
Lecture 15: Instruction Level Parallelism

-- 5-stage Pipeline Extension, ILP Introduction,
Compiler Techniques and ~~Branch Prediction~~

CSCE 513 Computer Architecture

Department of Computer Science and Engineering

Yonghong Yan

yanyh@cse.sc.edu

<https://passlab.github.io/CSCE513>

Topics for Instruction Level Parallelism

- **5-stage Pipeline Extension, ILP Introduction, Compiler Techniques, and Branch Prediction**
 - C.5, C.6
 - 3.1, 3.2
 - ~~Branch Prediction, C.2, 3.3~~
- **Dynamic Scheduling (OOO)**
 - 3.4, 3.5
- **Hardware Speculation and Static Superscalar/VLIW**
 - 3.6, 3.7
- **Dynamic Superscalar, Advanced Techniques, ARM Cortex-A53, and Intel Core i7**
 - 3.8, 3.9, 3.12
- **SMT: Exploiting Thread-Level Parallelism to Improve Uniprocessor Throughput**
 - 3.11

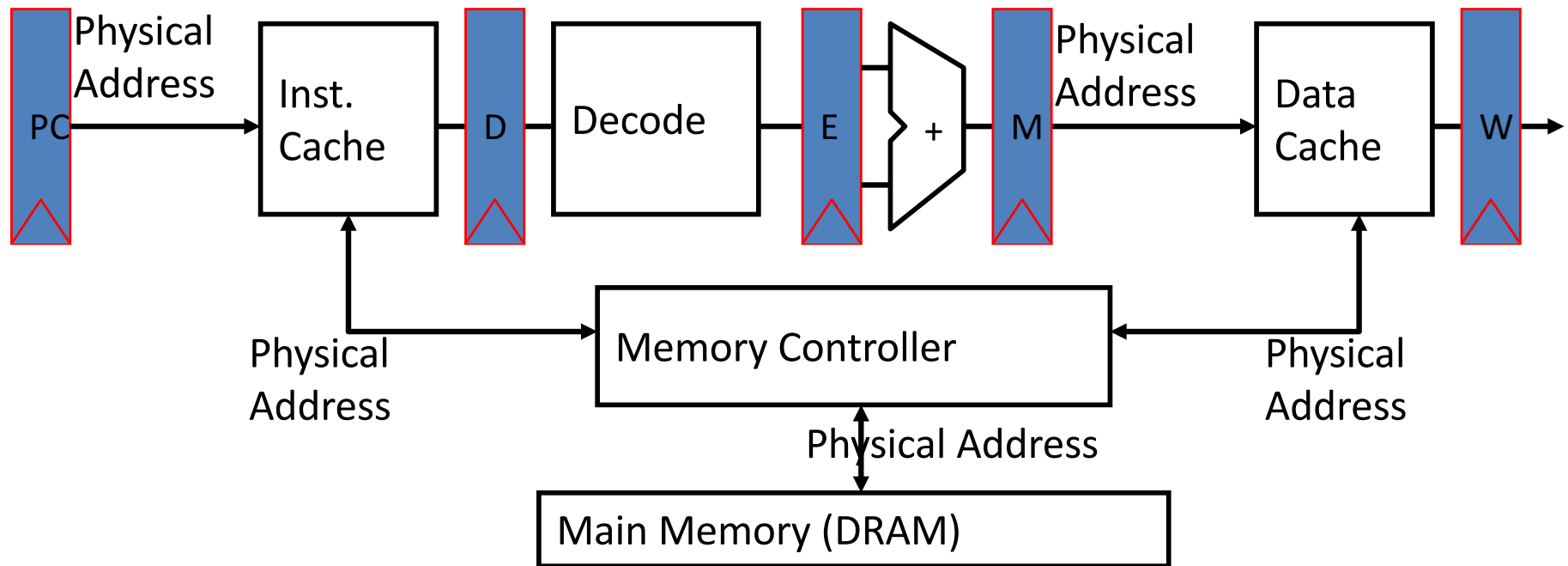
Extending 5-stage Integer Pipeline to Handle Multicycle Operations

Textbook: CAQA C.5 and C.6

C

Complex Pipelining: Motivation

- Why would we want more than our in-order pipeline?



Complex Pipelining: Motivation

Pipelining becomes complex when we want high performance in the presence of:

- Long latency or partially pipelined floating-point units
 - Not all instructions are floating point or integer
- Memory systems with variable access time
 - For example cache misses
- Multiple arithmetic and memory units

Floating Point Representation

- IEEE standard 754

Value = $(-1)^s * 1.\text{mantissa} * 2^{(\text{exp}-127)}$

Exponent = 0 has special meaning

IEEE Floating Point Representation



IEEE Double Precision Floating Point Representation

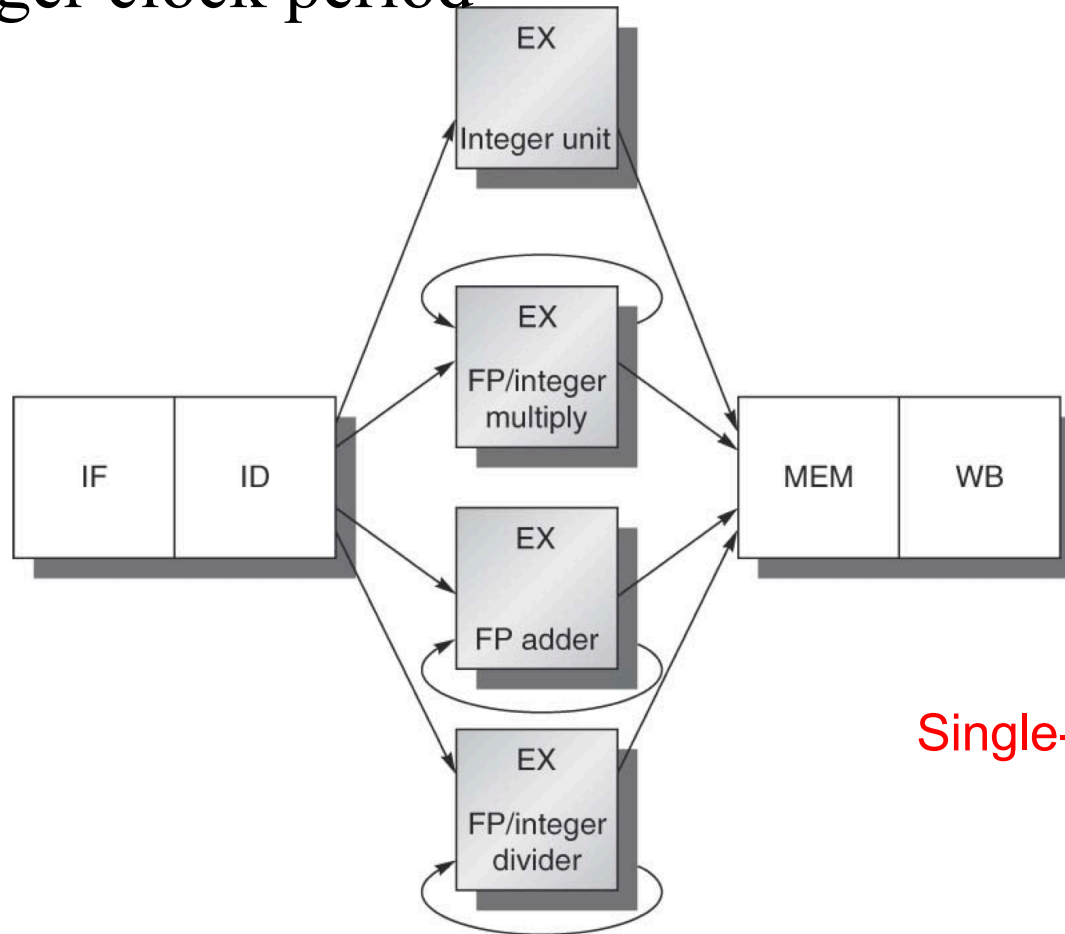


Floating-Point Unit (FPU)

- Much more hardware than an integer unit
 - A simple FPU takes 150,000 gates. Verification complex. Some exceptions specific to floating point.
 - Integer FU to the order of thousands
- Common to have several FPU's
 - Some integer, some floating point
- Common to have different types of FPU's: Fadd, Fmul, Fdiv, ...
- An FPU may be pipelined, partially pipelined or not pipelined
- To operate several FPU's concurrently the FP register file needs to have more read and write ports

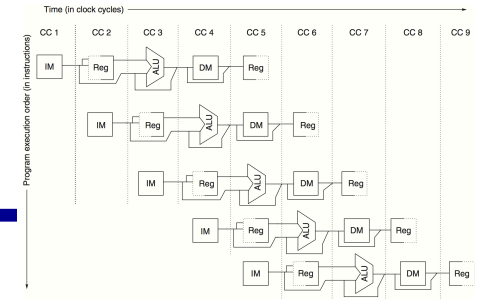
Unpipelined FP EXE Stage

- FP takes loops to compute
- Much longer clock period



Single-cycle FPU is a bad idea

Latency and Interval



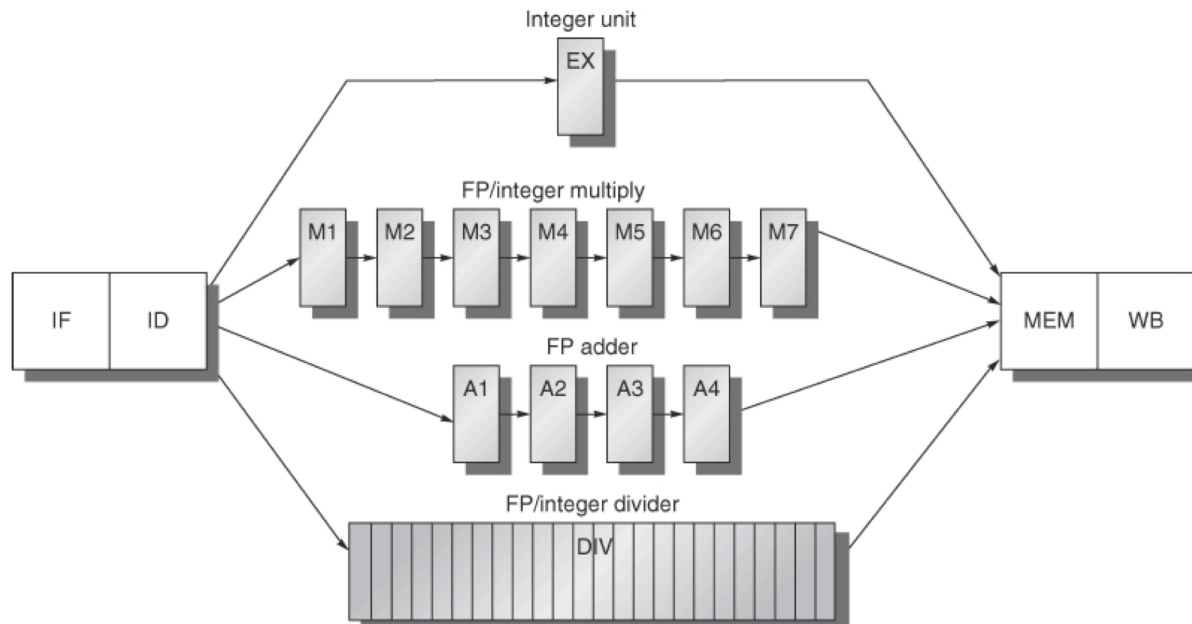
- Latency
 - The number of intervening cycles between an instruction that produces a result and an instruction that uses the result.
 - Usually the number of stages after EX that an instruction produces a result
 - **ALU Integer 0, Load latency 1**
- Initiation or repeat interval

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

Figure C.34 Latencies and initiation intervals for functional units.

