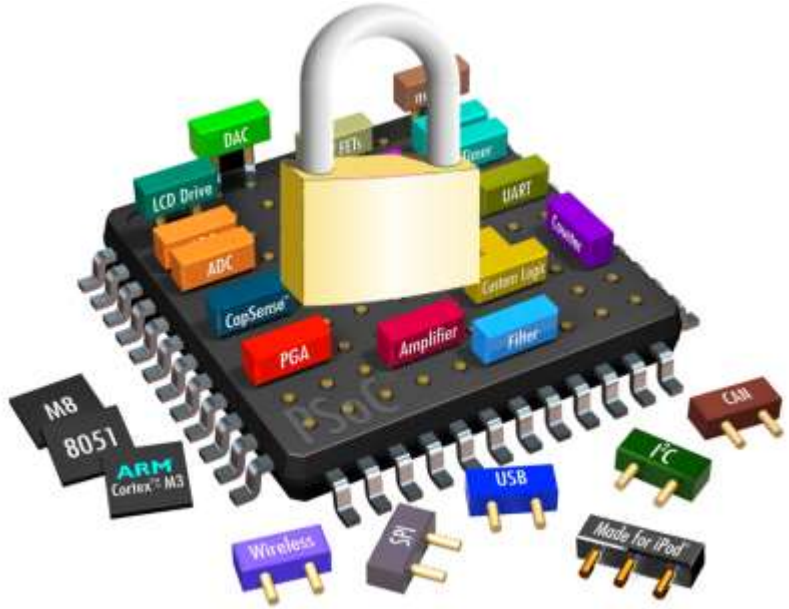
Advanced Computer Architecture

Instruction-Level Parallelism and Its Exploitation



Understanding Instruction-Level Parallelism

All processors since about 1985 use pipelining to overlap the execution of instructions and improve performance. This potential overlap among instructions is called *instruction-level parallelism* (ILP), since the instructions can be evaluated in parallel.

Dynamic Approach

Relies on hardware to discover and exploit parallelism dynamically at runtime

Dominates desktop and server markets (Intel Core series)

Static Approach

Relies on software technology to find parallelism at compile time

Common in personal mobile devices where energy efficiency is key (ARM Cortex-A8)

Despite enormous efforts, aggressive compiler-based approaches have not been successful outside of scientific applications. Understanding ILP limitations remains important in balancing ILP and thread-level parallelism.

Loop-Level Parallelism

The amount of parallelism available within a *basic block* (straight-line code sequence with no branches except at entry/exit) is quite small. For typical MIPS programs, the average dynamic branch frequency is between 15% and 25%, meaning only three to six instructions execute between branches.

The simplest way to increase ILP is to exploit parallelism among iterations of a loop, called *loop-level parallelism*.

```
for (i=0; i<=999; i=i+1)
    x[i] = x[i] + y[i];
```

Every iteration of this loop can overlap with any other iteration. We can convert loop-level parallelism into instruction-level parallelism by unrolling the loop either statically by the compiler or dynamically by the hardware.

An important alternative method for exploiting loop-level parallelism is using SIMD in both vector processors and GPUs, which can dramatically reduce instruction count by processing multiple data items in parallel.

Data Dependences

Name Dependences

- A name dependence occurs when two instructions use the same register or memory location, called a name, but there is no flow of data between the instructions associated with that name.

Antidependence

```
Read X
Write X
```

Output Dependence

```
Write X
Write X
```

- Because a name dependence is not a true dependence, instructions involved in a name dependence can execute simultaneously or be reordered, if the name (register number or memory location) used in the instructions is changed so the instructions do not conflict.

Data Hazards

A hazard exists when there is a name or data dependence between instructions close enough that their overlap during execution would change the order of access to the operand. We must preserve program order where it affects program outcome.

Suppose instruction i is executed before instruction j.

