อนไขการตรวจสอบและให้กระโดดย้อนกลับไปต้นลูป การทำนาย Branch แบบไม่กระโดดจะถูกต้องเป็นส่วนมากในการกรณีดังแบบนี้

แต่ถ้าเป็นลูปแบบเงื่อนไขอยู่ด้านบน (Top Loop) เช่น

```

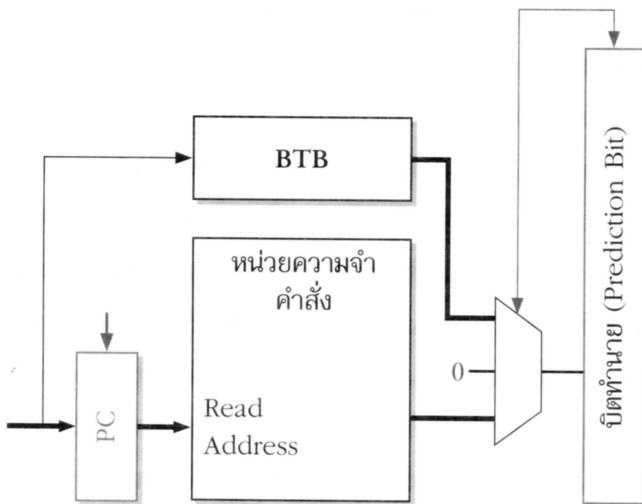
Loop: beq $1,$2,Out
      1nd loop instr
      .
      .
      .
      last loop instr
      j Loop
Out: fall out instr
  
```

จะทำให้ทำนายผิดทุกรอบตั้งแต่รอบแรกจนถึงรอบสุดท้าย

2. การทำนายแบบกระโดด (Predict Taken) จะเป็นการทำนายโดยเอาตำแหน่งที่จะกระโดดไปมาทำงาน แต่ว่าการทำนายแบบนี้จะต้องการ Stall อย่างน้อย 1 ไซเกล เนื่องจากต้องการคำนวนหาตำแหน่งที่จะกระโดดไป แต่ว่าถ้ามีการใช้แคชมาช่วยจดจำตำแหน่งที่จะกระโดดไป อาจจะลดเวลาส่วนการคำนวนตำแหน่งนี้ไปได้ ถ้าเป็นไปปีไลน์ที่ลึกซึ้งจะทำให้ Stall สำหรับคำสั่ง Branch มีมากขึ้น ดังนั้น การทำนาย Branch แบบอื่นๆ อาจจะช่วยได้เพิ่มขึ้น

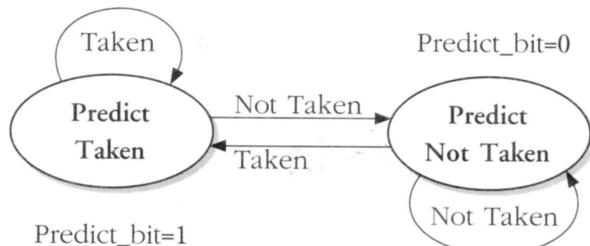
3. การใช้ Dynamic Branch Prediction เป็นการทำนายตำแหน่งที่จะกระโดดไประหว่างการทำงานของโปรแกรม โดยเก็บข้อมูลระหว่างการทำงานเพื่อใช้ประกอบการทำนาย ในการทำงานนั้นจะใช้ BHT (Branch History Table) เป็นบัฟเฟอร์ที่ใช้เก็บผลของการทำนายสำหรับแต่ละคำสั่งกระโดดในรอบก่อนหน้า โดยในขั้นตอนการเฟตช์จะใช้บัฟเฟอร์นี้ จะเอาค่า 4 บิตบนของรีจิสเตอร์ PC มาเป็นตัวชี้ในบัฟเฟอร์นี้เพื่อทำการค้นหาในบัฟเฟอร์ว่าคำสั่งนี้จะเป็นคำสั่ง Branch หรือไม่ และผลของคำสั่ง Branch อันนี้ในรอบก่อนหน้านี้เคยเป็นอย่างไร การทำนายผิดจะไม่มีผลต่อความถูกต้องของโปรแกรม แต่มีผลต่อเวลาในการทำงานทั้งหมด เพราะเมื่อทำนายผิด จะต้องทำการ Flush ไปปีลน์และไปเริ่มการทำงานไปปีลน์ใหม่ (Restart) และจะทำการบันทึกผลการทำนายในบัฟเฟอร์ BHT นี้ การทำนายนี้จะเกิดในขั้นตอนการตัดหลังจากขั้นตอนการเฟตช์ และเมื่อถูกว่าคำสั่งนั้นเป็นคำสั่ง Branch

สำหรับ BHT ใช้เป็นที่เก็บผลการทำนายท่านั้น ไม่ได้เก็บตำแหน่งของคำสั่งที่จะกระโดดไป จึงต้องใช้ Branch Target Buffer (BTB) เก็บตำแหน่งของคำสั่งที่จะกระโดดไป รูปที่ 6.37 แสดงการเปลี่ยนแปลง Data Path เมื่อเพิ่ม BTB (Branch Target Buffer) โดยนำ 4 บิตบนของ PC มาคั้นใน BTB ด้วย BTB จะใช้เก็บตำแหน่งของคำสั่งที่จะกระโดดไป ในบัฟเฟอร์ IF/ID จะเพิ่มการเก็บบิตการทำนาย (Prediction Bit) โดย Prediction Bit จะเป็นตัวเลือกตำแหน่งของคำสั่งใน MUX ดังแสดงในรูปที่ 6.37 จะเห็นว่าในกรณีนี้เราต้องการพอร์ตจำนวน 2 พอร์ตในการอ่านคำสั่งจากหน่วยความจำคำสั่ง จากรูปจะเห็นว่า BTB ยังถูกเขียนไปพร้อมๆ กับการอ่านคำสั่งในหน่วยความจำคำสั่งด้วย ในรูปแบบนี้แม้ว่าจะทำนายผิดพลาดก็จะไม่มี Stall เกิดขึ้น



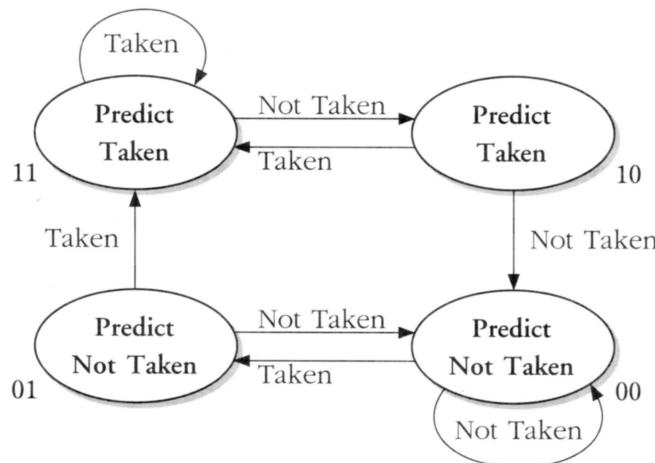
รูปที่ 6.37 Data Path ที่เปลี่ยนแปลงเมื่อเพิ่ม BTB