Pipelined FP EXE

- Increased stall for RAW hazards



MUL.D	IF	ID	<i>M1</i>	M2	M3	M4	M5	M6	M7	MEM	WB
ADD.D		IF	ID	<i>A1</i>	A2	A3	A4	MEM	WB		
L.D			IF	ID	<i>EX</i>	MEM	WB				
S.D				IF	ID	<i>EX</i>	<i>MEM</i>	WB			

Figure C.36 The pipeline timing of a set of independent FP operations. The stages in italics show where data are needed, while the stages in bold show where a result is available. The ".D" extension on the instruction mnemonic indicates double-precision (64-bit) floating-point operations. FP loads and stores use a 64-bit path to memory so that the pipelining timing is just like an integer load or store.

Breaking Our Assumption of Integer Pipeline

- The divide unit is not fully pipelined
 - structural hazards can occur
 - need to be detected and stall incurred.
- The instructions have varying running times
 - the number of register writes required in a cycle can be > 1
- Instructions no longer reach WB in order
 - Write after write (WAW) hazards are possible
 - Note that write after read (WAR) hazards are not possible, since the register reads always occur in ID.
- Instructions can complete in a different order than they were issued (out-of-order complete)
 - causing problems with exceptions
- Longer latency of operations
 - stalls for RAW hazards will be more frequent.

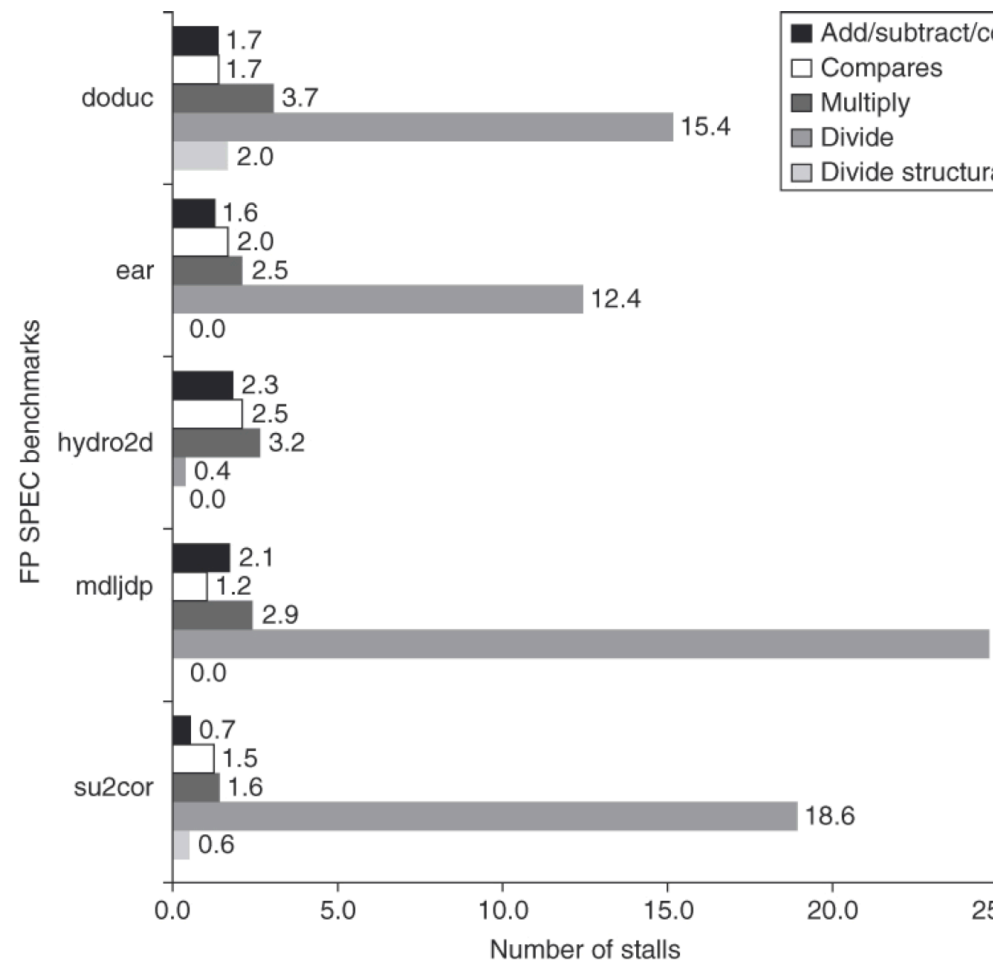
Hazards and Forwarding for Longer-Latency Pipeline

Instruction	Clock cycle number																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
L.D F4,0(R2)	IF	ID	EX	MEM	WB												
MUL.D F0,F4,F6		IF	ID	Stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADD.D F2,F0,F8			IF	Stall	ID	Stall	Stall	Stall	Stall	Stall	Stall	A1	A2	A3	A4	MEM	WB
S.D F2,0(R2)					IF	Stall	Stall	Stall	Stall	Stall	Stall	ID	EX	Stall	Stall	Stall	MEM

Figure C.37 A typical FP code sequence showing the stalls arising from RAW hazards. The longer pipeline substantially raises the frequency of stalls versus the shallower integer pipeline. Each instruction in this sequence is dependent on the previous and proceeds as soon as data are available, which assumes the pipeline has full bypassing and forwarding. The S.D must be stalled an extra cycle so that its MEM does not conflict with the ADD.D. Extra hardware could easily handle this case.

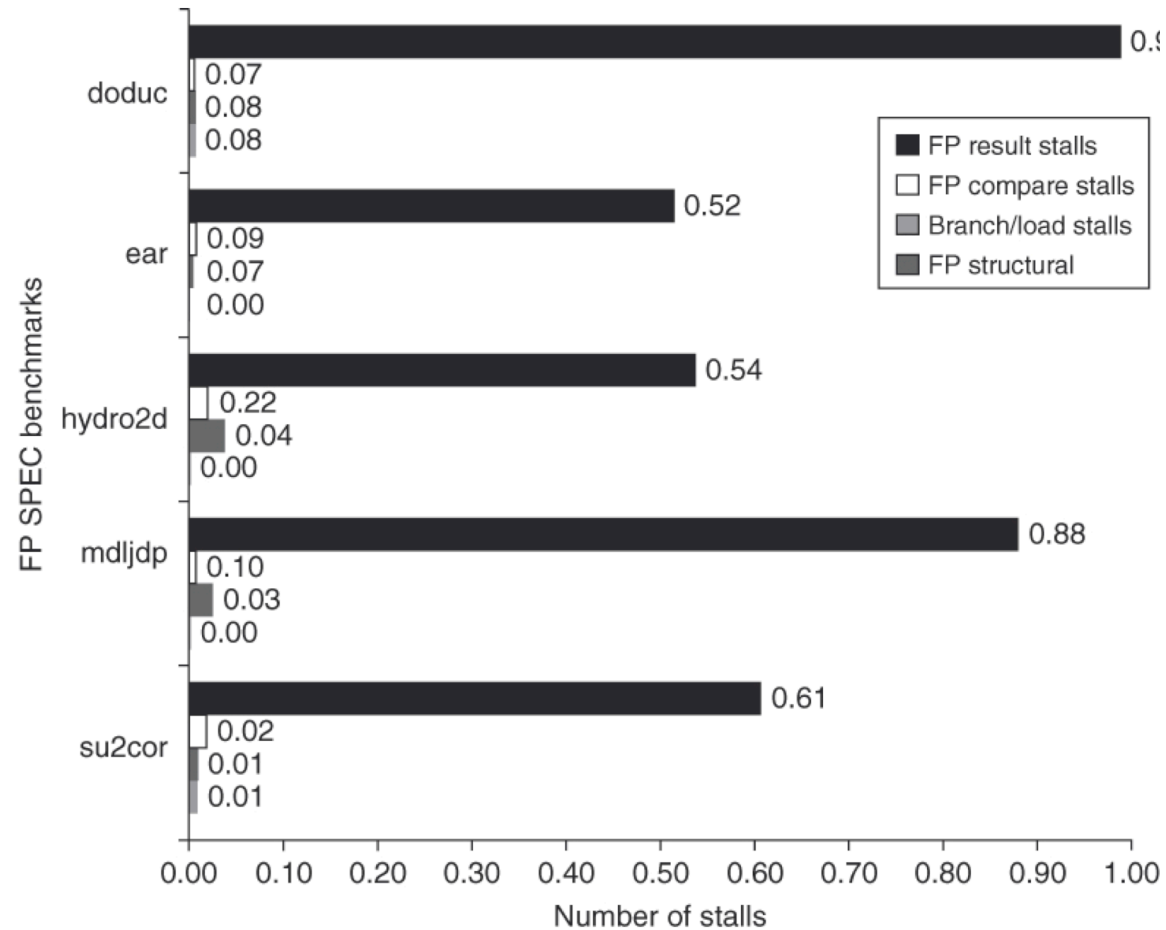
Stalls of FP Operations

- SPEC89 FP
- Latency average
- FP add, subtract, or convert
 - 1.7 cycles, or 56% of the latency (3 cycles).
- Multiplies and divides
 - 2.8 and 14.2, respectively, or 46% and 59% of the corresponding latency.
- Structural hazards for divides are rare
 - since the divide frequency is low.



Stalls per FP Operation

- The total number of stalls per instruction
 - ranges from 0.65 for su2cor to 1.21 for doduc, with an average of 0.87.



- FP result stalls dominate in all cases, with an average of 0.71 stalls per instruction, or 82% of the stalled cycles.