RAW (Read After Write)

j tries to read a source before i writes it, getting the old value

Most common hazard, corresponds to true data dependence

WAW (Write After Write)

j tries to write an operand before i writes it

Corresponds to output dependence

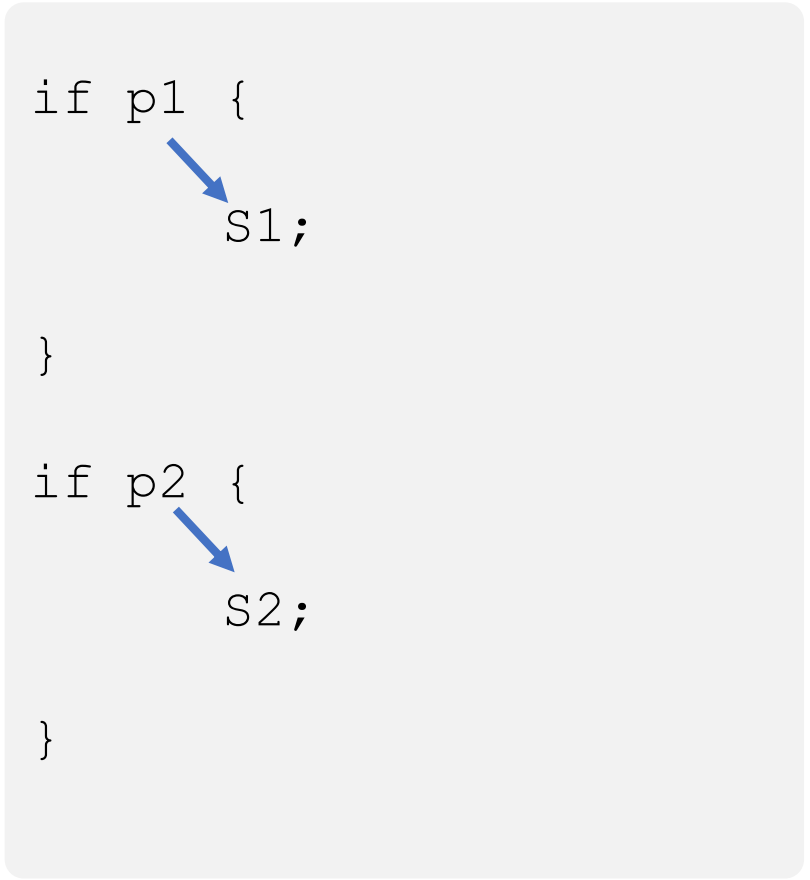
WAR (Write After Read)

j tries to write a destination before it's read by i

Arises from antidependence (name dependence)

Control Dependences

- A control dependence determines the ordering of an instruction, i, with respect to a branch instruction so that instruction i is executed in correct program order and only when it should be.



Example Code

```
for (i=0; i<=999; i=i+1)
    x[i] = x[i] + c;
```

```
Loop:   L.D     F0,0(R1)        ;F0=array element
        ADD.D   F4,F0,F2        ;add scalar in F2
        S.D     F4,0(R1)        ;store result
        DADDUI  R1,R1,#-8       ;decrement pointer
        BNE     R1,R2,Loop      ;branch R1!=R2
```

Loop Unrolling

Unroll the loop by 4 assuming R2 is a multiple of 32

```
Loop:   L.D     F0,0(R1)
        ADD.D   F4,F0,F2
        S.D     F4,0(R1)
        DADDUI  R1,R1,#-8
        BNE     R1,R2,Loop
```

```
Loop:   L.D     F0,0(R1)
        ADD.D   F4,F0,F2
        S.D     F4,0(R1)
        L.D     F6,-8(R1)
        ADD.D   F8,F6,F2
        S.D     F8,-8(R1)
        L.D     F10,-16(R1)
        ADD.D   F12,F10,F2
        S.D     F12,-16(R1)
        L.D     F14,-24(R1)
        ADD.D   F16,F14,F2
        S.D     F16,-24(R1)
        DADDUI  R1,R1,#-32
        BNE     R1,R2,Loop
```

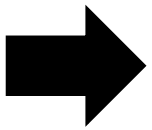
Delay

Producing result	Using result	Latency in clock cycle
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
FP ALU op	Branch	1
Load double	FP ALU op	1

```
Loop:   L.D     F0,0(R1)
        stall
        ADD.D   F4,F0,F2
        stall
        stall
        S.D     F4,0(R1)
        DADDUI  R1,R1,#-8
        stall
        BNE     R1,R2,Loop
```

