Instruction Level Parallelism and Its Dynamic Exploitation

Instruction Level Parallelism achived from overlapping executions of multiple instructions.
Increasing amount of parallelism requires reducing the effect of hazards and using compilers.

Performance in pipeline CPI= Ideal CPI + Structural hazards stalls + RAW salls + WAR stalls + WAW stalls + Control stalls

Different techniques to reduce R.H.S:

- Control stalls by Loop unrolling and speculation and dynamic branch prediction
- RAW stalls by dynamic scheduling with scoreboarding, basic pipeline scheduling, speculation

Different techniques to reduce R.H.S:

- Ideal CPI by issuing multiple instructions, compiler dependency analysis, software pipelining, and trace scheduling
- WAR and WAW stalls by dynamic scheduling and register renaming and speculation

Concept of ILP

Amount of parallelism in a straight line sequence of code.

Limitations: with a branch frequency of 15%, means on average there will be only 6-7 instructions between any pair of branches.

This means that maximum amount of overlap is much less than 6.

We must find parallelism outside each basic block (among blocks).

Finding Parallelism Among Itterationso f a Loop

Example:

```
for (i=1; i<=1000; i++)
    X[i] = X[i] + Y[i];
```

There is a loop parallelism (1000) that we need to convert it to ILP.

Method: Loop Unrolling using compiler or hardware techniques.

Other Method: Use Vector Processing.

Tradeoffs: Increasing size of code and using more registers.

Key to improvements,**resolve dependency** between instructions.

N. Mekhiel

Dependences

Two instructions that are independent are parallel and could be executed in parallel.

Types of Dependences:

1-Data Dependences

1- Instruction i produces results that is used by j
2-instruction j dependens on k and k depends on i (indirect dependency)
Example:

```
1 Loop: LD    F0, 0(R1) ; F0= element
2       ADDD F4, F0, F2 ; Add scalar to the elemen
3       SD   0(R1), F4  ; store new value to eleme

Data dependency between I1, I2 (F0)
and between I2,I3 (F4)

4       SUBI R1,R1, #8 ; Decrement pointer
5       BNE  R1, R2, Loop ;  branch if R1 != R2

Data dependency between 4, 5 (R1)
```

The presence of dependences indicates a potential for a hazard, but actual hazard and length of stall is a property of pipeline.

Example of Data dependence with dependent chain:

```
1-Loop:  LD  F0, 0(R1)
2-       ADDD F4, F0,F2
3-       SD 0(R2), F4
4-       SUBI R1, R1, #8
5-       LD F6, 0(R1)
6-       ADDD F8, F6, F2
7-       SD 0(R1), F8
8-       SUBI R1, R1, #8
```

Data dependency between Inst4, Inst5, Inst7, Inst8 for R1

Compiler removes these dependences by computing intermidiate values of R1 and adjusting the offeset of LD, SD and decrementing R1 removing SUBI. R1= R1 -8 (LD F6, -8(R1),..)

N. Mekhiel

Name Dependences

Two types:

1-WAR hazard (antidependence)

When instruction j writes to register or memory before instruction i reads it

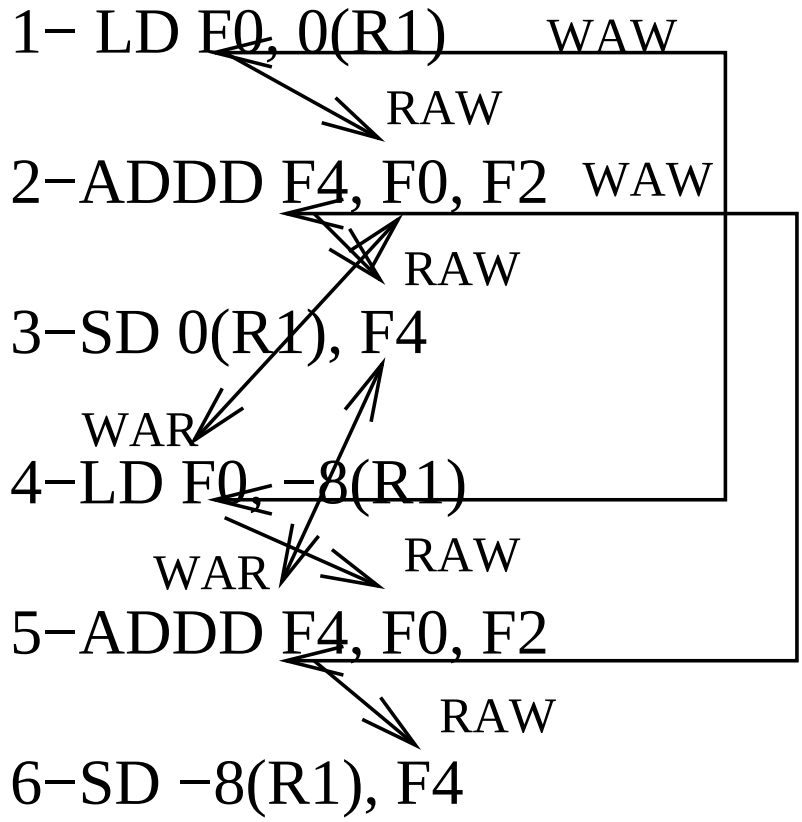
2-WAW hazard (output dependence)

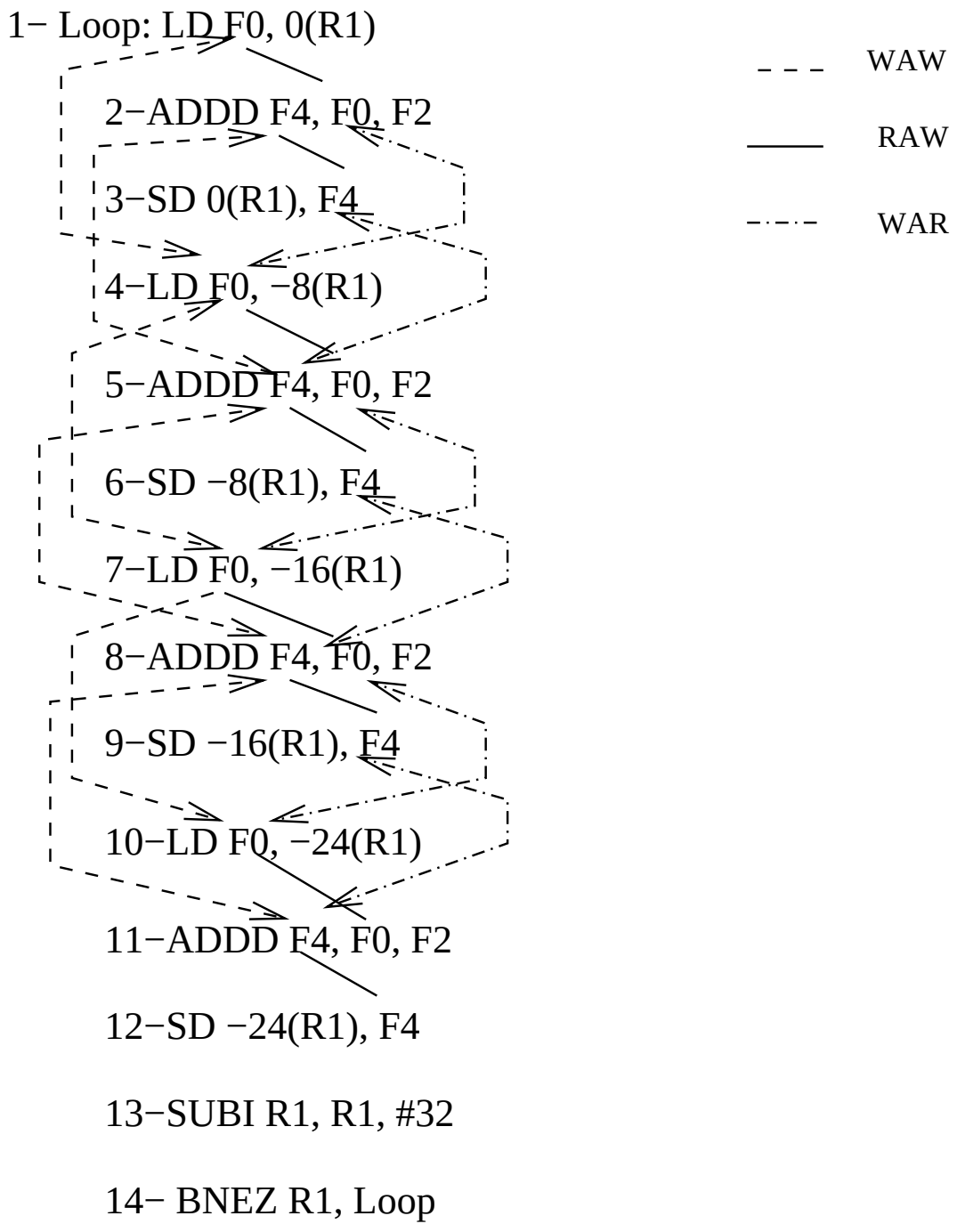
When j writes to same register or memory before i writes to it.

