CA 指令集并行

1 静态指令流调度

1.1 循环展开


```
for (i=1; i<=1000; i++)
        x(i) = x(i) + s;
```

Loop:	LD	F0,0(R1)	;F0=vector element
	ADDD	F4,F0,F2	;add scalar from F2
	SD	0(R1),F4	;store result
	SUBI	R1,R1,8	;decrement pointer 8B (DW)
	BNEZ	R1,Loop	;branch R1!=zero
	NOP		;delayed branch slot

```
1 Loop:  LD    F0,0(R1)     ;F0=vector element
2        stall
3        ADDD  F4,F0,F2     ;add scalar in F2
4        stall
5        stall
6        SD    0(R1),F4     ;store result
7        SUBI  R1,R1,8      ;decrement pointer 8B (DW)
8        stall               ;
9        BNEZ  R1,Loop      ;branch R1!=zero
10       stall               ;delayed branch slot
```

IF ID EX MEM

Stall → IF ID

Branch 使用R1

产生结果的指令	使用结果的指令	所需延时
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

10 clocks: 是否可以通过调整代码顺序使stalls减到最小

```
1 Loop:  LD      F0,0(R1)
2        SUBI    R1,R1,8
3        ADDD    F4,F0,F2
4        stall
5        BNEZ    R1,Loop ;delayed branch
6        SD      8(R1),F4      ;altered when move past SUBI
```

Swap BNEZ and SD by changing address of SD

6 clocks: 通过循环展开4次是否可以提高性能?

```
1 Loop:  LD      F0,0(R1)
2        LD      F6,-8(R1)
3        LD      F10,-16(R1)
4        LD      F14,-24(R1)
5        ADDD    F4,F0,F2
6        ADDD    F8,F6,F2
7        ADDD    F12,F10,F2
8        ADDD    F16,F14,F2
9        SD      0(R1),F4
10       SD      -8(R1),F8
11       SUBI    R1,R1,#32
12       SD      16(R1),F12
13       BNEZ    R1,LOOP
14       SD      8(R1),F16       ; 8-32 = -24
```

- 代码移动后
  - SD移动到SUBI后，注意偏移量的修改
  - Loads移动到SD前，注意偏移量的修改

14 clock cycles, or 3.5 per iteration

(True) 数据相关（Data dependencies） RAW

另一种相关称为名相关（ name dependence） :
两条指令使用同名参数(register or memory location) 但不交换数据 WAR, WAW

最后一种相关称为控制相关(control dependence)

循环间相关（3/4）-循环变换

OLD:

```
for (i=1; i<=100; i=i+1) {
        A[i] = A[i] + B[i];    /* S1 */
        B[i+1] = C[i] + D[i];} /* S2 */
```

S2与下一次S1放一起

NEW:

```
 A[1] = A[1] + B[1];
 for (i=1; i<=99; i=i+1) {
        B[i+1] = C[i] + D[i];
        A[i+1] = A[i+1] + B[i+1];
 }
        B[101] = C[100] + D[100];
```

循环间无相关

循环间相关（4/4）-Dependence Distance

通常循环间相关呈现为递推关系

```
for (i=1; i<N; i++) A[i] = A[i-1] + B[i];
```

相关的距离可能大于1

```
for (i=4; i<N; i++) A[i] = A[i-4] + B[i];
```

可以通过循环展开增加循环内的并行性

```
for (i=4; i<N; i=i+4)
{    A[i]  = A[i-4] + B[i];
     A[i+1] = A[i-3] + B[i+1];
     A[i+2] = A[i-2] + B[i+2];
     A[i+3] = A[i-1] + B[i+3];
 }
```

2　记分牌策略（动态指令流调度）

允许乱序执行（out-of-order execution） => 乱序完成（out-of-order completion）

记分牌技术要点(2/2)

(无方向路径)

- Out-of-order completion => WAR, WAW hazards?
- WAR: 一般解决方案 （在不采用寄存器重命名的情况下）
  - 对操作排队
  - 仅在读操作数阶段读寄存器
- WAW: 检测到相关后，停止发射前一条指令，直到前一条指令完成
- 提高效率的前提：需要有多条指令进入执行阶段=>必须有多个执行部件或执行部件是流水化的
- 记分牌保存相关操作和状态
- 指令执行过程：IF, ISSUE, RO, EX, WR

记分牌控制的四阶段(1/2)

IF取指 → 放入指令窗口（大小？）

- 1.Issue—指令译码，检测结构相关
  - 准入条件：顺序发射，并且没有结构相关，没有WAW相关
  - 如果当前指令所使用的功能部件空闲，并且没有其他活动的指令使用相同的目的寄存器 (WAW)，记分牌发射该指令到功能部件，并更新记分牌内部数据，如果有结构相关或WAW相关，则该指令的发射暂停，并且也不发射后继指令，直到相关解除。

- 2. Read operands—没有数据相关时，读操作数
  - 准入条件：没有RAW相关。
  - 如果先前已发射的正在运行的指令不对当前指令的源操作数寄存器进行写操作，或者一个正在工作的功能部件已经完成了对该寄存器的写操作，则该操作数有效。操作数有效时，记分牌控制功能部件读操作数，准备执行。
  - 记分牌在这一步动态地解决了RAW相关，指令可能会乱序执行。

- **3.Execution—取到操作数后执行 (EX)**
  - **准入条件：** RO后准入
  - 接收到操作数后，功能部件开始执行
  - 当计算出结果后，通知记分牌，可以结束该条指令的执行。

RO 一进入就可以开始 EX

- **4.Write result—finish execution (WR)**
  - **准入条件：** 没有WAR相关
  - 一旦记分牌得到功能部件执行完毕的信息后，记分牌**检测WAR相关**，如果没有WAR相关，就写结果，如果有WAR 相关，则暂停该条指令。
  - Example:

```
DIVD   F0,F2,F4
ADDD   F10,F0,F8
SUBD   F8,F8,F14
```

  - CDC 6600 scoreboard 将暂停 SUBD 直到ADDD 读取操作数后，才进入WR段处理。

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21	61	62
ADDD	F6	F8	F2	13	14	16	22

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

基本假设：
1、5个FU
2、Load - 1个Cycle
Multi - 10个Cycle
Add - 2个Cycle
Divide - 40个Cycle

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
62	FU									

- 顺序issue; 乱序 execute & 乱序 commit
- 问题：Branch指令怎么办?