Loop Unrolling and Scheduling

```
Loop:   L.D      F0,0(R1)
        stall
        ADD.D    F4,F0,F2
        stall
        stall
        S.D      F4,0(R1)
        L.D      F6,-8(R1)
        stall
        ADD.D    F8,F6,F2
        stall
        stall
        S.D      F8,-8(R1)
        L.D      F10,-16(R1)
        stall
        ADD.D    F12,F10,F2
        stall
        stall
        S.D      F12,-16(R1)
        L.D      F14,-24(R1)
        stall
        ADD.D    F16,F14,F2
        stall
        stall
        S.D      F16,-24(R1)
        DADDUI   R1,R1,#-32
        stall
        BNE      R1,R2,Loop
```



```
Loop:   L.D      F0,0(R1)
        L.D      F6,-8(R1)
        L.D      F10,-16(R1)
        L.D      F14,-24(R1)
        ADD.D    F4,F0,F2
        ADD.D    F8,F6,F2
        ADD.D    F12,F10,F2
        ADD.D    F16,F14,F2
        S.D      F4,0(R1)
        S.D      F8,-8(R1)
        S.D      F12,-16(R1)
        S.D      F16,-24(R1)
        DADDUI   R1,R1,#-32
        stall
        BNE      R1,R2,Loop
```

Key Transformations for Loop Unrolling

1

Identify Independent Iterations

Determine that loop iterations are independent except for maintenance code

2

Register Renaming

Use different registers to avoid unnecessary constraints from name dependencies

3

Code Simplification

Eliminate extra tests and branches; adjust loop termination and iteration code

4

Memory Analysis

Determine that loads and stores from different iterations are independent by analyzing memory addresses

5

Instruction Scheduling

Schedule code while preserving dependencies needed for correct results

Limitations of Loop Unrolling

Diminishing Returns

Each additional unroll provides less overhead reduction

A decrease in the amount of overhead is amortized with each unroll

Code Size Growth

Unrolling increases code size substantially

May cause instruction cache miss rate to increase

Register Pressure

Aggressive unrolling and scheduling increases the number of live values

May create register shortage that eliminates performance gains

Branch Predictors

- Branch predictors are specialized digital circuits inside modern CPUs that guess the outcome of conditional branches (like if-then-else statements) before the actual result is known.
- Their primary goal is to keep the CPU's instruction pipeline full and avoid costly stalls by speculatively executing the most likely path.
- Without prediction, the CPU would need to wait for each branch decision to be fully resolved, potentially wasting 10-20 clock cycles in modern processors with long pipelines.

Branch Prediction Strategies

Static Prediction

Uses simple fixed rules without history:

- Always predict branch not taken
- Predict backward branches (loops) as taken
- Predict based on branch opcode

Advantages: No hardware tables needed, power efficient

Dynamic Prediction

Uses runtime history stored in hardware tables:

- Two-bit saturating counters track branch behavior
- Pattern history tables remember sequences
- Tournament predictors combine multiple strategies