There is no data being transferred between instruction i, j. We can use register renaming to resolve this.

It is difficult to detect same name for memory access. For example 0(R2) and 20(R3) might reference the same memory location.

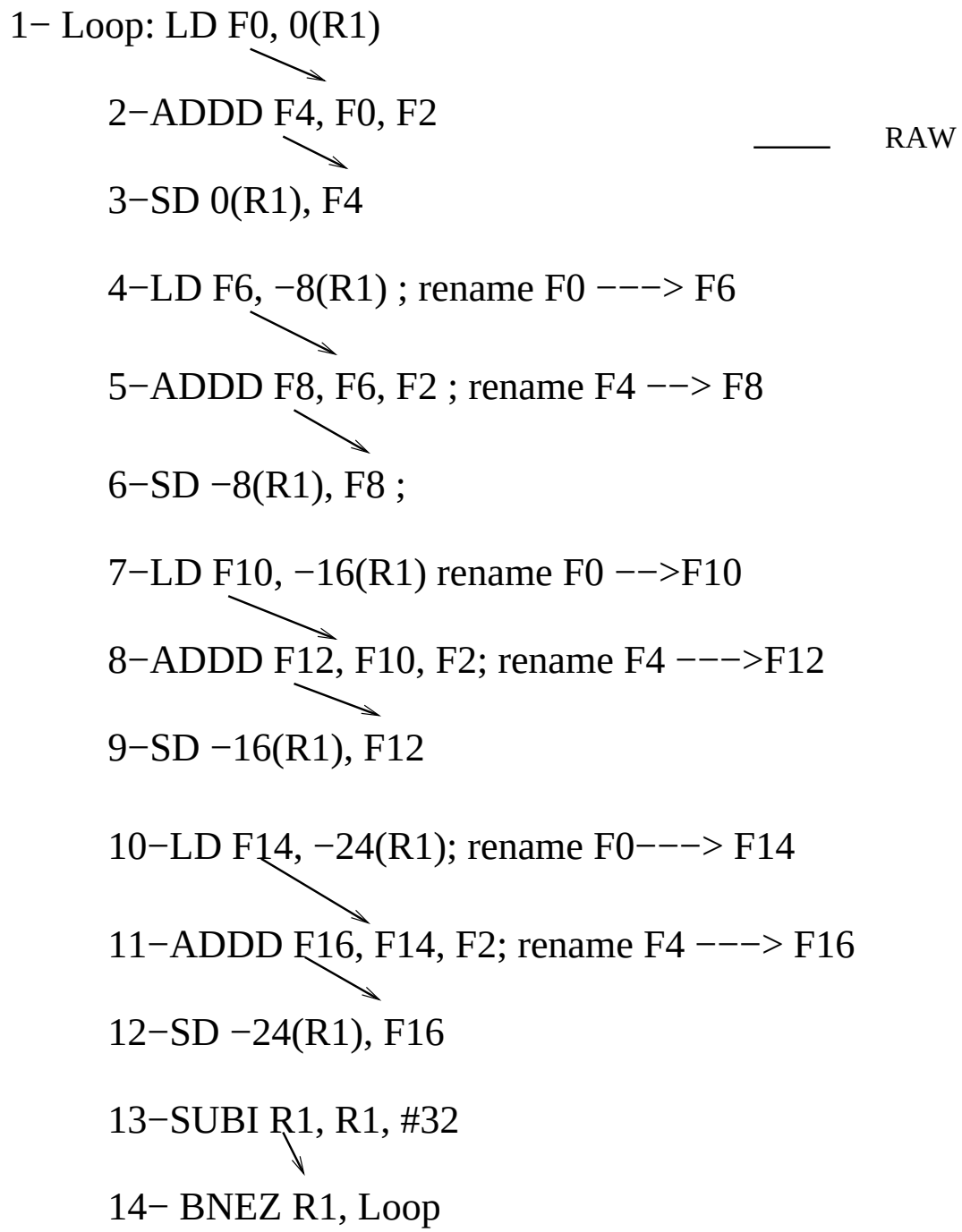
Example: With loop unrolling, no scheduling, indicate both data and name dependences. Show how renaming reduces name dependences




1− Loop: LD F0, 0(R1)
2−ADDD F4, F0, F2
3−SD 0(R1), F4
4−LD F0, −8(R1)
5−ADDD F4, F0, F2
6−SD −8(R1), F4
7−LD F0, −16(R1)
8−ADDD F4, F0, F2
9−SD −16(R1), F4
10−LD F0, −24(R1)
11−ADDD F4, F0, F2
12−SD −24(R1), F4
13−SUBI R1, R1, #32
14− BNEZ R1, Loop
WAW
RAW
WAR