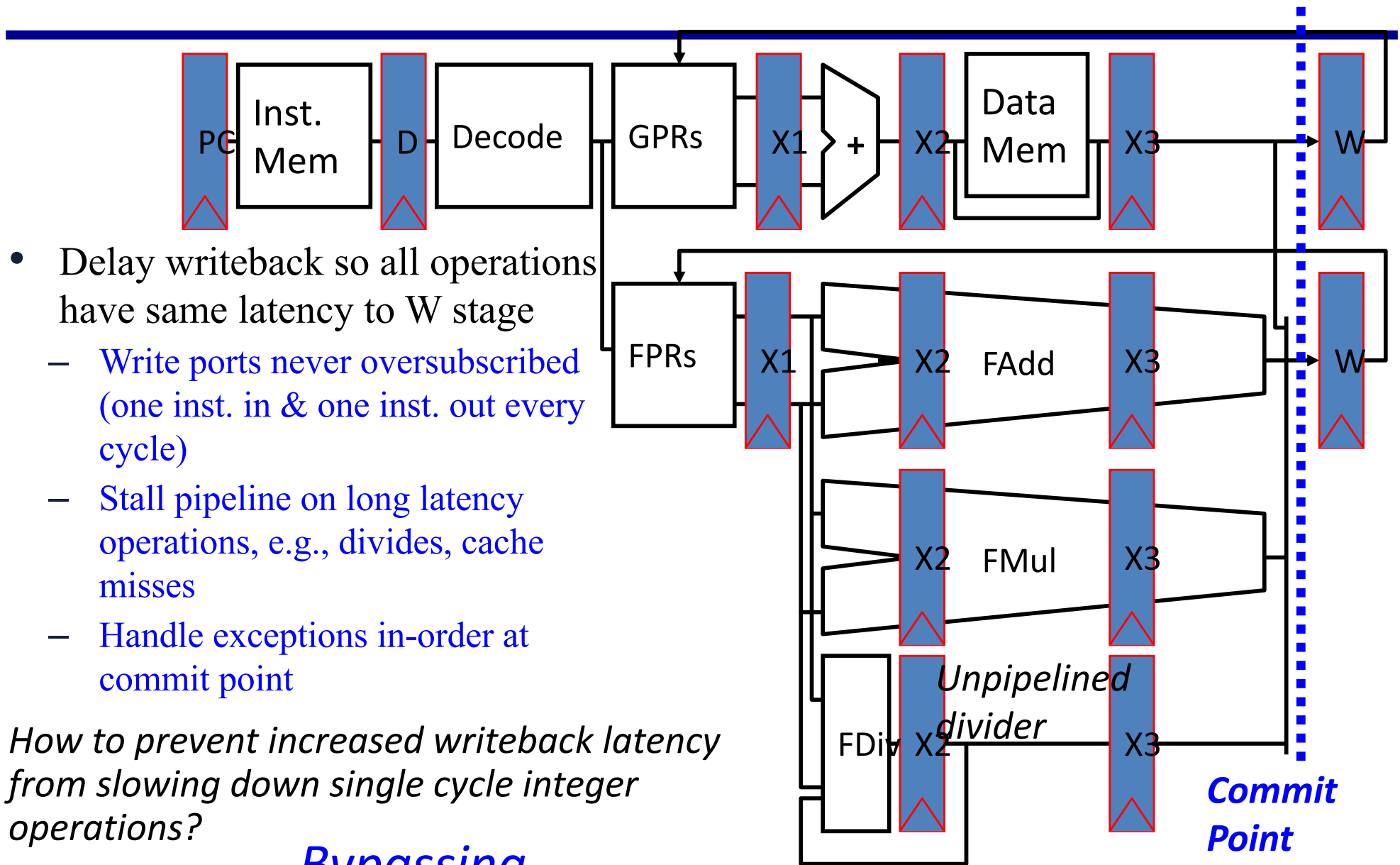
Problems Arising From Writes

- If we issue one instruction per cycle, how can we avoid structural hazards at the writeback stage and out-of-order writeback issues?
- WAW Hazards**

Instruction	Clock cycle number										
	1	2	3	4	5	6	7	8	9	10	11
MUL.D F0,F4,F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
ADD.D F2,F4,F6				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
L.D F2,0(R2)							IF	ID	EX	MEM	WB

Figure C.38 Three instructions want to perform a write-back to the FP register file simultaneously, as shown in clock cycle 11. This is *not* the worst case, since an earlier divide in the FP unit could also finish on the same clock. Note that although the MUL.D, ADD.D, and L.D all are in the MEM stage in clock cycle 10, only the L.D actually uses the memory, so no structural hazard exists for MEM.

Complex In-Order Pipeline



- Delay writeback so all operations have same latency to W stage
 - Write ports never oversubscribed (one inst. in & one inst. out every cycle)
 - Stall pipeline on long latency operations, e.g., divides, cache misses
 - Handle exceptions in-order at commit point

How to prevent increased writeback latency from slowing down single cycle integer operations?

Bypassing

Floating-Point ISA

- Interaction between floating-point datapath and integer datapath is determined by ISA
- RISC-V ISA
 - separate register files for FP and Integer instructions
 - the only interaction is via a set of move/convert instructions (some ISA's don't even permit this)
 - separate load/store for FPR's and GPR's (general purpose registers) but both use GPR's for address calculation
 - FP compares write integer registers, then use integer branch

Realistic Memory Systems

Common approaches to improving memory performance:

- Caches - single cycle except in case of a miss
=>stall
- Banked memory - multiple memory accesses
=> bank conflicts
- split-phase memory operations (separate memory request from response), many in flight
=> out-of-order responses

Latency of access to the main memory is usually much greater than one cycle and often unpredictable

Solving this problem is a central issue in computer architecture

Multiple-Cycles MEM Stage

- MIPS R4000
- IF: First half of instruction fetch; PC selection actually happens here, together with initiation of instruction cache access.
- IS: Second half of instruction fetch, complete instruction cache access.
- RF: Instruction decode and register fetch, hazard checking, and instruction cache hit detection.
- EX: Execution, which includes effective address calculation, ALU operation, and branch-target computation and condition evaluation.
- DF: Data fetch, first half of data cache access.
- DS: Second half of data fetch, completion of data cache access.
- TC: Tag check, to determine whether the data cache access hit.
- WB: Write-back for loads and register-register operations.

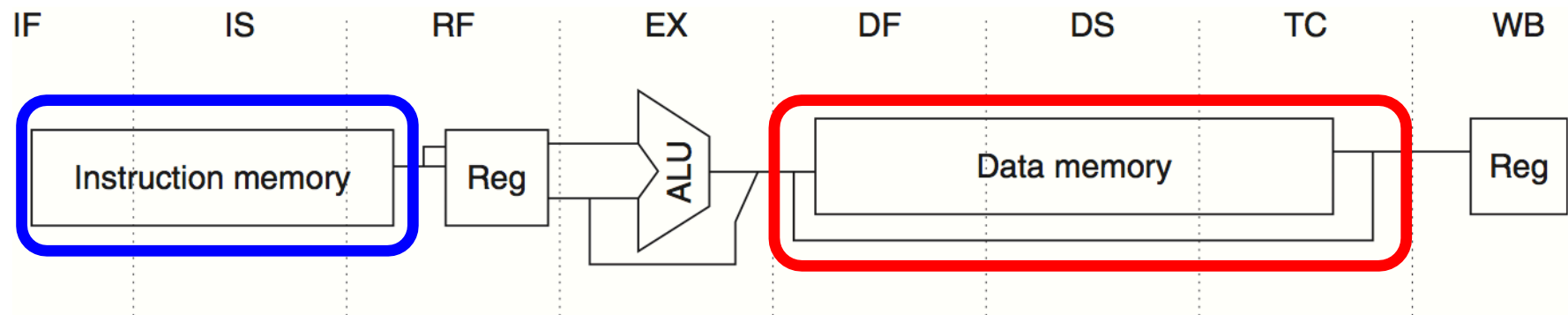
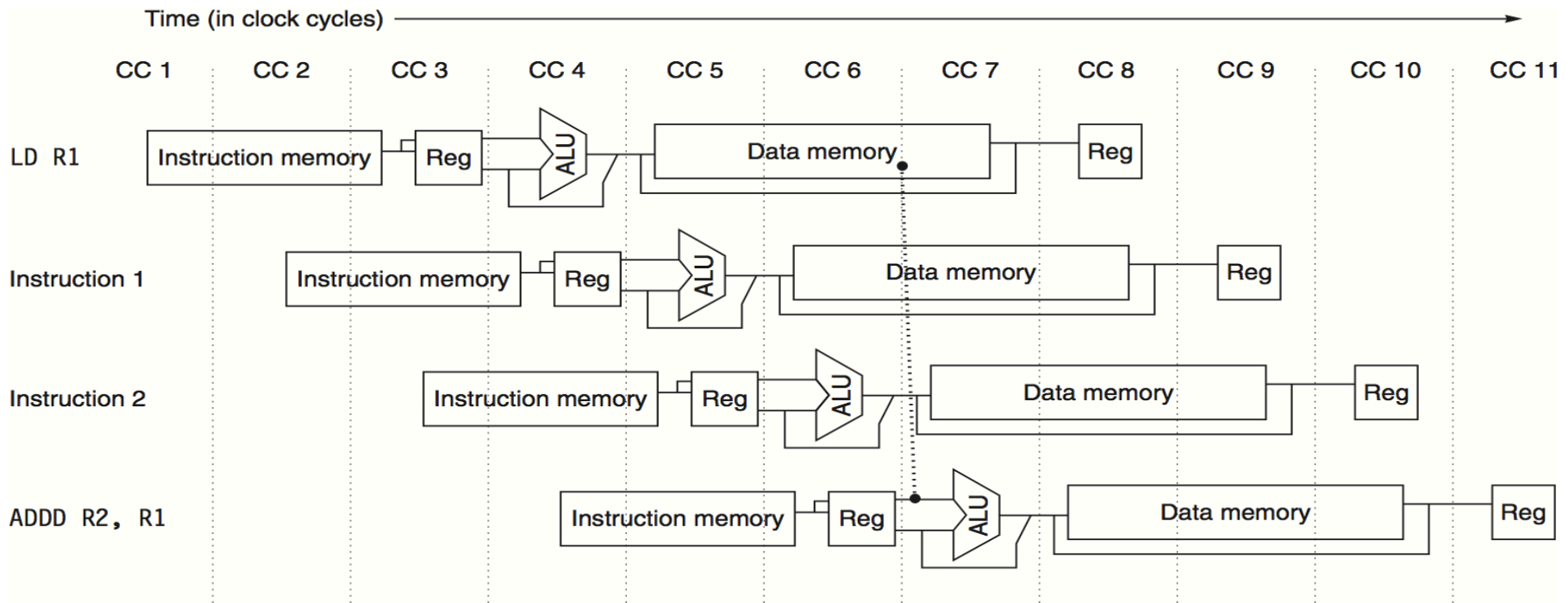


Figure C.41 The eight-stage pipeline structure of the R4000 uses pipelined instruction and data caches. The pipeline stages are labeled and their detailed function is described in the text. The vertical dashed lines represent the