Advantages: Higher accuracy, adapts to program behavior

Advanced Techniques

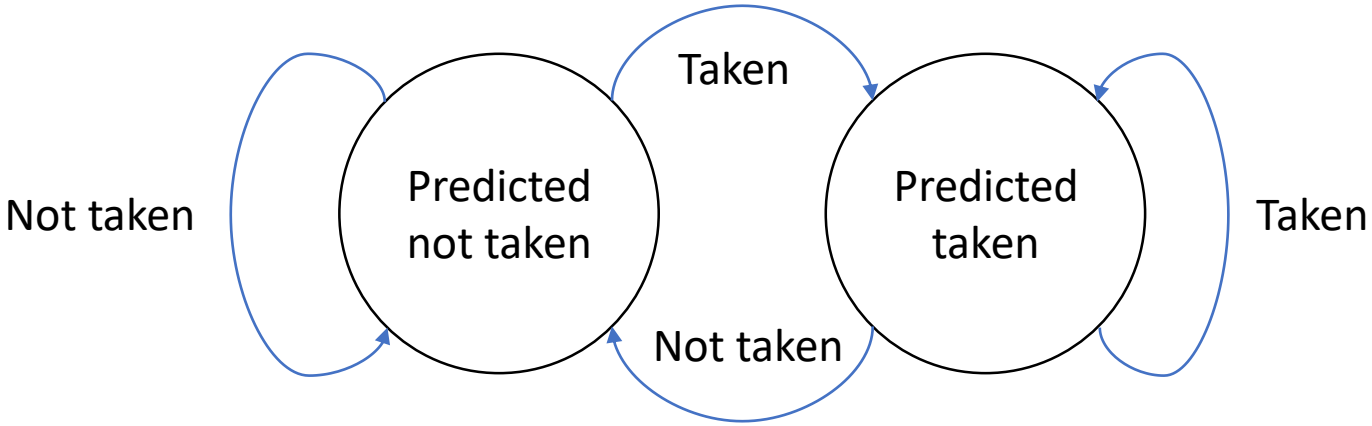
Modern CPUs combine multiple approaches:

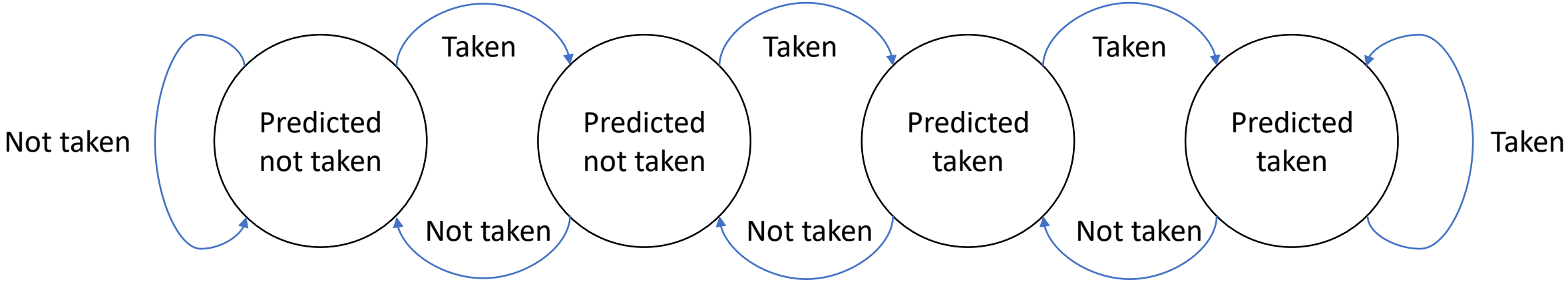
- Branch target predictors (where to jump)
- Neural network predictors (AMD, Apple)
- TAGE predictors with multiple history lengths

The better the predictor, the fewer pipeline stalls and higher CPU efficiency

Branch prediction continues to evolve with each CPU generation, with designers constantly seeking the optimal balance between accuracy, power consumption, and silicon area. Modern predictors can achieve over 95% accuracy on most workloads.