N. Mekhiel

With Register Renaming

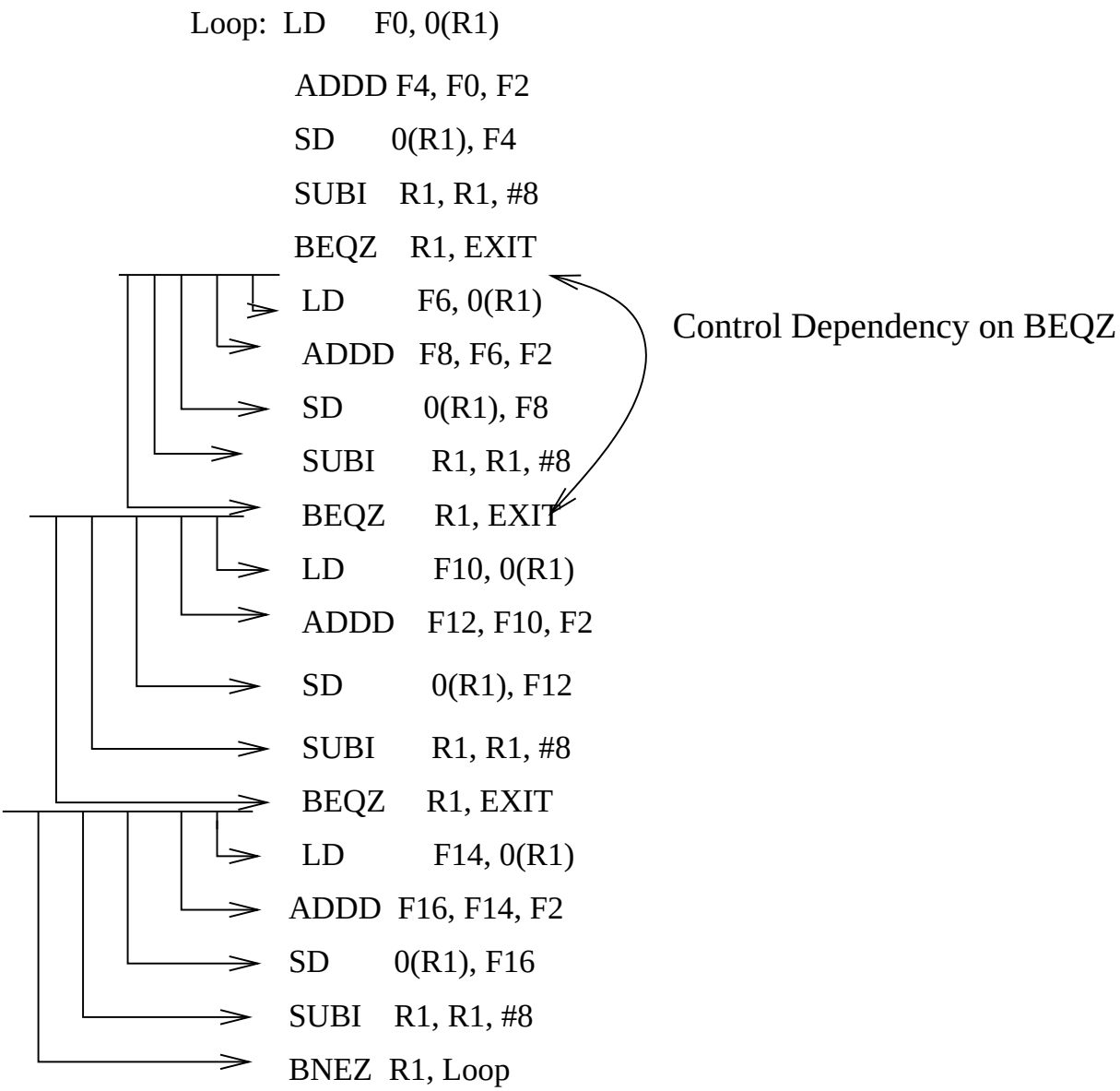


Control Dependences

Control dependences: It determines if a set of instructions should be executed or not.
Control dependences must be preserved.

```
Example:

1- if P1 {Si}
       ;

Instructions on Si should only be executed if P1 i

 We cannot move an instruction that is control dep
branch to before branch (cannot move instructions
before if).


2-    X;
  if P2 {S2;}

Instruction that is not control dependent cannot b
after branch.
Cannot move statement  before (X;) if  to into the
```

Example: Indicate control dependences for loop unrolling with overhead SUBI, BEQZ



If R1 =4*8, We can remove 3 intermediate SUBI, BNEZ, and remove Control dependences.

Preserving Control Dependency By:

1-In order instruction execution

2- Detection of Control hazards and stall

Preserving control dependences assures program correctness in two ways:

1-Preserve exception behavior

Example:

```
BEQZ  R2, L1
LW    R1, 0(R2)
```

Assume that control dependence is not preserved, and we move LW before branch.

LW may cause an exception (memory violation), if branch is taken, the above exception will never occur.

2-Preserve data flow:

branches make data flow dynamic,
hard to predict in static code and
cannot be gauranteed with data dependency.

Example:

```
          ADD    R1, R2, R3
          BEQZ   R4, L
          SUB    R1, R5, R6
   L:     OR R7, R1, R8
```

If branch is taken R1=R2+R3,
 IF not R1=R5-R6

Dynamic Scheduling To Deal with Data Hazards

Without dynamic scheduling, we used compiler to minimize hazards using static scheduling.

dynamic scheduling uses Hardware to rearrange instructions to reduce stalls.

Advantages of Dynamic Scheduling:

- Deals with dependences which are only known at run time.
- Simplifies Compiler
- Allows code compiled for one pipeline to run on a different pipeline.

Dynamic Scheduling allows out of order Instruction Completion making Precise Exceptions difficult.

N. Mekhiel

Example:

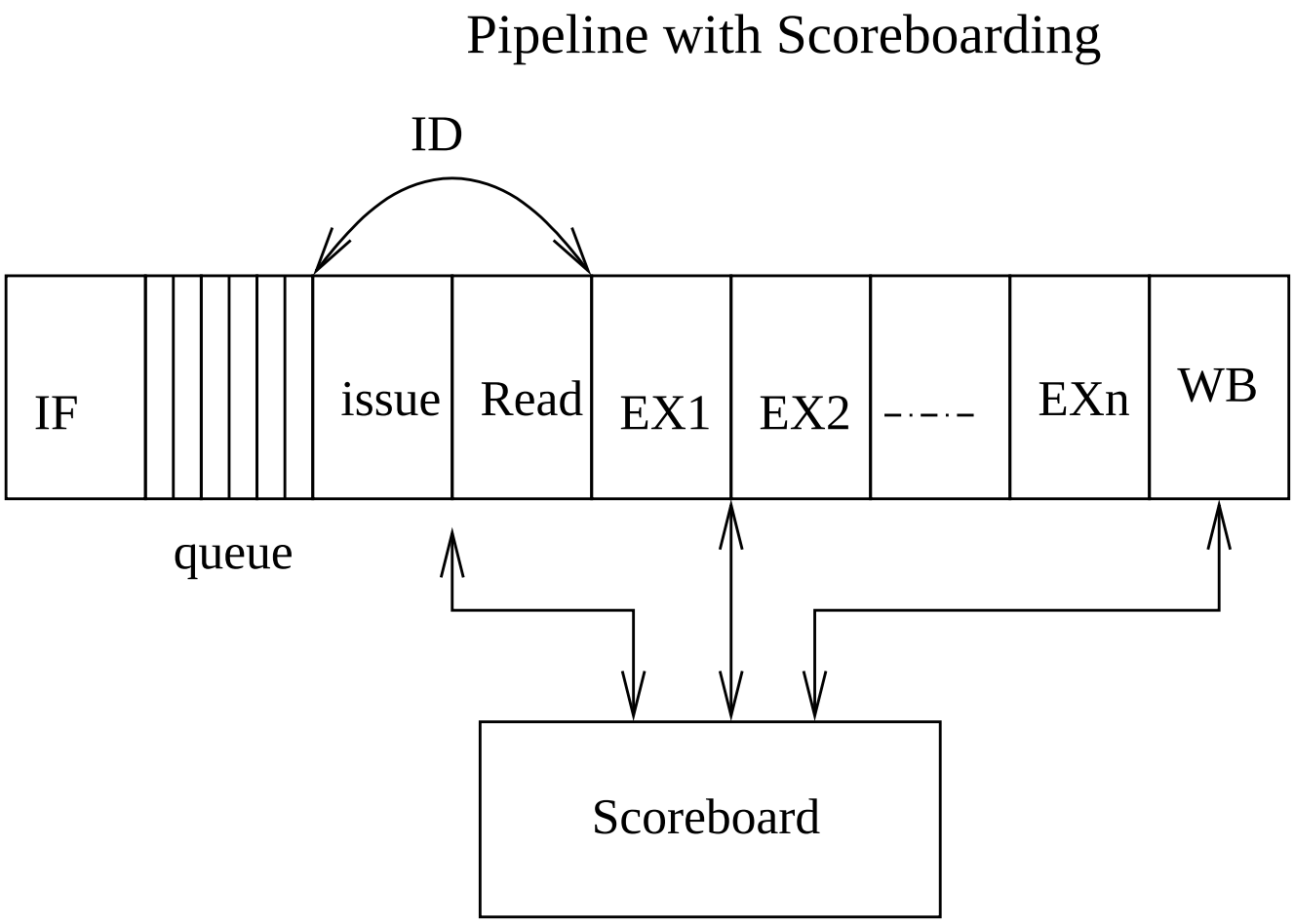
```
DIVD   F0, F2, F4
ADDD   F10, F0, F8 ;RAW dependency on F0
SUBD   F12, F8, F14
```

Normal pipeline will stall pipeline waiting for DI
to complete.

SUBD is not data dependent on anything.

With dynamic scheduling SUBD can complete early ou
of order.

Concept of Dynamic Scheduling Using Scoreboarding



**Pipeline: IF; Queue, Decode(Issue, Read Opera
Execute; Write Back**

N. Mekhiel

Dynamic Scheduling:

- Instruction queue allows multiple instructions to be available for issuing (if one instruction stall, others are available).
- Decode stage is split in two stages:-
  - 1-Issue: Decode instruction and check for structural hazards
  - 2-Read operand: Wait until no data hazards (RAW)
- Execute stage might take multiple clock cycles
- Write Back to write results
- Scoreboard: monitors hazards, dependency, available FUs, operands, and change pipeine to execute instructions as early as possible.

Scoreboarding creats a new hazard:

It could creat a WAR hazard that did not exist in simple DLX pipeline.

