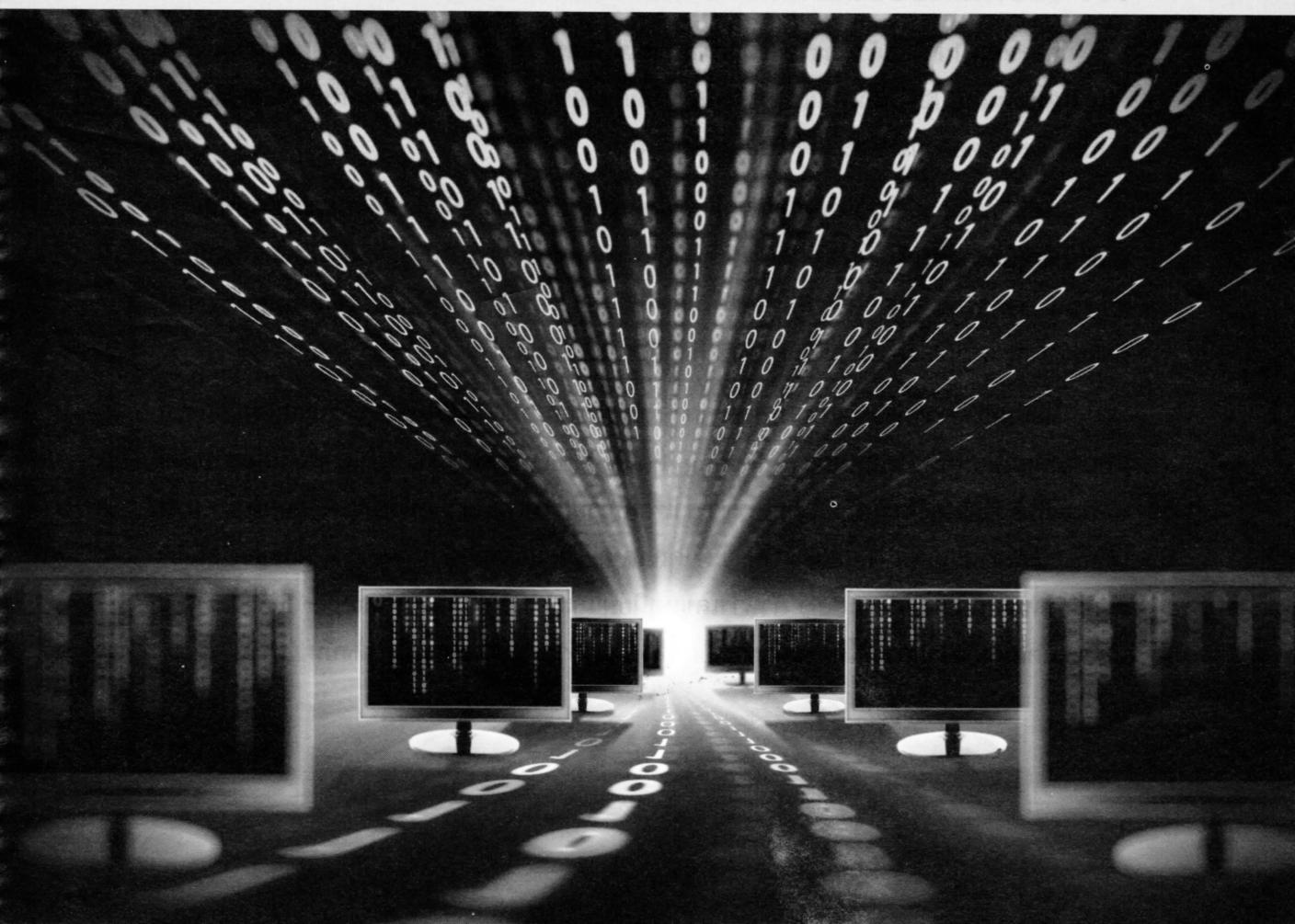


7

เทคโนโลยีสำหรับไปoline ขั้นสูง



การทำงานแบบไปปีลайнนั้นมีอุปสรรคที่สำคัญ ได้แก่ Data Hazard และ Control Hazard ซึ่งเป็นลิสท์ที่หลีกเลี่ยงไม่ได้ ในบทที่แล้วได้กล่าวถึงวิธีการต่างๆ เพื่อลด Stall ที่เกิดจาก Hazard เหล่านั้น ซึ่งเป็นวิธีการทางซอฟต์แวร์หรือการใช้ตัวแปลงภาษาช่วยในการจัดแต่งโค้ด และการเพิ่มหน่วย Forward เพื่อให้ส่งค่าไปยังโอดรันด์ของ ALU ได้เร็วขึ้น แต่ปัญหา Hazard ดังกล่าว ยังคงอยู่ และถ้าพิจารณาไปปีลайнรูปแบบอื่นๆ ปัญหาดังกล่าวอาจจะทำให้เกิด Stall เป็นจำนวนมากกว่าในกรณีของเครื่องแบบ MIPS ก็เป็นได้

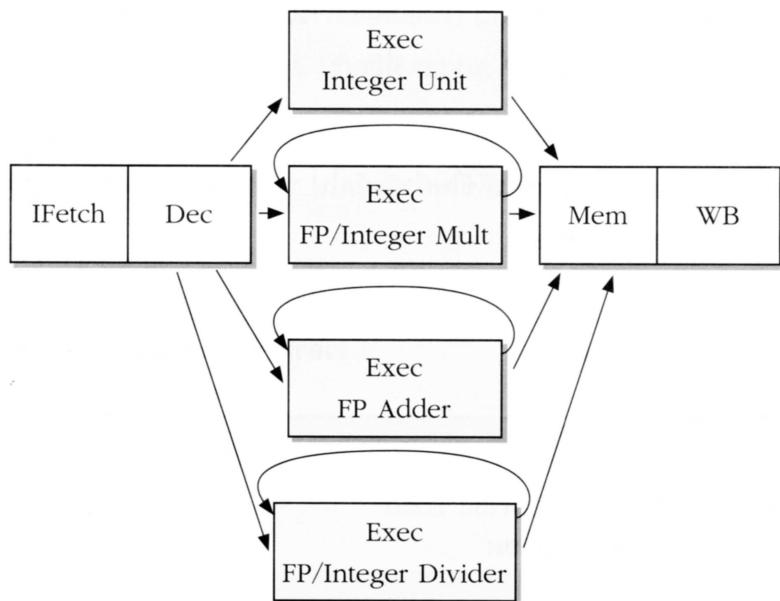
ในบทนี้จะกล่าวถึงลักษณะของไปปีลайнรูปแบบอื่นๆ ที่จะเป็นไปได้ที่รองรับการทำงานแบบขนาดได้มากขึ้น วิธีการจัดลำดับที่ช่วยในการทำงานให้ลดการเกิด Stall ทั้งแบบ Static และแบบ Dynamic รวมไปถึงการทำนายการกระโดดเพื่อลดการเกิด Stall จากคำสั่งแบบ Branch ด้วย

7.1 ไปปีลайнที่รองรับการทำงานหลายหน่วย

ในความเป็นจริงแล้ว การประมวลผลภายในคอมพิวเตอร์ให้หน่วยประมวลผลหลายหน่วย เช่น ในเทคโนโลยีปัจจุบันที่เป็นแบบ 2-core เป็นต้น แท้จริงแล้วในแต่ละ core ยังมีการทำงานแบบไปปีลайн โดยอาจจะมีหลายๆ ไปปีลайнอีกด้วย

ในที่นี้จะกล่าวถึงไปปีลайнที่มีหน่วยคำนวณหลายหน่วยก่อน เนื่องจากที่กล่าวมาในบทที่แล้ว ได้อธิบายต้นแบบที่มีเพียง ALU หน่วยเดียวที่ทำหน้าที่คำนวณ แต่เพื่อเพิ่มประสิทธิภาพในการคำนวณ เรายังเพิ่มหน่วยคำนวณมาช่วยในกรณีที่ ALU อาจจะต้องใช้หลายไฟล์ในการทำงาน ตัวอย่างเช่น กรณีของการคำนวณคำสั่งที่เกี่ยวกับเลขทศนิยม เป็นต้น

ในตัวอย่างการทำงานแบบไปปีลайн เราอาจจะแยกการคำนวณแบบเลขจำนวนเต็มออกจากแบบทศนิยมโดยใช้หน่วยทศนิยม (Floating Point Unit) มาช่วยทำงานในคำสั่งที่เกี่ยวกับเลขทศนิยมก็ได้ ดังแสดงในรูปที่ 7.1 จะเห็นว่าเราเพิ่มหน่วยคำนวณหลายหน่วย ซึ่งมีทั้งหมด 4 หน่วย ได้แก่ หน่วยจำนวนเต็ม (Integer Unit) ตัวคูณเลขทศนิยมและจำนวนเต็ม (FP/Integer Mult) ตัวบวกเลขทศนิยม (FP Adder) ตัวหารเลขทศนิยมและจำนวนเต็ม (FP/Integer Divider) ตั้งนั้นในขั้นตอนหลังจากการตีโค้ดจะทำการอึกซีคิวต์ในหน่วยที่เหมาะสมกับคำสั่งนั้น



รูปที่ 7.1 ตัวอย่างไปป์ไลน์แบบหลายหน่วย

ในรูปที่ 7.1 แสดงตัวอย่างไปป์ไลน์ที่ประกอบด้วยหลายหน่วยคำนวณ บางหน่วยคำนวณ จะใช้หลายไบเกิลในการทำงานได้ ได้แก่ FP/ Integer Mult, FP Adder, FP/Integer Divider ดังแสดงโดยลูกภาษาในไบเกิลของหน่วย FP ต่างๆ เมื่อกำหนดให้การทำงานของแต่ละหน่วย FP มากกว่า 1 ไบเกิล ความเป็นไปได้ของการเกิด Stall ระหว่างคำสั่ง FP ในหน่วยต่างๆ จะมีมากขึ้น แต่ละกรณีจะมีจำนวน Stall ไม่เท่ากันอีกด้วย จากในรูปที่ 7.1 ประกอบด้วย

1. Integer Unit สำหรับคำสั่งแบบ Load/Store และคำสั่งแบบ ALU/Branch
2. ตัวคูณสำหรับเทคนิคเต็ม และตัวคูณสำหรับจำนวนเต็ม (FP/Integer Mult)
3. ตัวบวกสำหรับเทคนิคบวก-ลบเลข techniques และการแปลงเลข
4. ตัวหารสำหรับเทคนิคและจำนวนเต็ม (FP และ Integer Divider)

สำหรับการทำงานแบบหลายหน่วยนี้ ต้องพิจารณาปัจจัยต่อไปนี้

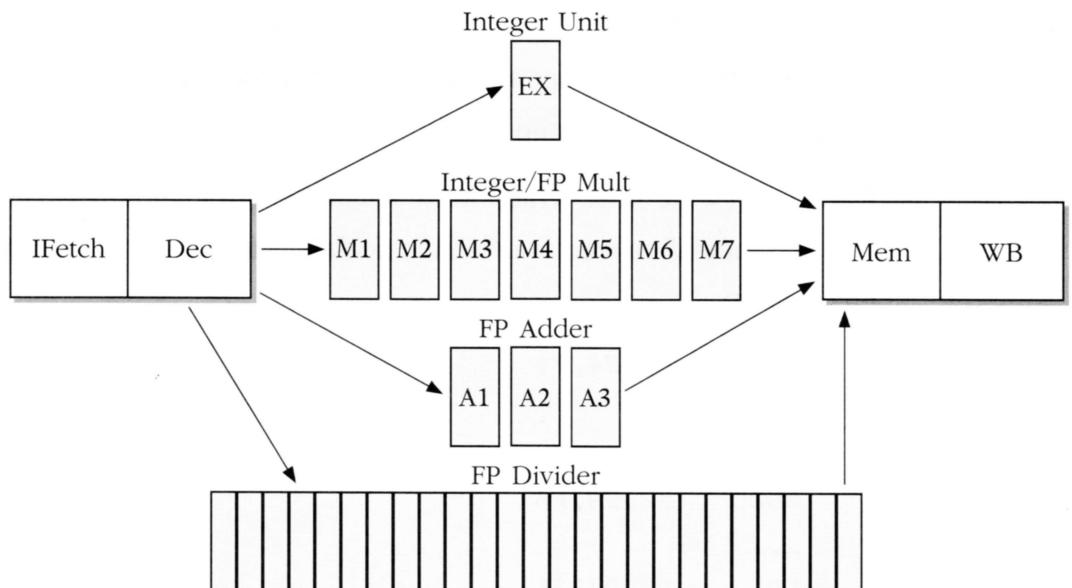
Initiation Interval คือจำนวนไบเกิลที่ต้องการระหว่างการส่งคำสั่ง (Issue) 2 คำสั่งที่เป็นประเภทเดียวกัน ว่าจะต้องห่างกันกี่ไบเกิล ทั้งนี้เพื่อไม่ให้เกิด Structure Hazard เช่น ระหว่างการ

Issue 2 คำสั่งในการทำ FP Add (ในขั้นตอนการอีกซีคิวต์) Initial Interval หมายถึง เวลาขั้นต่ำที่ต้องการในการใช้หน่วย FP Adder ที่ติดกัน 2 ครั้ง

Latency คือจำนวนไบเกิลที่ต้องการสำหรับหน่วยคำนวนนั้นๆ ในการประมวลผลเพื่อให้ได้ผลลัพธ์ และการจะนำผลลัพธ์ไปใช้ในคำสั่งถัดไป ว่าคำสั่งถัดไปนี้จะต้องอยู่ห่างอย่างน้อยกี่ไบเกิล ดังตัวอย่างในตารางด้านล่าง

หน่วย	เวลาที่ใช้ (Latency) (ไบเกิล)	Initiation Interval (ไบเกิล)
Integer Unit	0	1
Memory Unit สำหรับคำสั่ง Load เลขจำนวนเต็มและศูนย์	1	1
FP Adder	3	1
FP/Integer/Mult	6	1
FP Divider	24	25