Dynamic Predictors





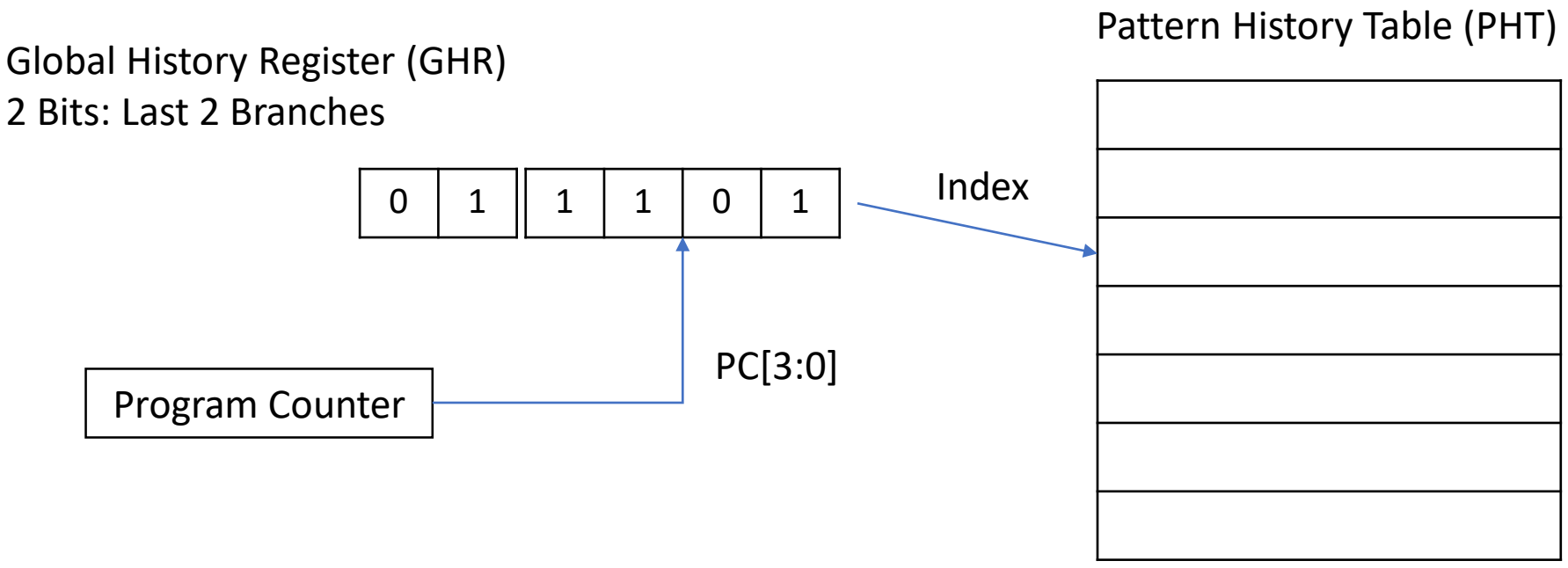
Branch Correlation

- Basic 2-bit predictors use only the recent behavior of a single branch, missing important correlations between different branches in the code.

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

```
        DADDIU R3,R1,#-2
        BNEZ   R3,L1          ;branch b1 (aa!=2)
        DADD   R1,R0,R0       ;aa=0
L1:     DADDIU R3,R2,#-2
        BNEZ   R3,L2          ;branch b2 (bb!=2)
        DADD   R2,R0,R0       ;bb=0
L2:     DSUBU  R3,R1,R2       ;R3=aa-bb
        BEQZ   R3,L3          ;branch b3 (aa==bb)
```

Two-Level Predictors



Example

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

```
        DADDIU R3,R1,#-2
0x04:   BNEZ   R3,L1          ;branch b1 (aa!=2)
        DADD   R1,R0,R0       ;aa=0
L1:     DADDIU R3,R2,#-2
0x10:   BNEZ   R3,L2          ;branch b2 (bb!=2)
        DADD   R2,R0,R0       ;bb=0
L2:     DSUBU  R3,R1,R2       ;R3=aa-bb
0x1C:   BEQZ   R3,L3          ;branch b3 (aa==bb)
```

GHR

0	0	1	1	0	0
0	1	1	1	0	0
1	0	1	1	0	0
1	1	1	1	0	0

→ Always Taken

Intel Core i7 Branch Predictor

Two-Level Design

Smaller first-level predictor for single-cycle prediction, larger second-level predictor as backup

Combined Approach

Each level combines three predictors:

- Simple two-bit predictor
- Global history predictor
- Loop exit predictor (counts iterations)

Additional Units

Separate predictors for indirect branches and return addresses

Example

```
        DADDIU R3,R1,#-2
0x04:   BNEZ   R3,L1          ;branch b1 (aa!=2)
        DADD   R1,R0,R0       ;aa=0
L1:     DADDIU R3,R2,#-2
0x10:   BNEZ   R3,L2          ;branch b2 (bb!=2)
        DADD   R2,R0,R0       ;bb=0
L2:     DSUBU  R3,R1,R2       ;R3=aa-bb
0x1C:   BEQZ   R3,L3          ;branch b3 (aa==bb)
```