Example:

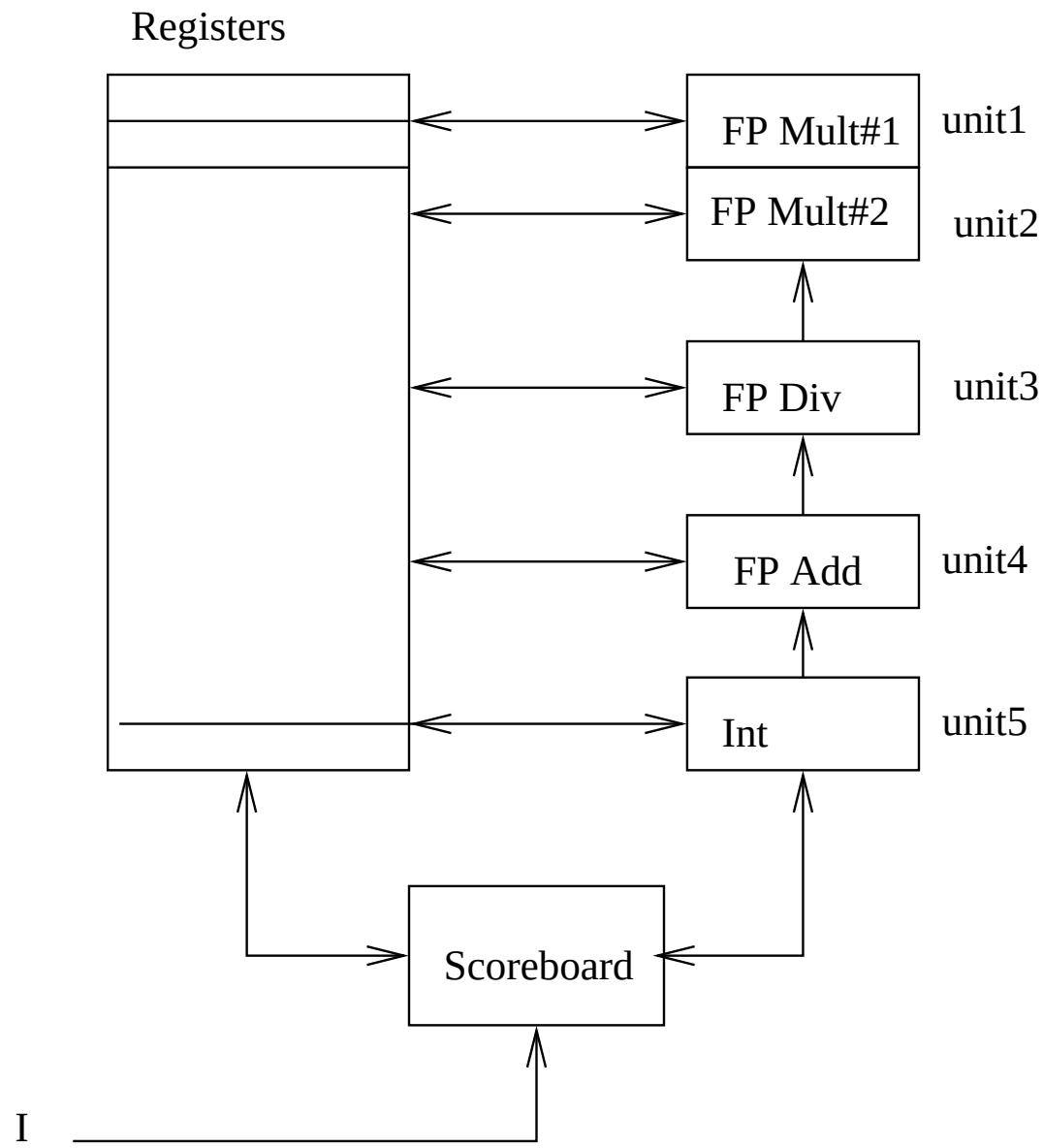
```
DIVID F0,  F2, F4
ADDD  F10, F0, F8;  RAW for F0
SUBD  F8,  F8, F14;  WAR for F8
```

If we let SUBD completes early before ADDD,
we have WAR hazard.
Scoreboard must detect this hazard.

If destination register for SUBD =F10, we have WAW
Scoreboard must also check for WAW hazards.

Multiple Function Units to allow execution of multiple instructions.

Scoreboarding with Multiple Function Units



Latency of MULT= 10 cycles; DIV = 40 cycles; ADDD= 2 cycles, Int=0 cycle.

Four Stages of Scoreboard Control:

- **1-ISSUE:** Decode instruction and check for structural hazards
  - a-Functional Unit is free
  - No active instruction has the same destination (WAW). If not, stall issuing any instruction (in order issuing), wait until a,b are okay.
- **2- Read Operands:**

  Wait until no data hazards, then read operands (RAW)
  - a-no active instruction is going to wite to operand.
  - no function unit is currently writing to this operand

  If a,b okay instruction may be sent to execute OUT OF ORDER.

Stages of Scoreboard

- **3-Execution:** Operate on operands and might take multiple cycles. When result is ready, it notify scoreboard for completion.

- **4- Write results:**
  - a-Scoreboard check for WAR (DEST is not used as source for other pending instructions)

    Example:

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

    Instruction SUBD will not write to F8 until ADDD reads F8.
  - b-Stall until all pending instructions read their sources.

Parts of Scoreboard

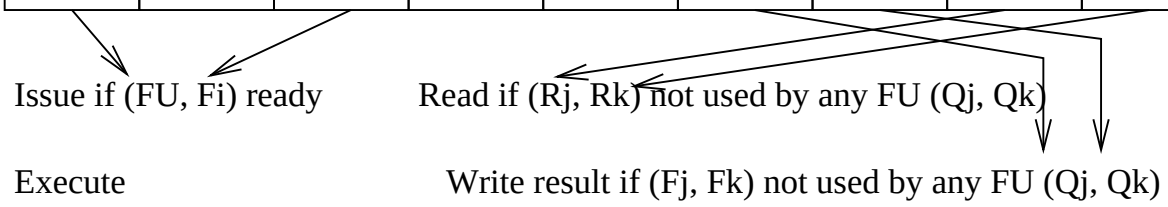
Three Parts of Scoreboard

1−Instruction Status in Pipeline

Issue	Read Operands	Execute Complete	Write result

2−Function Units Status

Busy	OP	Fi	Fj	Fk	Qj	Qk	Rj	Rk
FU is busy or not	operation in FU	dest. Reg	source Reg1	source Reg2	FU that produces Fj	FU that produces Fk	Flag if Fj ready	Flag if Fk ready

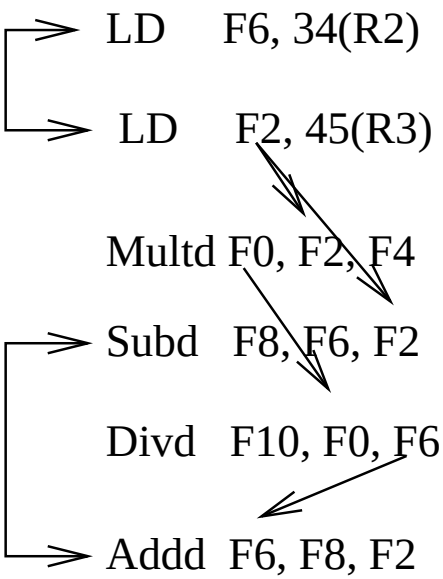


3−Register result status

Reg	F0		F8			
FU	DIV		ADD			

N. Mekhiel

CYCLE 5



Issue	Read	Complete	Write result
1	2	3	4
5	6	7	8
6	F2 9	19	20
7	F2 9	11	12
8	F0 21	61	62
13	14	15	F6 22

Add Latency = 2 cycles

Mult Latency = 10 cycles

Div Latency = 40 cycles

Int Latency = 0 cycles

N. Mekhiel

CYCLE 5

	Issue	Read	Complete	Write result
LD F6, 34(R2)	1	2	3	4
LD F2, 45(R3)	5			
Multd F0, F2, F4				
Subd F8, F6, F2				
Divd F10, F0, F6				
Addd F6, F8, F2				

Name	busy	op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Int	LD	F2	R3					Yes
	Mult1								
	Mult2								
	Add								
	Div								

Reg	F0	F2	F4	F6		F30
Fu		Int				

(N. Mekhiel)

CYCLE 8

	Issue	Read	Complete	Write result
LD F6, 34(R2)	1	2	3	4
LD F2, 45(R3)	5	6	7	8
Multd F0, F2, F4	7			
Subd F8, F6, F2	8			
Divd F10, F0, F6				
Addd F6, F8, F2				

Name

busy	op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Int	LD	F2		R3				Yes
Mult1	Mult	F0	F2	F4	Int (F2)		N (F2)	Y
Mult2								
Add	Sub	F8	F6	F2		Int (F2)	Y	N (F2)
Div	Div	F10	F0	F6	Mult		N (F0)	Y

Reg	F0	F2	F4	F6	F8	F10		F30
Fu	Mult	Int			Add	Div		

(N. Mekhiel)

Reducing Branch Penalties with Dynamic Hardware Prediction

Motivations :-

- Reduce control dependences and extract more ILP.
- Reduce effect of branches in multiple issuing (branch comes n times faster).
- Branch penalty has very dramatic effect in fast machines (Amdahl's law).