สำหรับ Integer Unit ใช้เวลาในการคำนวน (Latency) เท่ากับ 0 เพราะว่าคำสั่งที่ติดกันถัดไปสามารถใช้ผลลัพธ์ได้ทันที (เมื่อมีการ Forward ดังที่ได้กล่าวถึงในบทที่แล้ว) ส่วน Load ใช้ Latency เท่ากับ 1 เพราะว่าผลของการ Load จะใช้ได้กับคำสั่งที่ห่างออกไป 1 คำสั่ง หรือต้องการ 1 Stall นั่นเอง ในโครงสร้างนั้น ตัวดำเนินการ (Operation) ส่วนใหญ่ใช้อะเปอแรนต์ในขั้นตอนการอีกซีคิวต์ ดังนั้น จะทำให้ใช้เวลาในการคำนวนเท่ากับจำนวนขั้นที่มีห้องหมวดหลังจากขั้นตอนการอีกซีคิวต์นั่นเอง สำหรับคำสั่ง Store ค่าจะถูกใช้ได้หลังจาก 1 ไบเกิลถัดมา ถ้า FP Adder มีขั้นตอนการอีกซีคิวต์ใช้เวลาห้องหมวด 4 ไบเกิล ดังนั้น มี Latency เป็น 3 ไบเกิล FP Mult มี 7 ขั้นตอน ดังนั้น มี Latency = 6 ไบเกิล สำหรับในตัวอย่างนี้ FP Divider มี 1 ขั้นตอน แต่ใช้ 24 ไบเกิล



รูปที่ 7.2 ตัวอย่างขั้นต่างๆ แบบหลายไชเกิล

ในรูปที่ 7.2 แสดงหน่วยคำนวนหลายๆ หน่วย แต่ละหน่วยแบ่งเป็นขั้นๆ จะเห็นว่าเมื่อไปป์ไลน์เปลี่ยนไป จะต้องเพิ่มรีจิสเตอร์ชั่วคราวระหว่างขั้นตอน เช่น A1/A2, A2/A3, A3/A4 สำหรับ FP Adder และในทำนองเดียวกัน สำหรับ FP Mult นอกจากนี้ ต้องมีการเชื่อมรีจิสเตอร์ ID/EX ในขั้นตอนการตีโค้ดกับขั้นตอนการอักษีคิวต์ EX และขั้นตอนการ DIV, M1, A1 ด้วย ในขั้นตอนที่เปลี่ยนกรอบสี่เหลี่ยมเป็นขั้นตอนที่ได้ผลลัพธ์การทำงานของคำสั่งนั้นๆ

MULTD	Fetch	Dec	M1	M2	M3	M4	M5	M6	M7	Mem	WB
ADDD		Fetch	Dec	A1	A2	A3	A4	Mem	WB		
LD		Fetch	Dec	EX	Mem	WB					
SD		Fetch	Dec	EX	Mem	WB					

รูปที่ 7.3 ตัวอย่าง Sequence การทำงาน

ในรูปที่ 7.3 แสดงไชเกิลที่ได้ผลลัพธ์ในแต่ละคำสั่ง เมื่อใช้สถาปัตยกรรมแบบนี้ จะเห็นว่าถ้าคำสั่ง SD ต้องการใช้ผลลัพธ์ของคำสั่งก่อนหน้าในขั้นตอนการตีโค้ด จะต้องมีการใส่ Stall เข้าไปจำนวนมาก ดังนั้น สำหรับไปป์ไลน์ที่ลีกชิ้นแบบนี้ต้องพิจารณา Hazard และการ Forward เป็นกรณีๆ ไป เช่น

1. หน่วย Divider ไม่ใช้รูปแบบไปปีลайнทำให้มีโอกาสเกิด Structural Hazard
2. มีหลายคำสั่งที่ทำงานอยู่ในเวลาหนึ่ง และมีความยาวคำสั่งไม่เท่ากัน มีโอกาสเกิดการเชื่อมากกว่า 1 ครั้งใน 1 ไซเคิล
3. มีโอกาสเกิด (Write-After-Write) WAW คำสั่งที่เกิดก่อนเขียนทับผลของคำสั่งที่เกิดหลัง เพราะคำสั่งที่เกิดอาจจะก่อนเสร็จที่หลัง ดังโค้ดซึ่งเขียนไปยัง F2 ทั้งคำสั่ง ADDD และคำสั่ง LD

```
MULTD    F0, F4, F6
...
...
ADD     F2, F4, F6
...
LD      F2, 0 (R2)
```

4. คำสั่งอาจจะเสร็จในลำดับที่แตกต่างจากลำดับที่ส่งคำสั่นหรือ Issue คำสั่งออกไป
5. เนื่องจากไปปีลайнมีหลายขั้น การเกิด Stall จาก RAW จะมีโอกาสมาก และจำนวน Stall จะเพิ่มขึ้นได้ ดังโค้ดด้านล่าง คำสั่ง ADDD จะเขียนไปยัง F2 แต่การเขียนเกิดขึ้นช้ากว่า เพราการทำคำสั่ง ADDD ใช้ Latency = 3 ไซเคิล แต่คำสั่ง Store ทำงานเสร็จลินก่อน

```
LD      F4, 0 (R2)
MULTD  F0, F4, F6
ADDD   F2, F0, F8
SD      0 (R2), F2
```

ในโค้ดยังต้องแก้ไขโดยการเลื่อน (Delay) การ Issue คำสั่ง Load ไปจนกว่าคำสั่ง ADDD จะเข้าสู่ขั้นการเข้าถึงหน่วยความจำ และจัดเก็บผลลัพธ์ของคำสั่ง ADDD ไว้ชั่วคราวก่อน แต่ยังไม่เขียนไปจริง และดูว่าถ้ามี Hazard เกิดขึ้น จะไม่ให้คำสั่ง ADDD ทำการเขียนคำตอบไป

สำหรับการแก้ปัญหา WAW จะต้องติดตามการใช้พอร์ตสำหรับการเขียน ณ ขั้นตอนการติดด้วยเมื่อผ่านขั้นตอนครั้งแรกแล้วและจะยับยั้งการทำงานไว้ และถ้าคำสั่งนั้นเขียนไปยังรีจิสเตอร์ที่เดียวกับคำสั่งที่กำลังทำงานอยู่ โดยจะให้ Stall ไป 1 ไซเคิล อาจจะพิจารณา Stall เมื่อคำสั่งนั้นกำลังจะเข้าสู่ขั้นการเข้าถึงหน่วยความจำหรือขั้นการเขียนค่ากลับ และเลือกเอาที่จะ Stall คำสั่งใดคำสั่งหนึ่งก็ได้

ในทั้ง 2 กรณี Hazard จะถูกตรวจสอบเมื่อเข้าสู่ขั้นตอนการที่ได้ด้วยว่าต้องมีการ Stall หรือไม่ คำสั่ง LD อาจจะใช้การใส่ Nop ไป สำหรับในตัวอย่างนั้น การตรวจสอบว่าคำสั่ง LD จะทำงานเสร็จก่อนคำสั่ง ADDD นั้นยาก เพราะว่าต้องรู้ว่าไปป์ไลน์นั้นลีกเท่าไร และรู้ว่าคำสั่ง ADDD ทำงานอยู่ขั้นใดแล้ว

สรุปได้โดยรวมว่าจะต้องเพิ่มการตรวจสอบดังนี้

1. ตรวจสอบ Structural Hazard โดยตรวจสอบว่าหน่วยคำนวณที่ต้องการว่างอยู่หรือไม่ ถ้าว่าง ก็จะ Issue คำสั่นนี้ได้
2. ตรวจสอบ RAW Hazard โดยดูว่าผลลัพธ์ที่ได้สามารถส่งไปยังคำสั่งถัดไปได้ก่อนหรือ มีการ Forward ผลลัพธ์ได้หรือไม่
3. ตรวจสอบ WAW Hazard โดยดูว่าคำสั่ง A1, ..., A4, D, M1, ..., M7 มีรีจิสเตอร์เก็บ ผลลัพธ์ตรงกันหรือไม่ ถ้าใช่ ต้อง Stall การ Issue คำสั่งที่เข้ามาที่หลังซึ่งอยู่ในขั้นเดียวกัน ในตัวอย่างไปป์ไลน์แบบนี้จะมีโอกาสเกิด Out-Of-Order-Completion กล่าวคือ

DIVF	F0, F2, F4
ADDF	F10, F10, F8
SUBF	F12, F12, F14

คำสั่ง SUBF จะทำงานเสร็จก่อนคำสั่ง DIVF เพราะว่าใช้จำนวนไซเกลน้อยกว่า แต่ถ้า คำสั่ง DIVF เกิด Exception ขึ้น คำสั่ง ADDF ซึ่งเสร็จก่อนจะเขียนผลลัพธ์ไปเรียบร้อยแล้ว และ จะไม่สามารถถูกค่ากลับคืนดังเดิมได้ ต้องแก้ไขโดยการเอาบัฟเฟอร์มาเก็บผลลัพธ์ของคำสั่ง ADDF ไว้ก่อน และรอให้คำสั่ง DIVF เสร็จจึงต่อยทำการเขียนจากบัฟเฟอร์ไปยังรีจิสเตอร์ นอกเหนือนี้แล้ว อาจจะทำการเก็บข้อมูลเกี่ยวกับคำสั่งต่างๆ และค่าในรีจิสเตอร์ PC เพื่อว่าเมื่อจัดการ Exception แล้วจะสามารถกลับมาทำงานชุดคำสั่งต่อได้เลย

สำหรับไปป์ไลน์ที่ลีกขั้นนั้นจำนวน Stall ระหว่างคำสั่ง LD และคำสั่งที่ใช้ผลของการโหลด เมื่อมีการ Forward แล้วนั้นอาจจะมากกว่า 1 ไซเกล ขึ้นอยู่กับขั้นตอนที่มีอยู่

ในการนี้ของ MIPS R4000 จะต้องใช้ 2 Stall ส่วนคำสั่ง Branch ต้องการ Delay Slot 1 ไซเกล และในกรณีการกระโดดจริง จะใช้ไซเกลในการ Stall 2 ไซเกลจริง ดังนั้น จะต้อง เลื่อนคำสั่ง Branch ไปทั้งหมด 3 ไซเกล เพราะเงื่อนไขการ Branch ถูกประเมินในขั้นตอนการ

ເລື້ອກທີ່ຄົວຕົວ ຄ້າເປັນກຣນີທີ່ໃຊ້ການທໍານາຍວ່າມີກາຮະໂດດ (Predict-Taken) ຈະທ້ອງການມີ Stall 2 ໄຂເກີລ ແລະ ໃນກຣນີຄ້າໄມ້ມີກາຮະໂດດເກີດເກີດເກີດ ຈະໃຊ້ການເລື່ອນຄຳສັ່ງ Branch ອອກໄປເພີ່ມ 1 ໄຂເກີລ ສໍາຫຼັບກຣນີຂອງໜ່ວຍທຄນີຍມຈະໃຊ້ລັກຜະນະດັ່ງນີ້

ແຕ່ລະຄຳສັ່ງແບບທຄນີຍມໃໝ່ເວລາ 2 – 112 ໄຂເກີລ ແລະ ແປ່ງເປັນບັນຕ່າງໆ ດັ່ງນີ້

A FP Adder: ຂັ້ນການບວກ Mantissa

D FP Divider: ຂັ້ນການหาร

E FP Mult: ຂັ້ນການທດສອບ Exception

M FP Mult: ຂັ້ນການຄູນຄົງແຮກ

N FP Mult: ຂັ້ນການຄູນຄົງໜັງ

R FP Add: ຂັ້ນການປັດເສຍ

S FP Add: ຂັ້ນການເລື່ອນຈຸດທຄນີຍມ

U: ຂັ້ນການ Unpack ຕັວເລີບທຄນີຍມ

ໂດຍແຕ່ລະຄຳສັ່ງແບບທຄນີຍມໃໝ່ຈຳນວນບັນຕ່າງໆ ກັນມີ Latency ຕ່າງກັນ ແລະ Initiation Interval ຕ່າງກັນ ດັ່ງແສດງໃນຮູບທີ່ 7.4 ຕາມຕ້ວຍ່ອດ້ານນັນ

ຄຳສັ່ງເກີຍກັນທຄນີຍມ	Latency	Initial Interval	ຂັ້ນຕອນທີ່ໃຊ້
ບວກລົບ	4	3	U, S + A, A + R, R + S
ຄູນ	8	4	U, E + M, M, M, M, N, N + A, R
หาร	36	35	U, A, R, D ²⁷ , D + A, D + R, D + A, D + R, A, R
ຄອດຮາກ	112	111	U, E, (A + R) ¹⁰⁸ , A, R
Negate	2	1	U, S + A, A + R, R + S
ຫາຄ່າສົມບຽບ	2	1	U, S + A, A + R, R + S
ເບຣີຍນເທືຍນ	4	2	U, A, R

ຮູບທີ່ 7.4 ການໃຊ້ໜ່ວຍທຄນີຍມໃນ MIPS R4000

ในรูปที่ 7.4 ซึ่งจะทำให้เกิด Stall ลักษณะดังนี้

1. Stall จากคำสั่ง Load เกิดจากการใช้ค่าที่คำสั่ง Load อ่านเข้ามาภายใน 1 – 2 ไซเกล ถัดมา
 2. Stall จากคำสั่ง Branch เป็น 2 ไซเกล และการหาคำสั่งมาทำงานระหว่างการเลือกการกระโดดออกไป
 3. Stall จากการใช้ผลจากคำสั่งที่ใช้หน่วยทศนิยม เป็น Stall แบบ RAW Hazard ในโอบอแ阮เดิร์บองหน่วยทศนิยม
 4. Stall จากการที่สองคำสั่งใช้หน่วยทศนิยมเดียวกัน เป็น Structural Hazard เพราะเกิดจากการใช้หน่วยคำนวณทศนิยมที่มีจำกัด

7.2 ความสัมพันธ์ของข้อมูลรูปแบบต่างๆ ที่มีผลต่อไปปีไลน์ และการจัดลำดับ

ในแนวคิดของไปปีลайнจัดได้ว่าเป็นลักษณะของการทำงานแบบบนระดับคำสั่ง (Instruction-Level Parallelism (ILP)) ซึ่งหมายถึงว่า ใน เวลาหนึ่งๆ จะมีคำสั่งหลายคำสั่งทำงานอยู่ ดังได้กล่าวถึงไปแล้วในบทที่ 6 จะเห็นว่าปัญหาของไปปีลайнที่เกิดจาก Hazard รูปแบบต่างๆ ซึ่ง มาจากการออกแบบ Data Path และเมื่อมีหน่วยคำนวนทำงานพร้อมกันหลายหน่วย ก็เป็นไปได้ที่ จะมีความต้องการข้อมูลของหน่วยต่างๆ เหล่านั้นเกิดขึ้นอย่างพร้อมกัน แต่ไม่สามารถตอบสนองได้ ดังนั้น การทำงานของคำสั่งบางคำสั่งอาจจะต้องรอ แม้ว่าจะมีหน่วยคำนวนที่ว่างอยู่ก็ตาม