3 Tomasulo算法

Tomasulo Algorithm vs. Scoreboard

- **控制和缓存：分布在各部件中 vs. 集中在记分牌**
  - FU 缓存称“Reservation Stations”; 保存待用操作数
- **寄存器重命名：Tomasulo 有 vs. Scoreboard无**
  - 指令中的寄存器在RS中用寄存器值或指向RS的指针代替（称为register renaming）
  - 避免 WAR, WAW hazards
- **定向路径：Tomasulo 有 vs. Scoreboard无**
  - 传给FU的结果从RS来而不是从寄存器来
  - FU的计算结果通过Common Data Bus 以广播方式发向所有功能部件
- **控制相关处理：Tomasulo分支可跨越 vs. Scoreboard不可跨越**
  - 可以跨越分支，允许FP操作队列中FP操作不仅仅局限于基本块
- **Load和Store部件也看作带有RS的功能部件**

投机、回滚

Tomasulo 算法的三阶段

- **1. Issue—从FP操作队列中取指令**
  - 如果RS空闲(no structural hazard)，则控制发射指令和操作数(renames registers). **消除WAR，WAW相关**
- **2. Execution—operate on operands (EX)**
  - 当两操作数就绪后，就可以执行
    如果没有准备好，则监测Common Data Bus 以获取结果。通过推迟指令执行**避免RAW相关**
- **3. Write result—finish execution (WB)**
  - 将结果通过Common Data Bus传给所有等待该结果的部件;
    标识RS可用
- **数据通信：功能部件产生结果的传送**
  - 通常的数据总线: data + destination （“go to” bus)
  - Common data bus: data + source（“come from” bus)
    - 64 bits 数据线 + 4 bits 功能部件源地址（FU source address)
    - 产生结果的部件如果与RS中等待的部件匹配，就接收数据
    - 广播方式传送

注意：Load操作在EXE阶段分两步

2、Execute

FP Operation

wait until: (RS[r].Qj=0) and (RS[r].Qk=0)

Action or bookkeeping:

computer result: Operands are in Vj and Vk

Load-store step1 地址计算顺序读出

wait until: RS[r].Qj =0 & r is head of load-store queue

Action or bookkeeping:

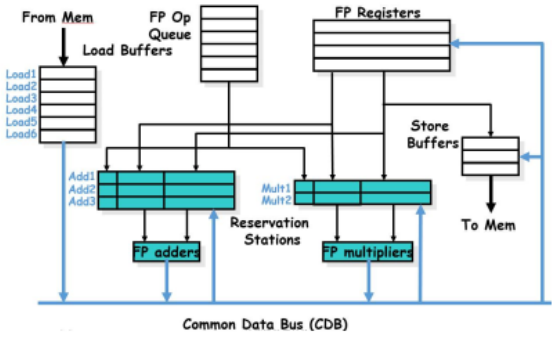
RS[r].A ← RS[r].Vj + RS[r].A;

Load step2

wait until: Load Step1 complete

Action or bookkeeping:

Read from Mem[RS[r].A]



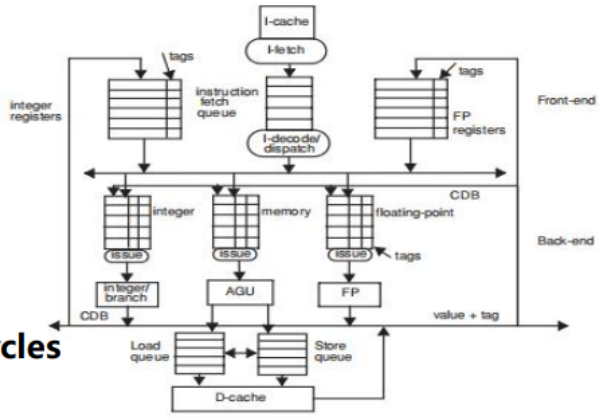
- **为什么scoreboard/6600所需时间较长?**
  - 结构冲突
  - WAR,WAW冲突
  - 没有定向技术

Tomasulo v. Scoreboard (IBM 360/91 v. CDC 6600)

流水化的功能部件	多个功能部件
(6 load, 3 store, 3 +, 2 x/÷)	(1 load/store, 1 + , 2 x, 1 ÷,...)
指令窗口大小: 较大	较小
有结构冲突时不发射	有结构冲突时不发射
WAR: 用寄存器重命名避免	stall 来避免
WAW:用寄存器重命名避免	停止发射
从FU广播结果写寄存器方式	
Control: RS集中式scoreboard	

Tomasulo Loop Example

Loop:	LD	F0, 0(R1)
	MULTD	F4, F0, F2
	SD	F4, 0(R1)
	SUBI	R1,R1,#8
	BNEZ	R1 Loop



- 假设循环3次，设Multiply执行阶段4 cycles
- 访存操作分为计算地址和访存两阶段
  - 计算地址需1个cycle
  - 第1次load时Cache未命中，访存需7个cycles (cache miss)，第2次以后均命中，访存操作需1cycles
- 访存顺序的约定：
  - 所有访存指令1个计算地址队列，实际访存操作时分为load队列和store队列
  - 计算访存地址按序，Load操作之间按序，Store操作按序
  - Load访存操作如果与store访存操作没有冲突，可以先行
- 为清楚起见，下面我们也列出SUBI, BNEZ的时钟周期

Summary-Loop Example

ITER	Inst.	i	j	k	Issue	Exec-start	Exec-End	Cache	WR(CDB)
1	LD	F0	0	R1	1	2		3~9	10
1	MULTD	F4	F0	F2	2	11	14		15
1	SD	F4	0	R1	3	4	-	16	
1	SUB				4				
1	BNEZ				5				
2	LD	F0	0	R1	6	7		11	12
2	MULTD	F4	F0	F2	7	13	16		17
2	SD	F4	0	R1	8	9		18	
2	SUB				9				
2	BNEZ				10				
3	LD	F0	0	R1	11	12		13	14
3	MULTD	F4	F0	F2	16	17	20		21
3	SD	F4	0	R1	17	18		22	
3	....								

本例访存约定：
- 顺序计算访存地址
- 顺序Load访存
- 顺序Store访存
- Load访存可以跨越Store访存先行（Load的地址与Store地址不冲突时）

- TIPS： 不同的存储器访问序的约定会产生不同结果。
- 其他约定?
  - 例如：分别有LoadBuffer和StoreBuffer，但计算地址和实际访存buffer合并；顺序Load访存、顺序Store访存，Load访存可以跨越Store访存先行
  - 例如：简单约定所有访存指令按序执行（计算地址队列和实际访存buffer合并，并且顺序访存）；

- **Reservations stations: 寄存器重命名，缓冲源操作数**
  - 避免寄存器成为瓶颈
  - 避免了Scoreboard中无法解决的 WAR, WAW hazards
  - 允许硬件做循环展开
- **不限于基本块(快速解决控制相关)**
- **贡献**
  - Dynamic scheduling （动态指令流调度）
  - Register renaming （寄存器重命名）
  - Load/store disambiguation （存储器访问歧义消解）

- **动态指令流调度硬件方案可以用硬件进行循环展开**
- **如何处理精确中断?**
  - Out-of-order execution -> out-of-order completion!
- **如何处理分支?**
  - **用硬件做循环展开必须快速解决控制相关问题**

精确中断是指在指令和指令之间如果出现了中断/异常，那么处理器要确保中断/异常之前的所有指令都执行完毕，而中断/异常之后的所有指令都没有执行，然后处理器把中断发生时的处理器状态给保存下来或是呈现给程序员看。