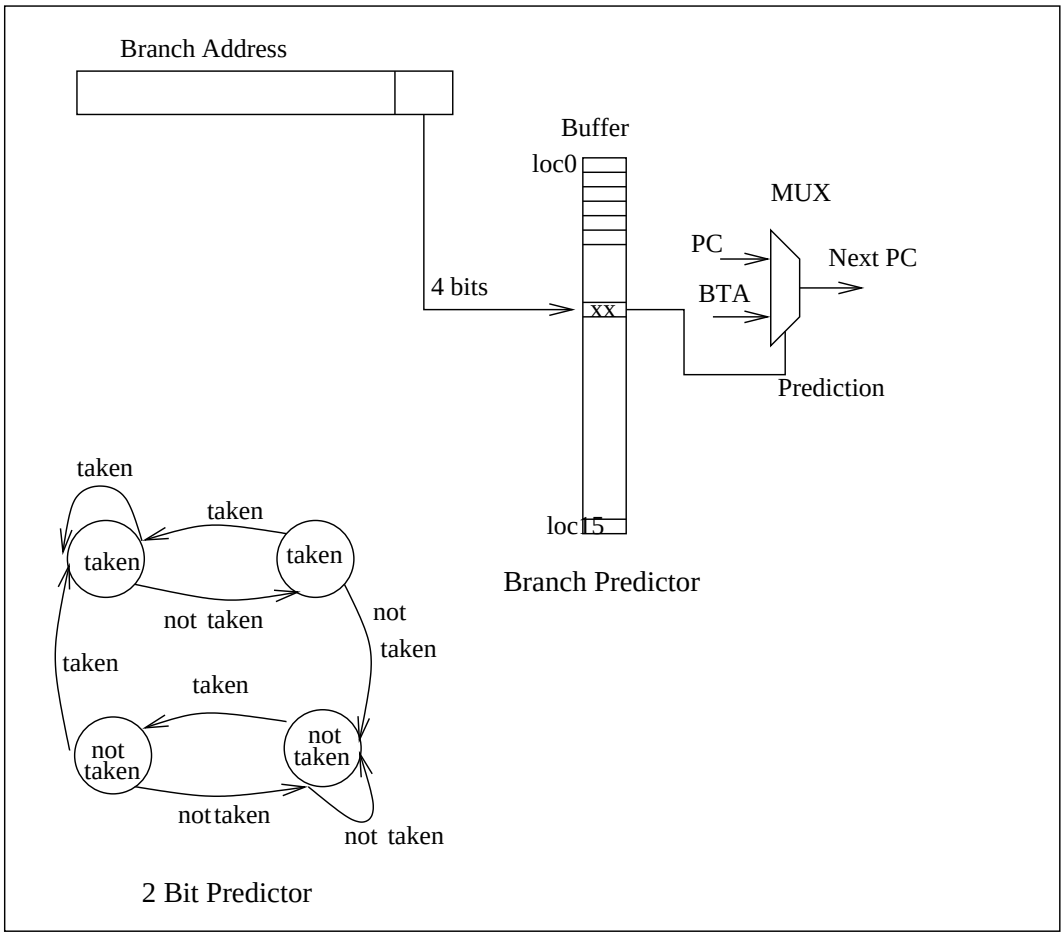
Background:

Static prediction schemes used to deal with branches (taken/not taken) does not depend on dynamic behavior of branch.
Delayed branch scheme uses compiler to schedule useful instruction (static).
We need more accurate branch prediction and has to follow dynamically the branch bhavior.

Concept: Use hardware to dynamically predict branch oucome, early.

Branch Prediction Buffer

Uses buffer (memory) to store recent branch behavior (1 bit/2 bit) for prediction.
The buffer is indexed by the lower portion of branch instruction address.
It uses Multiplexer to select BTA or PC depending on (value of 1 bit/2 bit) prediction attached to each branch address.



N. Mekhiel

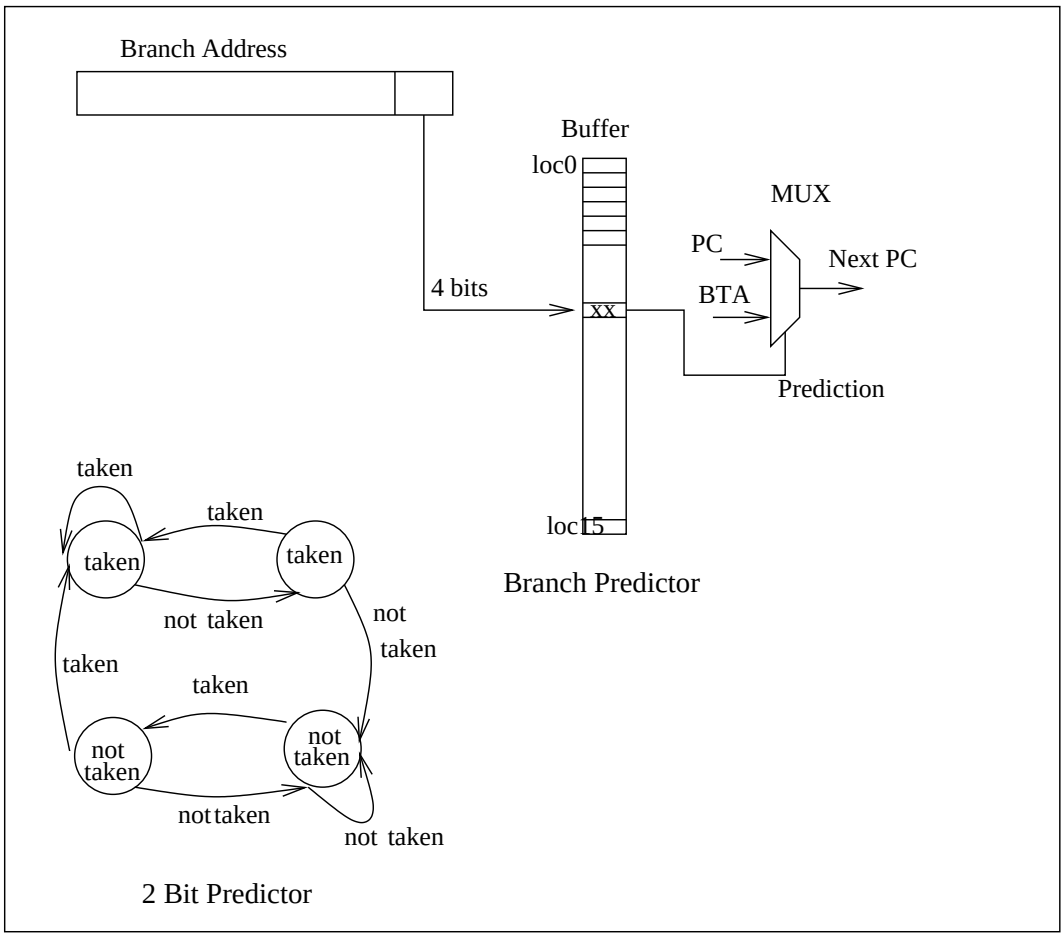
One bit prediction scheme has performance shortcoming as it is likely to predict incorrectly twice rather than once.

```
Example:
  L1: -----
      -----
  L2: ----
     br L2
     -----
     br  L1
```

```
assume a loop and branch always taken, then
when not taken, we predict wrong (1 time),
and buffer is updated with 1 bit=not taken.
Next time, we enter the loop, the branch
should be taken but we predict not
taken, wrong again for the 2nd time.
```

2 Bit Prediction Scheme

Prediction must miss twice before it changes the prediction.