โดยปกติแล้ว คำสั่งในโปรแกรมจะต้องถูกทำงานตามลำดับเพื่อให้ได้ผลลัพธ์ที่ถูกต้อง การจัดลำดับการทำงานของคำสั่งเหล่านี้เรียกว่า Scheduling ซึ่งเป็นหน้าที่ของตัวแปลภาษาที่ต้องสร้างลำดับของคำสั่งที่จะถูกป้อนเข้าไปเพื่อทำการเฟดท์ ลำดับที่ได้จากตัวแปลภาษานี้เรียกว่า ลำดับแบบ Static (Static Schedule) เนื่องจากลำดับถูกสร้างไว้ก่อนจะมีการทำงานจริง นอกจากนี้ ยังมีการจัดลำดับแบบ Dynamic (Dynamic Schedule) ซึ่งจะตัดสินลำดับคำสั่งระหว่างการทำงาน ซึ่งอาจจะปรับแก้ลำดับที่ได้จากตัวแปลภาษาระหว่างการทำงานภายใต้พิธีเพื่อลด Stall ที่ไม่สามารถกำจัดได้จาก Static Schedule ในการลด Stall ด้วยตัวแปลภาษาที่จัดลำดับคำสั่งนี้ เรียกว่า เป็นวิธีการทางซอฟต์แวร์ และวิธีการแบบ Dynamic ต้องอาศัย hardware เข้าช่วย วิธีการ

แบบ Dynamic ได้ถูกใช้ในคอมพิวเตอร์ทั่วไป เช่น Pentium, AMD, IBM เป็นต้น ดังนั้น จะเห็นได้ว่าทั้งฮาร์ดแวร์และซอฟต์แวร์มีส่วนสำคัญในการลด Stall ในการทำงานทั้งสิ้น

ในการจัดลำดับคำสั่งนี้จะต้องพิจารณาถึงความสัมพันธ์ (Dependency) ซึ่งถูกแบ่งออก 4 ประเภท ได้แก่

1. **Data Dependence** เป็นความสัมพันธ์จากข้อมูล โดยเกิดจากการใช้อ่านข้อมูลที่ถูกเปลี่ยนแปลงแล้ว หรือเรียกว่า Read-After-Write Hazard (RAW) จัดว่าเป็น True Dependence เพราะเป็นความสัมพันธ์ที่มีอยู่ในโปรแกรมเกิดจากการเขียนโดยเดียวและความหมายของโค้ดนั้นเอง

Add r1, r2, r4
Sub r4, r1, r4

ในตัวอย่างเป็นความสัมพันธ์จากการใช้รีจิสเตอร์ r1 ที่ถูกเขียนโดยคำสั่ง Add ตามเวลา ซึ่งเป็นไปตามลำดับของโค้ดจริง ในการปรับแต่งจัดลำดับการทำงานนั้นจะต้องคำนึงถึงความสัมพันธ์แบบนี้ไว้ เพราะต้องการรักษาความหมายของโปรแกรมเอาไว้ให้ทำงานได้ถูกต้อง

2. **Anti-Dependence** เป็นความสัมพันธ์ของข้อมูล เนื่องจากประโยชน์คัดมาจะเขียนทับข้อมูลที่ประโยชน์ก่อนหน้าอ่าน เรียกว่าเป็น Write-After-Read (WAR) บางครั้งจัดว่าเป็นความสัมพันธ์โดยชื่อ (Name Dependence) เพราะเป็นความสัมพันธ์ที่เกิดจากการใช้ชื่อของรีจิสเตอร์ที่ซ้ำกัน ในตัวอย่างด้านล่างเป็นรีจิสเตอร์ r1 โดยคำสั่งแรกใช้ r1 เพื่ออ่าน และคำสั่งถัดมาใช้ r1 ในการเขียน และถ้าคำสั่งหลังเสร็จก่อน ค่าใน r1 จะเปลี่ยนไป ทำให้คำสั่ง Sub อ่านโอเปอเรนด์ r1 เป็นค่าใหม่ ดังนั้น ถ้าคำสั่ง Add ถัดมาเปลี่ยนชื่อรีจิสเตอร์ที่ใช้เป็นชื่ออื่นที่ไม่ซ้ำ ก็จะไม่มีปัญหานี้เกิดขึ้น

Sub r4, r1, r3
Add r1, r3, r3
Mul r6, r1, r7

3. **Output Dependence** จัดเป็นความสัมพันธ์แบบชื่อเช่นกัน แต่เกิดจากการเขียนไปยังรีจิสเตอร์ชื่อเดียวกัน ดังตัวอย่างนี้เรียกว่าเป็นประเภท Write-After-Write Hazard (WAW) การแก้ปัญหาอาจจะทำได้โดยการเปลี่ยนรีจิสเตอร์ที่ใช้งานในคำสั่ง Add ให้ไปใช้รีจิสเตอร์ตัวอื่นแทน

```

Sub r1, r4, r5
Add r1, r2, r3
Mul r6,r1,r7

```

สำหรับการทำงานของไปป์ไลน์รูปแบบอื่นๆ เช่น แบบมีไปป์ไลน์หลายหน่วย จะก่อให้เกิดความสัมพันธ์ของข้อมูลรูปแบบอื่นๆ เช่น WAR, WAW ด้วย ซึ่งจะไม่พบในรูปแบบที่ง่ายในบทที่ 6

4. **Control Dependence** เป็นความสัมพันธ์ที่เกิดจากประโยชน์แบบเงื่อนไข เช่น ประโยชน์ S1 จะถูกทำเมื่อเงื่อนไข p1 เป็นจริง เรียกว่า S1 เป็น Control Dependence กับเงื่อนไข p1 และในตัวอย่างนี้ S2 ก็จะเป็น Control Dependence กับเงื่อนไข p2 เท่านั้น

```

if p1{
    S1;
}
if p2{
    S2;
}

```

ในการจัดลำดับโค้ดนั้น จะต้องคำนึงถึงความสัมพันธ์ที่เกิดจากประโยชน์แบบเงื่อนไขด้วย รวมไปถึงพฤติกรรมเกี่ยวกับการทำงานกรณีมี Exception เกิดขึ้นด้วย เพราะว่าถ้ามีการเปลี่ยนแปลงลำดับการทำงานแล้ว แม้ว่าผลลัพธ์จะถูกต้องในกรณีปกติ แต่ถ้ามี Exception เกิดขึ้น การจัดการลำดับการทำงานต้องพิจารณากรณีการเกิด Exception นี้ และลำดับการจัดการ Exception จะต้องเหมือนการทำงานแบบไม่ใช่ไปป์ไลน์ และไม่ก่อให้เกิด Exception ใหม่เพิ่มเติมด้วย

พิจารณาตัวอย่างนี้

```

DADDU R2,R3,R5
BEQZ R2,L1
LW R1,0(R2)
L1:

```

ถ้าเราย้ายคำสั่ง LW ไปไว้ก่อนคำสั่ง BEQZ แม้ว่าคำตอบจะยังเหมือนเดิม แต่การเกิด Exception จะมีพฤติกรรมต่างกัน เพราะว่าคำสั่ง LW อาจจะก่อให้เกิด Exception ในลักษณะของ

Page Fault เพราะเมื่อจัดการ Exception นี้เสร็จสิ้นแล้ว สถานะของหน่วยความจำอาจจะเปลี่ยนไปได้ ดังนั้น กรณีลักษณะของสถานะของหน่วยความจำที่ได้จะต่างจากรูปแบบที่ไม่ได้มีการย้ายคำสั่ง LW ไป ซึ่งอาจจะให้ผลการทำงานต่อๆ ไปไม่เหมือนกัน

ในการจัดลำดับการทำงาน จะเน้นว่าต้องรักษาการไหลของข้อมูล (Data Flow) ไว้ให้เหมือนเดิม เพื่อรักษาความถูกต้องของโปรแกรม พิจารณาการไหลของข้อมูลภายในรีจิสเตอร์ R1 ตามโค้ดด้านล่าง

DADDU	<u>R1</u> , R3, R5
BEQZ	R4, L
DSUBU	<u>R1</u> , R5, R6
L : ...	
OR	R7, <u>R1</u> , R8

ในการจัดลำดับนี้จะต้องคำนึงถึงชาร์ดแวร์ของไปป์ไลน์ และความล้มเหลวของข้อมูลในโค้ดเพื่อให้ได้ลำดับคำสั่งที่ทำงานได้ในจำนวนไข่เกิลที่น้อยที่สุดหรือเร็วที่สุดนั่นเอง อีกนัยหนึ่งจะต้องทำการลด Stall ให้หมดไปหรือให้เหลือน้อยที่สุด เมื่อไปป์ไลน์มีลักษณะที่แตกต่างกัน เช่น ความลึกแตกต่างกัน หน่วยคำนวนต่างๆ แตกต่างกัน ขั้นตอนแตกต่างกัน ก็จะทำให้ Stall ที่เกิดขึ้นในแต่ละกรณีแตกต่างกันไป อย่างที่ทราบแล้วว่า ถ้าไปป์ไลน์ลึกมากๆ และมีหลายหน่วยคำนวนโอกาสที่จะเกิด Stall จะมีได้มาก และจำนวน Stall ที่เกิดขึ้นแต่ละกรณีก็อาจจะใช้หลายไข่เกิลได้พิจารณาตัวอย่างโดยต่อไปนี้

$$\begin{aligned} a &= b + c \\ d &= e - f \end{aligned}$$

ถ้าเขียนเป็นคำสั่งรูปที่แบบ MIPS จะได้ดังนี้

LW	Rb, b
LW	Rc, c
ADD	Ra, Rb, Rc
SW	a, Ra
LW	Re, e
LW	Rf, f
SUB	Rd, Re, Rf
SW	d, Rd

จะเห็นว่า Stall จะเกิดขึ้นระหว่างคำสั่ง LW Rc และ ADD ... Rc และระหว่างคำสั่ง LW Rf และคำสั่ง ADD ... Rf

ถ้าเปลี่ยนได้เป็นลำดับดังนี้

LW	Rb, b
LW	Rc, c
<u>LW</u>	<u>Re, e;</u> สลับคำสั่งเพื่อเลี่ยง Stall
ADD	Ra, Rb, Rc
LW	Rf, f
<u>SW</u>	<u>a, Ra;</u> สลับคำสั่งเพื่อเลี่ยง Stall
SUB	Rd, Re, Rf
SW	d, Rd

จะเห็นว่าคำสั่งที่ได้จัดเร้นไว้หน้าได้ถูกยกย้ายมาเพื่อใช้ช่อง Stall ของคำสั่ง LW Rc และคำสั่ง Lw Rf 2 ตัวที่มีปัญหา ทำให้ลดเวลาการทำงานไปได้ 2 ไซเกิล

พิจารณาสถาปัตยกรรมลักษณะดังนี้

รูปแบบ คำสั่งผลิตผลลัพธ์	รูปแบบ คำสั่งที่ใช้ผลลัพธ์	Latency (ไซเกิล)	จำนวน Stall ระหว่างไซเกิล
FP ALU	FP ALU	4	3
FP ALU	Store ค่า Double	4	2
Load ค่า Double	FP ALU op	1	1
Load ค่า Double	Store ค่า Double	1	0
คำสั่งเกี่ยวกับจำนวนเต็ม	คำสั่งเกี่ยวกับจำนวนเต็ม	1	0

ในคอลัมน์แรกแสดงประเภทของคำสั่งที่ให้ผลลัพธ์ คอลัมน์ที่ 2 แสดงประเภทของคำสั่งที่ใช้ผลลัพธ์นั้น และคอลัมน์ที่ 3 แสดง Latency ระหว่าง 2 คำสั่งรูปแบบนี้ ในคอลัมน์ที่ 4 แสดงจำนวน Stall ที่ต้องการเมื่อ 2 คำสั่งนี้อยู่ติดกัน

พิจารณาลูปในภาษา C ดังนี้

```
for (i = 1; i <= 1000; i = i + 1)
    y [i] = y[i] + s;
```

แปลงเป็นคำสั่งใน MIPS ดังนี้ สมมติให้เลขหน่วยมีขนาด 8 ไบต์

Loop:	L.D	F0, 0 (R3)	; F0 array element ของ Y
	ADD.D	F4, F0, F2	
	S.D	0 (R3), F4	
	DADDUI	R3, R3, #-8	; ลดค่า pointer
	BNE	R3, R2, Loop	; branch R1 != R2

จะเห็นว่าในการทำงานของโค้ดดังกล่าวจะมีความล้มเหลวและ Stall เกิดขึ้นดังแสดงด้านล่าง Stall เกิดขึ้นในไซเกิลที่ 2, 4, 5, 8 และ 10

ไซเกิลที่		
Loop:	L.D	F0, 0 (R3) 1
	<u>stall</u>	2
	ADD.D	F4, F0, F2 3
	<u>stall</u>	4
	<u>stall</u>	5
	S.D	F4, 0 (R3) 6
	DADDUI	R3, R3, #-8 7
	<u>stall</u>	8
	BNE	R3, R2, Loop 9
	<u>stall</u>	10

ในการทำงานดังกล่าว จะเห็นว่าจาก LD ไปยัง ADD ต้องมี 1 Stall และจาก ADD ไปยัง SD ที่ต้องรอ F4 ต้องการอีก 2 Stall สมมติว่าการคำนวณหาตำแหน่งที่จะกระโดยไปทำในขั้นตอนการต่อไป จึงมี 1 Stall ให้คำสั่ง BNE ในการทำงานหั้งลูป 1 รอบ จะใช้หั้งหมด 10 ไซเกิล แต่ถ้าเปลี่ยนลำดับการทำงานเป็นดังโค้ดด้านล่างจะใช้หั้งหมดเพียง 7 ไซเกิล แต่สั้นมากกว่าการย้าย

คำสั่ง SD มากข้างล่างจะต้องปรับการอ้างถึงหน่วยความจำให้เหมาะสมด้วย

```

Loop:    L.D      F0, 0 (R3)
          DADDUI   R3, R3, #-8
          ADD.D    F4, F0, F2
          stall
          BNE      R3, R2, Loop ;delay branch
          S.D      F4, 8 (R3)