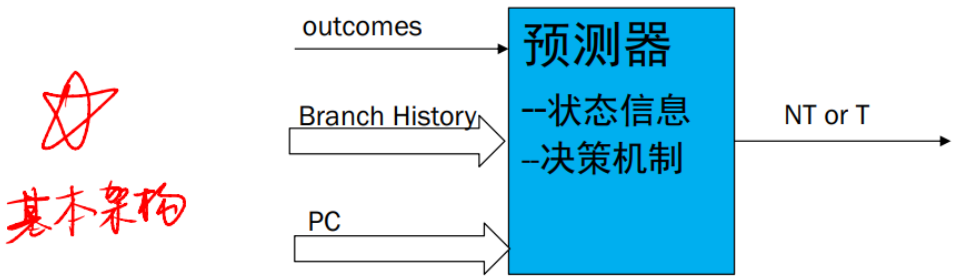
- **乱序完成加大了实现精确异常的难度**
  - 在前面指令还没有完成时，寄存器文件中可能有后面指令的运行结果.
  - 如果这些前面的指令执行时有异常产生，怎么办?
  - 例如：DIVD F10, F0, F2
    SUBD F4, F6, F8
    ADDD F12, F14, F16
- **需要“rollback” 寄存器文件到原来的状态:**
  - 精确异常的含义:
    - 该地址之前的所有指令都已完成
    - 其后的指令还都没有完成
- **实现精确异常的技术：顺序完成（或提交）**
  - 即提交指令完成的顺序必须与指令发射的顺序相同

- **控制相关：**
  - 由条件转移或程序中断引起的相关，也称全局相关。
  - 控制相关对流水线的吞吐率和效率影响相对于数据相关要大得多
    - 条件指令在一般程序中所占的比例相当大
    - 中断虽然在程序中所占的比例不大，但中断发生在程序中的哪条指令，发生在一条指令执行过程中的哪个功能段都是不确定的
- **处理条件转移和异常引起的控制相关的关键问题：**
  - 要确保流水线能够正常工作
  - 减少因断流引起的吞吐率和效率的下降

- **假设对于一条有K段的流水线，由于条件分支的影响，在最坏情况下，每次分支“跳转”将造成k-1个时钟周期的断流。假设条件分支在一般程序中所占的比例为p，采用静态分支预测“不跳转”策略，条件“跳转”的概率为q。试分析分支对流水线的影响。**
- **结论：条件转移指令对流水线的影响很大，必须采取相关措施来减少这种影响。**

$$T = \frac{n}{Kt + (n-1)\Delta t + npq(k-1)\Delta t}$$

预测器的基本结构及输入输出

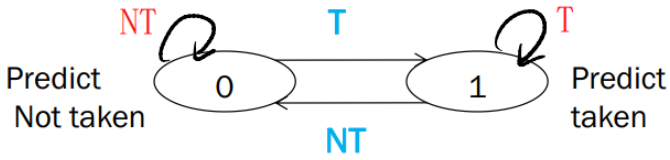


基本架构

- **根据转移历史(和PC)来选择预测器**
- **由预测器的状态决定预测值(输出)**
- **根据实际结果(outcomes)更新预测器的状态信息**

- **动态分支预测：预测分支的方向在程序运行时刻动态确定**
- **需解决的关键问题是：**
  - 如何记录转移历史信息
  - 如何根据所记录的转移历史信息，预测转移的方向（跳转或不跳转）
- **主要方法**
  - 基于BPB(Branch Prediction Buffer)或BHT(Branch History Table)
    - 1-bit BHT和2-bit BHT
    - Correlating Branch Predictors （GAp or PAp）
    - Tournament Predictors: Adaptively Combining Local and Global Predictors
  - High Performance Instruction Delivery（优化取指令带宽）
    - BTB
    - Return Address Predictors
    - Integrated Instruction Fetch Units （单独的取指部件连接到流水线的其他部分，其中集成了分支预测器、指令预取、指令Cache的存取和缓存等）
- **Performance = *f*(accuracy, cost of misprediction)**
  - Misprediction　　Flush Reorder Buffer

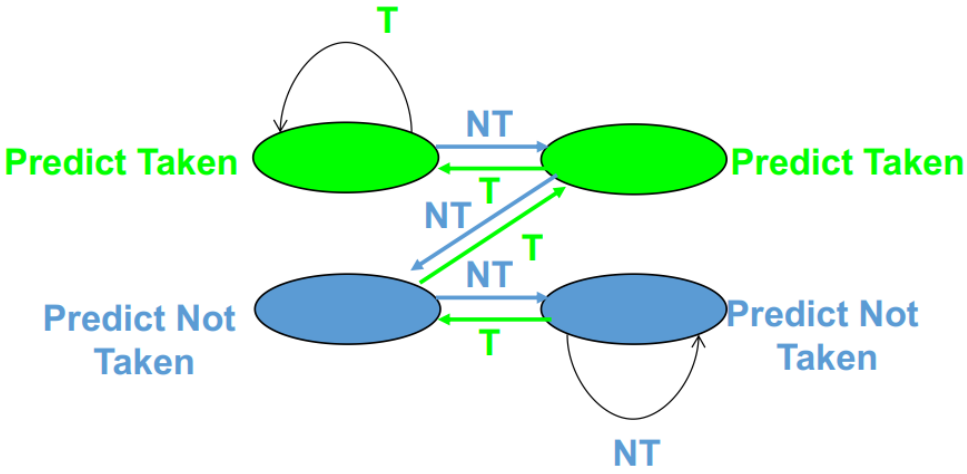
1-bit BHT



- **术语:**
  - Not taken | taken 跳转|不跳转 （成功|失败）
  - 预测准确率（Accuracy），预测错误率(Misprediction)
- **Branch History Table:**
  - 分支指令的PC的低位索引
  - 该表记录上一次转移是否成功
  - 不做地址检查
  - 1-bit BHT

- **问题: 在一个循环中, 1-bit BHT 将导致2次分支预测错误**
  - 假设一循环次数为10次的简单程序段
  - 最后一次循环： 前面预测“跳转”，最后一次需要退出循环
  - 首次循环：前面预测为“不跳转”，这次实际上为成功

2-bit BHT



- **解决办法: 2位记录分支历史**
- **Blue: stop, not taken (不跳转)**
- **Green: go, taken （跳转）**

分支间存在关联的情况举例(1/2)

- 假设d的初始值序列为0，1，2
- b1 如果分支“不跳转”，b2一定也分支“不跳转”。
- 前面基本的1-bit 2-bit预测器都没法利用这一点

→ 两级预测器

```
if (d==0)d=1;
if (d==1) d=0;
```

翻译为汇编指令

```
      BNEZ R1,L1     ;branch b1(d!=0)
      ADDI R1,R0,#1  ;d==0, so d=1
L1:   ADDI R3,R1,#-1
      BNEZ R3,L2     ;branch b2(d!=1)
```

Initial value of d	d==0?	b1	Value of d before b2	d==1?	b2
0	yes	not taken	1	yes	not taken
1	no	taken	1	yes	not taken
2	no	taken	2	no	taken

FIGURE 3.10 Possible execution sequences for a code fragment.

- **假设d的初始值在2和0之间切换。T:“跳转”，NT:“不跳转”**
- **用1-bit预测器，b1和b2的初始设置为预测NT**

```
if (d==0) d=1;
if (d==1) d=0;
```

```
      BNEZ R1,L1     ;branch b1(d!=0)
      ADDI R1,R0,#1  ;d==0, so d=1
L1:   ADDI R3,R1,#-1
      BNEZ R3,L2     ;branch b2(d!=1)
```

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

FIGURE 3.11 Behavior of a one-bit predictor initialized to not taken. T stands for taken, NT for not taken.