ตัวอย่างการใช้การทำนายแบบ 1-Bit Prediction พิจารณา Bottom Loop สมมติให้เริ่มต้นค่า Predict\_bit=0 ในรอบแรก ตัวทำนายจะทำนายผิด เนื่องจากรอบแรกต้องกระโดดกลับไปต้นลูป และถ้ายังทำอยู่ในลูป ก็จะทำนายถูกทุกครั้ง เมื่อออกจากลูปจะทำนายผิดอีกครั้ง และเปลี่ยนค่า Predict\_bit=0 ถ้าลูปทำงาน 10 รอบก็จะทำนายถูกต้อง 90% ในการทำนายแบบ 1 บิตนี้ เปรียบเสมือนการใช้เครื่องยนต์สถานะจำガัด ดังแสดงในรูปที่ 6.38 กำหนดสถานะเริ่มต้นอยู่ที่ Predict Not Taken ถ้าอยู่ในสถานะ Predict Taken คือจะให้อาต์พุตค่า Predict\_bit=1 คือการทำนายว่ากระโดด และถ้าอยู่ในสถานะ Predict Not Taken คือจะให้อาต์พุตค่า Predict\_bit=0 คือการทำนายว่าไม่กระโดด ถ้าอยู่ในสถานะ Predict Not Taken และถ้ามีการทำนายผิด กล่าวคือ Input เป็น Not Taken จะเปลี่ยนคำทำนายในครั้งต่อไป โดยจะเปลี่ยนสถานะเป็น Predict Not Taken อีกนัยหนึ่ง ถ้าทำนายผิด ก็จะเปลี่ยนคำทำนายเป็นตรงกันข้ามในรอบถัดไป



รูปที่ 6.38 State Machine ของการทำนาย 1-Bit Prediction

รูปที่ 6.39 แสดงกรณีใช้ 2 บิตช่วยเก็บผลการทำนายก่อนหน้า นั่นคือจะเปลี่ยนบิตคำทำนายเมื่อทายผิด 2 ครั้ง ซึ่งตัวทำนายแบบนี้จะสามารถเพิ่มเติมโดยใช้กีบิตก็ได้



รูปที่ 6.39 ไดอะแกรมสถานะของการทำนายใช้ 2 บิต

## ■ 6.6 การจัดการความผิดปกติ และการขัดจังหวะ

ความผิดปกติ (Exception) หรือการขัดจังหวะ (Interrupt) จะมีลักษณะการทำงานเหมือนคำสั่ง Jump จัดได้ว่าอาจจะก่อนเกิด Control Hazard ได้ รูปแบบของความผิดปกติมีหลายรูปแบบ ได้แก่

- การเกิดโอเวอร์โหลดของคำสั่งที่เกี่ยวกับคณิตศาสตร์แบบ R-Type
- การทำงานของคำสั่งที่ไม่มีในระบบ (Undefined Instruction)
- การร้องขอการใช้อุปกรณ์อินพุตเอาต์พุต
- การเกิดร้องขอการใช้บริการของระบบปฏิบัติการ เช่น Page Fault, TLB Exception
- ฮาร์ดแวร์ทำงานผิดปกติ

ในการเกิดข้อผิดพลาดเหล่านี้ ไปป์ไลน์ต้องทำการหยุดการทำงานคำสั่งที่มี Exception ดังกล่าว โดยต้องทำให้คำสั่งที่เกิดก่อนคำสั่งที่มี Exception นั้นทำงานเสร็จไปก่อน และทำการ Flush คำสั่งที่อยู่ในไปป์ไลน์หลังจากคำสั่งที่มี Exception นี้ นอกจากนี้ ต้องเก็บสถานะการทำงาน

ปัจจุบันໄว້ ເພື່ອວ່າຫຼັງຈາກຈັດການແກ້ໄຂ Exception ນັ້ນ (Exception Handler) ແລ້ວ ສາມາຄົດືນຄ່າສະານະເດີມແລະທຳການຕ່ອໄປໄດ້ ໃນກາຮັບກາຮັບທຳການໄປຈັດການ Exception ນັ້ນເປັນໜ້າທີ່ຂອງຮະບົບປົງປັດກາຮູ້ເວຼົອຂ້ວແມ່ເຊີນ

ລັກຄະນະກາຮັດ Exception ມີ 2 ຮູ່ແບບ ໄດ້ແກ່

1. ກາຮັດກາຮັດຈັດຈະກຳແບບ Asynchronous ແ່ນ ເກີດຈາກເຫດຖາກຮົມກາຍນອກ ອາຈະເກີດຮ່ວມທຳການຕໍ່າ ກີ່ໄດ້ ດັ່ງນັ້ນ ຄໍາສິ່ງທີ່ກຳລັງທຳການອູ່ໃນໄປປ່າໄລນົມຄວະທຳການ ໄທ້ສໍາເຮົາຈ່ອນຈະສົ່ງທົດກາຮັບທຳການໄປຢັງຮະບົບປົງປັດກາຮູ້ເວຼົອເພື່ອຈັດກາຮັດຈັດຈະກຳຈະເປັນກາຮູ້ເຫັນວ່າ ຊ້ວຍກາວແລະຫຼັງຈາກຈັດກາຮັດແລ້ວກີ່ກຳລັນມາທຳການຕ່ອໄປ
2. Traps (ຫຸ້ນ ຂອງ Exception) ເປັນກາຮັບທຳການແບບ Synchronous ເກີດຈາກເຫດຖາກຮົມກາຍນອກເຫັນກັນ ກາຮັດກາຮັດຕ້ອງອາຫັນ Exception Handler ດັ່ງນັ້ນ ຈະຕ້ອງຫຼຸດກາຮັບທຳການຂອງຄໍາສິ່ງທີ່ກຳລັງທຳການອູ່ໃນໄປປ່າໄລນົມກ່ອນຈະສົ່ງທົດກາຮັບທຳການໄປຢັງຮະບົບປົງປັດກາຮູ້ເວຼົອເພື່ອໃຫ້ Exception Handler ຈັດກາຮັດຂໍ້ອັດພລາດນັ້ນ ຄໍາສິ່ງທີ່ມີ Exception ອາຈະຖູກທຳຫ້າຫຼັງຈາກຈັດກາຮັດຂໍ້ອັດພລາດແລ້ວ ພ້ອມວ່າຈະທຳການຕ່ອໂຫຼວງວ່າຈະຫຼຸດກາຮັບທຳການໄປເລີຍກີ່ໄດ້

**ຕັວຢ່າງທີ່ 6.1 ກາຮັດ Exception ຂອງຄໍາສິ່ງໃນ MIPS ຈາກໂຄຮງສ້າງຂອງ MIPS 5 ບັນດອນໃນຮູບທີ່ 6.7 ພິຈາລະນາກາຮັດException ໄດ້ແກ່**

- ໂເວຼົອໄຟລ໌ກ່ອງກາຮັດການຕໍ່ານວນທາງຄົນຕາສຕົວເຊີງເກີດໃນບັນດອນກາຮັດເອົກເຊີກວົດ ເປັນກາຮັດເປັນແບບ Synchronous
- ມີຄໍາສິ່ງທີ່ມີມີໃນຮະບົບ (Undefined Instruction) ເກີດຂຶ້ນໃນບັນດອນກາຮັດໄດ້ ເປັນກາຮັດເປັນແບບ Synchronous
- ເກີດ TLB Page Fault ເກີດໃນບັນດອນກາຮັດເພື່ອແຕ່ງແລະບັນດອນກາຮັດເຫັນທີ່ມີມີມີໃນຮະບົບ ເປັນກາຮັດເປັນແບບ Synchronous
- ເກີດກາຮັດຂອອິນພຸດ-ເຄາດພຸດໃນບັນດີ່າ ກີ່ໄດ້ ເປັນກາຮັດເປັນແບບ Asynchronous