```

■ 7.3 การใช้เทคนิคขยายลูป (Unroll)

จากโค้ดข้างต้น จะพบว่าถ้าต้องการลดจำนวนไบเกิลต่อรอบลงอีกจะไม่สามารถทำได้เนื่องจากโค้ดในลูปมีเหลือเพียง 4 คำสั่ง ซึ่งไม่สามารถหาคำสั่งอื่นมาท่อน Stall ในไบเกิลที่ 4 – 5 ได้ สิ่งที่ต้องการทำต่อไปคือ การขยายโค้ดในลูปเพื่อให้มีจำนวนคำสั่งมากขึ้น และเปิดโอกาสนำคำสั่งเพิ่มเติมเหล่านั้นมาท่อน Stall ในไบเกิลเหล่านี้ วิธีการขยายลูปนี้เรียกว่า การ Unroll หรือ Unfold ลูปนั้นเอง ในตัวอย่างต่อไปเป็นการขยายลูปออกมา 4 รอบ คำสั่ง LD จะต้องการ 1 Stall และคำสั่ง ADD จะต้องการ 2 Stall

1. Loop: L.D F0, 0 (R3)
3. ADD.D F4, F0, F2
6. S.D F4, 0 (R3) ; เอา SUB, BNEZ ออก
7. L.D F6, -8 (R3)
9. ADD.D F8, F6, F2
12. S.D F8, -8 (R3) ; เอา SUB, BNEZ ออก
13. L.D F10, -16 (R3)
15. ADD.D F12, F10, F2
18. S.D F12, -16 (R3) ; เอา SUB, BNEZ ออก
19. L.D F14, -24 (R3)
21. ADD.D F16, F14, F2
24. S.D F16, -24 (R3) ; เอา SUB, BNEZ ออก
25. DADDUI R3, R3, #--32
26. BNE R3, R2, Loop ;branch R1 != R2

ดังนั้น ในการทำงานทั้งลูปที่ขยายนี้จะต้องทำการปรับการอ้างถึงข้อมูลในแต่ละรอบให้เหมาะสมด้วย จากโค้ดที่ Unroll นี้จะเห็นว่าจะใช้เวลาทั้งหมด 27 ไบเกิลต่อ 4 รอบ ซึ่งจะเท่ากับ 6.75 ไบเกิลต่อ 1 รอบ ในที่นี้หลังจาก Unroll จะยังไม่มีการจัดลำดับใดๆ ระหว่างคำสั่ง LD และ ADD.D ยังมี Stall อยู่ร่วมทั้งหมด 3 Stall ดังนั้น ในการขยายลูปจะขยายออกมากกว่ารอบก็ได้ และจะทำให้ลดจำนวนไบเกิลที่ใช้ต่อรอบได้ นอกจากนี้ การขยายลูปยังเป็นการขยายขนาดของโค้ดอีกด้วย การขยายลูปจึงต้องทำให้ได้ปริมาณที่เหมาะสม เพื่อไม่ให้ขยายขนาดของโค้ดมากจนเกินไป ทำให้สิ้นเปลืองที่เก็บโค้ด และอาจจะไม่เพิ่มประสิทธิภาพในการทำงาน

```

1. Loop: L.D      F0,0(R1)
2.       L.D      F6,-8(R1)
3.       L.D      F10,-16(R1)
4.       L.D      F14,-24(R1)
5.       ADD.D    F4,F0,F2
6.       ADD.D    F8,F6,F2
7.       ADD.D    F12,F10,F2
8.       ADD.D    F16,F14,F2
9.       S.D      F4,0(R1)
10.      S.D     F8,-8(R1)
11.      DADDUI   R1,R1,#-32
12.      S.D      F12,16(R1) ;32 - 16
13.      BNE     R1,R2,Loop ;branch R1 != R2
14.      S.D      F16,8(R1) ;8 - 32

```

ในโค้ดนี้ เมื่อจัดลำดับคำสั่งเรียบร้อยแล้ว เพื่อข่อน Stall ที่เกิดขึ้น จึงทำให้ใช้เวลา 14 ไบเกิล ต่อ 4 รอบ หรือ 3.5 ไบเกิลต่อ 1 รอบนั่นเอง

กล่าวโดยสรุป ในการขยายลูปจะต้องพิจารณาลิสต์ต่างๆ ดังนี้

- ลูปที่จะขยายนั้นต้องมีการทำงานแต่ละรอบที่ไม่เข้าหากัน หรือระหว่างรอบของการทำงาน ในแต่ละรอบสามารถทำงานได้โดยอิสระจากกัน
- หลังจากขยายแล้ว ต้องปรับการใช้งานรีจิสเตอร์ให้เหมาะสมเพื่อกำจัด WAW และ WAR ด้วย
- ต้องปรับการทดสอบการสิ้นสุดลูปให้เหมาะสม และจำนวนรอบที่ทำให้ถูกต้อง

4. พิจารณาการใช้หน่วยความจำตัวแหน่งต่างๆ ของคำสั่ง LD และคำสั่ง SD ปรับออฟเซตให้ถูกต้องในลูปที่ขยายแล้ว
5. จัดลำดับโค้ดในลูปใหม่ที่ได้เพื่อข่อน Stall ต่างๆ

การขยายลูปจะเป็นการเพิ่มขนาดของโค้ดทั้งหมด แต่จะเป็นการเปิดโอกาสให้ตัวแปลภาษาหาคำสั่งมาข่อน Stall ได้มากขึ้น โดยจะข่อน Stall ได้ทั้งในส่วนของ Data Dependency และ Control Dependency ด้วย

จำนวนรอบที่ถูกขยายออกมากต้องพิจารณาให้เหมาะสม โดยเฉพาะกับระบบที่มีข้อจำกัดสำหรับขนาดของหน่วยความจำโค้ด นอกจากนี้ การขยายลูปออกมากจำนวนมากจะทำให้สิ้นเปลืองการใช้รีจิสเตอร์ในข้อที่ 2 ถ้าระบบมีจำนวนรีจิสเตอร์จำกัดจะทำให้มีปัญหาเรื่องการเปลี่ยนชื่อรีจิสเตอร์ได้ และทำให้ไม่ได้โค้ดที่มีประสิทธิภาพเต็มที่

การขยายลูปนั้นเป็นการเปิดโอกาสให้ตัวแปลภาษาເเอกสารคำสั่งมาใส่ไว้ใน Delay Slot เพื่อข่อน Control Dependency นอกจากนี้ยังมีวิธีอีกหนึ่ง ที่กำจัด Control Dependence ในชาร์ดแวร์ด้วยดังได้กล่าวถึงไปแล้วในบทที่ 6

■ 7.4 Dynamic Scheduling โดยใช้ Tomasulo

ในการจัดลำดับแบบ Dynamic จะพิจารณาการจัดลำดับในการป้อนคำสั่งไปในไปป์ไลน์เพื่อลด Stall ที่ไม่สามารถกำจัดได้โดยการจัดลำดับโค้ดตัวอย่างด้วยตัวแปลภาษาของ และลด Stall ที่อาจจะเกิดระหว่างทำงาน เช่น Stall จากแคช Miss โดยอาจจะไปทำงานคำสั่งอื่นไปพลางๆ นอกจากนี้จะเอื้อประโยชน์ในการนำโค้ดที่แปลงแล้วไปใช้สำหรับไปป์ไลน์รูปแบบหนึ่ง และนำมาทำงานบนไปป์ไลน์อีกรูปแบบโดยไม่ต้องแปลงใหม่ด้วย

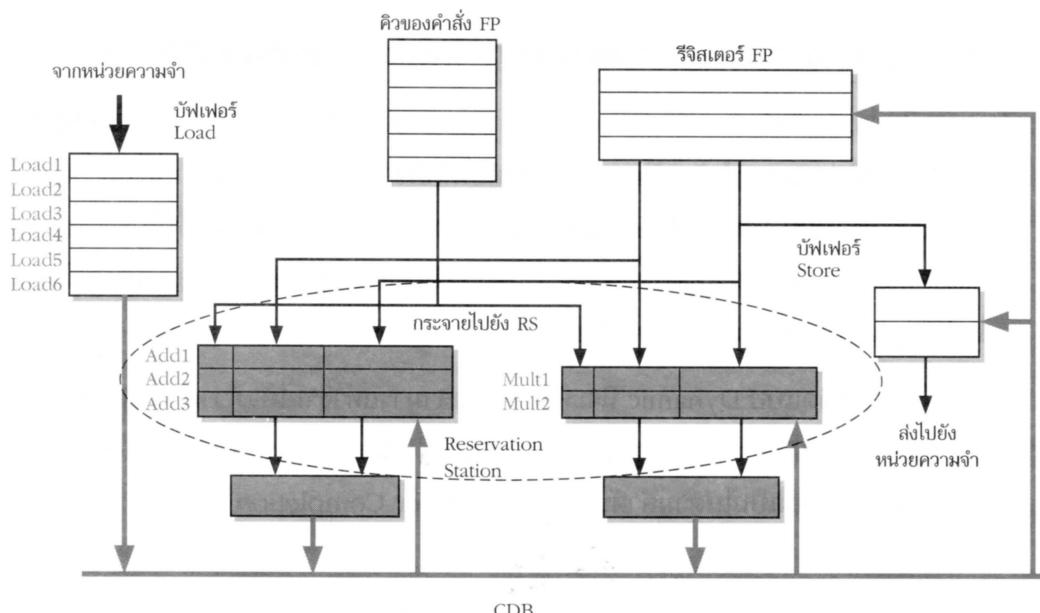
ในแนวคิดแบบ Dynamic นี้จะทำให้คำสั่งสามารถทำงานได้แม้ว่าคำสั่งก่อนหน้ายังไม่เสร็จสิ้น อาจจะทำให้เกิดการอึซซิคิวต์แบบไม่ตามลำดับ (Out-Of-Order Execution) และคำสั่งจะทำงานเสร็จแบบไม่เป็นไปตามลำดับ (Out-Of-Order Completion) ได้

DIVD	<u>F0</u> , F2, F4
ADDD	F14, <u>F0</u> , F8
SUBD	F12, F8, F14

ในโค้ดนี้ คำสั่ง DIVD จะใช้เวลานาน แต่อาจจะทำให้คำสั่ง ADDD สามารถทำงานได้ถ้าไม่ได้มีความเกี่ยวข้องกับคำสั่ง DIVD และถ้ายังมีหน่วยคำนวณที่รองรับ และคำสั่ง ADDD อาจจะเสร็จก่อนคำสั่ง DIVD ได้ แต่ในโค้ดทั้งสองคำสั่งมีความลับพันธ์กันด้วย F0 สำหรับคำสั่ง SUB อาจจะเสร็จได้ก่อน เพราะไม่ได้เกี่ยวข้องกับคำสั่งก่อนหน้านี้เลย

ในการจัดลำดับแบบ Dynamic นั้นจะแยกขั้นตอนต่อไปเป็นขั้นตอนย่อย ซึ่งทำหน้าที่ตรวจสอบ Structure Hazard และตรวจสอบ RAW ได้แก่ ขั้นการ Issue และขั้นการอ่านโอเปอเรนด์ (Read Operand) โดยในขั้น Issue จะทำการตรวจสอบ Structure Hazard เพื่อดูว่าหน่วยคำนวณที่ต้องการใช้ว่างอยู่หรือไม่ และขั้นอ่านโอเปอเรนด์จะทำการจองกว่าโอเปอเรนด์ที่ต้องการพร้อมอยู่ในรีจิสเตอร์แล้ว เทคนิคดังกล่าวมีสมมติฐานอยู่ในวิธีการ Tomasulo ซึ่งเกิดขึ้นกับเครื่องคอมพิวเตอร์ในปี ค.ศ. 1966 เทคนิคนี้ในปัจจุบันก็ยังใช้กันอยู่ในเครื่องคอมพิวเตอร์ตระกูล Pentium, AMD, Power เป็นต้น ใน Tomasulo เองจะมีการกำจัด WAW, WAR ด้วยโดยการใช้เปลี่ยนชื่อรีจิสเตอร์ (Register Rename) ด้วย

ลักษณะของ Tomasulo ประกอบด้วยโครงสร้างพื้นฐานดังแสดงในรูปที่ 7.5



รูปที่ 7.5 โครงสร้างพื้นฐานของ Tomasulo

ในโครงสร้างนี้จะมีตัวบวก (FP Adder) และตัวคูณ (FP Multiplier) แต่ละหน่วยคำนวณโดยเริ่มต้น คำสั่งที่จะทำงานจะอยู่ในคิวคำสั่ง FP จากนั้นจึงถูกกระจายไปยังหน่วยตัวบวกหรือตัวคูณ ซึ่งจะมีคิวบัฟเฟอร์ (Queue) ของตนเองเพื่อเก็บคำสั่งและโอดีเพอแรนด์ที่จะทำงานในหน่วยของตน และมีหน่วยควบคุมของตนเองเป็นการควบคุมแบบกระจาย มีบัฟเฟอร์ที่กระจายสำหรับแต่ละบัฟเฟอร์เรียกว่า Reservation Stations (RS) จะเข้าไว้เก็บโอดีเพอแรนด์สำหรับแต่ละคำสั่งที่รอการใช้งาน รีจิสเตอร์ในแต่ละคำสั่งโอดีเพอแรนด์จะถูกเปลี่ยนเป็นค่าหรือเป็นพอยน์เตอร์ที่ซึ่ไปยังคำสั่งที่จะให้ค่านั้น พอยน์เตอร์ก็จะเป็นพอยน์เตอร์ซึ่ไปยัง Reservation Station และที่จะเก็บผลลัพธ์ที่ต้องการ จึงเท่ากับเป็นการเปลี่ยนชื่อทางอ้อม เพื่อกำจัด WAW, WAR และ Reservation Station นี้จะทำให้เลื่อนว่ามีรีจิสเตอร์เพิ่มขึ้นในส่วนที่ตัวแปลภาษาของไมโครชิปนั้น ผลลัพธ์จะถูกส่งจาก RS ไปยังหน่วยคำนวณโดยตรงไม่ผ่านรีจิสเตอร์ การส่งใช้ Common Data Bus (CDB) และเป็นการกระจายผลลัพธ์ที่ได้จากหน่วยคำนวณไปยังหน่วยอื่นที่รอผลนั้นอยู่ ตรงนี้เป็นการกำจัด RAW และเมื่อคำสั่งได้รับโอดีเพอแรนด์ที่ต้องการ จะเริ่มทำการอีกชีกิวต์ได้ สำหรับ Load/Store จะถูกพิจารณาว่าเป็นหน่วยคำนวณเด่นกัน และจะมีบัฟเฟอร์เก็บค่าของคำสั่งที่เข้ามาทำงานคำสั่งเกี่ยวกับจำนวนเต็มจะสามารถทำงานได้แม้ว่าอยู่ต่อหนาต่อบล็อกกัน (เป็นการใช้การทำงานแบบกระโดด) อีกนัยหนึ่ง