Accuracy of branch prediction scheme

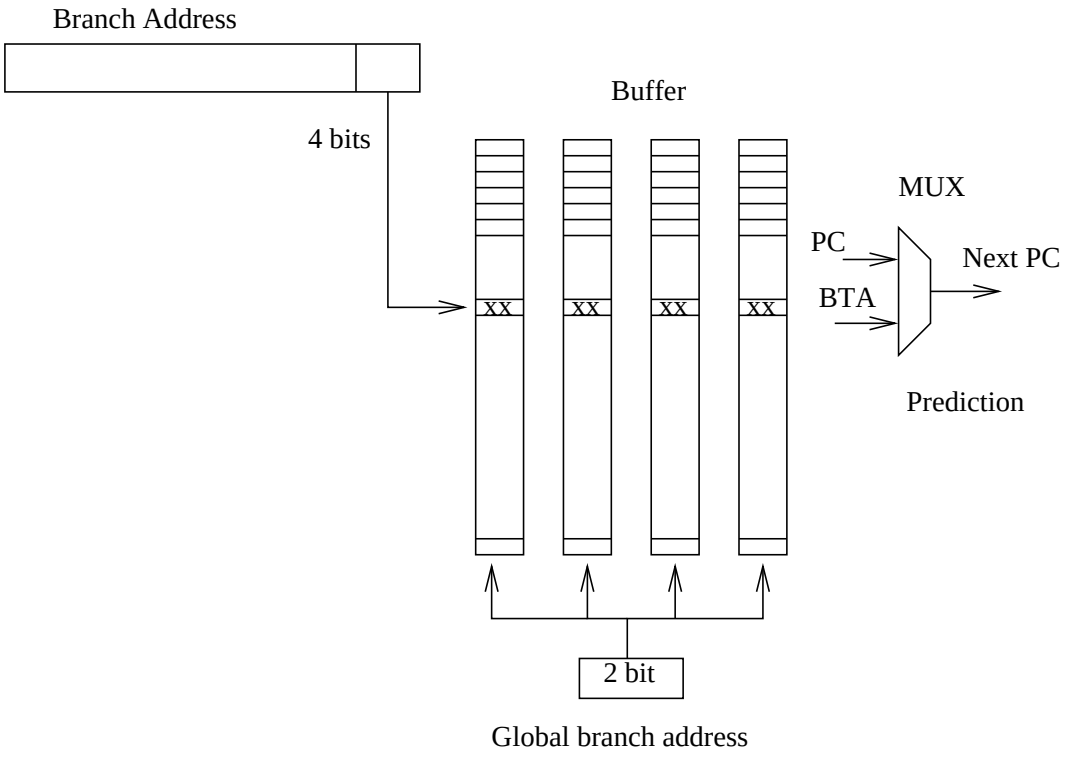
Using SPEC89, with prediction buffer of 4096 (12 bits idex), accuracy= 99% to 82%.

Performance depends on:- branch frequency, prediction accuracy and misprediction penalties (cycles).

N. Mekhiel

Improving Branch Prediction with Two Level Predictors

It uses extra 2 bits for global history to choose among 4 predictors.



Branch Predictor with Global History

Example: If branch prediction buffer has 8 Kbits, using (2,2), find number of branch enteries.

Number of enteries $= 8K \div (4 \times 2bits) =$ 1 K.

N. Mekhiel

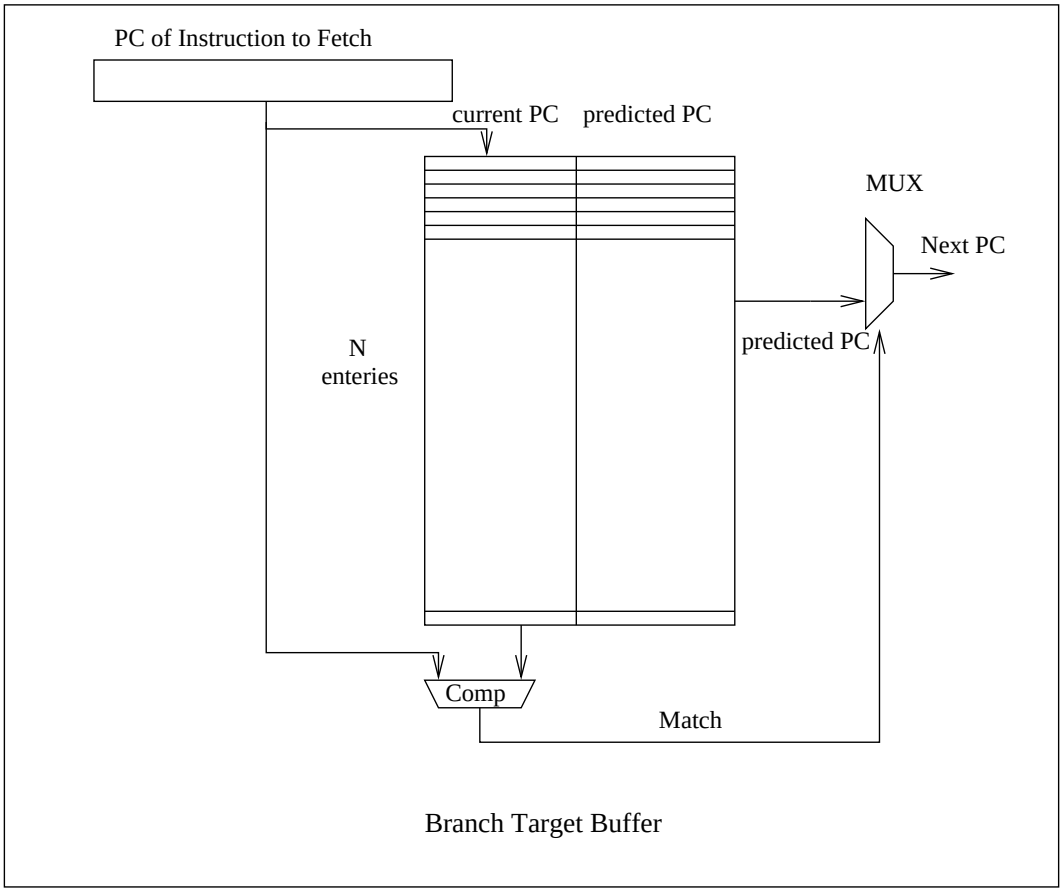
Reducing Branch Penalty Using Branch Target Buffers "BTB"

Motivations: even with very good prediction, if misprediction penalty is high, the performance of branches will be low.

Concept: reduce branch penalty by predicting address of next instruction after a branch. This reduce branch penalty to zero.

It uses a branch target buffer to store PC of instruction to fetch.

```
IF1       ID1    EX1    .....; Branch instruction
                                and BTB gives addre
           IF2    ID2      EX2  ; IF2 in time
```



If PC =Match , then PC=predicted PC from branch target buffer. If PC =No match , then instruction is not predicted to be a branch and use PC.

Improving Performance of BTB

Storing Target Instruction instead of target address, when BTB hit, get instructions from BTB not cache (can use larger buffer for multiple instruction storage), It allows branch folding to obtain zero cycle branches)

N. Mekhiel

Techniques to Detect Loop Level Parallelism

Must analyze source code for data dependences in the Loop or accross the itterations of Loop.

Data dependency detected if operand is written at some point and read at later point

```
Examples of data dependences:
Example1: Find if there is dependences in loop
 body and between different itterations of:
  for(i=1; i<=1000; i++)
    X[i]=X[i] +5;
There is dependency within the loop X[i], X[i]
There is no dependency between itterations
Example2:  find dependences in the following:
   for(i=1; i<=1000; i=i+1){
       A[i+1] = A[i] + C[i]; S1
       B[i+1] = B[i] + A[i+1]; S2
-There is dependency within Loop between S1,S2
using A[i+1]
-there is dependence between itteration i and i+1
S1 writes to A[i+1] and read at next itteration in
S1 as A[i]
-the same in  S2 for  B[i+1]
```

Example3: Find dependences in the following:

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

Itteration i:

```
    A[i] = A[i] + B[i];  S1
    B[i+1] = C[i] + D[i]; S2
```

Itteration i+1

```
     A[i+1] = A[i+1] + B[i+1];  S1
    B[i+2] = C[i+1] + D[i+1]; S2
```

B[i] makes S1 dependens on S2, but S2 does not depend on S1, there is no cyclic dependency and loops could be made parallel using LOOP TRANSFORMATION AS:

```
    A[1]  = A[1] + B[1];
    for(i=1; i<=99; i=i+1){
    B[i+1] = C[i] + D[i]; S2
    A[i+1] = A[i+1] + B[i+1];
                          }
   B[101] = C[100] = D[100];
```

ILP with Loop Unrolling

For ILP to work, we must find sequence of unrelated instructions, avoid pipeline stalls, separate dependent instruction from source by a distance equal to **pipeline latency**