- เกิดชาร์ดแวร์ทำงานผิดพลาดได้ในขั้นใดๆ ก็ได เป็นการจัดการเป็นแบบ Asynchronous

ความยากของการจัดการ Exception ในไปป์ไลน์ได้แก่สาเหตุต่างๆ ดังนี้

- Exception นั้นเกิดขึ้นระหว่างการทำงานคำสั่งหนึ่งๆ ในขั้นตอนการเข้าถึงหน่วยความจำหรือขั้นตอนการอีกซีดิวตี้
- คำสั่งที่เกิด Exception จะต้องถูกทำใหม่ตั้งแต่ต้นได้หลังจากการจัดการ Exception นั้นเสร็จสิ้น

ก่อนการเริ่มทำการคำสั่งใหม่นั้น ต้องทำการเก็บค่าตำแหน่งของคำสั่งที่มี Exception ก่อน เพื่อจะได้ทราบว่าเมื่อจัดการ Exception เรียบร้อยแล้ว จะได้กลับไปทำการคำสั่งในตำแหน่งใดใหม่ ถ้าคำสั่งที่ถูกทำใหม่อีกครั้งนั้น ไม่ใช่คำสั่งประเภท Branch ก็ไปทำการเฟตช์คำสั่งนั้นและคำสั่งถัดไปใหม่ ตามลำดับ ถ้าเป็นคำสั่งแบบ Branch ก็จะต้องตรวจสอบเงื่อนไขใหม่หลังจากการจัดการ Exception แล้ว ดังนั้น กล่าวได้ว่าสำหรับการทำงานเมื่อเกิด Exception ที่คำสั่ง I ได้แก่

- การทำให้คำสั่ง I ให้ไปรอดอยู่คิวเพื่อการเฟตช์เข้ามาในไปป์ไลน์ครั้งต่อไป
- ต้องทำการยับยั้งการเขียนทั้งหมดที่จะเกิดขึ้นเกี่ยวกับคำสั่งนั้นๆ และคำสั่งถัดไปในไปป์ไลน์ จนกว่าการจัดการ Exception เสร็จสิ้น
- หลังจากนั้นจะส่งการทำงานไปยังระบบปฏิบัติการให้จัดการ Exception และต้องจัดเก็บตำแหน่งคำสั่งไว้ระหว่างการเกิด Exception จะเพิ่มขึ้น ซึ่งต้องมากกว่าจำนวน Delay Slot อีกตัวอย่าง 1 ตัวด้วย

ถ้าใช้วิธี Delay Branch ในการจัดการคำสั่ง Branch คำสั่งที่อยู่ใน Delay Slot ก็อาจจะไม่สัมพันธ์กับคำสั่งที่ทำให้เกิดการ Exception ในขณะนั้น ดังนั้น จำนวนรีจิสเตอร์ PC ที่ต้องจัดเก็บตำแหน่งคำสั่งไว้ระหว่างการเกิด Exception จะเพิ่มขึ้น ซึ่งต้องมากกว่าจำนวน Delay Slot อีกตัวอย่าง 1 ตัวด้วย

เราจะเรียกไปป์ไลน์ที่หยุดการทำงานของคำสั่งทั้งหมดก่อนคำสั่งที่ทำให้เกิด Exception ไว้และจะเริ่มคำสั่งข้างหลังคำสั่งที่เกิด Exception ได เรียกการจัดการแบบนี้ว่าเป็น Precise Exception โดยทั่วไปแล้ว คำสั่งที่มี Exception จะไม่เปลี่ยนสถานะอยู่แล้ว

นอกจากนี้แล้ว อาจจะมีหลายๆ Exception เกิดขึ้นได้ภายใน 1 ไซเกิล การเกิด Exception ใน 1 ไซเกิลของ MIPS จะเห็นว่าถ้าไปปีลайнมีลักษณะต่างกัน การวิเคราะห์โอกาสการเกิด Exception ก็ต่างกันไป เช่น ในกรณีของ MIPS ใน 1 ไซเกิล อาจจะเกิด Page Fault ของหน่วยความจำข้อมูลในขั้นตอนการเข้าถึงข้อมูล ในขั้นตอนการเข้าถึงชีคิวต์อาจจะเกิดการໂອเวอร์โฟล์วของ ALU ขั้นตอนการดีโคดอาจจะเกิด Undefined Instruction ขั้นตอนการเฟตช์อาจจะเกิด Page Fault ของหน่วยความจำคำสั่ง

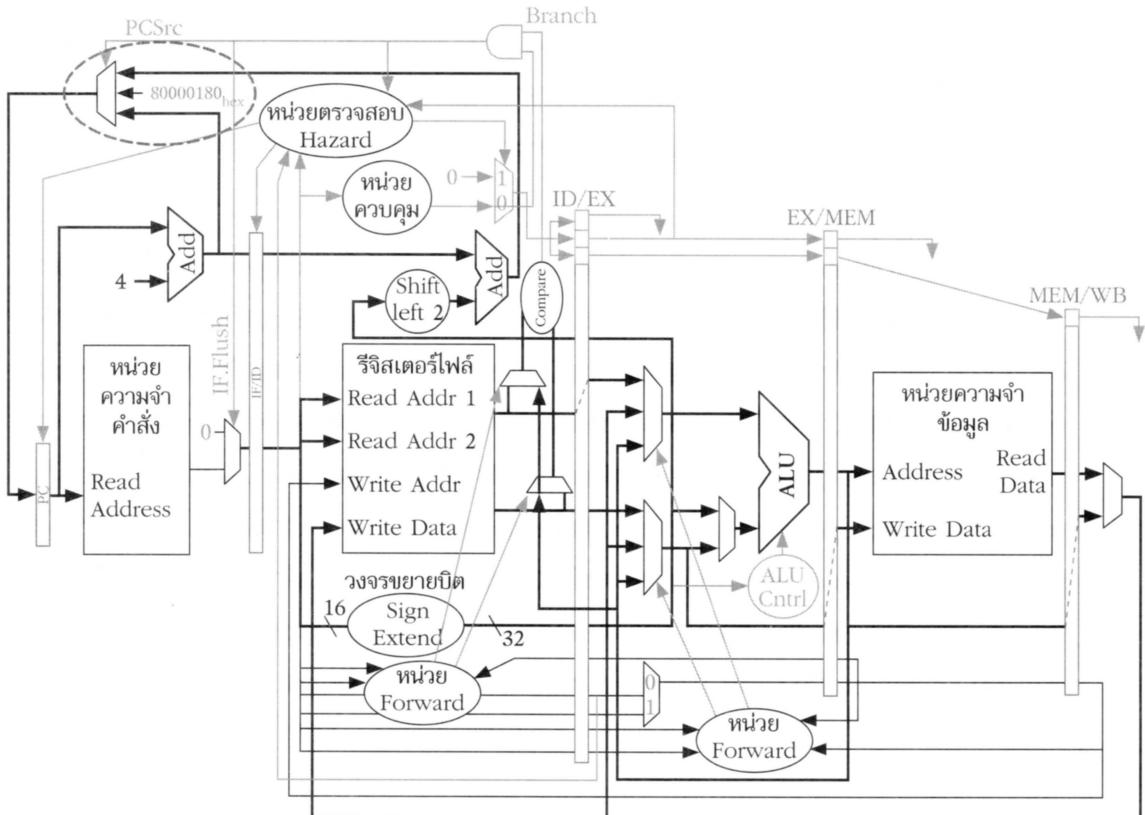
ในการจัดการ Exception หลายๆ ตัวแบบนี้ จะต้องมีกลไกการจัดการให้เป็นไปตามลำดับของคำสั่งที่เข้ามาในไปปีลайнด้วย เพราะต้องการให้ผลการทำงานเหมือนกับการทำงานแบบที่ไม่ใช้ไปปีลайн เช่น ถ้าคำสั่ง Load เกิด Page Fault ในขั้นตอนที่เข้าถึงหน่วยความจำข้อมูล และคำสั่งที่ตามมาเป็นคำสั่ง Add เกิด Page Fault ในขั้นตอนการเฟตช์นั้น Exception ของคำสั่ง Add จะเกิดขึ้นก่อน เพราะเกิดในไซเกิลที่ 2 แต่ Exception ของคำสั่ง Load จะเกิดในไซเกิลที่ 4 ตามความหมายของโปรแกรมคำสั่ง Load ต้องทำก่อนคำสั่ง Add ดังนั้น การจัดการ Exception ก็ต้องจัดการ Exception ของคำสั่ง Load ก่อนเข่นกัน