ใน RS แต่ละແຄواจะประกอบด้วยพิล็อต์ต่างๆ ดังนี้

Op: ระบุคำสั่งที่จะทำงาน เช่น บวก ลบ

V_j, V_k: หมายถึง ค่าของโอดีเพอแรนด์ที่จะใช้งานกับตัวดำเนินการนั้น

ในบัฟเฟอร์ของ Store จะมีค่า V เก็บไว้เพื่อเก็บค่าที่จะเก็บลงไปจริง

Q_j, Q_k: จะซึ่ไปยัง RS แต่ที่เป็นที่มาของค่าของรีจิสเตอร์นั้นๆ ถ้า Q_j, Q_k = 0 หมายถึงว่า โอดีเพอแรนด์นั้นพร้อมแล้ว หน่วยคำนวณนั้นจะสามารถเริ่มทำงานได้ และค่าของโอดีเพอแรนด์จะระบุในพิล็อต์ V_j, V_k

ในบัฟเฟอร์ของ Store จะมีพิล็อต์ Q_i สำหรับเก็บ RS ที่ให้ผลลัพธ์ที่จะทำคำสั่ง Store ถ้าระบุว่า Busy หมายถึง RS นั้นถูกใช้งานอยู่

Register Result Status จะเก็บชื่อหน่วยคำนวณที่จะเขียนค่าไปยังรีจิสเตอร์นั้น ถ้าเป็นเช่นร่าง หมายถึงว่าไม่มีคำสั่งใดๆ เขียนไปยังรีจิสเตอร์นั้น

ใน Tomasulo จะมีขั้นตอนต่างๆ ตามที่อธิบายดังนี้

1. **Issue** เป็นการอ่านคำสั่งมาจากคิวของ FP Op ถ้า RS ยังว่าง หมายถึงว่า ไม่มี Structure Hazard หน่วยควบคุมจะ Issue คำสั่งและส่งค่าของโอเปอเรนด์ที่ต้องการไปให้ (เป็นการเปลี่ยนชื่อรีจิสเตอร์ทางอ้อม)
2. **ເອັກສີຄົວຕີ** เป็นการทำโนดโดยดำเนินการกับโอเปอเรนด์เมื่อโอเปอเรนด์นั้นพร้อมแล้ว และถ้ายังไม่พร้อมก็จะคอยดูว่าค่าโอเปอเรนด์นั้นถูกคำนวณและส่งมาอย่างบัส CDB แล้ว หรือยัง
3. **Write Result** เสียงผลลัพธ์ตามขั้นเสียงค่ากลับ โดยเสียงไปยัง CDB เพื่อให้หน่วยอื่นๆ ที่รับผลนั้นอยู่นำໄປใช้ได้ และทำการบันทึกว่า RS นั้นๆ ว่างแล้ว

โดยทั่วไปแล้ว CDB จะส่งข้อมูลและแหล่งที่มาของข้อมูล โดยข้อมูลจะมีขนาด 64 บิต และแต่ละข้อมูลจะมีขนาด 64 บิต แต่แต่ละข้อมูลจะใช้กับหน่วยคำนวณอื่นๆ ที่ต้องรอคิวยอดลัพธ์นี้ ซึ่งเป็นการกระจายข้อมูลแบบ Broadcast

ในตารางข้างล่างแสดงโครงสร้างข้อมูลสำหรับการทำงานของ Tomasulo ตารางแรกแสดงสถานะคำสั่ง ในช่องจะระบุไขเกิลที่แต่ละคำสั่งเข้าสู่ขั้นต่างๆ ขั้น Issue ขั้น Exec Complete (ทำการເອັກສີຄົວຕີเสร็จในหน่วยนั้น) และขั้นเสียงผลลัพธ์

			สถานะคำสั่ง (ไขเกิลที่)		
คำสั่ง	j	k	Issue	Exec Complete	Write Result
LD F6	34+	R2			
LD F2	45+	R3			
MULTD F0	F2	F4			
SUBD F8	F6	F2			
DIVD F10	F0	F6			
ADDD F6	F8	F2			

ตารางต่อไปแสดงสำหรับหน่วย Load ซึ่งมี 3 หน่วยว่า่ว่างอยู่หรือไม่ ถ้าไม่ว่าง (Busy = No) กำลังทำงานกับแอดเดรสใด

	Busy	แอดเดรส
Load1	No	111000000000
Load2	No	
Load3	No	

ต่อไปเป็นตารางเก็บค่าสถานะของ RS สำหรับหน่วย Add ซึ่งมี 3 หน่วย และหน่วย Mult ซึ่งมี 2 หน่วย

Reservation Status						
		S1	S2	RS	RS	
	Busy	Op	Vj	Vk	Qj	Qk
Add1	No					
Add2	No					
Add3	No					
Mult1	No					
Mult2	No					

และสุดท้ายเป็นตารางเก็บค่าสถานะของรีจิสเตอร์ว่ามีหน่วยอะไรกำลังใช้งานอยู่บ้าง อาจจะอ้างอิงไปยังภาพของตาราง RS ก็ได้

	F0	F2	F4	F6	F8	F10	F12
หน่วยคำนวน							
ที่ใช้อยู่							

พิจารณาตัวอย่างต่อไปนี้ กำหนดให้ FP ADD, SUB ใช้เวลาทำงาน 3 ไซเกิล การคูณใช้เวลา 10 ไซเกิล และการหารใช้เวลา 40 ไซเกิล

เมื่อเริ่มต้น LD บรรทัดแรกถูกทำงาน ระบุโดยตัวเลขในช่วง Issue หมายถึงถูก Issue ในไซเกิลที่ 1 และจะเห็น Load Buffer และ Load1 ไม่ว่าง และมีค่า Address เป็น $34+R2$ บอกว่ามีมากองหน่วยความจำที่ต้องการ Load ใน Register Result Status มากกว่า F6 ต้องการผลจากหน่วย Load1 ดังตารางข้างล่าง

			สถานะคำสั่ง (ไซเกิลที่)		
คำสั่ง	j	k	Issue	Exec Complete	Write Result
LD F6	34+	R2	1		
LD F2	45+	R3			
MULTD F0	F2	F4			
SUBD F8	F6	F2			
DIVD F10	F0	F6			
ADDD F6	F8	F2			

	Busy	แอดเดรส
Load1	Yes	$34+R2$
Load2	No	
Load3	No	

	F0	F2	F4	F6	F8	F10	F12
หน่วยคำนวณที่ใช้อยู่				Load1			

ต่อมาในไชเกิลที่ 2 LD บรรทัดที่ 2 ถูก Issue ไชเกิลที่ 2 ไปยังผล Load2 ใน Register Result Status บอกว่ารีจิสเตอร์ F2 ต้องการผลจากหน่วย Load2 ต่อมาไชเกิลที่ 3 MULTD จะถูก Issue ไปยัง RS แล้ว Mult1 ระบุว่าต้องการโอบอเพนเดนต์จากหน่วย Load2 ทำให้ยังทำงานไม่ได้ และค่าอ่านได้จากรีจิสเตอร์ F4 ได้ระบุว่าต้องการเขียนไปยังรีจิสเตอร์ F0 ใน Register Result Status ในเวลาที่ LD บรรทัดแรกเข้าสู่ขั้นເອົກຫີຕົວດີ

เมื่อไชเกิลที่ 4 Load1 เสร็จสิ้น โดยจะเขียนค่ากลับลงไปในไชเกิลนี้ ส่วน Load2 เข้าสู่ขั้นເອົກຫີຕົວດີได้ เพราะหน่วย Load ทำคำสั่งแรกเสร็จแล้ว และคำสั่ง SUBD ถูก Issue ไปยัง RS แล้ว Add1 โดยต้องรออ่านโอบอเพนเดนต์จาก Load2 (Qk) และค่าจากหน่วยความจำที่ได้จาก LD บรรทัดแรก (Vj) ระบุใน Register Result Status ว่าต้องการเขียนไปยังรีจิสเตอร์ F8

ในไชเกิลที่ 5 LD คำสั่งที่ 2 ทำงานเสร็จสิ้น ในช่อง F2 เก็บค่า M(A2) หมายถึง ค่ามา จากหน่วยความจำและเดรส์ที่ระบุใน A2 (ซึ่งคือ 45(R3)) ในเวลาที่ DIVD ถูก Issue ไปยังผล Mult2 แต่ยังต้องรอโอบอเพนเดนต์จาก Mult1 อยู่ และตัวคูณเริ่มทำงาน จะเห็นใน Mult1 ที่ต้องมี การเริ่มนับเวลาการทำงานของตัวคูณแล้ว เพราะเริ่มทำงานโดยใช้โอบอเพนเดนต์ที่ได้จากคำสั่ง LD อันที่ 2 M(A2) และค่าในรีจิสเตอร์ F4 (R(F4)) ส่วน Add1 ก็เช่นกัน ก็ต้องมีการเริ่มนับเวลาการทำงาน โดยการทำคำสั่ง SUB ใช้ค่าโอบอเพนเดนต์จากหน่วยความจำ M(A1) และ M(A2) ในที่นี้ ทำคำสั่ง FP ADD ใช้ 3 ไชเกิล และตัวคูณใช้ 10 ไชเกิล ดังตารางข้างล่าง

สถานะคำสั่ง (ไชเกิลที่)					
			Issue	Exec Complete	Write Result
คำสั่ง	j	k			
LD F6	34+	R2	1	3	4
LD F2	45+	R3	2	4	5
MULTD F0	F2	F4	3		
SUBD F8	F6	F2	4		
DIVD F10	F0	F6	5		
ADDD F6	F8	F2			

	Busy	แอ็ดเดรส
Load1	No	
Load2	No	
Load3	No	

ในเวลาผ่านไปใช้เกลิที่ 6 คำสั่ง ADD ถูก Issue ต่อ เพราะว่า RS ยังร่วงโดยไปใช้ Add2 และระบุว่าต้องการรอโวเปอแรนด์จาก Add1 (Qj) และເອົາດ່າຈາກ LD บรรทัดที่ 2 มาใช้ (V_k) ระบุ Register Result Status ว่าໃຫ້ຮັບສະເຕອຣ໌ F6

ต่อมาในไฟเกลที่ 7 ในเวลานี้ SUBD ใน Add1 ทำงานเสร็จแล้ว ดังในขั้นເອົກຫີຄົວດໍາເລັດ

เมื่อ SUBD ทำการเอิกซ์คิวต์สเต็จ จะทำการเขียนผลไปยัง F8 และคำสั่ง ADDD ถูกเปลี่ยนค่าในโโคเดอร์ V_k เป็นค่าที่ได้จาก SUBD

ตารางข้างล่างแสดงสถานะเมื่อผ่านไป 9 ไซเกิล

คำสั่ง	j	k	สถานะคำสั่ง (ไซเกิลที่)		
			Issue	Exec	Complete
					Write Result
LD F6	34+	R2	1	3	4
LD F2	45+	R3	2	4	5
MULTD F0	F2	F4	3		
SUBD F8	F6	F2	4	7	8
DIVD F10	F0	F6	5		
ADDD F6	F8	F2			

	Busy	แออดเดรส
Load1	No	
Load2	No	
Load3	No	

Reservation Status

รอบ	เวลา	Reservation Status					
		Busy	Op	Vj	Vk	Qj	Qk
	Add1	No					
1	Add2	Yes	Add	(M-M)	M(A2)		
	Add3	No					
6	Mult1	Yes	MULTD	M(A2)	A(F4)		
	Mult2	No	DIVD		M(A1)	Mult1	

	F0	F2	F4	F6	F8	F10	F12
หน่วยคำนวณ ที่ใช้อยู่	Mult1	M(A2)		Add2	(M-M)	Mult2	

ในเวลาไซเกิลที่ 9 – 10 นี้ RS แต่ละແຕງยังใช้ออยู่ ในไซเกิลที่ 11 คำสั่ง ADDD ทำงานเสร็จแล้ว ดังนั้น Register Result Status ถูกเปลี่ยน และในไซเกิลที่ 16 คำสั่ง MULTD ได้ทำงานเสร็จสิ้น จะเห็นว่าเมื่อ MULTD ทำงานเสร็จสิ้น โอเปอเรนเดอร์ของ RS และ Mult2 ถูกเปลี่ยนใน Vj และ Register Result Status F0 เก็บค่าผลจากคำสั่ง MULTD ดังตารางข้างล่าง

คำสั่ง	j	k	สถานะคำสั่ง (ไซเกิลที่)		
			Issue	Exec Complete	Write Result
LD F6	34+	R2	1	3	4
LD F2	45+	R3	2	4	5
MULTD F0	F2	F4	3	15	16
SUBD F8	F6	F2	4	7	8
DIVD F10	F0	F6	5		
ADDD F6	F8	F2	6	10	11

	Busy	แอดเดรส
Load1	No	
Load2	No	
Load3	No	

Reservation Status

นาฬิกา	เวลา	Reservation Status					
		Busy	Op	S1	S2	RS	RS
	Add1	No					
	Add2	No					
	Add3	No					
40	Mult1	No					
	Mult2	Yes	DIVD	M*F4	M(A1)		
		F0	F2	F4	F6	F8	F10
หน่วยคำนวณ		M*F4	M(A2)		M+M+M	(M-M)	Mult2
ที่ใช้อยู่							

ส่วน DIVD ใน Mult2 จะເອີກຫີ່ຄົວຕໍ່ເສັ້ນໃນໄຊເກີລທີ 56 ແລະເບີຍນຜລັກພົງໃນໄຊເກີລທີ 57

สถานะคำสั่ง (ໄຊເກີລທີ)

คำสั่ง	j	k	Issue	Exec	Complete	Write Result
LD F6	34+	R2	1	3		4
LD F2	45+	R3	2	4		5
MULTD F0	F2	F4	3	15		16
SUBD F8	F6	F2	4	7		8
DIVD F10	F0	F6	5	56		57
ADDD F6	F8	F2	6	10		11