- **结论：这样的序列每次预测都错，预测错误率100%。**

Correlating Branches

- **基本思想：记为 （1，1）**
  - 用1位作为correlation位。记录最近一次执行的分支
  - 每个分支都有两个相互独立的预测位：一个预测位假设最近一次执行的分支"不跳转"时的预测位，另一个预测位是假设最近一次执行的分支"跳转"时的预测位。
- **最近一次执行的分支与要预测的分支可能不是同一条指令**

Prediction bits	Prediction if last branch not taken	Prediction if last branch taken
NT/NT	not taken	not taken
NT/T	not taken	taken
T/NT	taken	not taken
T/T	taken	taken

FIGURE 3.12 Combinations and meaning of the taken/not taken prediction bits. T stands for taken, NT for not taken.

Correlating Branches

- **Correlating 预测器的预测和执行情况**
- **显然只有在第一次d=2时，预测错误，其他都预测正确**
- **记为（1，1）预测器，即根据最近一次分支行为来选择一对1-bit预测器中的一个。**
- **更一般的表示为（m, n），即根据最近的m个分支，从2^m个分支预测器中选择预测器，每个预测器的位数为n**

```
      BNEZ R1,L1          ;branch b1(d!=0)
      ADDI R1,R0,#1       ;d==0, so d=1
L1:   ADDI R3,R1,#-1
      BNEZ R3,L2          ;branch b2(d!=1)
```

前一次不跳转看第一个，
前一次跳转看第二个

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	**NT**/NT	T	T/NT	NT/**NT**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T
2	T/**NT**	T	T/NT	NT/**T**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T

FIGURE 3.13 The action of the one-bit predictor with one bit of correlation, initialized to not taken/not taken. T stands for taken, NT for not taken. The prediction used is shown in bold.

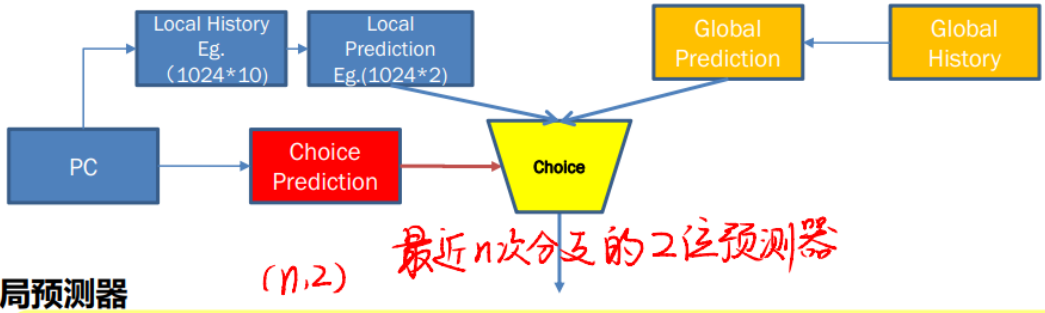
（上）每个分支有自己的预测器

- **关联预测器 也称为 两级预测器**
- **两级预测器的四种组合：** 依据两级的全局或局部属性
  - ①GAg：全局历史表和全局预测表；②GAp：全局历史表和单地址预测表
  - ③PAg：单地址历史表和全局预测器表；④PAp: 单地址历史表和单地址预测器表

Branch Prediction

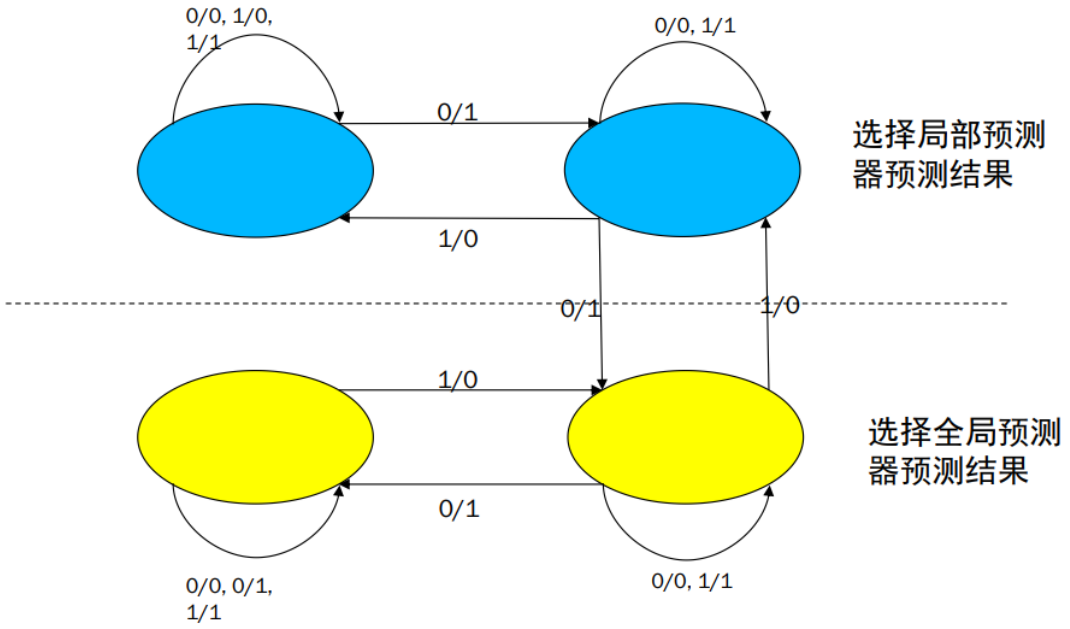
- **Basic 2-bit predictor:**
- **关联预测器(n,2):**
  - 两级全局预测器（GAp）
    - 每个分支有多个 2-bit 预测器
    - 根据**最近n次分支**的执行情况从2^n中选择预测器
  - 两级局部预测器(Local predictor) PAp
    - 每个分支有多个2-bit 预测器
    - 根据**该分支的最近n次分支**的执行情况从2^n中选择预测器
- **竞赛（组合）预测器(Tournament predictor):**
  - 例如：结合两级全局预测器和两级局部预测器

竞赛预测器（举例）



- **全局预测器**
  - 使用最近n次分支跳转情况来索引（2^n个entries），每个Entry是一个标准的2位预测器
- **两级局部预测器** ← 对该分支
  - 一个局部历史记录表（**Local History**）：使用PC的低m位索引（2^m个entries），每个entry有k位，记录该指令最近的k次分支跳转情况
  - 根据Local History选择的entry的k位，索引选择下一级（**Local Prediction**）的entries，这些entries由2位计数器构成，以提供本地预测。
- **选择器：**
  - 使用PC低m位索引，每个索引得到一个两位计数器，用来选择使用局部预测器还是使用全局预测器的预测结果。
  - **在设计时默认使用局部预测器，当两个预测器都正确或都不正确时，不改变计数器；当全局预测器正确而局部预测器预测错误时，计数器加1，否则减1。**