การจัดการกรณีการทำงานของ Exception ที่เกิดขึ้นไม่เป็นไปตามลำดับ (Out-Of-Order) ของคำสั่งนั้น โดยทั่วไปแล้วเราจะใช้รีจิสเตอร์พิเศษเก็บสถานะ (Status Vector) มาช่วย ซึ่งจะมีไว้สำหรับแต่ละคำสั่งเพื่อเกิด Exception ขึ้น การเขียนต่างๆ ที่จะเกิดขึ้นหลังจากคำสั่งนั้นและคำสั่งต่อไปจะต้องถูก Disable และจะมีการตรวจสอบ Exception Vector เมื่อทำงานในขั้น เผียบค่ากลับ ซึ่งจะทำให้การจัดการ Exception เป็นไปตามลำดับของคำสั่งในโปรแกรมเหมือนกับทำงานบนเครื่องที่ไม่ใช้ไปปีลайн กล่าวคือ Exception ที่เกิดจากคำสั่ง I จะต้องถูกจัดการก่อน Exception ของคำสั่ง I + 1 นั่นเอง

ในกรณีของ MIPS นั้น การปรับค่าสถานะต่างๆ ทั้งหมดเกิดขึ้นตั้งแต่ขั้นการเข้าถึงหน่วยความจำเป็นต้นไป ทำให้การจัดการ Precise Exception ทำได้ง่าย แต่สำหรับสถาปัตยกรรมบางแบบที่มีการเปลี่ยนแปลงค่าสถานะต่างๆ ก่อนหน้านั้น เมื่อมี Exception เกิดขึ้นและต้องหยุดการทำงานไปจัดการ Exception นั้น สถานะค่าต่างๆ ในเครื่องอาจจะเปลี่ยนไป ทำให้การจัดการเป็นแบบ Imprecise Exception ต้องมีการเพิ่มฮาร์ดแวร์มาช่วยแก้ไขเพื่อตรวจสอบและช่วยในการคืนค่าต่างๆ กัน

สำหรับการใช้ Delay Slot นั้นต้องระวังในกรณีของเครื่องที่มีลักษณะแบบ Condition Code ในกรณีของคำสั่งประเภท Branch ลักษณะแบบ Condition Code นั้นจะมีรีจิสเตอร์พิเศษในการเขตค่าสถานะของการเปรียบเทียบ เพราะคำสั่งที่มาใช้ใน Delay Slot อาจจะเปลี่ยนแปลงค่าในรีจิสเตอร์ Condition Code ไปได้ และต้องระวังว่าคำสั่งสุดท้ายที่เขตค่า Condition Code นั้นคือคำสั่งใดด้วย

ในรูปที่ 6.40 แสดง Data Path ในการออกแบบเพื่อจัดการ Exception นี้ ซึ่งจะต้องการรีจิสเตอร์เพิ่มเติมคือรีจิสเตอร์ Cause เพื่อระบุสาเหตุที่มาของ Exception สำหรับการจัดการ Exception ต่างๆ ที่เกิดขึ้นและต้องการสัญญาณ CauseWrite เพื่อควบคุมการเขียนรีจิสเตอร์ Cause ต้องการรีจิสเตอร์ EPC เพื่อเก็บแอดเดรสของคำสั่งที่มีปัญหาเหล่านี้ ยาร์ดแวร์จะต้องบันทึกแอดเดรสเหล่านี้ และต้องการสัญญาณ EPCWrite เพื่อควบคุมการเขียนรีจิสเตอร์ EPC โดยในตัวระบบปฏิบัติการต้องจัดการ Exception ให้สัมพันธ์กับคำสั่งนั้นๆ นอกจากนี้ ยังต้องการกลไกการโหลดค่าตำแหน่งของ Exception Handler เข้าไปยังรีจิสเตอร์ PC โดยการเพิ่ม MUX เข้าไปที่อินพุตของรีจิสเตอร์ PC และให้ขาอีกขาของอินพุตอ่านมาจากแอดเดรสของ Handler และต้องการกลไกการ Flush คำสั่งที่มี Exception นั้น และคำสั่งถัดมาในไปป์ไลน์ หลังจากการจัดการ Exception เสร็จสิ้นแล้ว



รูปที่ 6.40 Data Path เมื่อเพิ่มการจัดการการเกิด Exception

## ■ 6.7 ตัวอย่างการวิเคราะห์ประสิทธิภาพของไปป์ไลน์

โดยทั่วไปแล้ว ถ้าไปป์ไลน์มี 5 ขั้นตอน เราอาจคาดเดาได้ว่าจะต้องได้ความเร็วเพิ่มขึ้นเป็น 5 เท่า แต่ความเป็นจริงแล้วแต่ละขั้นตอนนั้นอาจจะใช้เวลาไม่เท่ากัน เมื่อนำมาทำเป็นรูปแบบไปป์ไลน์ จะต้องทำให้แต่ละขั้นให้เวลาเท่ากัน โดยอาจจะยืดเวลาการทำงานของบางขั้นตอนออกไป ดัง แสดงในตารางต่อไปนี้

ประเภทคำสั่ง	เฟตช์	อ่านค่า รีจิสเตอร์	ทำงานใน ALU	เข้าถึงหน่วย ความจำข้อมูล	เขียนผลลัพธ์สู่ รีจิสเตอร์	รวม (นาโนวินาที)
ALU Type	2	1	2	0	1	6
Load Word	2	1	2	2	1	8
Store Word	2	1	2	2		7
Branch	2	1	2			5
Jump	2					2

จะเห็นว่าในแต่ละขั้nob จะใช้เวลาไม่เท่ากัน คำสั่งแต่ละประเภทจะใช้เวลาทั้งหมดไม่เท่ากัน เพราะแต่ละคำสั่งใช้หน่วยต่างๆ ไม่เหมือนกัน และแต่ละขั้nob ของไปป์ไลน์จะต้องเพิ่มบัฟเฟอร์ และทำให้เกิด Overhead ในแต่ละขั้nob เท่ากับ 2 นาโนวินาที ลองเบริยบเทียบเวลาการทำงานของคำสั่ง Lw สามคำสั่งที่ไม่เกี่ยวข้องกัน ว่าในรูปแบบการทำงานแบบไปป์ไลน์จะใช้เวลาเท่าไร และแบบไม่ใช้ไปป์ไลน์จะใช้เวลาเท่าไร

ในรูปแบบไปป์ไลน์ ทำงานคำสั่ง Lw สามคำสั่งติดกันจะใช้เวลาทั้งหมด  $(2 + 2) \times (5 + 2) = 22$  นาโนวินาที