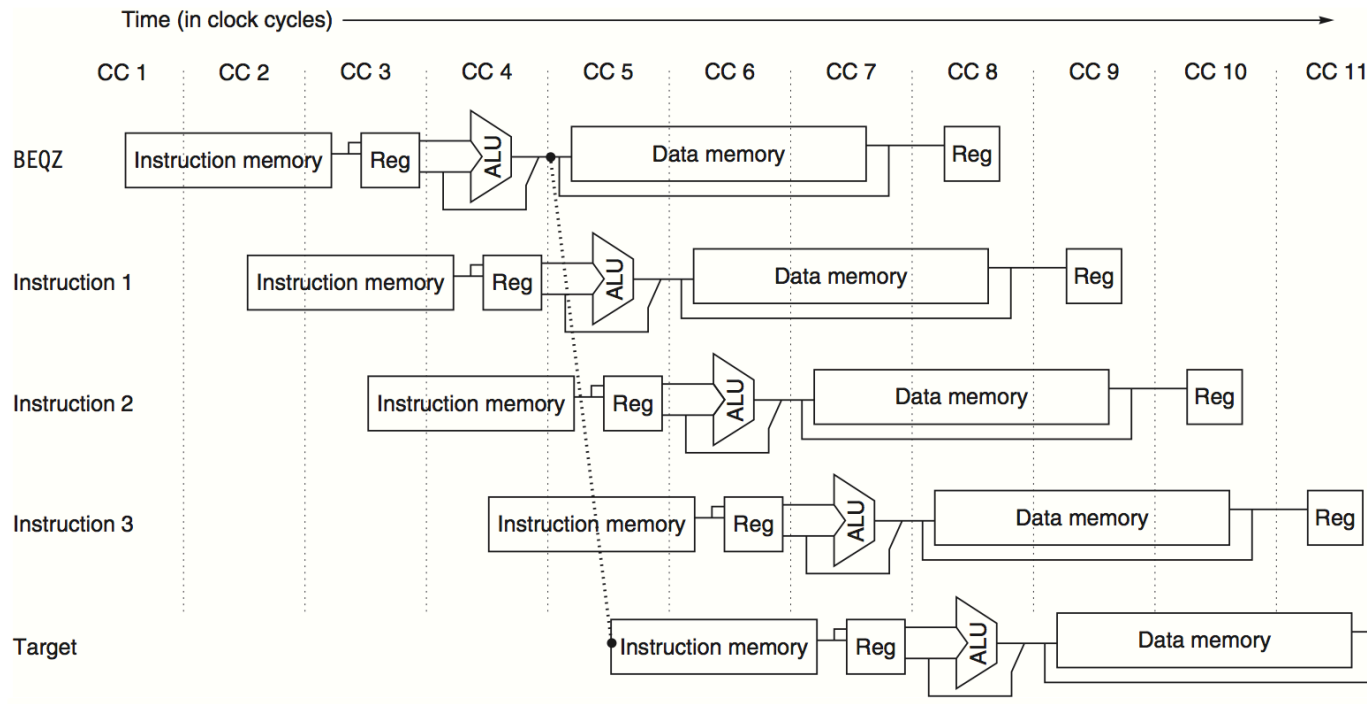
2-Cycles Load Delay



	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
LD R1,...	IF	IS	RF	EX	DF	DS	TC	WB	
DADD R2,R1,...		IF	IS	RF	Stall	Stall	EX	DF	DS
DSUB R3,R1,...			IF	IS	Stall	Stall	RF	EX	DF
OR R4,R1,...				IF	Stall	Stall	IS	RF	EX

Figure C.43 A load instruction followed by an immediate use results in a 2-cycle stall. Normal forwarding paths can be used after 2 cycles, so the DADD and DSUB get the value by forwarding after the stall. The OR instruction gets

3-Cycle Branch Delay when Taken



	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Branch instruction	IF	IS	RF	EX	DF	DS	TC	WB	
Delay slot		IF	IS	RF	EX	DF	DS	TC	WB
Stall			Stall	Stall	Stall	Stall	Stall	Stall	Stall
Stall				Stall	Stall	Stall	Stall	Stall	Stall
Branch target					IF	IS	RF	EX	DF

3.1 ILP: Concepts and Challenges

- Instruction-Level Parallelism (ILP): overlap the execution of instructions to improve performance.
- 2 approaches to exploit ILP
 - Rely on **hardware** to help discover and exploit the parallelism dynamically (e.g., Pentium 4, AMD Opteron, IBM Power), and
 - Rely on **software** technology to find parallelism, statically at compile-time (e.g., Itanium 2)
- Pipelining Review (branch taken, wasted cycles in **RED**)

	1	2	3	4	5	6	7	8	9	10	11	12	13
ld x5 -32(x4)	IF	ID	EXE	MEM	WB								
ld x6 -16(x4)		IF	ID	EXE	MEM	WB							
add x6 x5 x6			IF	-	ID	EXE	MEM	WB					
add x6 x6 x6					IF	ID	EXE	MEM	WB				
BNEZ x6 L1						IF	ID	EXE	MEM	WB			
add x10 x4 #10							IF	ID	EXE	MEM	WB		
add x11 x5 #10								IF	ID	EXE	MEM	WB	
L1: add x12 x5 #10									IF	ID	EXE	MEM	WB

Improving Instruction Level Parallelism (ILP)

$$\text{Pipeline CPI} = \text{Ideal pipeline CPI} + \text{Structural Stalls} + \text{Data Hazard Stalls} + \text{Control Stalls}$$

Technique	Reduces	Section
Forwarding and bypassing	Potential data hazard stalls	C.2
Delayed branches and simple branch scheduling	Control hazard stalls	C.2
Basic compiler pipeline scheduling	Data hazard stalls	C.2, 3.2
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences	C.7
Loop unrolling	Control hazard stalls	3.2
Branch prediction	Control stalls	3.3
Dynamic scheduling with renaming	Stalls from data hazards, output dependences, and antidependences	3.4
Hardware speculation	Data hazard and control hazard stalls	3.6
Dynamic memory disambiguation	Data hazard stalls with memory	3.6
Issuing multiple instructions per cycle	Ideal CPI	3.7, 3.8
Compiler dependence analysis, software pipelining, trace scheduling	Ideal CPI, data hazard stalls	H.2, H.3
Hardware support for compiler speculation	Ideal CPI, data hazard stalls, branch hazard stalls	H.4, H.5

Figure 3.1 The major techniques examined in Appendix C, Chapter 3, and Appendix H are shown together with

Instruction-Level Parallelism (ILP): Basic Blocks

- **BB**: a straight-line code sequence with **no branches** in except to the entry and, no branches out except at the exit;

```

w = 0;
x = x + y;
y = 0;
if( x > z)
{
    y = x;
    x++;
}
else
{
    y = z;
    z++;
}
w = x + z;
    
```

Source Code

```

w = 0;
x = x + y;
y = 0;
if( x > z)
    
```

```

y = x;
x++;
    
```

```

y = z;
z++;
    
```

```

w = x + z;
    
```

Basic Blocks

B1

```

w = 0;
x = x + y;
y = 0;
if( x > z)
    
```

B2

```

y = x;
x++;
    
```

B3

```

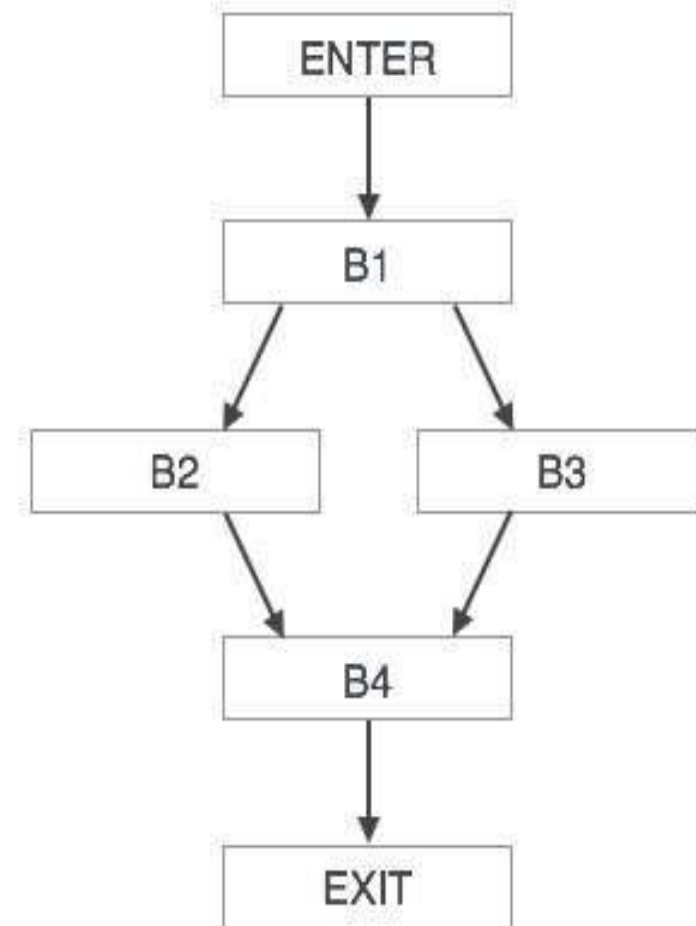
y = z;
z++;
    
```

B4

```

w = x + z;
    
```

Basic Blocks

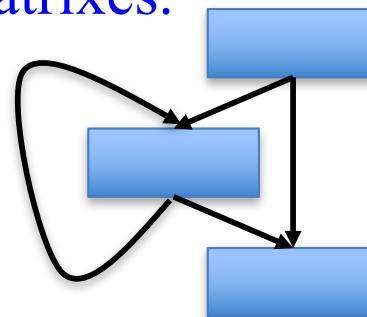


Flow Graph

Instruction-Level Parallelism (ILP)

- Inside a Basic Block (BB), ILP is quite small
- **Average dynamic branch frequency 15% to 25%**
 - 3 to 6 instrs execute between a pair of branches.
 - Plus instructions in BB likely to depend on each other.
- **To obtain substantial performance enhancements, we must exploit ILP across basic blocks. (ILP → LLP)**
 - **Loop-Level Parallelism:** to exploit parallelism among iterations of a loop. E.g., add two matrixes.