选择器状态转移图



- **局部预测器/全局预测器：**
  - 0/1：局部预测器预测错误，全局预测器预测正确
  - 1/0：局部预测器预测正确，全局预测器预测错误

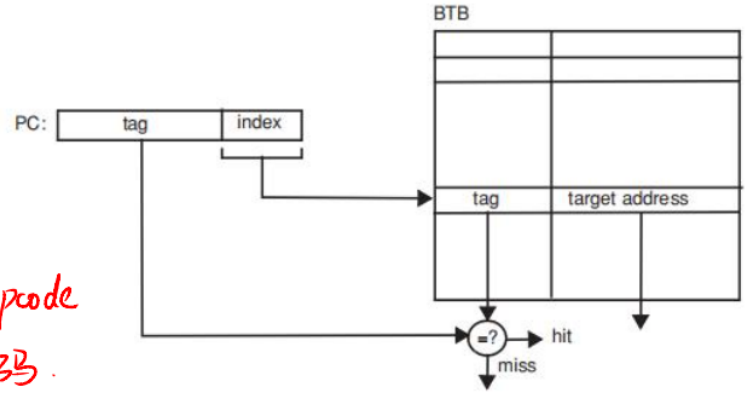
（2，n）（2，2）没看懂啊啊啊啊啊啊啊啊啊啊啊

4.1 BTB（预测目标地址）

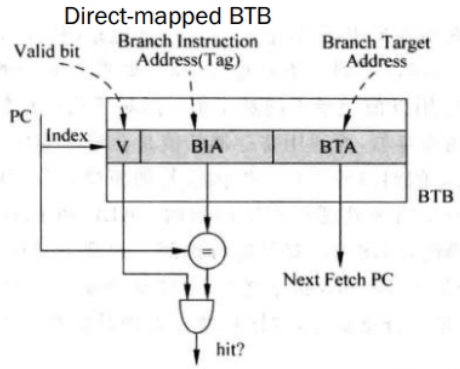
Branch Target Buffer (BTB)

- **BTB 小容量的Cache**
- **分支指令的地址作为BTB的索引，以得到分支预测地址**
  - 必须检测分支指令的地址是否匹配，以免用错误的分支地址
  - 从表中得到预测地址
  - 分支方向确定后，更新预测的PC

分支目标缓存

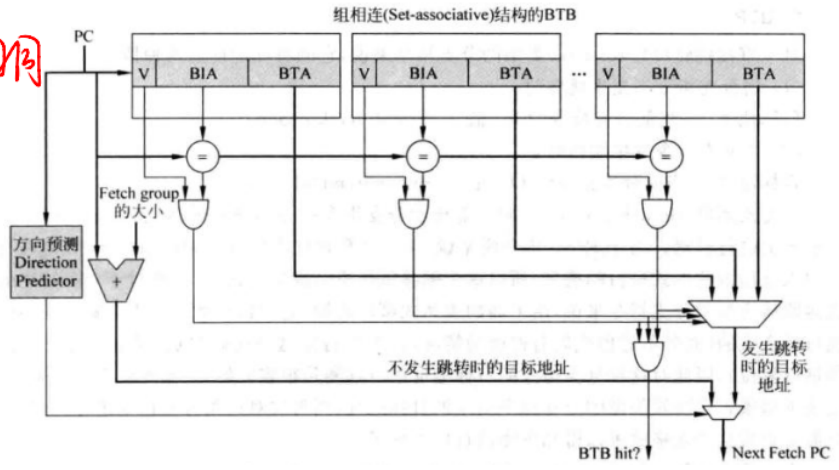


先对PC的opcode 解码. ↓判断 是否为分支



- **BTB本质上是Cache**
- **可以有多种组织方式，代价和性能不同**
  - 直接映像方式
  - 组相联方式
- **面向BTB的Cache组织优化**
  - 例如：缩短Tag的位数 （存储tag的部分位数，或通过运算缩短Tag的位数

和Cache命中方式相同 是否有hit



Branch Penalty：如果在BTB中命中，并且预测正确，则Penalty 为0，其他情况则Penalty为2

Return Address Predictors 举例

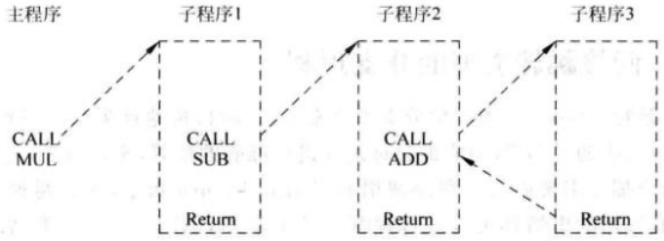


图 4.39 一个三级嵌套的子程序调用

4.40 执行三条 CALL 指令之后，RAS 中的值

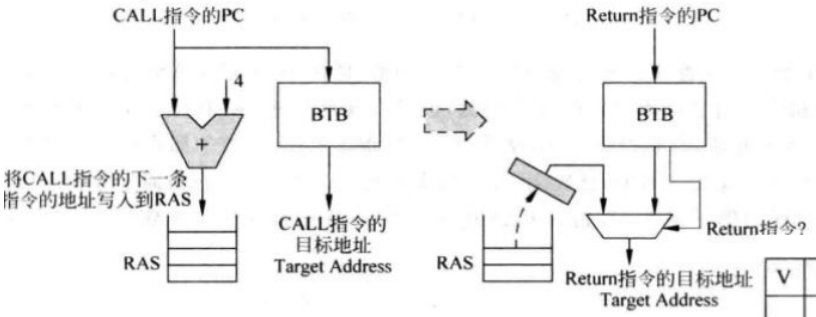


图 4.41 对 CALL/Return 指令进行分支预测

这个和中断服务很像

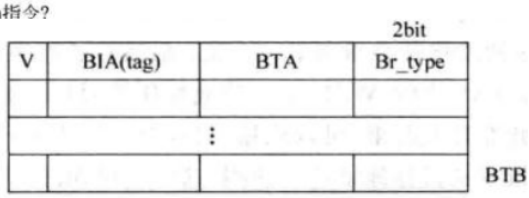


图 4.42 将指令的类型存储在 BTB 中

Tomasulo算法（支持推断执行）

主要差异:

- 增加了Reorder buffer（寄存器重命名）
- 删除了store buffer， 其功能集成在ROB中

硬件支持推断执行以及精确异常

- **支持推断执行的条件：具有“恢复”能力**
- **硬件缓存没有提交的指令结果: reorder buffer (ROB)**
  - 4 个域: 指令类型,目的地址, 值，ready域
  - Reorder buffer 可以作为操作数源 => 就像有更多的寄存器（与RS类似）
  - 当指令执行阶段完成后，用ROB的编号代替RS中的值
  - 增加指令提交阶段（Commit）
  - ROB提供执行完成阶段和提交阶段的操作数
  - 一旦结果提交，结果就写入寄存器
  - 在预测错误时，容易恢复推断执行的指令，或发生异常时，容易恢复状态

支持推断执行的 Tomasulo 算法的四阶段

- **1. Issue—get instruction from FP Op Queue**
  - 如果RS和ROB有空闲单元就发射指令。如果寄存器或ROB中源操作数可用，就将其发送到RS，目的地址的ROB编号也发送给RS
- **2. Execution—operate on operands (EX)**
  - 当操作数就绪后，开始执行。如果没有就绪，监测CDB，检查RAW相关
- **3. Write result—finish execution (WB)**
  - 将运算结果通过CDB传送给所有等待结果的FU以及**ROB单元**，标识RS可用
- **4. Commit—update register with reorder result**
  - 按ROB表中顺序，如果结果已有，就更新寄存器（或存储器），并将该指令从ROB表中删除
  - 预测错误或有异常（中断）时，刷新ROB
  - P191 Figure 3.14 (英文版)，P141 Figure 3-9 (中文版)
- **执行过程中需要检测CDB冲突**