ในรูปแบบไม่ใช้ไปป์ไลน์ คำสั่ง Lw สามคำสั่งติดกันจะใช้เวลาทั้งหมด  $8 \times 3 = 24$  นาโนวินาที จะได้ Speedup เท่ากับ  $24/22 = 1.09$  เท่า

ลองพิจารณาต่อไปว่า ถ้าเป็นคำสั่ง Lw ติดกันจำนวน 1,000 คำสั่งควรจะได้ Speedup เท่าไร

---

**ตัวอย่างที่ 6.2** สังเกตว่าใน MIPS คำสั่ง Branch และ Store ใช้ 4 ไซเกิล และคำสั่งอื่นๆ ใช้ 5 ไซเกิล

- ถ้าความตื้นของคำสั่ง Branch เท่ากับ 12% และคำสั่ง Store มีความตื้นของการใช้งานเท่ากับ 5% จงหา Overall CPI

จะได้ CPI รวม ดัง  $(0.17 \times 4) + (0.83 \times 5) = 4.82$

- ถ้าจะปรับปรุง CPI คือทำให้คำสั่งแบบ ALU เสร็จในขั้นเดียวเท่านั้นโดยความจำ (ทำให้ลดไป 1 ไบเกิล) ถ้าความถี่ของคำสั่งเกี่ยวกับ ALU เป็น 47% จงหา CPI รวม

$$\text{จะได้ } \text{CPI}_{\text{รวม}} = 0.17*4 + 0.47*4 + 0.36*5 = 4.36$$

$$\text{ดังนั้นจะได้ Speedup} = 4.82/4.36 = 1.1$$


---

**ตัวอย่างที่ 6.3** ถ้าเครื่องคอมพิวเตอร์แบบไม่ใช้ไปปีลินมี Cycle Time เป็น 1 นาโนวินาที และคำสั่งแบบ ALU Operation หรือคำสั่งแบบ Branch ทำงานใช้ 4 ไบเกิล และคำสั่งเกี่ยวกับหน่วยความจำใช้เวลา 5 ไบเกิล กำหนดให้ความถี่ของคำสั่งต่างๆ เป็นดังนี้

คำสั่งเกี่ยวกับ ALU มี 40% ของคำสั่งทั้งหมด คำสั่ง Branch มี 20% ของคำสั่งทั้งหมด และคำสั่งเกี่ยวกับหน่วยความจำมี 40% ของคำสั่งทั้งหมด ถ้าเพิ่มการใช้ไปปีลินทำให้เพิ่ม Overhead ต่างๆ ในส่วนของการเริ่มต้นการทำงานในแต่ละไบเกิลไป 0.2 นาโนวินาที อยากรابร่าจะได้ Speedup ในส่วนของ CPU Time สำหรับการใช้ไปปีลินเท่าไร

$$\text{CPU Time} = \text{Clock Cycle Time} \times \text{CPI}$$

$$= 1 \text{ นาโนวินาที} \times [(40\% + 20\%) * 4 + 5 * 40\%]$$

$$= 4.4 \text{ นาโนวินาที}$$

$$\text{จะได้ Speedup} = 4.4/(1 + 0.2) = 3.7 \text{ เท่า}$$


---

**ตัวอย่างที่ 6.4** ถ้าใช้หน่วยคำนวน 5 ตัวทำงานในแต่ละไบเกิลของการทำงานแบบไปปีลิน ซึ่งแต่ละหน่วยใช้เวลา 10 นาโนวินาที, 8 นาโนวินาที, 10 นาโนวินาที, 10 นาโนวินาที และ 7 นาโนวินาที ตามลำดับ การใช้ไปปีลินทำให้เพิ่มเวลาที่เป็น Overhead ไป 1 นาโนวินาที ให้หา Speedup เทียบกับรูปแบบไบเกิลเดี่ยว

สำหรับกรณีไม่ใช้ไปปีลิน ใช้เวลารวมทั้งหมด 1 ไบเกิล  $= 10 + 8 + 10 + 10 + 7 = 45$  นาโนวินาที

สำหรับ Cycle Time ของไปป์ไลน์ ขั้นตอนที่ใช้เวลามากที่สุดคือใช้ 10 นาโนวินาที รวมกับเวลา Overhead ซึ่งเท่ากับ 1 นาโนวินาที รวมเป็น 11 นาโนวินาที

$$\text{ดังนั้น Speedup} = 45/11 = 4.1 \text{ เท่า}$$


---

#### ตัวอย่างที่ 6.5 พิจารณากรณีของคำสั่ง Branch ดังนี้

คำสั่ง	ความถี่ (%)
Branch แบบย้อนกลับ (ของจำนวน Branch ทั้งหมด)	25%
Branch แบบ Taken (ของจำนวน Branch ทั้งหมด)	53%
คำสั่ง Branch (ของจำนวนคำสั่งทั้งหมด)	65%

สมมติว่า 90% ของการกระโดดเป็นแบบย้อนกลับที่เป็นแบบ Taken ให้หาเบอร์เข็นต์ของการกระโดดแบบไปข้างหน้าที่เป็น Taken สำหรับคำสั่ง Branch ถ้าเราทราบว่า

$$\begin{aligned} \text{เบอร์เข็นต์ของการกระโดดทั้งหมด} &= (\text{เบอร์เข็นต์ของการกระโดดแบบย้อนกลับที่เป็น} \\ &\quad \text{Taken} \times \text{เบอร์เข็นต์ของคำสั่ง Branch} \\ &\quad \text{แบบย้อนกลับ}) + (\text{เบอร์เข็นต์ของการกระโดด} \\ &\quad \text{แบบไปข้างหน้าที่เป็น Taken} \times \text{เบอร์เข็นต์ของ} \\ &\quad \text{คำสั่ง Branch แบบไปข้างหน้า}) \end{aligned}$$

$$53\% = (90\% \times 25\%) + (\text{เบอร์เข็นต์ของการกระโดดแบบ} \\ \text{ไปข้างหน้าที่เป็น Taken} \times 75\%)$$

ดังนั้น % ของการกระโดดแบบไปข้างหน้าที่เป็นแบบ

$$\text{Taken} = (53\% - 22.5\%)/75\% = 40.7\%$$


---

**ตัวอย่างที่ 6.6** ถ้ามีคำสั่งเกี่ยวกับการใช้ข้อมูลหน่วยความจำจำนวน 40% ของໂടด์ทั้งหมด และ CPI ของไปปีลน์แบบอุดมคติเป็น 1 สำหรับเครื่องที่มี Structural Hazard จะมีความถี่สัญญาณนาฬิกาสูงกว่าเครื่องที่ไม่มี Structural Hazard อยู่ 1.05 เท่า ถ้าพิจารณาเฉพาะ Structural Hazard อย่างเดียว เครื่องจะเร็วกว่ากัน

จากสูตร

$$\text{CPU Time} = \text{CPI} \times \text{Clock Cycle Time}$$

$$\text{สำหรับเครื่องที่มี Structural Hazard} = (1 + 0.4 \times 1) \times 1 / (1.05 \times \text{Clock Rate}$$