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



B1

```
w = 0;  
x = x + y;  
y = 0;  
if( x > z)
```

B2

```
y = x;  
x++;
```

B3

```
y = z;  
z++;
```

B4

```
w = x + z;
```

Basic Blocks

Data Dependences and Hazards


- Three data dependence: *data dependences* (true data dependences), *name dependences*, and *control dependences*.
 1. Instruction i produces a result that may be used by instruction j ($i \rightarrow j$), or
 2. Instruction j is data dependent on instruction k , and instruction k is data dependent on instruction i ($i \rightarrow k \rightarrow j$, dependence chain).
- For example, a code sequence

```
Loop: FLD      F0, 0(x1)      ;F0=array element
      FADD.D  F4, F0, F2     ;add scalar in f2
      FSD     F4, 0(x1)     ;store result
      ADDI   x1, x1, #-8     ;decrement pointer 8 bytes
      BNE   x1, x2, Loop    ;branch x1!=x2
```

True Data Dependence

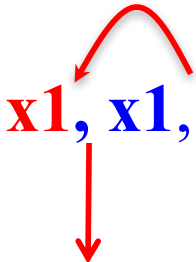
- Floating-point data part

```
Loop: FLD      F0, 0(x1) ;F0=array element
      FADD.D   F4, F0, F2 ;add scalar in f2
      FSD      F4, 0(x1) ;store result
```



- Integer data part

```
ADDI   x1, x1, #-8 ;decrement pointer
                          ;8 bytes (per DW)
BNE    x1, x2, Loop ;branch x1!=x2
```

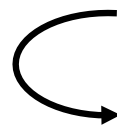


† This type is called a Read After Write (RAW) dependency.

True Data Dependence and RAW Hazards

- Instr_J is **data dependent** (aka **true dependence**) on Instr_I:

1) Instr_J tries to read operand before Instr_I writes it;

 I: **FLD F0, 0(x1) ;F0=array element**
J: **FADD.D F4, F0, F2 ;add scalar in f2**

2) Or Instr_J is data dependent on Instr_K which is dependent on Instr_I.

- If two instructions are data dependent, they cannot execute simultaneously or be **completely** overlapped.
- Data dependence in instruction sequence → data dependence in source code → **effect of original data dependence must be preserved.**
- If data dependence caused a hazard in pipeline, called a **Read After Write (RAW) hazard.**

True Data Dependencies → RAW Hazards for ILP

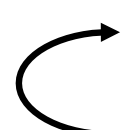
- **HW/SW must preserve program order**: instructions would execute in order if executed sequentially as determined by original source program.
 - **Dependences are a property of programs.**
- Presence of dependence indicates **potential** for a hazard, but actual hazard and length of any stall is property of the **pipeline**.
- Importance of the data dependencies.
 - 1) Indicates the possibility of a hazard;
 - 2) Determines order in which results must be calculated;
 - 3) Sets an upper bound on how much parallelism can possibly be exploited.
- **HW/SW goal: exploit parallelism by preserving program order only where it affects the outcome of the program.**

Detection of True Data Dependency

- Data value being dependent on between instructions either through **registers** or **through memory locations**.
- When the data flow occurs in a register
 - **Detecting the dependence is straightforward since the register names are fixed in the instrs within BB, interlock**
 - **More complicated between BB**
 - **branches intervene and correctness concerns force a compiler or hardware to be conservative.**
- Dependences that flow through memory locations are more difficult to detect,
 - **100(x4) and 20(x6) may be identical memory addresses.**
 - **The effective address of a load or store may change from one execution of the instruction to another**
 - **so that 20(x4) and 20(x4) may be different**

Name Dependence #1: Anti-dependence

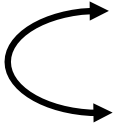
- **Name dependence**: when 2 instructions use same register or memory location, called a **name**, but **no flow of data between the instructions associated with that name**;
- 2 versions of name dependence (WAR and WAW).
- Instr_J writes operand *before* Instr_I reads it

 I: **sub r4,r1,r3**
 J: **add r1,r2,r3**
 K: **mul r6,r1,r7**

- Called an “anti-dependence” by compiler writers. This results from reuse of the name “r1”.
- If anti-dependence caused a hazard in the pipeline, called a **Write After Read (WAR) hazard**.

Name Dependence #2: Output dependence

- Instr_J writes operand before Instr_I writes it.

 I: **sub r1,r4,r3**
 J: **add r1,r2,r3**
 K: **mul r6,r1,r7**

- Called an “**output dependence**” by compiler writers. This also results from the reuse of name “**r1**”
- If anti-dependence caused a hazard in the pipeline, called a **Write After Write (WAW) hazard**.
- Instructions involved in a name dependence can execute simultaneously **if name used in instructions is changed** so instructions do not conflict.
 - **Register renaming resolves name dependence for regs;**
 - **Either by compiler or by HW.**

Control Dependencies

- Every instruction is control dependent on some set of branches
- **Control dependencies must be preserved to preserve program order.**

```
if p1 {  
    S1;  
};
```

```
if p2 {  
    S2;  
}
```

- S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1.

Control Dependence

- Two constraints imposed by control dependence
 1. An instruction that is dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch;
 2. An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch.
- Control dependence need not be preserved
 - Willing to execute instructions that should not have been executed
 - violating the control dependences, ok if can do so without affecting correctness of the program
- Not just branch or jump
 - Exception

Exception Behavior

- Preserving exception behavior
 - Any changes in instruction execution order must not change how exceptions are raised in program (\Rightarrow no new exceptions).

- Example

```
ADD    X2, X3, X4
BEQ    X2, X0, L1
Ld     X1, 0(X2)
```

L1:

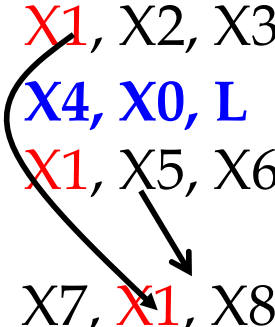
† Assume branches not delayed.

- Problem with moving LW before BEQZ even if branch is not taken?
 - LW may cause memory protection exception

Preserving Data Flow

- **Data flow**: actual flow of data values among instructions that produce results and those that consume them.
 - Branches make flow dynamic, determine which instruction is supplier of data.
- Example

```
ADD      X1, X2, X3
BEQ      X4, X0, L
SUB      X1, X5, X6
L:       ...
OR       X7, X1, X8
```



- X1 of OR depends on ADD or SUB?
 - Must preserve data flow on execution.

3.2 Basic Compiler Techniques for Exposing ILP

- This code, add a scalar to a vector

```
for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

- Assume following latencies for all examples
 - Ignore delayed branch in these examples

Instruction producing result	Instruction using result	Latency in cycles
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

Figure 3.2 Latencies of FP operations used in this chapter.

Latencies

- 5 stage pipeline
- Branches have one cycle delay
- Load → EXE-USE: 1 cycle delay

Instruction producing result	Instruction using result	Latency in clock cycles
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

Figure 3.2 Latencies of FP operations used in this chapter. The last column is the number of intervening clock cycles needed to avoid a stall. These numbers are similar to the average latencies we would see on an FP unit. The latency of a floating-point load to a store is 0 because the result of the load can be bypassed without stalling the store. We will continue to assume an integer load latency of 1 and an integer ALU operation latency of 0 (which includes ALU operation to branch).

FP Loop: Where are the Hazards?

- First translate into MIPS/RISC-V code
 - To simplify, assume 8 is lowest address
 - R1 stores the address of X[999] when the loop starts

L_c **for (i=999; i>=0; i=i-1)**
 x[i] = x[i] + s;

```
Loop:  fld      f0,0(x1)      //f0=array element
       fadd.d   f4,f0,f2     //add scalar in f2
       fsd     f4,0(x1)     //store result
       addi    x1,x1,-8     //decrement pointer
                               //8 bytes (per DW)
       bne     x1,x2,Loop   //branch x1≠x2
```

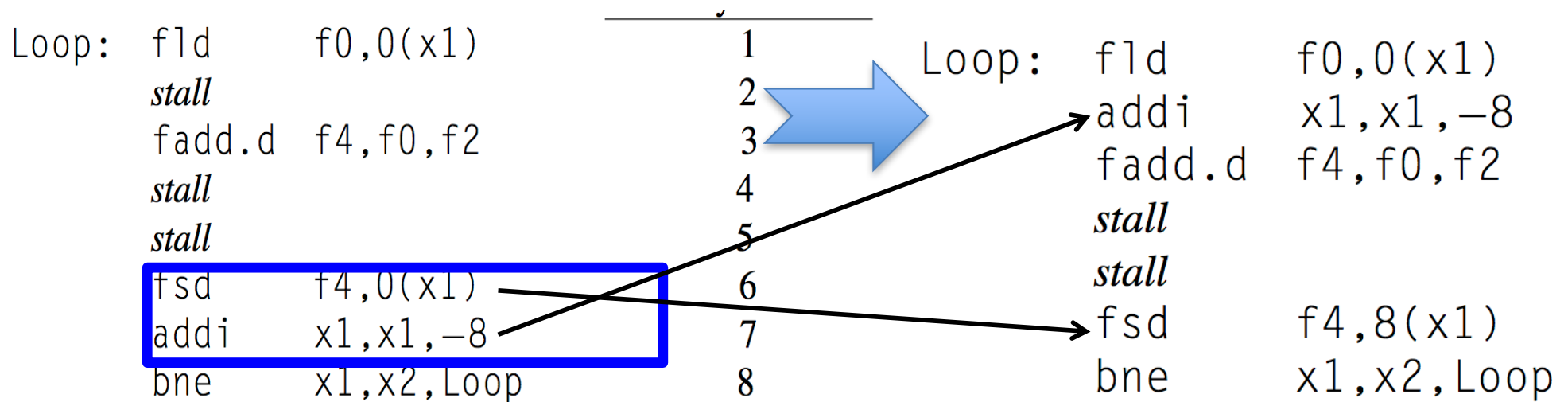
FP Loop Showing Stalls: V1

- **Example 3-1 (p.178)** *scheduled* and *unsc* delays from floating delayed branches.
- Answer

Instruction producing result	Instruction using result	Latency in clock cycles
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

		Clock cycle issued	
Loop:	f1d f0,0(x1)	1	
	<i>stall</i>	2	9 clock cycles, 6 for useful work Rewrite code to minimize stalls?
	fadd.d f4,f0,f2	3	
	<i>stall</i>	4	
	<i>stall</i>	5	
	fsd f4,0(x1)	6	
	addi x1,x1,-8	7	
	bne x1,x2,Loop	8	

Revised FP Loop Minimizing Stalls: V2



- Swap ADDI and FSD by changing address of FSD

† **7 clock cycles**

† **3 for execution (FLD, FADD.D, FSD)**

† **4 for loop overhead; How make faster?**

Unroll Loop Four Times: V3