	Busy	แอดเดรส
Load1	No	
Load2	No	
Load3	No	

	Reservation Status					
	Busy	Op	S1	S2	RS	RS
			Vj	Vk	Qj	Qk
Add1	No					
Add2	No					
Add3	No					
Mult1	No					
Mult2	Yes	DIVD	M*F4	M(A1)		

	F0	F2	F4	F6	F8	F10	F12
หน่วยคำนวน	M*F4	M(A2)		M+M+M	(M-M)	ค่าผลลัพธ์	
ที่ใช้อยู่							

จากตัวอย่างจะเห็นว่า Tomasulo นั้นใช้วิธีการเปลี่ยนชื่อรีสเตอร์โดยใช้ RS มากว่าย และอาจจะทำให้ลูปในรอบต่างๆ กันมาทำงานพร้อมกันได้ การใช้ RS จะทำให้สามารถ Issue คำสั่งได้ และทำงานได้กับค่าเก่าอยู่ เป็นการแก้ปัญหา WAR นอกจานี้วิธี Tomasulo ยังทำการตรวจสอบความพร้อมของโอเปอเรนเตอร์ก่อนทำงาน (RAW) ซึ่งเป็นการพิจารณาการให้ผลของข้อมูลระหว่างการทำงานไปด้วย

ข้อดีของ Tomasulo สรุปได้ดังนี้

1. เป็นการทำงานแบบกระจาย โดยจะกระจายการตรวจสอบ Hazard ไปยังหน่วยต่างๆ และใช้ CDB มากกว่า ถ้าหากคำสั่งในหน่วยคำนวนที่ต่างกันต้องการใช้ข้อมูลเดียวกัน

ก็สามารถทำงานได้เมื่อข้อมูลถูกส่งมาจาก CDB และถ้าเป็นบัสแบบปกติจะต้องรอการคิวการใช้รีจิสเตอร์นั่น

2. กำจัด WAW และ WAR Hazard ได้โดยใช้ RS เก็บค่าใน V_k, V_j

แต่มีข้อเสียในเรื่องของความซับซ้อนและ CDB เพราะประสิทธิภาพหั้งหมวดขึ้นอยู่กับ CDB นี้ และถ้าต้องการเพิ่มประสิทธิภาพจะต้องเพิ่มความเร็วในการส่งข้อมูลของ CDB ซึ่งจะเพิ่มความซับซ้อนไปอีก และในเรื่องการจัดการ Exception จะเป็นแบบ Imprecise ซึ่งยากขึ้น

■ 7.5 Tomasulo แบบ Speculative

ในการทำงานของ Tomasulo นั้น ได้แก้ปัญหาของ WAW และ WAR ไปแล้ว ยังคงเหลือปัญหาของ Control Dependence อยู่ที่ยังไม่ได้แก้ ถ้าหากสามารถทำนายผลของคำสั่งกระโดดได้ถูกต้อง (Speculative) โดยอาจจะใช้ชาร์ดแวร์ช่วย จะทำให้เพิ่มประสิทธิภาพของ ILP ได้มากกว่านี้ ถ้าสมมติว่าการทำนายของคำสั่ง Branch ถูกต้อง ก็จะทำการเฟตช์และ Issue รวมทั้งเอ็กซ์คิวต์คำสั่งเหมือนปกติ ในหัวข้อข้างต้นเราได้กล่าวถึงการจัดลำดับแบบ Dynamic เพียงอย่างเดียวที่ครอบคลุมแค่การเฟตช์และ Issue เท่านั้น โดยทั่วไปแล้ว ในการทำงานคำสั่งหนึ่งๆ นั้นจะทำการเอ็กซ์คิวต์ได้เมื่อโอเปอเรนเตอร์ของคำสั่งนั้นฯ พร้อมแล้ว การใช้ Speculation จะช่วยลด Stall จาก Control Hazard ได้

องค์ประกอบของชาร์ดแวร์สำหรับการใช้ Speculation ได้แก่ การใช้ทำนาย Branch แบบ Dynamic การใช้ Speculation เพื่อเอ็กซ์คิวต์คำสั่งได้ก่อน จะทราบผลของ Control Dependency ที่เกี่ยวกับคำสั่งนั้น ความสามารถในการยกเลิกการทำงานในกรณีที่ไม่เกิดการกระโดดจริง และการใช้การจัดลำดับแบบ Dynamic ก็เพื่อหากคำสั่งจากส่วนอื่นฯ มาจัดลำดับเพื่อลดจำนวน Stall จาก Control Dependency

ในการปรับ Tomasulo เพื่อรับ Speculation ได้แก่

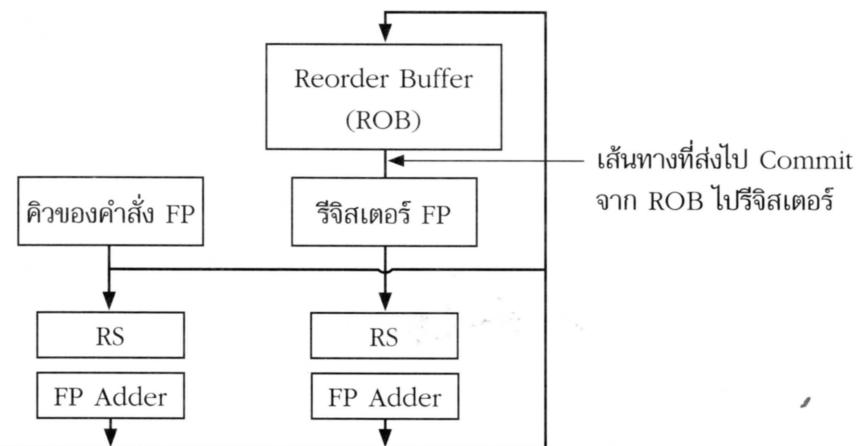
1. การเพิ่มขั้นตอนการ Commit เพื่อเขียนผลลัพธ์ไปยังรีจิสเตอร์และหน่วยความจำ เมื่อแนใจว่าคำสั่งนั้นต้องทำจริงๆ แล้ว คำสั่งนั้นจะไม่เป็นแบบคาดคะเนว่าจะทำอีกต่อไป
2. เมื่อคำสั่งนั้นต้องทำงานจริง คำสั่งนั้นจะไม่เป็นคำสั่งแบบ Speculative อีกต่อไป จึงจะต้องเขียนผลลัพธ์ลงไปในรีจิสเตอร์และหน่วยความจำ (เรียกว่า Commit) ดังนั้น จึงต้องการบันไฟฟอร์เพิ่มเพื่อเก็บผลลัพธ์ข่าวคราวก่อนทำการ Commit ตอนที่ยังเป็นคำสั่ง

Speculative อยู่ บัฟเฟอร์นี้เรียกว่า Reorder Buffer (ROB) การเพิ่ม ROB นี้จะทำให้เป็นการเพิ่มที่เก็บข้อมูลชั่วคราวด้วย

ในวิธี Tomasulo นั้น เมื่อคำสั่งเขียนผลลัพธ์ไปยังรีจิสเตอร์ไฟล์ คำสั่งอื่นๆ ต่อมา ก็จะใช้ผลลัพธันนี้ได้จากรีจิสเตอร์ไฟล์ ในการนี้ของ Speculative รีจิสเตอร์จะถูกเปลี่ยนแปลงค่า เมื่อเข้าสู่ขั้น Commit แล้ว และคำสั่งอื่นๆ อาจจะอ่านข้อมูลจาก ROB เพื่อประมวลผล ซึ่งจะทำให้คำสั่งเหล่านั้นเป็น Speculative ด้วย ROB จะเป็นที่พักข้อมูลเมื่อเดียวกับ RS และรอการ Commit เพื่อเขียนผลไปยังรีจิสเตอร์ แต่ละ ROB ประกอบด้วยพิล็อตต่างๆ ดังนี้

- ประเภทคำสั่ง ได้แก่ ประเภท Branch, Store, ALU Operation, Load
- โอลเปอแรนด์ปลายทาง (Destination) จะระบุรีจิสเตอร์และตำแหน่งหน่วยความจำที่ใช้
- ค่า (Value) ค่าของข้อมูลผลลัพธ์ก่อนจะ Commit
- แฟล็ก (Ready) ระบุว่าคำสั่งนั้นเสร็จสิ้นหรือยัง ถ้าเสร็จแล้วจะได้ค่าผลลัพธ์อยู่ในพิล็อต Value

หน้าที่ของ ROB จะเก็บคำสั่งในลำดับแบบ FIFO ในลำดับเดียวกับที่คำสั่งถูก Issue เมื่อคำสั่งทำงานเสร็จ ผลลัพธ์จะถูกเก็บไว้ใน ROB และอาจจะถูกส่งไปเป็นโอลเปอแรนด์ของคำสั่งอื่นๆ ตัว ROB นี้จะทำหน้าที่เหมือนกับ RS ที่มีรีจิสเตอร์ชั่วคราวเพิ่ม การ Commit จะตามลำดับคำสั่งที่อยู่ใน ROB ตั้งแต่ต้นคิวไป สำหรับผลลัพธ์ของคำสั่งที่เป็น Speculative จะจะถูกยกเลิกได้กรณีคำสั่ง Branch ถูกทำนายผิดหรือมี Exception เกิดขึ้น ดังแสดงในรูปที่ 7.6



รูปที่ 7.6 โครงสร้างส่วนที่เป็น ROB เพิ่มเข้ามา และการ Commit จะตามคิวลงไปยังรีจิสเตอร์ FP

ขั้นตอนต่างๆ ของวิวี Tomasulo จะเปลี่ยนแปลงบางขั้นตอนดังนี้

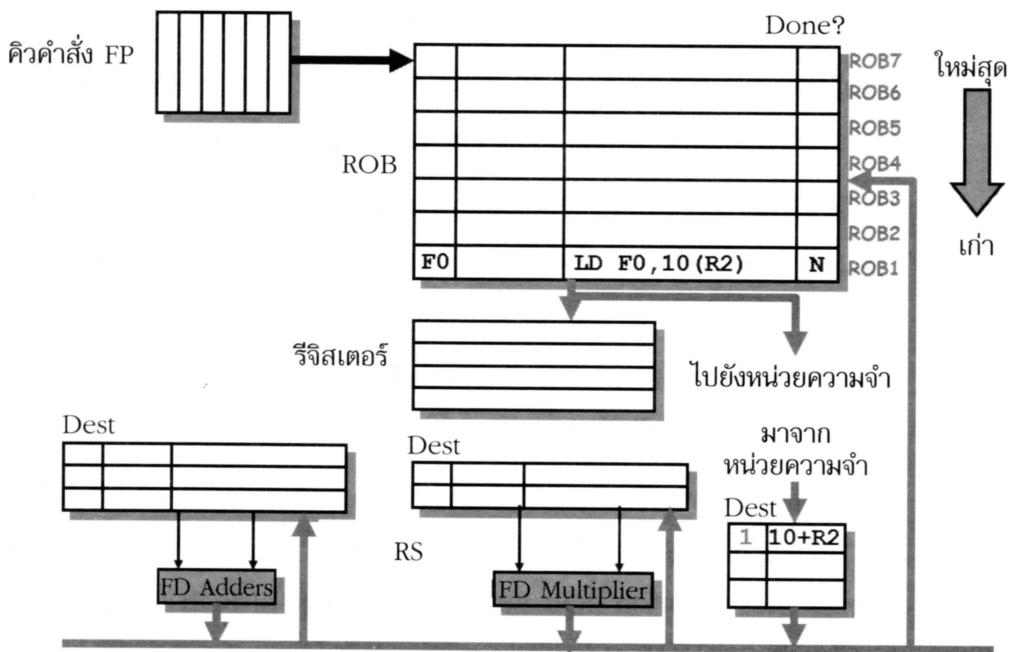
1. Issue จะอ่านคำสั่งจากคิวของคำสั่ง FP ถ้า RS และ ROB ว่าจะ Issue คำสั่งและส่งโอเปอเรนเดอร์ที่ต้องการไปยัง ROB ที่ใช้ขั้นตอนนี้เรียกว่า Dispatch
2. เอ็กซ์คิวต์ จะทำงานตามคำสั่งเมื่อโอเปอเรนเดอร์พร้อม ถ้ายังไม่พร้อมก็จะรอโดยดูจาก CDB และเมื่อโอเปอเรนเดอร์มาถึง RS เรียบร้อยแล้วจะทำงาน โดยตรวจสอบ RAW ก่อน บางครั้งเรียกว่า Issue
3. เขียนค่ากลับ (Write Result) เขียนผลลัพธ์ไปยัง CDB ส่งไปยังหน่วยต่างๆ และ ROB ที่รออยู่ และทำการบันทึกว่า RS แวนนี้ว่างแล้ว
4. Commit ทำการเขียนไปยังรีจิสเตอร์จริงจากผลที่เก็บใน ROB คำสั่งนั้นๆ จะถูก Commit เมื่อคำสั่งนั้นอยู่ที่หัวคิวของ ROB คำสั่ง Branch ที่มีการทำนายผิด ผลนั้นจะถูก Flush ออกจาก ROB และผลของคำสั่งถัดมาใน ROB ก็จะถูก Flush เนื่องกัน (เรียกว่า Graduation)

พิจารณาตัวอย่างต่อไปนี้

LD	F0, 10 (R2)
ADDD	F10, F4, F0
DIVD	F2, F10, F6
BNE	F2, L1
L1:	LD F4, 0 (R3)
	ADDD F0, F4, F6
ST	0 (R3), F4

ในรูปที่ 7.7 จะแยก RS ของแต่ละหน่วยออกจากกัน ตัวเลขข้างหน้าในช่อง Dest จะระบุ แควของ ROB ที่ผลิตผลลัพธ์ไปยังที่นั้น