แบบอุดมคติ)

$$= 1.4 \times (\text{Clock Cycle Time แบบอุดมคติ} / 1.05)$$

$$= 1.3 \text{ Clock Cycle Time แบบอุดมคติ}$$

ดังนั้น เครื่องที่มี Structure Hazard จะเร็วกว่า 1.3 เท่า

**ตัวอย่างที่ 6.7** ถ้า 30% ของคำสั่งทั้งหมดที่ทำงานเป็นคำสั่ง Load และคำสั่งที่ต่อจากคำสั่ง Load ใช้อิโอเปอแรนต์ ที่มาจากการคำสั่ง Load ทำให้ต้อง Stall ไป 1/2 ไซเคิล ในไปปีลน์แบบที่ไม่มี Stall เลย (ซึ่งมี CPI = 1) จะทำงานเร็วกว่าไปปีลน์แบบนี้เท่าไร

$$\text{CPI ของคำสั่งที่ต่อจากคำสั่ง Load คือ } 1.5 \times (1 + 0.5)$$

$$\text{CPI ของไปปีลน์แบบนี้คือ } (0.7 \times 1) + (0.3 \times 1.5) = 1.15$$

### ตัวอย่างที่ 6.8

คำสั่ง	เวลา (ไซเกิล)											
	1	2	3	4	5	6	7	8	9	10	11	12
Branch	IFetch	Dec	EX	Mem	WB							
คำสั่งต่อจาก Branch		IFetch	Stall	Stall	EX	Mem	WB					
คำสั่งต่อจาก Branch					IFetch	Dec	EX	Mem	WB			
คำสั่งต่อจาก Branch						IFetch	Dec	EX	Mem	WB		

ถ้าในระบบมี Stall เกิดขึ้น จะใช้เวลาเพิ่มไป 3 ไซเกิล จาก CPI เดิมที่เท่ากับ 1 และถ้าคำสั่งที่ทำให้เกิด Stall มีทั้งหมด 30% จะได้ CPI ใหม่เป็น  $0.7 \times 1 + 0.3 \times 4 = 1.9$  ซึ่งมากกว่าเดิมอยู่เกือบเท่าตัว

ตัวอย่างที่ 6.9 สมมติว่าไปป์ไลน์แบบ R4000 ต้องผ่าน 3 ขั้นตอนก่อนจึงจะรู้ตำแหน่งของคำสั่งที่จะไป และจะใช้เพิ่มอีก 1 ไซเกิล จึงจะทราบว่าจะกระโดดไปหรือไม่ ซึ่งทำให้ Penalty สำหรับคำสั่ง Branch เป็นดังนี้

Branch Scheme	Penalty กรณี Uncond	Penalty กรณี Untaken	Penalty กรณี Taken
Flush ไปป์ไลน์	2.0	3	3
ทำนายว่ากระโดด (Predict Taken)	2.0	3	2
ทำนายว่าไม่กระโดด (Predict Untaken)	2.0	0	3

ในช่อง Penalty Uncond แสดงจำนวนไขเกิลที่ต้องเสียไปเมื่อมีคำสั่งกระโดดแบบไม่มีเงื่อนไขเกิดขึ้น สำหรับแต่ละวิธีที่ใช้ ได้แก่ การ Flush ไปปีลน์ การทำนายว่ากระโดด และการทำนายว่าไม่กระโดด

ในช่อง Penalty Untaken แสดงจำนวนไขเกิลที่ต้องเสียไปเมื่อมีคำสั่ง Branch แต่เป็นแบบ Untaken สำหรับแต่ละวิธีที่ใช้ ได้แก่ การ Flush ไปปีลน์ การทำนายว่ากระโดด และการทำนายว่าไม่กระโดด

ในช่อง Penalty Taken แสดงจำนวนไขเกิลที่ต้องเสียไปเมื่อมีคำสั่ง Branch แต่เป็นแบบ Taken สำหรับแต่ละวิธีที่ใช้ ได้แก่ การ Flush ไปปีลน์ การทำนายว่ากระโดด และการทำนายว่าไม่กระโดด

ให้หา CPI ถ้าสมมติให้ความถี่ของคำสั่งต่างๆ เป็นดังนี้ 4%, 6%, 10% ตามลำดับ สำหรับกรณี Uncond, Conditional Untaken, Conditional Taken ตามลำดับ

วิธีการ	จำนวนไขเกิลของ Penalty กรณี Unconditional Branch	จำนวนไขเกิลของ Penalty กรณี Branch แบบ Untaken	จำนวนไขเกิลของ Penalty กรณี Branch แบบ Taken	รวม
ความถี่แต่ละกรณี	4%	6%	10%	20%
ใช้การ Flush ไปปีลน์	0.08	0.18	0.3	0.56
ทำนายว่า Taken	0.08	0.18	0.2	0.46
ทำนายว่า untaken	0.08	0	0.3	0.38

เปรียบเทียบกับไปปีลน์อุดมคติ:

การ Flush ไปปีลน์จะให้ CPI = 1.56

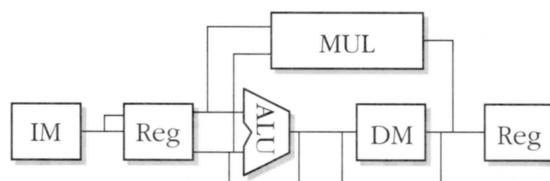
ดังนั้น ไปปีลน์อุดมคติจะเร็วกว่า 1.56 เท่า

และแบบ Predict Untaken จะเร็วกว่าแบบใช้การ Flush ไปปีลน์ =  $1 + (1.56 - 1.38)/1.38 = 1.13$  เท่า

## 6.8 ตัวอย่างไปป์ไลน์รูปแบบอื่นๆ

รูปที่ 6.41 แสดงตัวอย่างไปป์ไลน์แบบอื่นๆ ซึ่งแสดงให้เห็นถึงจำนวนหน่วยประมวลผลที่มากขึ้น เช่น มีตัวคูณ (MUL) เพิ่มเข้ามาช่วย ALU ทำงาน ดังนั้น คำสั่งที่เกี่ยวกับการคูณจะให้ตัวคูณทำได้โดย ไม่ต้องใช้ ALU และเวลาที่ใช้กับตัวคูณจะใช้ 2 ไซเคิล เพราะจากโครงสร้างจะเห็นว่าในกรณีของการใช้ตัวคูณจะไม่ต้องการทำงานในขั้นตอนการเข้าถึงหน่วยความจำข้อมูลแล้ว

ขั้นເອັກເຊີຕົວດີ + ເຂົາເລີ່ມໜ່ວຍຄວາມຈຳ ອີຣີ  
ขັ້ນເອັກເຊີຕົວດີດ້ວຍ MUL ໃຫ້ 2 ໄຊເຄີລ



รูปที่ 6.41 ไปป์ไลน์ที่มีหน่วยการคูณทำงานพร้อมกันกับ ALU

อีกด้วยในรูป 6.42 เป็นกรณีที่ใช้แบ่งการทำงานขั้นตอนที่ใช้หน่วยความจำข้อมูลเป็นสองขั้นตอน อาจจะด้วยเหตุที่ว่าการเข้าถึงหน่วยความจำข้อมูลจะซ้ำกับการเข้าถึงหน่วยความจำคำสั่งก็ได้



รูปที่ 6.42 ไปป์ไลน์ที่มีการแบ่งขั้นตอนของการเข้าถึงหน่วยความจำข้อมูลเป็นขั้นตอนย่อย

ในตัวอย่างของ ARM-7 ที่ใช้ในระบบฝังตัวมีจำนวน 3 ขั้นตอนคือ ขั้นตอนการเฟตช์ ขั้นตอนการเดшиฟ และขั้นตอนการເອັກເຊີຕົວດີ สำหรับขั้นตอนการເອັກເຊີຕົວດີจะทำงานทั้งการคำนวณของ ALU การเข้าถึงหน่วยความจำ และทำขั้นເປີຍຄ່າກລັບ