```
Loop: fld      f0,0(x1)
      fadd.d  f4,f0,f2
      fsd     f4,0(x1)      //drop addi & bne
      fld     f6,-8(x1)
      fadd.d  f8,f6,f2
      fsd     f8,-8(x1)    //drop addi & bne
      fld     f0,-16(x1)
      fadd.d  f12,f0,f2
      fsd     f12,-16(x1) //drop addi & bne
      fld     f14,-24(x1)
      fadd.d  f16,f14,f2
      fsd     f16,-24(x1)
      addi    x1,x1,-32
      bne     x1,x2,Loop
```

- 27 clock cycles ($6*4+3$), or 6.75 per iteration (Assumes R1 is multiple of 4) compared with 9 for unrolled/unscheduled

Unroll Loop Four Times

- 27 clock cycles ($6*4+3$), or 6.75 per iteration (Assumes R1 is multiple of 4) compared with 9 for unrolled/unscheduled
 - Reducing instrs for branch and loop bound calculation
 - Reduce branch stall
- Code size increases
 - 5 instructions to 14 instructions

Unrolling Loop in Real Program

- Do not usually know upper bound of loop.
- Suppose it is n , and we would like to unroll the loop to make k copies of the body.
- Instead of a single unrolled loop, we generate a pair of consecutive loops:
 - 1st executes $(n \bmod k)$ times and has a body that is the original loop;
 - 2nd is the unrolled body surrounded by an outer loop that iterates (n/k) times.
- For large values of n , most of the execution time will be spent in the unrolled loop.

Unrolled Loop That Minimizes Stalls: V4

```
Loop:  fld      f0,0(x1)
       fld      f6,-8(x1)
       fld      f0,-16(x1)
       fld      f14,-24(x1)
       fadd.d   f4,f0,f2
       fadd.d   f8,f6,f2
       fadd.d   f12,f0,f2
       fadd.d   f16,f14,f2
       fsd     f4,0(x1)
       fsd     f8,-8(x1)
       fsd     f12,16(x1)
       fsd     f16,8(x1)
       addi    x1,x1,-32
       bne     x1,x2,Loop
```

† **14 clock cycles**

Four Versions Compared

	Total Cycles (1000 Iterations)	Cycles Per Iterations	Code Sizes
V1: Original			
V2: Scheduled			
V3: Unrolled			
V4: Scheduled and Unrolled			

5 Loop Unrolling Decisions

- Requires understanding how one instruction depends on another and how the instructions can be changed or reordered given the dependences:
 1. Determine loop unrolling useful by finding that loop iterations were independent (except for maintenance code);
 2. Use different registers to avoid unnecessary constraints forced by using same registers for different computations;
 3. Eliminate the extra test and branch instructions and adjust the loop termination and iteration code;
 4. Determine that loads and stores in unrolled loop can be interchanged by observing that loads and stores from different iterations are independent;
 - Transformation requires analyzing *memory addresses* and finding that they do not refer to the same address.
 5. Schedule the code, preserving any dependences needed to yield the same result as the original code.

Limits to Loop Unrolling

- 3 Limits to Loop Unrolling
 1. Decrease in amount of overhead amortized with each extra unrolling.
 - ◆ Reducing the ratio of the portion that can not be optimized in Amdahl's Law.
 2. Growth in code size.
 - ◆ For larger loops, concern it increases the instruction cache miss rate.
 3. Register pressure: potential shortfall in registers created by aggressive unrolling and scheduling.
 - ◆ If not be possible to allocate all live values to registers, may lose some or all of its advantage.
- Loop unrolling reduces impact of branches on pipeline; another way is branch prediction.
 - We discuss it in section 3.3: Reducing Branch Costs with Prediction.

Summary

- Three kinds of data dependency
 - True data dependency
 - Name dependency
 - Control dependency
- Hazards from dependency
 - Stall the pipeline
- Compiler technology
 - Loop unrolling
 - Instruction Scheduling

When Safe to Unroll Loop?

- Example: Where are data dependencies?

(A,B,C distinct & nonoverlapping)

```
for (i=0; i<100; i=i+1) {  
    A[i+1] = A[i] + C[i];    /* S1 */  
    B[i+1] = B[i] + A[i+1]; /* S2 */  
}
```

1. S2 uses the value, A[i+1], computed by S1 in the same iteration.
2. S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1].

This is a “**loop-carried dependence**”: between iterations

- For our prior example, each iteration was distinct
 - In this case, iterations can't be executed in parallel, Right????

Does a loop-carried dependence mean there is no parallelism???

- Consider:

```
for (i=0; i < 8; i=i+1) {  
    A = A + C[i];      /* S1 */  
}
```

⇒ Could compute:

```
“Cycle 1”:  temp0 = C[0] + C[1];  
            temp1 = C[2] + C[3];  
            temp2 = C[4] + C[5];  
            temp3 = C[6] + C[7];
```

```
“Cycle 2”:  temp4 = temp0 + temp1;  
            temp5 = temp2 + temp3;
```

```
“Cycle 3”:  A = temp4 + temp5;
```

- Relies on associative nature of “+”.

3.3 Reducing Branch Costs with Prediction

- Because of the need to enforce control dependences through branch hazards and stall, branches will hurt pipeline performance.
 - Solution 1: loop unrolling → reduce branch instrs
 - Solution 2: by predicting how they will behave → reduce stalls
- SW/HW technology
 - SW: Static Branch Prediction, statically at compile time;
 - HW: Dynamic Branch Prediction, dynamically by the hardware at execution time.

Static Branch Prediction

- Appendix C showed scheduling code around *delayed branch*.
 - Reorder code around branches, need to predict branch statically when compile.
- Another and simplest scheme is to predict a branch as taken.
 - Average misprediction = untaken branch frequency = 34% SPEC. Unfortunately, from very accurate (59%) to highly accurate (9%).

```

Loop:   L.D      F0,0(R1)
        ADD.D   F4,F0,F2
        S.D     F4,0(R1)
        DADDUI  R1,R1,#-8

        BNE    R1,R2,Loop
    
```

999/1000 is correct for 1000 iterations

- More accurate scheme predicts branches using **profile information** collected from earlier runs, and modify prediction based on last run.

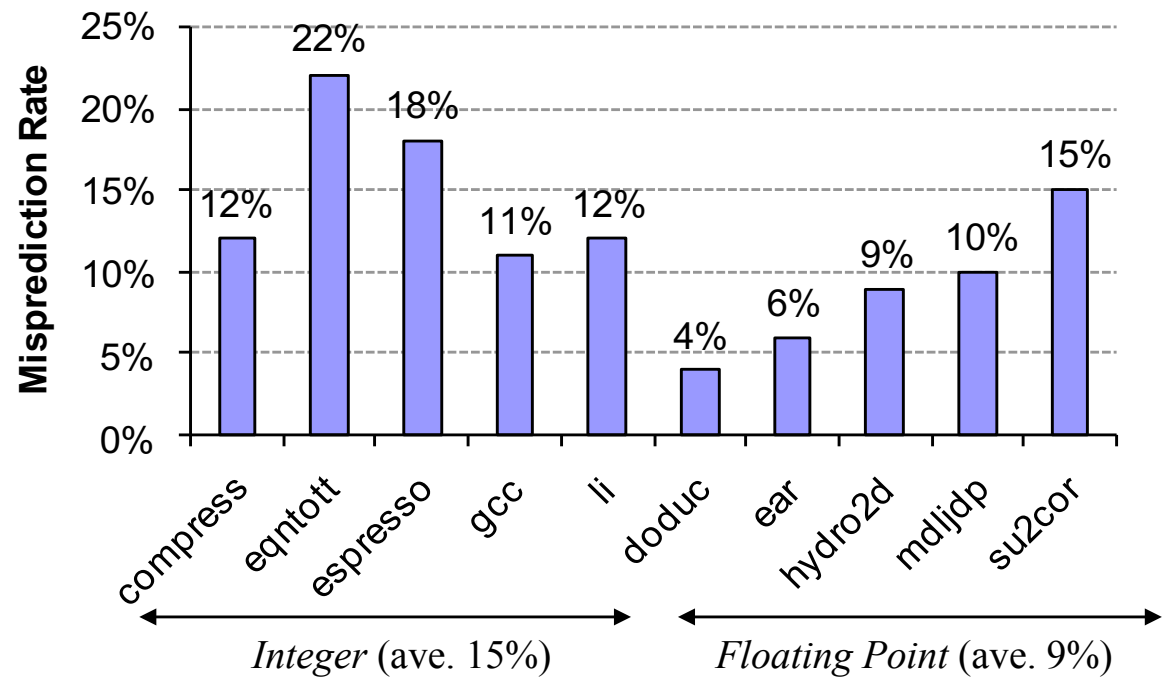


Figure C.17 The result of predict-taken in SPEC92

How It Works In Compiler (GCC)

Built-in Function: long `__builtin_expect` (*long exp, long c*)

You may use `builtin_expect` to provide the compiler with branch prediction information. In general, you should prefer to use actual profile feedback for this (`-fprofile-arcs`), as programmers are notoriously bad at predicting how their programs actually perform. However, there are applications in which this data is hard to collect.

The return value is the value of *exp*, which should be an integral expression. The semantics of the built-in are that it is expected that *exp* == *c*. For example:

```
if ( __builtin_expect (x, 0))
    foo ();
```

indicates that we do not expect to call `foo`, since we expect `x` to be zero. Since you are limited to integral expressions for *exp*, you should use constructions such as

```
if ( __builtin_expect (ptr != NULL, 1))
    foo (*ptr);
```

when testing pointer or floating-point values.

Collect Branch Statistics