Tomasulo With Reorder Buffer-Summary

Instruction	Issue	Exec Comp	WriteBack	Commit
LD F6, 34(R2)	1	2	3	4
LD F2, 45(R3)	2	3	4	5
MULT F0, F2, F4	3	5~14	15	16
SUBD F8, F6, F2	4	5~6	7	17
DIVD F10, F0, F6	5	16~55	56	57
ADDD F6, F8, F2	6	8~9	10	58

顺序发射、乱序执行、乱序完成、顺序提交

两种Tomasulo算法比较 (三阶段vs.四阶段)

Loop	L.S F0, 0(R1)
	L.S F1, 0(R2)
	ADD.S F2, F1, F0
	S.S F2, 0(R1)
	ADDI R1,R1, #4
	ADDI R2,R2, #4
	SUBI R3,R3,#1
	BNEZ R3, Loop

- **假设：**
  - Load和store部件：计算访存地址 需要 2 cycle；对Cache访问 需要 1个cycle
  - 浮点ADD执行：需要6个cycle
  - Store操作内部分解为两个操作操作：S.S-A 计算访存地址；S.S-D 对Cache访问
  - 其他整型类执行：需要2个cycle

Tomsasulo算法执行示例（无预测）

		Issue	Exe Start	Exe End	Cache	CDB	备注
I1	L.S F0, 0(R1)	1	2	3	(4)	(5)	
I2	L.S F1, 0(R2)	2	3	4	(5)	(6)	
I3	ADD.S F2,F1,F0	3	7	12	---	(13)	等待F1
I4	S.S-A F2, 0(R1)	4	5	6	---	---	
I5	S.S-D F2,0(R1)	5	14	15	(16)	---	等待F2
I6	ADDI R1,R2, #4	6	7	8	---	(9)	
I7	ADDI R2, R2,#4	7	8	9	---	(10)	
I8	SUBI R3, R3, #1	8	9	10	---	(11)	
I9	BNEZ R3, Loop	9	12	13	---	(14)	等待R3的值
I10	L.S F0, 0(R1)	15	16	17	(18)	(19)	等待I9
I11	L.S F1, 0(R2)	16	17	18	(19)	(20)	
I12	ADD.S F2,F1,F0	17	21	26	---	(27)	等待F1

Tomsasulo算法执行示例（有预测）

		Issue	Exe Start	Exe End	Cache	CDB	Commit	备注
I1	L.S F0, 0(R1)	1	2	3	4	(5)	6	
I2	L.S F1, 0(R2)	2	3	4	5	(6)	7	
I3	ADD.S F2,F1,F0	3	7	12	---	(13)	14	等待F1
I4	S.S-A F2, 0(R1)	4	5	6	---	---		
I5	S.S-D F2,0(R1)	5	14	15	16	---	(17)	等待F2
I6	ADDI R1,R2, #4	6	7	8	---	(9)	(18)	
I7	ADDI R2, R2,#4	7	8	9	---	(10)	(19)	
I8	SUBI R3, R3, #1	8	9	10	---	(11)	(20)	
I9	BNEZ R3, Loop	9	14	15	---	(16)	(21)	等待R3的值，若第12拍或第13拍进入EXE段，则WR阶段（CDB争用）分别与I10，I11存在冲突
I10	L.S F0, 0(R1)	10	11	12	13	(14)	(22)	
I11	L.S F1, 0(R2)	11	12	13	14	(15)	(23)	
I12	ADD.S F2,F1,F0	12	16	21	---	(22)	(24)	等待F1

使用ROB保持机器的精确状态

- **ROB维持了机器的精确状态，允许投机（推测）执行**
  - 直到确认无异常 然后进入提交阶段
  - 直到确定分支预测正确进入提交阶段
  - 如果有异常或预测错误
    - 刷新ROB、 RS和寄存器结果状态表
- **存储器操作使用类似的方法**
  - Memory Ordering Buffer （MOB）
    - Store操作的结果先存放到MOB中，然后提交阶段按存储操作的程序序提交

Summary-Tomasulo小结 #1/3

- **Reservations stations: 寄存器重命名，缓冲源操作数**
  - 避免寄存器成为瓶颈
  - 避免了Scoreboard中无法解决的WAR, WAW hazards
  - 允许硬件做循环展开
  - 不限于基本块(快速解决控制相关)
- **Reorder Buffer:**
  - 提供了撤销指令运行的机制
  - 指令以发射序存放在ROB中
  - 指令顺序提交
- **分支预测对提高性能是非常重要的**
  - 推断执行: 在控制相关还没有解决情况下，就开始执行
  - 推断执行利用了ROB撤销指令执行的机制
    - 处理预测错误时，撤销 推测执行的指令
  - 基于BHT的分支预测技术 （预测分支方向）
  - 基于BTB的分支预测技术 （预测分支目标地址）

1 存储器访问冲突05-06（不太看得懂）

Load Ordering. Store Ordering

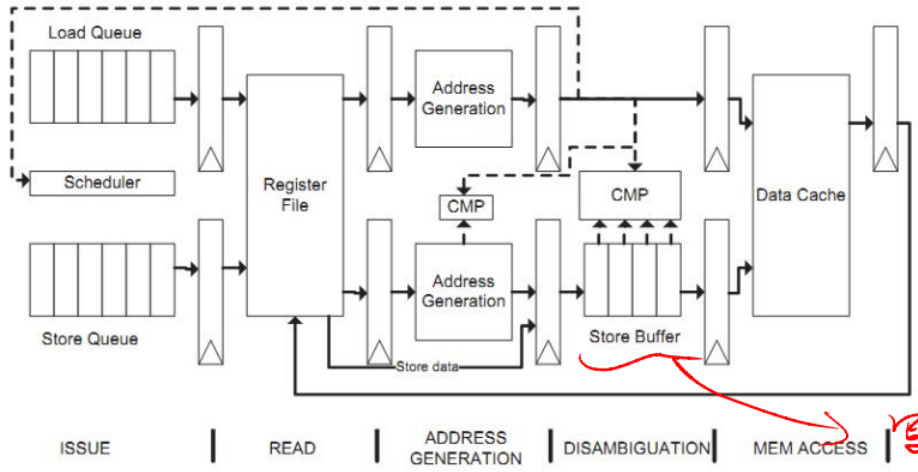


FIGURE 6.7: Schematics of the AMD K6 pipeline to implement a load-ordering store-ordering memory disambiguation policy.