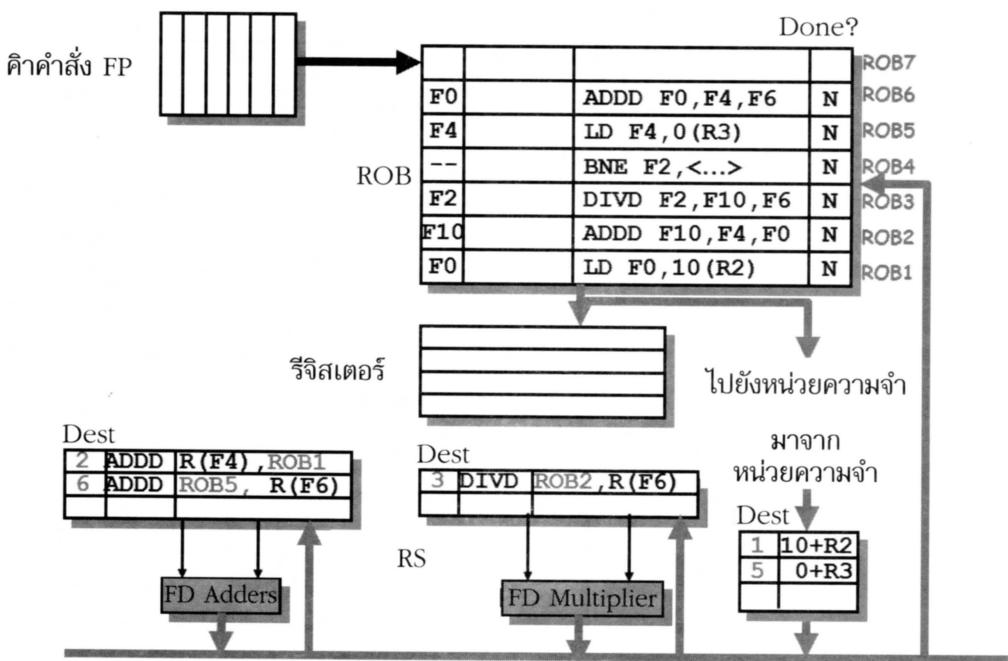
เมื่อเริ่มต้นคำสั่ง LD จะเข้าสู่ ROB ก่อนหน่วยจัดการหน่วยความจำจะระบุตำแหน่งหน่วยความจำที่จะเอาข้อมูลได้แก่แอดเดรส 10+R2



รูปที่ 7.7 โครงสร้างเมื่อเติม ROB เมื่อคำสั่ง LD เข้าไป

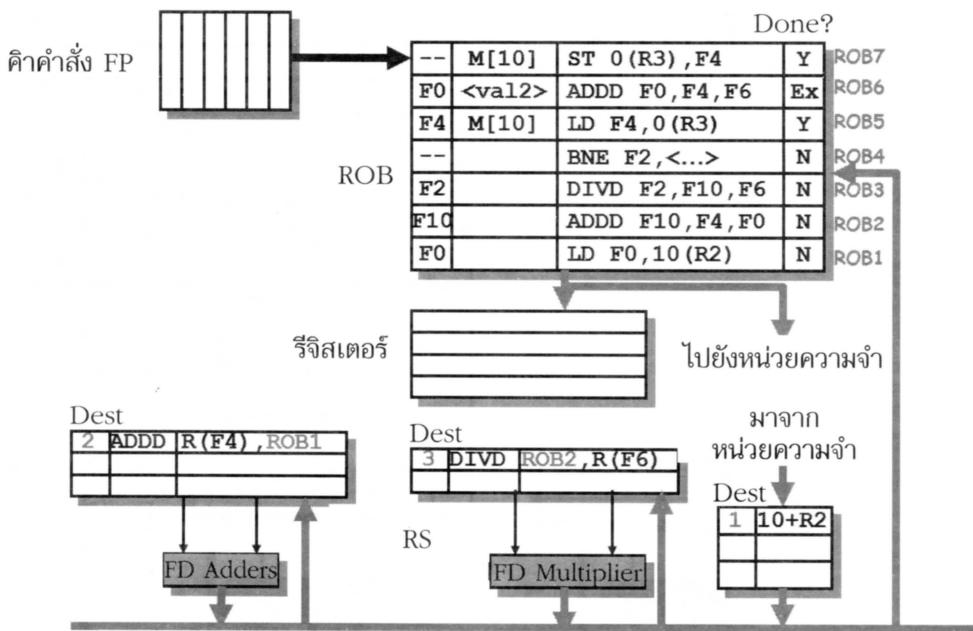
จากนั้นคำสั่ง ADD จะเข้าไปยัง ROB และเข้าไปยัง RS ของ FP Adder ก็จะระบุว่าต้องการโอเปอเรนเดอร์จาก ROB1 เพราะต้องการ F0

เมื่อ DIVD เข้าไปยัง ROB และเข้าไปยังค่าว RS 例外ของ FP MULT ระบุว่าต้องการโอเปอเรนเดอร์จาก ROB2 (F10) ต่อมาเมื่อคำสั่ง BNE เข้าไป คำสั่งถัดมา ได้แก่ LD และ ADDD ซึ่งจะถูกยกเว้น Speculative เพราะยังไม่แน่ใจว่า Branch จะกระโดดจริงหรือไม่ คำสั่ง LD ใช้ RS 例外ที่ 2 และคำสั่ง ADDD ใช้ RS 例外ที่ 2 ระบุว่าต้องเขียนผลลัพธ์ไปยัง ROB 例外ที่ 6 และต้องการโอเปอเรนเดอร์จาก例外 ROB5



รูปที่ 7.8 เมื่อ BNE เข้าไปแล้ว LD เข้าไปอีกรอบ

เวลาต่อมาคำสั่ง ST ของรอบต่อไปก็เข้าไปใน ROB และต่อมาคำสั่ง LD และ ST ก็ทำงานเสร็จ (Done เป็น Yes) และต่อมา ADDD ในแกวที่ 6 ก็ทำการอีกชีวิต ดังแสดงในรูปที่ 7.9



รูปที่ 7.9 ADDD ทำงานเสร็จใน ROB

แต่สังเกตว่าเมื่อ LD, ST กำลังจะ Commit อาจจะเกิด Structure Hazard ตรงหน่วยความจำ เนื่องจากการเขียนไปยังหน่วยความจำและอ่านจากหน่วยความจำต้องใช้หน่วยความจำหน่วยเดียวกัน

ในตัวอย่างของ Tomasulo นั้นจำเป็นต้องหลีกเลี่ยงการเกิด Conflict จากการใช้งานหน่วยความจำ การปรับค่าในหน่วยความจำนั้นต้องทำตามลำดับของการเกิดขึ้น ถ้าคำสั่ง SW อยู่ที่ต้นคิวของ ROB คำสั่งนี้จะทำให้เกิดการเปลี่ยนแปลงสถานะในหน่วยความจำก่อน ซึ่งเป็นการแก้ปัญหา WAR และ WAW สำหรับ Tomasulo

สำหรับ RAW นั้น ใน Tomasulo นี้จะแก้โดยพิจารณาจากคำสั่ง LW โดยไม่ให้คำสั่งเริ่มทำงาน ถ้าคำสั่ง SW ก่อนหน้าที่ไปยังตำแหน่งเดียวกันยังทำงานไม่เสร็จสิ้น และจะรักษาลำดับของการทำงานไม่ได้การทำงานของ LD จะเป็นผลจากคำสั่ง SW ก่อนหน้านี้

สำหรับวิธีการ Precise Interrupt/Exception นั้น จะอ้างอิงหลักการการทำงานเสร็จสิ้นตามลำดับ (In-Order Completion) และการ Commit ตามลำดับ (In-Order Commit) กล่าวคือ จะให้คำสั่งแต่ละคำสั่งเสร็จตามลำดับการทำงาน และจะ Commit ตามลำดับเข่นกัน ในการ

จัดการ Exception ก็จะต้องจัดการตามลำดับของคำสั่งที่ทำงานที่เกิด Exception ขึ้น ถ้าคำสั่งใน ROB เกิด Exception หนึ่ง Exception ก็จะอยู่ใน ROB ตามลำดับเช่นกัน

■ 7.6 VLIW และ Superscalar

วิธีการอื่นๆ ที่จะทำให้ลด CPI ให้น้อยลงกว่า 1 นั้น อาจจะต้องใช้เทคนิคแบบ Multiple-Issue ซึ่งใช้วิธีการต่อไปนี้ เช่น

1. ใช้การจัดลำดับแบบ Static ในโปรเซสเซอร์แบบ Superscalar
2. ใช้การจัดลำดับ Dynamic ในโปรเซสเซอร์แบบ Superscalar
3. ใช้โปรเซสเซอร์แบบ VLIW (Very Long Instruction Word)

ในวิธีการของ Superscalar Processor นั้นจะอาศัยการ Issue คำสั่งมากกว่า 1 ในหนึ่งไบเกลซึ่งอาจจะเป็นจำนวนไม่คงที่ได้ใน 1 ไบเกล ทำให้เกิดการทำงานแบบ In-Order สำหรับวิธีจัดลำดับแบบ Static และอาจจะเกิดการทำงานที่ทำงานแบบ Out-of-Order ถ้าใช้วิธีจัดลำดับแบบ Dynamic แต่ในวิธีการของ VLIW Processor นั้นจะใช้จำนวนคำสั่งที่คงที่และรูปแบบที่แน่นอนซึ่งจะถูก Pack รวมกันอยู่ ซึ่งจะบ่งบอกถึงตัวรีช่องการทำงานแบบบนงานเดียว (หลักการนี้ใช้ใน Intel/ HP Itanium เป็นต้น)

ความหมายของ VLIW นั้น จะทำการรวมคำสั่งเข้าไว้ด้วยกันเรียกว่าเป็น Packet ใน IA-64 หรือเรียกว่า Molecule ใน Transmeta วิธีการนี้จะต้องพิจารณาข้อตี-ข้อเสียระหว่างขนาดของโค้ดและการถอดรหัสเอง การรวมคำสั่งหลายๆ คำสั่งนั้นทำให้คำสั่งหนึ่งมี Packet ยาว ต้องการตัวถอดรหัสที่ขับขอนกว่า คำสั่งเหล่านั้นจะทำงานรวมกันแบบบนงานซึ่งหมายถึงว่าคำสั่งเหล่านั้นต้องไม่มี Dependency ต่อกันในรูปแบบใดๆ เช่น ถ้าคำสั่งใน Packet หนึ่งๆ ประกอบด้วย

- คำสั่งแบบ Integer จำนวน 2 คำสั่ง
- คำสั่งแบบทศนิยมจำนวน 2 คำสั่ง
- คำสั่งเกี่ยวกับหน่วยความจำจำนวน 2 คำสั่ง
- คำสั่ง Branch จำนวน 1 คำสั่ง

ดังนั้น ใน 1 Packet นี้จะมีห้องหมุด 7 คำสั่ง และถ้าต้องใช้ 16-24 บิตต่อ 1 พิล็อต ห้องหมุด 1 Packet รวมกันจะใช้ 112-168 บิต นอกจากนี้ แล้วยังต้องการตัวแปลภาษาที่สนับสนุนโดยเลือกคำสั่งมาร่วมเป็น Packet เดียวกันเพื่อให้เกิดการทำงานแบบขนาดน้อย ในกรณีนี้อาจต้องอาศัยการ Unroll มาช่วยเพื่อขยายโค้ดและเพิ่มคำสั่งทางเลือกที่จะหยิบมาใส่ใน Packet และอาจจะต้องพิจารณาคำสั่งจากหลายๆ บล็อกในกรณีที่มีคำสั่ง Branch หลายคำสั่งเพื่อนำมาร่วมเข้าด้วยกัน

ดังตัวอย่างเช่น

```

1. Loop: L.D      F0,0(R1)
2.       L.D      F6,-8(R1)
3.       L.D      F10,-16(R1)
4.       L.D      F14,-24(R1)
5.       ADD.D    F4,F0,F2
6.       ADD.D    F8,F6,F2
7.       ADD.D    F12,F10,F2
8.       ADD.D    F16,F14,F2
9.       S.D      F4,0(R1)
10.      S.D     F8,-8(R1)
11.      S.D     F12,16(R1) ;32 - 16
12.      DADDUI  R1,R1,#--32
13.      BNE     R1,R2,Loop ;branch R1 != R2
14.      S.D     F16,8(R1) ;8 - 32

```

ในโค้ดนี้จะถูกขยายลูปออกมา 4 รอบ และจะใช้เวลา 3.5 ไบเกิลต่อ 1 รอบ ถ้านำ VLIW มาใช้ และใช้รูปแบบ Packet ข้างต้น ต้องขยายลูปออกมา 7 รอบ จัดลำดับโค้ดได้ดังข้างล่าง

คำสั่งเกี่ยวกับ หน่วยความจำ 1	คำสั่งเกี่ยวกับ หน่วยความจำ 2	คำสั่ง FP 1	คำสั่ง FP 2	คำสั่ง Integer/Branch
LD F0, 0 (R1)	LD F6, -8 (R1)			
LD F10, -16 (R1)	LD F14, -24 (R1)			
LD F18, -32(R1)	LD F22, -40 (R1)	ADDD F4, F0, F2		ADDD F8, F6, F2
LD F26, -48 (R1)		ADDD F12, F10, F2		ADDD F16, F14, F2
		ADDD F20, F18, F2		ADDD F24, F22, F2
SD 0 (R1), F4	SD -8 (R1), F8	ADDD F28, F26, F2		
SD -16 (R1), F12	SD -24 (R1), F16			SUBI R1, R1, #56
SD 24 (R1), F20	SD 16 (R1), F24			BNEZ R1, Loop
SD 8 (R1), F28				

จะเห็นว่าแต่ละແຕງແສດງถึงว่าในแต่ละໄชเกิลใช้ตัวดำเนินการอะไรบ้าง บางครั้งอาจจะหาคำสั่งมา Pack ในແຕງเดียวกันไม่ได้ เพราะความสัมพันธ์ในໄชเกิลที่ 3 จากເเลັນທີ່จะເຫັນວ່າໃນຕ້ອງຢ່າງສາມາດทำงานໄດ້ພຽມກັນถึง 4 คำสั่ง ເລັນທີ່ລາກແສດງถึงความສັມພັນຂອງການໃໝ່ງວຽກຈົບຕະຫຼາດທີ່ເກີດຂຶ້ນຈາກຄວາມສັມພັນຂອງຂໍ້ມູນຕາມໂອເປວແນຣດ (F0 กັບ F0 และ F4 กັບ F4) ເພື່ອປະກອບກັບຍາຍລຸບອອກມາຈະທຳໃຫ້ໄດ້ 9 ໄชເກີລ ອ້ອງໃໝ່ 1.3 ໄชເກີລຕ່ອຮອບ ຜຶ່ງຕິກວ່າເດີມ 1.8 ເທົ່າ ເທົ່າກັບການໂດຍເຄີ່ຍ 2.5 คำสั่งຕ່ອໄງເກີລ ຈະເຫັນວ່າໃນລັກໝະນະ VLIW ນີ້ຕ້ອງການຈຳນວນວຽກຈົບຕະຫຼາດໃໝ່ງວຽກເພີ່ມຂຶ້ນ