```
gcc -Wall -fprofile-arcs -ftest-coverage cov.c
```

where `cov.c` is the name of the program file. This creates an instrumented [executable](#) which contains additional instructions that record the number of times each line of the program is executed. The option `-ftest-coverage` adds instructions for counting the number of times individual lines are executed, while `-fprofile-arcs` incorporates instrumentation code for each branch of the program. Branch instrumentation records how frequently different paths are taken through 'if' statements and other conditionals. The [executable](#) must then be run to create the coverage data. The data from the run is written to several files with the extensions `.bb` `.bbg` and `.da` respectively in the current [directory](#). This data can be analyzed using the `gcov` command and the name of a source file:

Dynamic Branch Prediction

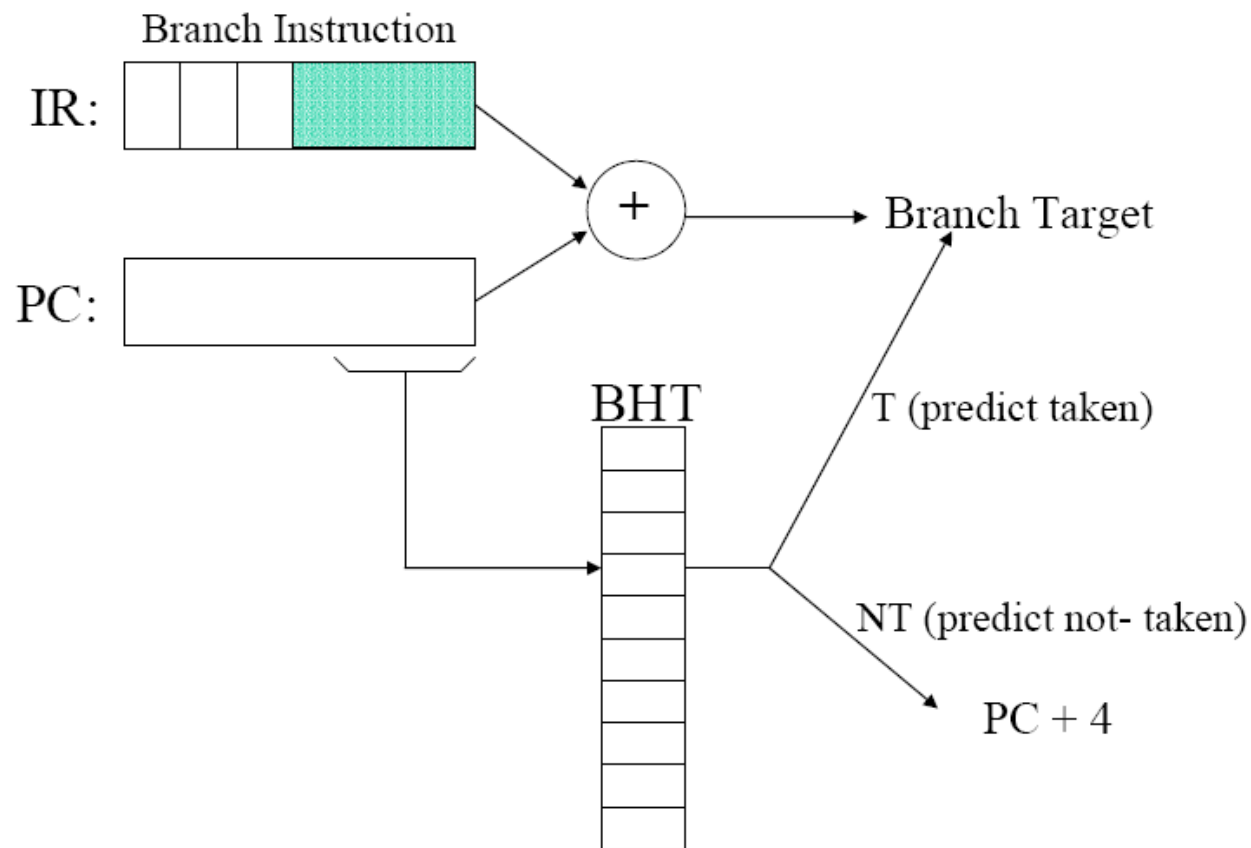
- Why does prediction work?
 - Underlying *algorithm* has regularities;
 - *Data* that is being operated on has regularities;
 - *Instruction sequence* has redundancies that are artifacts of way that humans/compiler think about problems.
- Is dynamic branch prediction better than static branch prediction?
 - Seems to be;
 - There are a small number of important branches in programs which have dynamic behavior.

Dynamic Branch Prediction

- Performance = $f(\text{accuracy, cost of misprediction})$
- **Branch History Table (also called Branch Prediction Buffer):**
lower bits of PC address index table of 1-bit values.
 - Says whether the branch was recently taken or not;
 - No address check.
- Problem: in a loop, 1-bit BHT will cause two mispredictions (average is 9 in 10 iterations before exit).
 - End of loop case, when it exits instead of looping as before;
 - First time through loop on *next* time through code, when it predicts exit instead of looping.

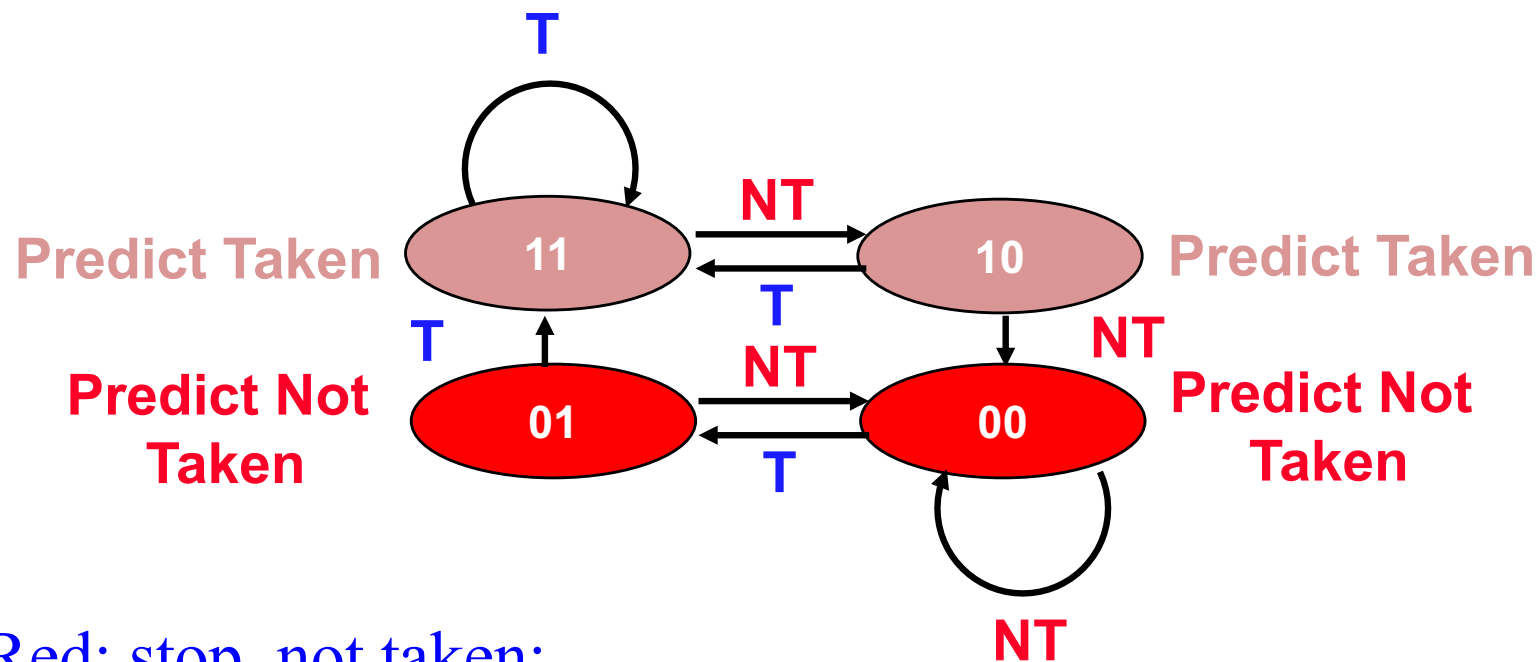
Basic Branch Prediction Buffers

- a.k.a. Branch History Table (BHT) - Small direct-mapped cache of T/NT bits.



Dynamic Branch Prediction

- Solution: 2-bit scheme where change prediction only if get misprediction *twice*.



- Red: stop, not taken;
- Blue: go, taken;
- Adds *hysteresis* to decision making process.

2-bit Scheme Accuracy

- Mispredict because either:
 - Wrong guess for that branch;
 - Got branch history of wrong branch when index the table.
- 4,096 entry table

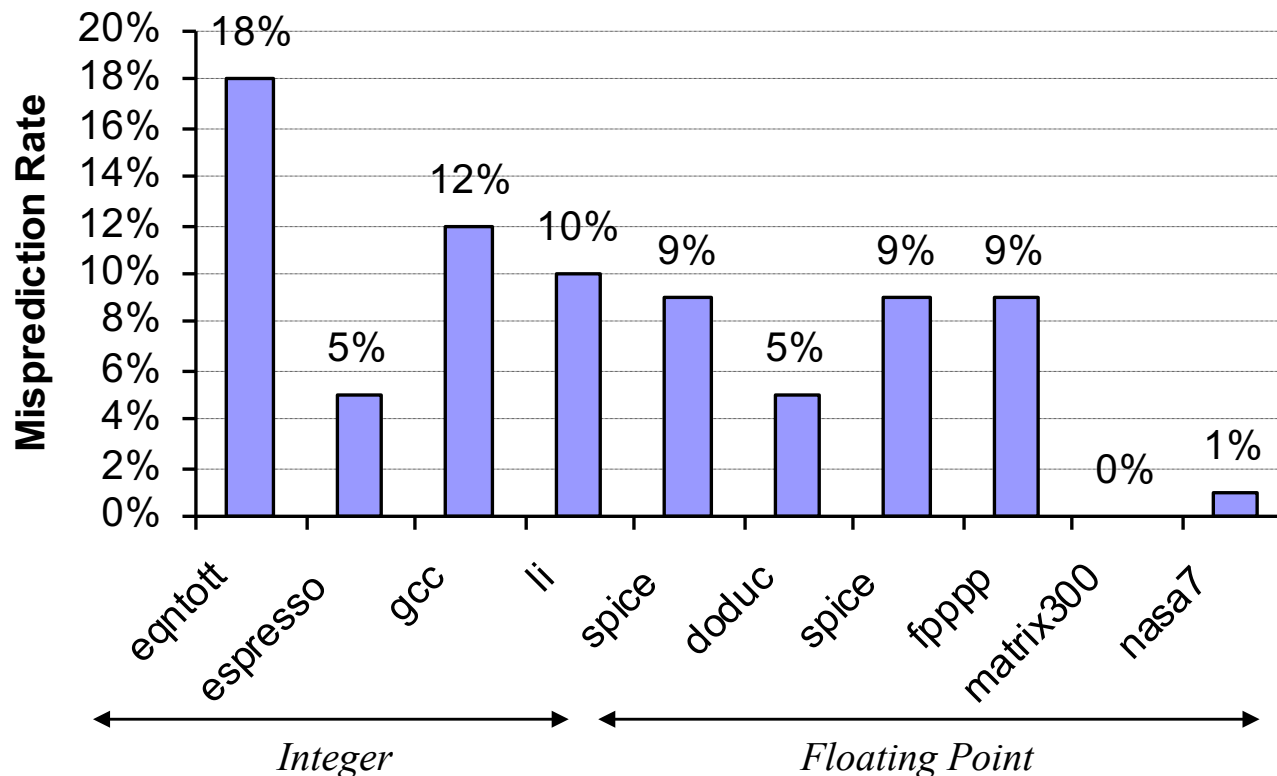
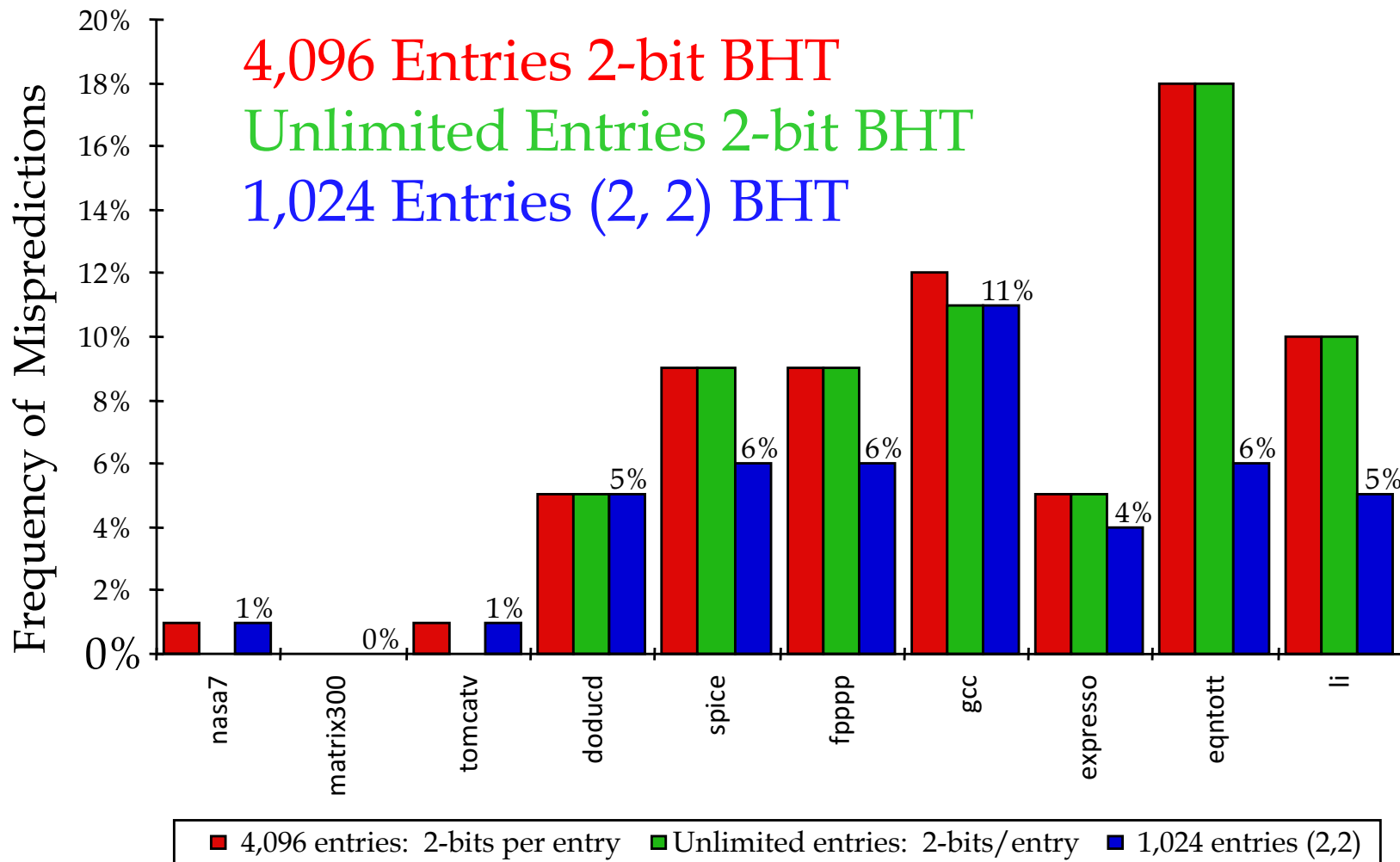


Figure 2.5 The result of 2-bit scheme in SPEC89

2-bit Scheme Accuracy

- The accuracy of the predictors for *integer programs*, which typically also have higher branch frequencies, is lower than for the *loop-intensive scientific programs*.
- Two ways to attack this problem
 - Large buffer size;
 - Increasing the accuracy of the scheme we use for each prediction.
- However, simply increasing the number of bits per predictor without changing the predictor structure also has little impact.
 - Single branch predictor V.S. correlating branch predictors.

Accuracy of Different Schemes



Improve Prediction Strategy By Correlating Branches

- Consider the worst case for the 2-bit predictor