Load queue: 以程序序存储load指令，按照FIFO方式流出
Address generation: 生成存储器访问有效地址
Store queue: 以程序序存储store指令，按照FIFO方式流出
Store buffer: 以程序序保留store操作（有效地址，值），一直到其成为最早的store操作，才实际更新存储器

- **Question: 给定一个指令序列，store，load 这两个操作是否有关？即下列代码是否有相关问题?**

```
Eg:    st  0(R2),R5
       .....
       ld  R6,0(R3)
```

- **我们是否可以较早启动ld?**
  - Store的地址可能会延迟很长时间才能得到.
  - 我们也许想在同一个周期开始这两个操作的执行.
- **两种方法:**
  - No Speculation: 不进行load操作，直到我们确信地址 0(R2) ≠ 0(R3)
  - Speculation: 我们可以假设他们相关还是不相关 (called “dependence speculation”)，如果推测错误通过ROB来修正

Tomsasulo算法执行示例（Total Ordering - 无推断）

		Issue	Exe Start	Exe End	Cache	CDB	备注
I1	LD F0, 0(R1)	1	2	3	（4）	（5）	
I2	MULTD F4, F0, F2	2	6	11	--	（12）	等待F0 （I1）
I3	SD-A, F4, 0(R1)	3	4	5	---	---	
I4	SD-D F4, 0(R1)	4	13	14	(15)	---	等待F4 (I2)
I5	SUBI R1, R1, #8	5	6	7	---	(8)	
I6	BNEZ R1, Loop	6	9	10		(11)	等待R1 (i5)
I7	LD F0, 0(R1)	12	14	15	(16)	(17)	考虑CDB冲突，从14拍开始计算地址
I8	MULTD F4, F0, F2	13	18	23	--	(24)	
I9	SD-A, F4, 0(R1)	14	15	16	---	---	
I10	SD-D F4, 0(R1)	15	25	26	(27)	(28)	
I11	SUBI R1, R1, #8	16	17	18	---	(19)	
I12	BNEZ R1, Loop	17	20	21	---	(22)	

- **Load和store：计算访存地址 2 cycle；对Cache访问 1个cycle**
- **浮点操作执行：6个cycle ； 其他整型类：2个cycle**

Load ording，store ording - 分支预测

		Issue	Exe Start	Exe End	Cache	CDB	commit	备注
I1	LD F0, 0(R1)	1	2	3	（4）	（5）	6	
I2	MULTD F4, F0, F2	2	6	11	--	（12）	13	等待F0 （I1）
I3	SD-A, F4, 0(R1)	3	4	5	---	---	14	
I4	SD-D F4, 0(R1)	4	13	14	(15)	---	16	等待F4 (I2)
I5	SUBI R1, R1, #8	5	6	7	---	(8)	17	
I6	BNEZ R1, Loop	6	11	12		(13)	18	CDB冲突
I7	LD F0, 0(R1)	7	8	9	(10)	(11)	19	
I8	MULTD F4, F0, F2	8	12	17	--	(18)	20	
I9	SD-A, F4, 0(R1)	9	10	11	---	---	21	
I10	SD-D F4, 0(R1)	10	19	20	(21)	(22)	23	等待F4 (I8)
I11	SUBI R1, R1, #8	11	12	13	---	(14)	24	
I12	BNEZ R1, Loop	12	15	16	---	(17)	25	

- **Load和store：计算访存地址 2 cycle；对Cache访问 1个cycle**
- **浮点操作执行：6个cycle ； 其他整型类：2个cycle**

Partial Ordering （MIPS R10000）

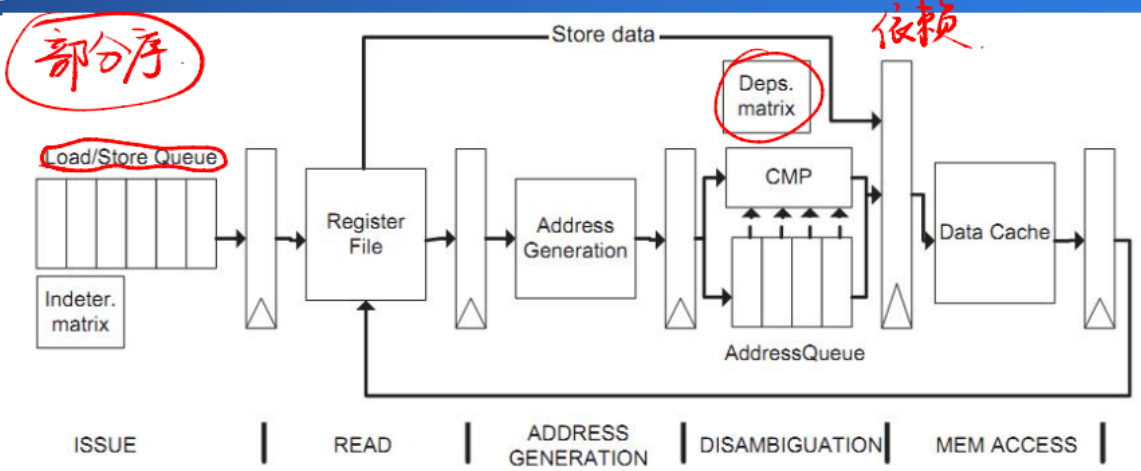


FIGURE 6.8: Schematics of the MIPS R10000 pipeline to implement the partial ordering memory disambiguation policy.

Load/store queue: 长度为16的队列，存储load/store指令，直到其操作数准备好。
Address generation: 计算存储器操作的有效地址

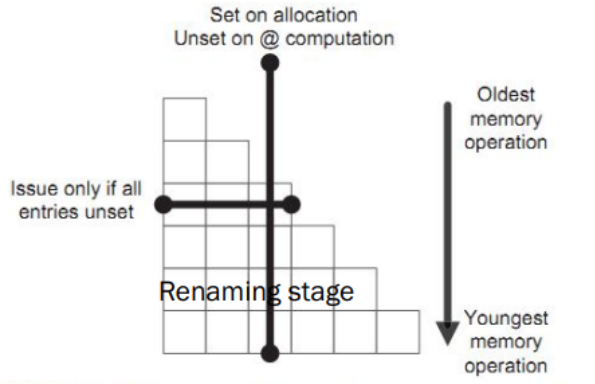


FIGURE 6.9: Example of a 6-entry indetermination matrix.

FIGURE 6.10: Example of a 6-entry dependency matrix.

- **Indetermination matrix: 16x16 矩阵（half），每行每列代表队列中的存储器操作。当有存储器操作进入队列时，对应列置位。当存储器指令计算有效地址时，对应列复位。计算有效地址的条件之一，该操作对应行中非对角线元素为0**
- **Dependency matrix: 16x16 矩阵，每行每列对应load/store队列的存储器操作。Load操作如果依赖于前面的store，则将该load操作的行中对应该store的列置位。Store操作更新存储器时，将该store操作对应的列复位。只有当load操作对应的行全0时，方可执行load操作**
- **Address queue:保存访问cache的loads/store操作的地址。如果是load操作，除了保存地址，还需要比较所有在该load操作之前的store的地址，如果匹配，则对dependency matrix对应位置位。**

2 Superscalar

- **是否能够使CPI < 1? 多发射处理器**
- **两种基本方法: Superscalar 、VLIW**
- **Superscalar:**
  - 每个时钟周期所发射的指令数不定（1 - 8条）
  - 由编译器或硬件完成调度
  - IBM PowerPC, Sun UltraSparc, DEC Alpha, HP 8000
  - 该方法对目前通用计算是最成功的方法

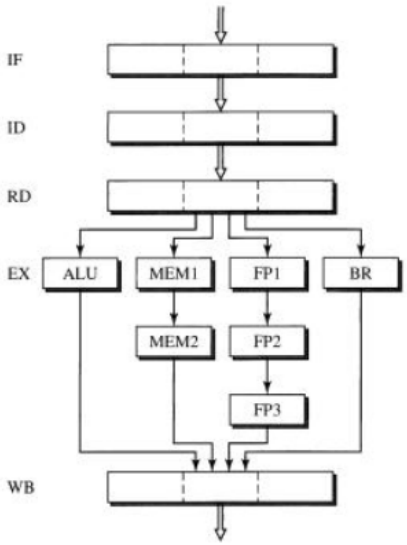


Figure 4.5
A Diversified Parallel Pipeline with Four Execution Pipes.