นอกจากนี้ ยังมีตัวอย่างของเป็นของ XScale ที่เป็นไปป์ไลน์มี 7 ขั้นตอน มีรายละเอียดของแต่ละขั้นตอน ขั้นตอนการเฟตช์แบ่งเป็น 2 ขั้นตอนย่อย ขั้นแรกปรับค่า PC เข้าถึง BTB และเริ่มเข้าถึงหน่วยความจำคำสั่ง ขั้นตอนการเดшиฟ ทำการเข้าถึงรېਜິສເທອຣ ขั้นต่อไปเป็นการ Shift จากนั้นเป็นขั้นตอนการເອັກເຊີຕົວດີโดยใช้ ALU และขั้นตอนเข้าถึงหน่วยความจำข้อมูลมี 2 ขั้นตอนย่อย

โดยในขั้นที่ 2 ทำการเขียนหน่วยความจำและเขียนรีจิสเตอร์

ในตัวอย่างของ MIPS R400 ประกอบด้วย 8 ขั้นตอน ได้แก่

- IF เฟตช์ครีงแรกของคำสั่ง ปรับค่าในรีจิสเตอร์ PC และเริ่มต้นของการเข้าถึงแคช
- IS อ่านครีงหลังของคำสั่ง และเสร็จสิ้นการเข้าถึงแคชที่เก็บคำสั่ง
- RF ผลตรหัสคำสั่ง และทำเฟตช์ค่าในรีจิสเตอร์ ตรวจสอบ Hazard และตรวจสอบการ Hit ของแคชคำสั่ง
- EX ทำการอีกชีคิวต์ คำนวนหา Effective Address ดำเนินการคำนวนทางคณิตศาสตร์ และคำนวนตำแหน่งที่จะกระโดดไป และคำนวน Condition Code ของการ Branch
- DF เฟตช์ข้อมูล และทำการเข้าถึงแคชข้อมูลในครีงแรก
- DS ครีงหลังของการเข้าถึงข้อมูล เฟตช์ข้อมูล และเสร็จสิ้นการเข้าถึงแคชที่เก็บข้อมูล
- TC TagCheck เพื่อตรวจดูว่า Hit หรือ Miss ในแคช
- WB เป็นการเขียนค่ากลับ สำหรับคำสั่ง Load และคำสั่งที่ไม้อเปอแรนด์ แบบรีจิสเตอร์-รีจิสเตอร์

นอกจากนี้ยังมีตัวอย่างของ Intel Itanium 2 ซึ่งมีทั้งหมด 8 ขั้นตอน (ที่มา: <http://www.intel.com>) IA-32 เป็นสถาปัตยกรรมที่ประกอบด้วยไปปีลอน์ 8 ขั้นตอน สามารถทำงานได้ 6 คำสั่งแบบบานานต่อ 1 ไซเกิล ส่วน IPG/ROT เป็นส่วนของ Front-end มีหน้าที่ทำการ Prefetch คำสั่ง เป็นการอ่านคำสั่งเข้ามาล่วงหน้า และหมาย Branch ส่วน EXP, REN, REG จะทำการขยายอเปอแรนด์ แปลงชื่อรีจิสเตอร์ และเข้าถึงรีจิสเตอร์ ส่วนขั้นตอน EXE เป็นการอีกชีคิวต์ ขั้น DET จะดูแลเรื่องการจัดการ Exception และ WRB คือขั้นของการเขียนค่ากลับ

## ■ 6.9 สรุป

ในการทำงานของคำสั่งหนึ่งๆ อาจจะใช้หลายไซเกิลในการทำงาน หากต้องการทำให้การทำงานเร็วขึ้นต้องมีการจัดโครงสร้างทางฮาร์ดแวร์ใหม่ให้มีการใช้ทรัพยากร่วมกันให้มีประสิทธิภาพสูงสุด ในบทนี้ได้กล่าวถึงเทคนิคของไปปีลอน์ซึ่งเป็นการทำงานแบบบานานในระดับคำสั่ง ที่เป็นการ

นำเอาหน่วยฯ คำสั่งมาทำงานทับซ้อนกันในช่วงเวลาหนึ่ง และคำสั่งเหล่านี้จะอยู่ในขั้นตอนที่ต่างกัน มีการปรับปรุง Data Path จากแบบหลายขาเกลิลที่มีอยู่เดิม และเพิ่มบัพเฟอร์เก็บการทำงานแต่ขั้นของแต่ละคำสั่ง แต่อย่างไรก็ได้ การใช้ไปป์ไลน์จะไม่ได้ประสิทธิภาพสูงสุด เพราะมี Hazard ของไปป์ไลน์อยู่ Hazard ของไปป์ไลน์มี 3 แบบ จึงต้องมีการปรับโครงสร้างทาง Data Path ของไปป์ไลน์เพื่อการแก้ปัญหา Hazard เหล่านี้ เป็นการเพิ่มความซับซ้อนของฮาร์ดแวร์ นอกจากนี้ ประเด็นสำคัญอีกอย่าง ได้แก่ การจัดการ Exception ในไปป์ไลน์เพื่อให้การทำงานโดยรวมถูกต้อง ในบทนี้ยังได้กล่าวถึงตัวอย่างการคำนวณประสิทธิภาพของไปป์ไลน์ และแสดงตัวอย่างการไปป์ไลน์รูปแบบต่างๆ ด้วย

## คำความท้ายบท

1. การทำงานแบบไปป์ไลน์ต่างจากแบบหลาຍไชเกิลอย่างไร
2. ไปป์ไลน์อุดมคติครั้งมี CPI เท่าไร
3. โครงสร้าง Data Path ของ MIPS เมื่อเป็นการทำงานแบบไปป์ไลน์จะต่างจากแบบหลาຍไชเกิลอย่างไร
4. Hazard ของไปป์ไลน์มีอะไรบ้าง จงอธิบาย
5. จากโคลด์ด้านล่าง จงระบุประเภทของ Data Hazard ที่อาจจะเกิดได้
  - LD R2, 1000 (R1)
  - DADD R3, R2, #10
  - DSUB R3, R1, #4
  - DSUB R4, R3, #8
  - SW R4, 2000 (R1)

พิจารณา MIPS ไปป์ไลน์ และพิจารณาคำสั่งต่อไปนี้

```

LW   R1, 300 (R2)
LW   R3, 400 (R4)
ADD R2, R1, R3
SUB R1, R4, #4
SW   500 (R1), R2
  
```

- 5.1 จงระบุ Hazard ที่อาจจะเกิดขึ้นได้ทั้งหมด และระบุประเภทของ Hazard
- 5.2 ในไขเกิลที่ 4 คำสั่งใดกำลังทำงานอยู่บ้าง และอยู่ในขั้นตอนอะไร
- 5.3 การทำงานทั้งหมดนี้จะมีไขเกิล และมี Stall ทั้งหมดเท่าไร
- 5.4 ถ้ากำหนดให้เครื่องนี้ทำงานที่ความถี่ 1 GHz โปรแกรมนี้จะใช้เวลาเท่าไร

5.5 สมมติให้มีการปรับปรุง Data Path โดยกำหนดให้มีการ Forward ได้เต็มรูปแบบ โปรแกรมดังกล่าวจะใช้เวลาทำงานกี่ขาเกล และจะใช้เวลาเท่าไร

## 6. พิจารณาโค้ดดังข้างล่าง

```

Loop : LD R1,100(R2)
        LD R3,200(R2)
        DADD R4,R3, R1
        SD R4,300(R2)
        DSUB R2,R2,#4
        BNEZ R2,Loop
    
```