```

if (aa==2)
    aa=0;
if (bb==2)
    bb=0;
if (aa != bb) {

```

if the first 2 fail then the 3rd will always be taken.

† Single level predictors can never get this case.

- Correlating predictors or 2-level predictors

- Correlation = what happened on the last branch
 - Note that the last correlator branch may not always be the same.
- Predictor = which way to go
 - 4 possibilities: which way the last one went chooses the prediction.
 - (Last-taken, last-not-taken) × (predict-taken, predict-not-taken)

```

DSUBUI R3, R1, #2
BNEZ R3, L1
DADD R1, R0, R0
L1:
DSUBUI R3, R2, #2
BNEZ R3, L2
DADD R2, R0, R0
L2:
DSUBU R3, R1, R2
BEQZ R3, L3

```

This branch is based on the Outcome of the previous 2 branches.

Correlated Branch Prediction

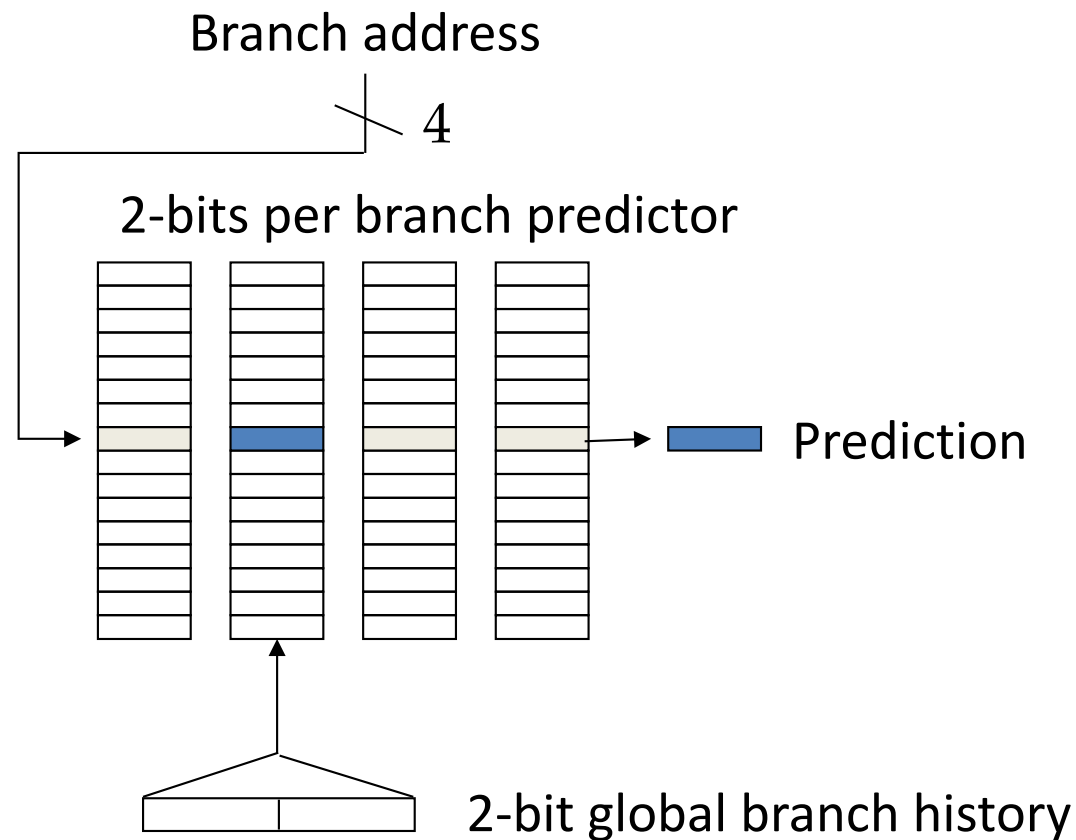
- Idea: record m most recently executed branches as taken or not taken, and use that pattern to select the proper n -bit branch history table.
- In general, (m, n) predictor means record last m branches to select between 2^m history tables, each with n -bit counters.
 - Thus, old 2-bit BHT is a $(0, 2)$ predictor.
- Global Branch History: m -bit shift register keeping T/NT status of last m branches.
- Each entry in table has m n -bit predictors.
- Total bits for the (m, n) BHT prediction buffer:

$$\text{Total_memory_bits} = 2^m \times n \times 2^p$$

- 2^m banks of memory selected by the global branch history (which is just a shift register)
 - e.g. a column address;
- Use p bits of the branch address to select row;
- Get the n predictor bits in the entry to make the decision.

Correlating Branches

- (2, 2) predictor
 - Behavior of recent 2 branches selects between four predictions of next branch, updating just that prediction.



Example of Correlating Branch Predictors

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

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

Example: Multiple Consequent Branches

```

if (d==0)
    d=1;
if (d==1)
    ...
L2:

```

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

if(d == 0) ;not taken
 d=1;
 else ;taken
 if(d==1) ;not taken
 else ;taken

If b1 is not taken, then b2 will be not taken

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.

1-bit predictor: consider d alternates between 2 and 0. All branches are mispredicted

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 1-bit predictor initialized to not taken. T stands for taken, NT for not taken.

Example: Multiple Consequent Branches

```

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

if(d == 0) ;not taken
d=1;
else ;taken
if(d==1) ;not taken
else ;taken

2-bits prediction : prediction if last branch not taken/ and prediction if last branch taken

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

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

(1,1) predictor - 1-bit predictor with 1 bit of correlation: last branch (either taken or not taken) decides which prediction bit will be considered or updated

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 1-bit predictor with 1 bit of correlation, initialized to not taken/not taken. T stands for taken, NT for not taken. The prediction used is shown in bold.

Branch Prediction with Neural Networks

- [1] D. Jimenez and C. Lin, “Dynamic branch prediction with perceptrons”, *Proc. of the 7th Int. Symp. on High Perf. Comp. Arch (HPCA-7)*, 2001.
- [2] D. Jimenez and C. Lin, “Neural methods for dynamic branch prediction”, *ACM Trans. on Computer Systems*, 2002.
- [3] A. Seznec, “Revisiting the perceptron predictor”, *Technical Report*, IRISA, 2004.
- [4] A. Seznec. An optimized *2bcgskew* branch predictor. Technical report Irisa, Sep 2003.
- [5] G. Loh. The frankenpredictor. In *The 1st JILP Championship Branch Prediction Competition (CBP-1)*, 2004
- [6] K. Aasaraai and A. Baniasadi Low-power Perceptrons
- [7] A. Seznec. The **O-GEometric History Length** branch predictor
- [8] M. Monchiero and G. Palermo The Combined Perceptron Branch Predictor
- [9] F. Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan, 1962.

Summary

- Branch Prediction
 - Static compiler-based prediction
 - Dynamic hardware-based prediction
 - Branch history table + Branch Target Buffer