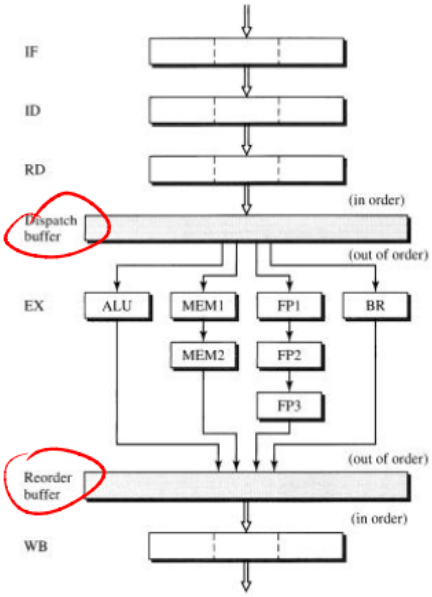


Figure 4.9
A Dynamic Pipeline of Width $s = 3$.

Superscalar MIPS

- **Superscalar MIPS: 每个时钟周期发射2条指令，1 条FP指令和一条其他指令**
  - 每个时钟周期取64位; 左边为Int , 右边为FP
  - 只有第一条指令发射了，才能发射第二条
  - 需要更多的寄存器端口，因为如果两条指令中第一条指令是对FP的load操作（通过整数部件完成），另一条指令为浮点操作指令，则都会有对浮点寄存器文件的操作
- **原来1 cycle load 延时在Superscalar中扩展为3条指令**

Type	Pipe Stages
Int. instruction	IF ID EX MEM WB
Fp. Instruction	IF ID EX MEM WB
Int. instruction	IF ID EX MEM WB
Fp. Instruction	IF ID EX MEM WB
Int. Instruction	IF ID EX MEM WB
Fp. Instruction	IF ID EX MEM WB

Review: 具有最小stalls数的循环展开优化

1 Loop:	LD	F0,0(R1)
2	LD	F6,-8(R1)
3	LD	F10,-16(R1)
4	LD	F14,-24(R1)
5	ADDD	F4,F0,F2
6	ADDD	F8,F6,F2
7	ADDD	F12,F10,F2
8	ADDD	F16,F14,F2
9	SD	0(R1),F4
10	SD	-8(R1),F8
11	SUBI	R1,R1,#32
12	SD	16(R1),F12
13	BNEZ	R1,LOOP
14	SD	8(R1),F16 ; 8-32 = -24

LD to ADDD: 1 Cycle
ADDD to SD: 2 Cycles

14 clock cycles, or 3.5 per iteration

采用Superscalar技术的循环展开

Integer instruction	FP instruction	Clock cycle
Loop: LD F0,0(R1)		1
LD F6,-8(R1)		2
LD F10,-16(R1)	ADDD F4,F0,F2	3
LD F14,-24(R1)	ADDD F8,F6,F2	4
LD F18,-32(R1)	ADDD F12,F10,F2	5
SD 0(R1),F4	ADDD F16,F14,F2	6
SD -8(R1),F8	ADDD F20,F18,F2	7
SD -16(R1),F12		8
SUBI R1,R1,#40		9
SD +16(R1),F16		10
BNEZ R1,LOOP		11
SD ＋8(R1),F20		12

- 循环展开5次以消除延时 (+1 due to SS)
- 12 clocks, or 2.4 clocks per iteration (1.5X)

多发射的问题

- **如果Integer和FP操作很容易区分组合，那么对这类程序在下列条件满足的情况下理想CPI= 0.5：**
  - 程序中50% 为FP 操作
  - 没有任何相关
- **如果在同一时刻发射的指令越多，译码和发射就越困难**
  - 即使是同一时刻发射2条 =>需检查2个操作码，6个寄存器描述符，检查是发射1条还是2条指令。
- **VLIW**
  - 指令字较长可以容纳较多的操作
  - 根据定义,VLIW中的所有操作是由编译时刻组合的，并且是相互无关的，也就是说：可以并行执行
  - 例如 2 个整数操作，2个浮点操作，2个存储器引用，1个分支指令
    - 每一个操作用16 到 24 位 表示 => 共7*16 = 112 bits 到 7*24 = 168 bits wide
  - 需要用编译技术调度来解决分支问题

基于VLIW的循环展开

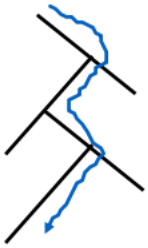
Memory reference 1	Memory reference 2	FP operation 1	FP op. 2	Int. op/ branch	Clock
LD F0,0(R1)	LD F6,-8(R1)				1
LD F10,-16(R1)	LD F14,-24(R1)				2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2		3
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2		4
		ADDD F20,F18,F2	ADDD F24,F22,F2		5
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2			6
SD -16(R1),F12	SD -24(R1),F16			SUBI R1,R1,#48	7
SD 16(R1),F20	SD 8(R1),F24				8
SD -0(R1),F28				BNEZ R1,LOOP	9

Unrolled 7 times to avoid delays
7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)
Average: 2.5 ops per clock, 50% efficiency

LD to ADDD: 1 Cycle
ADDD to SD: 2 Cycles

注: 在VLIW中，一条超长指令有更多的读写寄存器操作(15 vs. 6 in SS)

Trace Scheduling



- **消除分支的一种策略**
- **两步：**
  - Trace Selection
    - 搜索可能最长的直线型代码（由一组基本块构成）（通过静态预测或profile技术）(trace)
  - Trace Compaction
    - 将trace中的指令拼装为若干条VLIW 指令
    - 需要一些保存环境的代码，以防预测错误
- **由编译器撤销预测错误造成的后果（恢复寄存器的原值）**

HW推断执行(Tomasulo) vs. SW (VLIW) 推断执行

- **HW 确定地址冲突**
- **HW 分支预测较好，预测准确率较高**
- **HW 可支持精确中断模型**
- **HW 不必执行保存环境和恢复环境的指令**
- **SW 推断执行比HW推断执行硬件成本小**

Superscalar vs. VLIW

- **Superscalar**
  - 代码量较小
  - 二进制兼容性好
- **VLIW**
  - 译码、发射指令的硬件设计简单
  - 需要访问更多寄存器，一般使用多个寄存器文件

Superscalar 的动态调度（2/2）

- **用Tomasulo如何发射两条指令并保持指令序**
  - 假设有1 浮点操作，1个整数操作
  - Tomasulo控制器一个控制整型操作发射，一个控制浮点型操作发射
- **如果每个周期发射两条不同（整型类、浮点类）的指令，比较容易保持整型类操作序，浮点类操作序**
  - FP的Loads操作可能引起整型操作发射和浮点操作发射的相关
- **存储器引用问题：其中一种解决方案**
  - 将load的保留站组织成队列方式，操作数必须按指令序读取
  - Load操作时检测Store队列中Store的地址以防止RAW冲突
  - Store操作时检测Load队列的地址，以防止WAR相关
    - 允许Store操作跨越Load操作（非投机方式）
  - Store操作按指令序进行，防止WAW相关