ປັບປຸງຫາຂອງ VLIW ໃນຍຸດແຮກໆ ມີດັ່ງນີ້

- ການເພີ່ມຂອງขนาดໂຄଡ ເພະຕົ້ງການກຳນົດເພື່ອໃໝ່ມີຄຳສັ່ງມາຮວມເຂົ້າດ້ວຍກັນໂດຍການຍາຍລຸບ ແລະ ຄ້າ Packet ຂອງ VLIW ໄນເຕີມ ໝາຍຖື່ງ ຈະມີບີຫົ່ນທີ່ວ່າງໆ ທີ່ຈະເສີຍເປົ່າ ແລະ ທຳໄໝທີ່ມີໜ່ວຍຄໍານວານທີ່ຈະວ່າງງານໃໝ່ໄງເກີລນັ້ນ
- ການทำงานໃນລັກໝະນະ Lock-Step ຕື່ອ ຄ້າມີ Stall ເກີດທີ່ໜ່ວຍຄໍານວານໄດ້ກົດຕາມ ຖຸກຄຳສັ່ງໃໝ່ໄງເກີລເດີຍກັນຕ້ອງຫຼຸດຮອ ເພະຕົ້ງການ Synchronized ກັນ ຜຶ່ງເປັນຄວາມຍາກກັບຕົວແປລກາຫາເອງໃນການກຳນົດການກະລຸນາ ຮຸມທັງກຣັນທີ່ການເກີດແຜ່
- Compatibility ຂອງໃບນາຣີໂຄଡ ເຄື່ອງແບບ VLIW ທີ່ຕ່າງກັນກີ່ຈະໃໝ່ຈຳນວນໜ່ວຍຄໍານວານໃນ Packet ຕ່າງກັນ ທຳໄໝໃບນາຣີໂຄଡໄໝ Compatibile ກັນ

ตัวอย่างของ Intel/HP IA-64 ในบทที่แล้วเป็นรูปแบบคำสั่งแบบ EPIC ประกอบด้วย รีจิสเตอร์จำนวนมากถึง 256 ตัว ซึ่งเป็นรีจิสเตอร์เก็บเลขจำนวนเต็มขนาด 64 บิตจำนวน 128 ตัว และรีจิสเตอร์เก็บเลข浮นิยมขนาด 82 บิตจำนวน 128 ตัว สำหรับ Itanium จะมี 6 คำสั่ง ในหนึ่ง Packet มีไปปีลน์ 10 ขั้น และสำหรับ Itanium 2 มี 6 คำสั่งใน Packet และมีไปปีลน์ 8 ขั้น เป็นต้น

ในการรองรับ VLIW ฮาร์ดแวร์จะต้องสามารถเฟตช์คำสั่งหลายๆ คำสั่งพร้อมกัน กล่าวคือ จะต้องเพิ่มแบนด์วิดธ์ในการเฟตช์คำสั่งด้วย ซึ่งต้องอาศัยการทำงานทำนายตำแหน่งของคำสั่งที่จะใช้งาน และส่งตำแหน่งนั้นมาเพิ่ม โดยต้องมีการเฟตช์ล่วงหน้า อันนี้อาศัยเทคนิคของ BTB เข้าช่วยในการทำงาน ดังได้กล่าวถึงไปแล้วในบทที่ผ่านมา นอกจากนี้อาจมีการใช้บัฟเฟอร์สำหรับการ Prefetch เข้าช่วยร่วมกับการทำงาน และมีบัฟเฟอร์หรือแคชในการเก็บคำสั่งที่ทำการ Prefetch เข้ามา

เทคนิคการทำงานยื่นๆ ได้แก่ วิธี Value Prediction เป็นการทำงานค่าที่จะได้เป็นผลลัพธ์ของคำสั่ง เช่น ในกรณีของ Load ที่อาจจะไม่ได้เปลี่ยนแปลงค่าข้อมูล นอกจากนี้ ยังมีการทำงานใช้ Alias ของตำแหน่งหน่วยความจำเดียวกัน สำหรับกรณีของ RAW ของคำสั่ง LD และ SW หรือกรณี WAW ของคำสั่ง SD สองคำสั่งติดกัน ซึ่งการทำงาน Alias ได้มีโปรเซสเซอร์บางตัว ได้นำวิธีนี้ไปใช้แล้ว

■ 7.7 เครื่องเวกเตอร์ (Vector Machine)

ในการทำงานแบบขนาดลักษณะอื่น ได้แก่ การทำงานแบบเวกเตอร์ซึ่งเป็นแบบไปปีลน์แบบลึกและแต่ละคำสั่งไม่ขึ้นแก่กัน มีความถี่สัญญาณนาฬิกาสูง คำสั่งแต่ละคำสั่งที่ทำงานจะเป็นแบบเวกเตอร์ ซึ่งมีรูปแบบการใช้หน่วยความจำที่ตายตัว คำสั่งจะไม่มีคำสั่งประเภท Branch เลย ตัวอย่างเช่น การบวกเวกเตอร์เก็บตัวเลข 64 ตัว

ตัววัดประสิทธิภาพของเวกเตอร์ ได้แก่

- R_{∞} ระบุถึงอัตราการทำงานในหน่วย MFLOPS ของเวกเตอร์ขนาดไม่จำกัด ถ้า R_n คือ อัตราการทำงานในหน่วย MFLOPS ของเวกเตอร์ขนาด n
- $N_{1/2}$ หมายถึง ความยาวของเวกเตอร์ที่จะทำให้ได้ค่า R_{∞} ใช้เป็นตัววัดเวลาทำงานเริ่มต้น (Start-Up Time) หรือเวลาที่ทำให้หน่วยคำนวณเวกเตอร์เต็ม

- N_v หมายถึง ความยาวของเวกเตอร์ที่จะทำให้คำสั่งรูปแบบเวกเตอร์นั้นทำงานเร็วกว่าแบบ Scalar ใช้เป็นตัววัด Start-Up Time และการทำงานของตัวเลข Scalar เทียบกับเวกเตอร์

ในการวัดเวลาการทำงานของเวกเตอร์จะขึ้นกับความยาวของเวกเตอร์ ความสัมพันธ์ของข้อมูล และ Structure Hazard

- Initiation Rate หมายถึงอัตราที่หน่วยคำนวณจะให้งานแต่ละค่าในเวกเตอร์ ให้คิดเท่ากับจำนวน Lane (โดยทั่วไปมี 1-2 Lane)
- Convoy คือจำนวนคำสั่งแบบเวกเตอร์ที่สามารถเริ่มทำได้ในໄสเกิลเดียวกันในกรณีไม่มี Structure Hazard หรือ Data Hazard
- Chime คือเวลาที่ใช้ในการทำงานสำหรับหนึ่งการดำเนินการแบบเวกเตอร์ (Vector Operation)

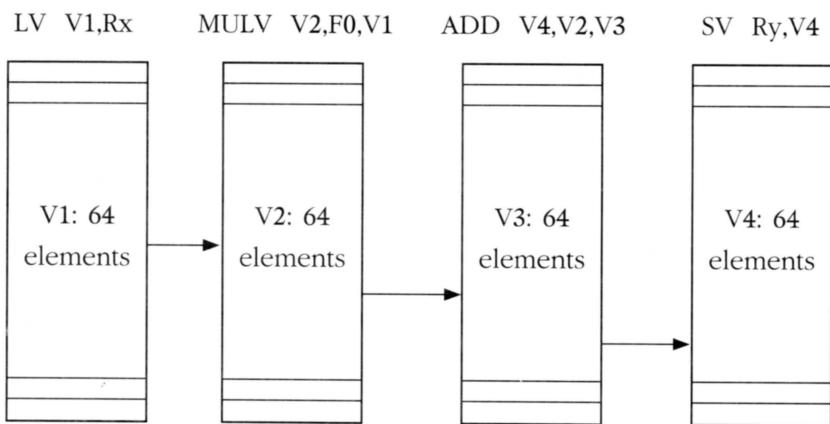
โดยทั่วไปแล้ว จำนวน m Convoy จะใช้เวลา m Chime ถ้าความยาวของเวกเตอร์เท่ากับ n จะใช้เวลาประมาณ $m \times n$ ໄสเกิล

ในตัวอย่างข้างล่าง

LV	<u>V1</u> , Rx
MULV	V2, F0, <u>V1</u>
LV	V3, Ry
ADDV	V4, V2, V3
SV	Ry, V4

ในโค้ด V1 ถูกส่งข้อมูลจากคำสั่ง LV ไปยัง MULV และ V2 ถูกส่งจากคำสั่ง MULV ไปยังคำสั่ง ADDV และ V4 ถูกส่งจากคำสั่ง ADDV ไปยังคำสั่ง SV

ในกรณีกำหนด 1 Lane, 4 Convoy, Vector Length เท่ากับ 64 จะได้ $4 \times 64 = 256$ ໄสเกิล หรือใช้เวลาในการคำนวณ 4 ໄสเกิลต่อ 1 ผลลัพธ์ ดังแสดงในรูปที่ 7.10



รูปที่ 7.10 การทำงานแบบเวกเตอร์ตามตัวอย่างโอดด์

ในการจัดการคำสั่งแบบ LD และคำสั่ง SW สามารถใช้เทคนิคต่างๆ มาช่วย เพื่อให้สามารถเข้าถึงหน่วยความจำได้พร้อมกันหลาย Word ในรอบเดียวกัน เช่น การใช้ Unit Stride คือเข้าถึงเป็นพื้นที่ที่ติดกันเป็นแบบ Non-Unit Stride ใช้หลักการของ Data Bank มาช่วย หรือใช้วิธีการ Indexed (ในลักษณะเดียวกับการ Scatter/Gather ข้อมูล ใช้หลักการเดียวกับ Sparse Matrix) ตัวอย่างของการใช้ Interleaving แสดงดังรูปที่ 7.11



รูปที่ 7.11 ตัวอย่างการ Interleave ของบล็อกในหน่วยความจำ

ในรูปที่ 7.11 เลือกใช้การ Modulo 4 แต่อาจจะใช้เลขจำนวนเฉพาะในการแบ่ง Bank ของหน่วยความจำเพื่อให้การเข้าถึงหน่วยความจำมีประสิทธิภาพ

โดยทั่วไปแล้ว ถ้าเบรี่ยบเทียบเครื่องแบบเวกเตอร์กับเครื่องแบบทั่วไปนั้นมีความแตกต่าง ดังแสดงในตารางด้านล่าง

Scalar	Vector
จำนวนตัวดำเนินการเท่ากับ N ต่อ 1 รอบ ขนาดวงจรเท่ากับ $O(N^2)$	จำนวนตัวดำเนินการเท่ากับ N ต่อ 1 รอบ ขนาดวงจรเท่ากับ $O(N + eN^2)$

ในปัจจุบันมีเครื่องเวกเตอร์ใหม่ๆ ได้แก่ CrayX1 ใช้สถาปัตยกรรมแบบ MIPS และ NEC Earth Simulator ทำงานได้ถึง 40 TFLOPs มี CMOS เวกเตอร์ ให้ลดจำนวน 640 โหนด

นอกจากนี้ ในการทำงานแบบขนาดนั้น ถ้าเป็นรูปแบบเวกเตอร์ที่เรียกว่าเป็น SMP เป็นแบบ มีการใช้หน่วยความจำร่วมกันแบบ Shared Memory แต่ยังมีการทำงานแบบขนาดรูปแบบอื่นๆ อีก เช่น แบบหน่วยความจำแบบกระจาย เป็นต้น

■ 7.8 ส魯ป

นอกจากการทำงานแบบไปป์ไลน์พื้นฐานแล้ว ในคอมพิวเตอร์ทั่วไปซึ่งใช้หน่วยคำนวณหลายหน่วย อาจจะใช้หลายไปป์ไลน์ ในบทนี้ก็ถว่าถึงเทคนิคการทำงานแบบขนาดขั้นสูง ดังแต่การใช้หลายหน่วยคำนวณ การทำจัดลำดับทั้งแบบ Static และ Dynamic เทคนิคการขยายลูป และตัวอย่างการคำนวณเวลาที่ใช้ต่อรอบ รวมทั้งกล่าวถึงวิธีการ Tomasulo ทั้งแบบ Speculative ซึ่งเป็นการคาดคะเนในรูปแบบต่างๆ ลักษณะการทำงานแบบ Superscalar และ VLIW การทำงานของเครื่องเวกเตอร์ ซึ่งเป็นพื้นฐานในการทำงานแบบขนาดขั้นสูงทั้งสิ้น

คำถ้ามท้ายบท

1. การทำงานแบบหลายๆ หน่วยคำนวนเป็นอย่างไร มีข้อดีอย่างไร
2. Latency คืออะไร ต่างจาก Initial Interval อย่างไร
3. Anti-Dependence และ Output Dependence ต่างกันอย่างไร
4. Tomasulo แบบมี ROB ต่างจากแบบไม่มี ROB อย่างไร
5. Chrime และ Convoy คืออะไร
6. Superscalar ต่างจาก VLIW อย่างไร
7. จงยกตัวอย่างโค้ดที่ทำงานแบบเวกเตอร์ได้
8. พิจารณาโค้ดต่อไปนี้

```
for (i = 0; i < 100 ;i++)
    A[i] = B[i] - C[i];
```

- 8.1 จงเขียนโค้ดตั้งกล่าวใน MIPS และขยายลูปมา 2 รอบ คำนวนหาจำนวนนี้เกิลที่ใช้ต่อรอบ
- 8.2 จงจัดลำดับโค้ดเพื่อลดจำนวนนี้เกิลที่ใช้ต่อรอบ อาจจะมีการขยายลูปเพิ่มขึ้น
- 8.3 จงจัดลำดับโค้ดสำหรับเครื่อง Superscalar ที่มีหน่วย FP Adder จำนวน 2 ตัว และหน่วย Memory Unit จำนวน 2 ตัว
9. จงอธิบายว่าการขยายลูปให้ประโยชน์อย่างไร และมีข้อเสียอย่างไร

10. จงแปลงโค้ดในข้อที่ 8 เป็นแบบเวกเตอร์
11. จงทดลองใช้ได้ดีในข้อที่ 8 ทำงานโดยอาศัยการทำงานแบบ Tomasulo