กำหนดให้ค่าเริ่มต้นของ R2 เท่ากับ 40

พิจารณาไปป์ไลน์ MIPS ที่มี 5 ขั้นตอน และการเปลี่ยนแปลงรูปแบบคำสั่งดังนี้

- เพิ่มรูปแบบการบวกระหว่างໂອເປອແຮນດังนี้
- DADDReg3, Mem [Reg1], Reg2; Reg3 = Mem[Reg1] + Reg2

6.1 จงอธิบายถึงการเปลี่ยนแปลงรูปแบบดังกล่าวกับตัวอย่างโค้ดข้างต้น โค้ดข้างต้นจะเปลี่ยนเป็นอย่างไร กรณีได้ใช้รูปแบบดังกล่าวได้ และกรณีได้ไม่สามารถใช้ได้

6.2 จงอธิบายถึงการเปลี่ยนแปลง Data Path ของ MIPS เมื่อทำการเปลี่ยนแปลงดังกล่าว และอธิบายถึงการเปลี่ยนแปลงรูปแบบคำสั่งที่อาจจะเกิดขึ้น

6.3 สมมติให้การเปลี่ยนแปลงโครงสร้าง Data Path ดังกล่าวทำให้เวลาของรอบสัญญาณนาฬิกาของไปป์ไลน์มากขึ้น 10% ในกรณีของโค้ดข้างต้น จะมี Speedup หรือไม่ ถ้ากำหนดให้ CPI ของคำสั่งทั่วไปเท่ากับ 1 ทุกคำสั่ง ยกเว้นคำสั่ง Load มี CPI เท่ากับ 2 และรูปแบบคำสั่งใหม่นี้มี CPI เท่ากับ 3 จงอธิบายพร้อมแสดงการคำนวณ

## 7. พิจารณาตัวอย่างโปรแกรมต่อไปนี้

```

Loop : LD R1,100(R2); R1 = mem[100+R2]
        DADDI R1,R1,#1; R1 = R1+1
        SD 100(R2),R1;
        DADDI R2,R2,#4
        DSUB R4,R3,R2
        BNEZ R4,Loop
    
```

สมมติค่าเริ่มต้นของ R3 เป็น R2 + 396 พิจารณาไปปีลอน MIPS แบบ 5 ขั้นตอน

7.1 จงระบุ Data Hazard ที่เกิดขึ้นทั้งหมด

7.2 หากกำหนดให้มีการใช้พอร์ตการอ่านของหน่วยความจำเพียง 1 พอร์ต จงระบุคำสั่งในโค้ดที่อาจจะก่อให้เกิด Structure Hazard

7.3 โค้ดดังกล่าวใช้เวลาทั้งหมดกี่ไซเกิล เมื่อคิด Stall ที่เกิดจาก Hazard แล้ว (กำหนดให้ไม่มีการใช้ Forwarding ใดๆ เลย และสมมติเมื่อ Structure Hazard ได้) และจะได้ CPI เท่าไร

7.4 หากมีการใช้ Fully Forwarding จะใช้เวลาการทำงานโดยเด็ดขาดทั้งหมดเท่ากับกี่ไซเกิล และคำนวณ CPI ที่ได้ (ไม่ต้องคิด Structure Hazard)

7.5 จากข้อที่ 7.3 สมมติเครื่องดังกล่าวมีความถี่เท่ากับ 100 เมกะเฮิรตซ์ จะใช้เวลาทำงานเท่าไร และจะมี Speedup เท่าไร เมื่อเทียบกับไปปีลอนในอุดมคติที่ไม่มี Hazard ใดๆ เลย

## 8. พิจารณา Benchmark ชุดหนึ่งที่มีคำสั่งต่างๆ ดังแสดงในตารางด้านล่าง

ประเภทคำสั่ง	จำนวน
คำสั่งเกี่ยวกับตัวดำเนินการทาง ALU	40%
คำสั่ง Branch แบบมีเงื่อนไขและไม่มีเงื่อนไข	20%
คำสั่ง Load/Store	20%
อื่นๆ	10%

สมมติให้จากการทดลองพบว่าคำสั่ง Branch ที่เป็นแบบ Taken Branch มีจำนวน 35% ของคำสั่ง Branch ที่มีอยู่ทั้งหมด และแบบ Untaken Branch มีแบบ 50% ของจำนวนคำสั่ง Branch ทั้งหมด ที่เหลือเป็น Unconditional Branch พิจารณาตัวอย่าง MIPS ที่ประกอบด้วย 5 ขั้นตอน กำหนดให้การตรวจสอบเงื่อนไขและการคำนวณแสดงเดรஸปลายทางทำงานอยู่ในขั้นตอนการอ่านชีคิวต์ จงคำนวณหา Speedup เมื่อเทียบกับไปป์ไลน์แบบอุดมคติในกรณีต่างๆ ดังนี้

8.1 เมื่อใช้การทำนายแบบ Taken เข้าช่วย

8.2 เมื่อใช้การทำนายแบบ Untaken

8.3 เมื่อใช้ Delay Slot โดยที่มีจำนวน 60% ของ Delay Slot ที่ถูกเติมด้วยคำสั่งที่มีประโยชน์ กับการทำงานของโค้ดโปรแกรม

9. พิจารณาโค้ดต่อไปนี้ ในภาษาชั้นสูง

```
for i := 1 to 10 do
    A = B + C
    D = A - E
```

9.1 จงแปลงโค้ดดังกล่าวเป็น MIPS กำหนดให้หน่วยความจำเก็บตัวแปรที่ตำแหน่งเริ่มต้น 1,000 ตัวแปร แต่ละตัวมีขนาด 2 ไบต์ เป็นจำนวนเต็ม สำหรับสถาปัตยกรรม MIPS โดยที่ 1 Word = 32 บิต โค้ดนี้ใช้กีฬากิลในการทำงาน

9.2 โค้ดดังกล่าวมี Data Hazard เกิดขึ้นที่ใช้กิลได้บ้าง ถ้าใช้ Fully Forwarding จะใช้เวลาทั้งหมดเท่าไร

9.3 โค้ดดังกล่าวมี Control Hazard เกิดขึ้นที่ใช้กิลได้บ้าง ถ้าต้องปรับโค้ดโดยใช้ Delay Slot จะปรับได้อย่างไร

10. พิจารณาการทำนาย Branch แบบใช้ 2 บิต และเริ่มต้นการทำนายว่ากระโดด โค้ดในข้อที่ 9 จะทำนายผิดกี่ครั้ง เทียบกับการใช้ 1 บิต และให้เปรียบเทียบการเขียนโค้ดแบบ Top Loop เทียบกับ Bottom Loop เทียบจำนวนครั้งที่ทำผิด สมมติให้ใช้โครงสร้างไปป์ไลน์แบบ MIPS โดยผลการการทดสอบเงื่อนไขการกระโดดทราบในขั้นต่อไป และพิจารณาถ้าใช้การทำนาย Branch

### 11. พิจารณาโปรแกรมประกอบด้วยคำสั่งต่อไปนี้

ประเภทคำสั่ง	ความถี่
คำสั่ง Load/Store	30%
คำสั่ง ALU	40%
คำสั่ง Branch แบบมีเงื่อนไขแบ่งเป็น Taken Branch (15%) Untaken Branch (10%)	25%
คำสั่ง JUMP	5%

อย่างทราบว่าโปรแกรมดังกล่าวจะใช้จำนวนไบเกิลในการทำงานทั้งหมดเท่าไรทั้งสองแบบ

11.1 ใช้การคำนายแบบ Untaken Branch

11.2 ใช้การคำนายแบบ Taken Branch