FP Pipeline Latency

Inst. Producing result	**Inst. Using it**	latency
FP ALU OP	ALU OP	3
FP ALU OP	store double	2
load double	FP ALU OP	1
load double	store double	0

Examples of Loop Unrolling

```
  for(i=1; i<=1000; i++)
    x[i] = x[i] + s;
 assume  M[R1]=x[1000], F2=s
Loop: LD F0, 0{R1) ; F1= x[i]
      ADDD F4, F0, F2 ; F4= x[i] + s
      SD 0(R1), F4    ; x[i]=F4
      subi R1, R1, #8 ; i-1
      BNEZ R1, Loop
```

Performance without unrolling loop:

```
  for(i=1; i<=1000; i++)
    x[i] = x[i] + s;
      Instruction          latency
Loop: LD F0, 0{R1) ;       1
      STALL  1 CYCLE
      ADDD F4, F0, F2 ;    3
      STALL 2 CYCLES
      SD 0(R1), F4;        6
      SUBI R1, R1, #8 ;    7
      STALL 1 CCLE         8
      BNEZ R1, Loop  ;     9
      STALL 1 CYCLE        10

 NUMBER OF CLOCKS= 10 CYCLES

Using Scheduling:
 Loop: LD F0, 0(R1)        1
       SUBI R1, R1, #8     2
       ADDD F4, F0, F2     3
       BNEZ R1, Loop       4
       STALL 1 CYCLE       5
       SD =8(R0), F4       6
Number of Cycles = 6 (more than 50% gain)
```

Loop Unrolling 4 times

```
      Instruction          latency
Loop: LD F0, 0{R1) ;       1
      STALL  1 CYCLE
      ADDD F4, F0, F2 ;    3
      STALL 2 CYCLES
      SD 0(R1), F4;        6
-------------------------------  1st
      LD F6   -8(R1)
      ADDD F8, F6, F2
      Stall  2 cycles
      SD -8(R1), F8
----------------------------12    2nd
      LD F10   -16(R1)
      ADDD F12, F10, F2
      Stall  2 cycles
      SD -16(R1), F12
----------------------------18    3rd
```

Unrolling the Loop for 4th tie

```
      LD F14   -24(R1)
      ADDD F16, F14, F2
      Stall  2 cycles
      SD -24(R1), F16
      SUBI R1, R1, #32
      STALL        1 CYCLE
      BNEZ R1, Loop  ;
      STALL   1 CYCLE
Total=                    27 for 4 loops (6.75 per
```

Loop Unrolling + Scheduling

```
Loop: LD F0, 0(R1) ; F0= x[1000-4i]
      LD F6, -8(R0)
      LD F10, -16(R1)
      LD F14, -24(R1)
      ADDD F4, F0, F2 ; F4= x[1000-4i] + S
      ADDD F8, F6, F2
      ADDD F12, F10, F2
      ADDD F16, F14, F2
      SD 0(R1), F4;     x[1000-4i]=F4
      SD -8(R1), F8
      SUBI R1, R1, #32
      SD -16(R1), F12
      BNEZ R1, Loop
      SD +8(R1), F16;  x[1000-4i-3]=F16
Total #cycles=4x3+2=14, or 14/4=3.5 cycles
9/3.5 almost 300% gain in performance
```

Multiple Issue

To improve ideal CPI, make it less than 1 (IPC instructions per clock cycle).
Two types:-

- **Superscalar**
- **VLIW** (very long instruction word)

1-Superscalar

- Issue variable number of instructions per clock.
- Instructions must be independent, no more than one memory reference per clock
- Variable instruction issuing, and dynamic issuing

Simple DLX version: Integer and FP Operations could be issued simultaneously.

- 1-Fetch and Decode 2 instructions 64 bits per cycle
- 2-Hardware will issue 2nd instruction (FP) if ist instruction can issue (dynamically).

DLX Superscalar

- 3-Need to pipelined FP Units or use multiple FP Function Units (Issuing of FP Inst 1/cycle).
- 4-Using different Register Sets for Integer and FP.
- 5-Hazards must be detected if having Int Instruction is a FP load, FP Inst. has dependency on load Inst (RAW).
- 6-Problem with DLX Load latency of one clock cycle= cost 3 instructions (1+2) and branch delay latency of 1 cycle = 3 instructions.

```
Find Performance of DLX Superscalar for the follow
Loop: LD F0 0(R1)
      ADDD F4, F0, F2
      SD 0(R1), F4
      SUBI R1, R1, #8
      BNEZ R1, Loop
Assume Loop unrolled 5 times.
```

Instruction DLX	SUPERSCALAR Int Inst.	FP Inst.	Clock
loop: LD F0, 0(R1)	loop: LD F0, 0(R1)		
LD F6, −8(R1)	LD F6, −8(R1)		2
LD F10, −16(R1)	LD F10, −16(R1)	ADD F4, F0, F2	3
LD F14, −24(R1)	LD F14, −24(R1)	ADD F8, F6, F2	4
LD F18, −32(R1)	LD F18, −32(R1)	ADD F12, F10, F2	5
ADD F4, F0, F2	SD(0(R1), F4	ADD F16, F18, F2	6
ADD F8, F6, F2	SD −8(R1), F8	ADD F20, F18, F2	7
ADD F12, F10, F2	SD −16(R1), F12		8
ADD F16, F18, F2	SD −24(R1), F16		9
ADD F20, F18, F2	SUBI R1, R1, # 40		10
SD(0(R1), F4	BNZ R1, loop		11
SD −8(R1), F8	SD 8(R1), F20		12
SD −16(R1), F12			13
SD −24(R1), F16			14
SUBI R1, R1, # 40			15
BNZ R1, loop			16
SD 8(R1), F20			17

Superscalar gain= 17/12=1.41

Performanc of superscalar $= 12 \div 5 = 2.4$ cycles per loop.

Performance gain $= 17 \div 12 = 1.41$

Superscalar could use dynamically scheduled instructions with scoreboarding (Tomasulo's algorithm) to improve ILP.

VLIW Approach

- Reduces hardware needed for multiple issue superscalar
  Superscalar hardware is needed to examine opcode of multile instructios, and registes to determine if it can issue them.

- VLIW use multiple independent functional Units and one long instruction with the help of compiler to form it (static scheduling).

- VLIW instruction might have 2 Int operations + 2 FP operations + 2 memory operations and a branch. Instruction length could be 112 or 168 bits.

Example
Assume 2 mem operations, 2 FP , 1 Int 1 branch in VLIW cycle. Using loop unrolling for 7 times, find the performance improvement for VLIW

VLIW					
Int Inst.	Int Inst.	FP Inst.	FP Inst.	Int Inst.	Clock
loop 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)	ADD F4, F0, F2	ADD F8, F6, F2		3
LD F26, −48(R1)		ADD F12, F10, F2	ADD F16, F18, F2		4
		ADD F20, F18, F2	ADD F24, F22, F2		5
SD(0(R1), F4	SD −8(R1), F8	ADD F28, F26, F2			6
SD −16(R1), F12	SD −24(R1), F16				7
SD −32(R1), F20	SD −40(R1), F24			SUBI R1, R1, # 56	8
SD 8(R1), F28				BNZ R1, loop	9

Performance gain=17/9=2

Perfomance of VLIW= 9 Cycles for 7 loop iterations=1.28 cycles/loop compared to 2.4 for superscalar or 3.5 with scheduling and loop unrolling.

Limitations of Multiple Issue Processors

- Inherent limitations of ILP in progrms (very serious limitation). Not enough parallel operations.
- Complexity of hardware implementations, large increase in memory bandwidth, register bandwidth (multiple ports).
- Superscalar with dynamic scheduling complicates design
- VLIW increases code size, if instructions are not full (not enough ILP) causes a waste in instruction encoding. Ccahe miss in VLIW causes all function units to stall (all instructions in VLIW are synchronized).
- Binary code compatability is a major problem for VLIW (comes from using different number of function Units, or instructions).

Pentium 4 Processor Microarchitecture

- 90 nm Process technology
- Execution trace cache
- 2X frequency execution core
- Hyper-Threading (SMT)
- New SSE3 Instructions (streaming SIMD Extension)
- Higher frequency with extended pipeline (3.4 GHz)

Overview of NETBURST (P4) microarchitecture

Execution trace cache

Out-of-order core

Rapid Execution

Store to load forwarding

Trace Cache:

- Instruction cache called execution trace cache
- Stores decoded instructions in form of uops
- UOPS could be accessed repeatedly like cache
- No Decoding , trace cache takes decoded uops from decoder
- trace is assembled from multiple UOPS (up to 6) which are sequentially predicted from path of program including target of branches. In instruction cache, only branch instruction is delivered with delay but not from taeget.
- 3 UOPS per cycles
- When trace cache miss, fetch and decode from L2 cache
- Trace cache holds up to 12 K UOPS

Instructions that cannot be encoded in trace cache, use indirect CALLS and sequence them from Microcode ROM as indirect CALLS, and software prefetch.

Out of Order Core:

- Extract parallelism from code stream (UOPS)
- Schedule as many UOPS as possble for execution per each clock cycle
- Scheduler tracks input register operands when ready to execute and exeution resources available
- Can dispatch 6 UOPS
- ALU can schedule on each 1/2 cycles
- Uses two dispatch ports for load and store

Execution Engine:

- Executes up to 6 UOPS per cycle:
  - 2 Int ALU
  - 1 Complex Int unit
  - load and store address generation units (AGU)
  - complex FP/media unit
  - FPmedia move unit
- INT ALU executes at double clock speed
- Execute 1 load and 1 store every clock from L1 (16 KB, 8 way, 64 B) Data cache
- Parallel DTLB and L1 to provide low latency from L1

Store to Load Forwarding:

- Could forward data to be stored to L1 directly to load using a store forwarding buffer (SFB)
- This is important because store is done at a later stage in a deep pipeline
- It uses feedback, MUX , SFB and control logic similar to data forwarding

Branch Prediction:

- Uses Static branch prediction (simple)
- prediction at decode time (early)
- If branch is backwards, predict it taken and if forward predict it not taken
- modify prediction for loop ending branch (with branch distance is less than a threshold)
- must flush pipeline if branch miss predicted

Memory System:

- Unified (I and D) L2 cache system size= 1 MB, 8 way, 128 B
- Use prefetch instructions for data to L2 and page table entries to DTLB
- Hardware prefetching by a predictor for stream of data

Hyper-Threading:

- Allows one physical processor to appear as two logical processors
- Two software threads can execute simultaneously eliminating contex switching overhead penalty
- **Changes in microarchitecture to support Hyper Threading:**
  - Increase number of outstanding L1 load misses from 4 to 8

- Increase queue size between front end and alloation/rename logic
- Page table walk can occur at same time as memory access
- Page table walk that misses all caches and go to DRAM does not block other page table walk from being intiated
- Trace cache respond faster to stalling events and dedicate all its resources to the thread that is not stalled
- Uses extra bit in virtual tag for each logical processor to prevent conflicts in L1 cache when the two logical processors has a matched virtual tag (encourage true sharing but disallow false sharing).

SSE 3 Instructions:

- For Integer conversion
- Complex arithmetic
- Video encoding
- Graphics
- Thread synchronization

Pentium 4 Performance

Improvements due to speed and Hyper Threading of about 20% (limted why?).

EXAMPLES:

A-13 Scoreboard stage other than execute takes 1 c
MUL takes 3 cycles, ADD, SUB each take 1 cycle. As
function units and two multiply function units. Fi
processor pipeline stages in executing:

	Issue	Read	Execute	W
MUL F0, F6, F4	1	2	2+3=5	6
SUB F8, F0, F2	2	7(MUL F0)	8	9
ADD F2, F10, F2	3	4	5	8

CH4-5
List all dependency(true=RAW), (ou=WAW) and (anti=
in the following:

```
for(i=2; i<100; i=i+1)
{
 a[i]   = b[i] + a[i}  ; S1
 c[i-1] = a[i] + d[i}  ; S2
 a[i-1] = 2*b[i] ;       S3
 b[i+1] = 2*b[i];        S4
```

```
}
Next loop
   a[i+1]   = b[i+1] + a[i+1}  ; S1
 c[i] = a[i+1] + d[i+1}  ; S2
 a[i] = 2*b[i+1] ;       S3
 b[i+2] = 2*b[i+1];        S4

within loop RAW S1, S2
out of loop RAW S4, S1 and S4, S3

out of loop WAW S1,S3

out of loop WAR S3 2ND LOOP, S1,S2 FIRST LOOP
```

```
Example 4-8
The following loop Y[i]=axX[i] + Y[i]
loop LD F0, 0(R1); F0=X[i]
     MUL F0,F0,F2; F0=a*X[i]
     LD F4, 0(R2); F4=Y[i]
     ADD F0,F0,F4; F0=a*X[i]+Y[i]
     SD 0(R2), F0; Y[i]=a*X[i]+Y[i]
     SUBI R1,R1,#8 ; i+1 for X[]
     SUBI R2,R2,#8 ;
     BNZ R1, loop

a-single-issue , unroll loop, and schedule it
Assume ALU-ALU=3, ALU -SD=2, LD ALU=1
loop LD F0, 0(R1)
     LD F6, -8(R1)
     LD F12, -16(R1)
     LD F18, -24(R1)
     MUL F0, F0, F2
     MUL F6,F6,F2
     MUL F12,F12,F2
     MUL F18,F18,F2
     LD F4, 0(R2)
     LD F8, -8(R2)
```

```
      LD F10, -16(R2)
      LD F14, -24(R2)
      ADD F0,F0,F4
      ADD F6,F6,F8
      ADD F12,F12,F10
      ADD F18,F18,F14
      SUBI R1,R1,#32
      SUBI R2,R2,#32
      SD 32(R2), F0
      SD 24(R2), F6
      SD 16(R2), F12
      BNZ R1, loop
      SD  8(R2)
TIME/LOOP =23/4=5.75 CYCLES
```

```
assume dual-issue processor

loop LD F0, 0(R1)
     LD F6, -8(R1)
     LD F12, -16(R1)   MUL F0, F0, F2
     LD F18, -24(R1)   MUL F6,F6,F2
     LD F4, 0(R2)      MUL F12,F12,F2
     LD F8, -8(R2)     MUL F18,F18,F2
     LD F10, -16(R2)   ADD F0,F0,F4
     LD F14, -24(R2)   ADD F6,F6,F8
     SUBI R1,R1,#32    ADD F12,F12,F10
     SUBI R2,R2,#32    ADD F18,F18,F14
     SD 32(R2), F0
     SD 24(R2), F6
     SD 16(R2), F12
     BNZ R1, loop
     SD 8(R2), F18

total cycles=15/4=3.75
```

Example 4-9

a)find number of cycles

```
bar: LD F2,0(R1)          1
                          stall
     MUL F4,F2,F0         3
     LD F6,0(R2)          4
                          stall 2 cycles MUL to ADD
     ADD F6,F4,F6         7
                          STALL 2 CYCLES ALU-STORE
     SD 0(R2),F6          10
     ADDI R1,R1,#8        11
     ADDI R2,R2,#8        12
     SGTI R3,R1,#800      13
                          STALL 1 CYCLE R3
     BEQZ R3, bar         15
                         STALL FOR BRANCH 1 CYCLE
TOTAL=16 CYCLES PER LOOP
```

```
b) single issue with loop unrolling 4 times + sche
bar LD F2,0(R1)
    LD F4,8(R1)
    LD F6,16(R1)
    LD F8,24(R1)
    MUL F2,F2,F0
    MUL F4,F4,F0
    MUL F6,F6,F0
    MUL F8,F8,F0
    LD F10,0(R2)
    LD F12,8(R2)
    LD F14,16(R2)
    LD F16,24(R2)
    ADD F2,F2,F10
    ADD F4,F4,F12
    ADD F6,F6,F14
    ADD F8,F8,F16
    SD 0(R2),F2
    SD 8(R2),F4
    ADDI R1,R1,#32
    ADDI R2,R2,#32
    SGTI R3,R1,#800
    SD -16(R2),F6
```

```
    BEQZ R3, bar
    SD -8(R2), F8
```

CYCLES=24/4=6 Cycles

c-using VLIW, unroll loop 4 times

mem	mem	FP	FP	INT
LD F2	LD F4			
LD F6	LD F8			
LD F10	LD F12	MLT F2	MLT F4	
LD F14	LD F16	MLT F6	MLT F8	
				ADI R1,32
				ADI R2,32
		ADD F2	ADD F4	
		ADD F6	ADD F8	
				SGT R3
SD F2	SD F4			BEQZ
SD F6	SD F8			

Total cycles=11/4=2.75 cycles