GHR PC[3:2]

0	0	0	1

0000	0
0001	0
0010	0
0011	0
0100	0
0101	0
0110	0
0111	0
1000	0
1001	0
1010	0
1011	0
1100	0
1101	0
1110	0
1111	0

Predicted NT

Example

```
        DADDIU R3,R1,#-2
0x04:   BNEZ   R3,L1          ;branch b1 (aa!=2)
        DADD   R1,R0,R0       ;aa=0
L1:     DADDIU R3,R2,#-2
0x10:   BNEZ   R3,L2          ;branch b2 (bb!=2)
        DADD   R2,R0,R0       ;bb=0
L2:     DSUBU  R3,R1,R2       ;R3=aa-bb
0x1C:   BEQZ   R3,L3          ;branch b3 (aa==bb)
```

GHR		PC[3:2]	
0	1	0	1

0000	0
0001	1
0010	0
0011	0
0100	0
0101	0
0110	0
0111	0
1000	0
1001	0
1010	0
1011	0
1100	0
1101	0
1110	0
1111	0

Actually T

Example

```
        DADDIU R3,R1,#-2
0x04:   BNEZ   R3,L1          ;branch b1 (aa!=2)
        DADD   R1,R0,R0       ;aa=0
L1:     DADDIU R3,R2,#-2
0x10:   BNEZ   R3,L2          ;branch b2 (bb!=2)
        DADD   R2,R0,R0       ;bb=0
L2:     DSUBU  R3,R1,R2       ;R3=aa-bb
0x1C:   BEQZ   R3,L3          ;branch b3 (aa==bb)
```

GHR PC[3:2]

0	1	0	0

0000	0
0001	1
0010	0
0011	0
0100	0
0101	0
0110	0
0111	0
1000	0
1001	0
1010	0
1011	0
1100	0
1101	0
1110	0
1111	0

Predicted NT

Example

```
        DADDIU R3,R1,#-2
0x04:   BNEZ   R3,L1          ;branch b1 (aa!=2)
        DADD   R1,R0,R0       ;aa=0
L1:     DADDIU R3,R2,#-2
0x10:   BNEZ   R3,L2          ;branch b2 (bb!=2)
        DADD   R2,R0,R0       ;bb=0
L2:     DSUBU  R3,R1,R2       ;R3=aa-bb
0x1C:   BEQZ   R3,L3          ;branch b3 (aa==bb)
```

GHR		PC[3:2]	
1	1	0	0

0000	0
0001	1
0010	0
0011	0
0100	1
0101	0
0110	0
0111	0
1000	0
1001	0
1010	0
1011	0
1100	0
1101	0
1110	0
1111	0

Actually T

Example

```
        DADDIU R3,R1,#-2
0x04:   BNEZ   R3,L1          ;branch b1 (aa!=2)
        DADD   R1,R0,R0       ;aa=0
L1:     DADDIU R3,R2,#-2
0x10:   BNEZ   R3,L2          ;branch b2 (bb!=2)
        DADD   R2,R0,R0       ;bb=0
L2:     DSUBU  R3,R1,R2       ;R3=aa-bb
0x1C:   BEQZ   R3,L3          ;branch b3 (aa==bb)
```

GHR PC[3:2]

1	1	1	1

0000	0
0001	1
0010	0
0011	0
0100	1
0101	0
0110	0
0111	0
1000	0
1001	0
1010	0
1011	0
1100	0
1101	0
1110	0
1111	0

Predicted NT

Example

```
        DADDIU R3,R1,#-2
0x04:   BNEZ   R3,L1          ;branch b1 (aa!=2)
        DADD   R1,R0,R0       ;aa=0
L1:     DADDIU R3,R2,#-2
0x10:   BNEZ   R3,L2          ;branch b2 (bb!=2)
        DADD   R2,R0,R0       ;bb=0
L2:     DSUBU  R3,R1,R2       ;R3=aa-bb
0x1C:   BEQZ   R3,L3          ;branch b3 (aa==bb)
```

GHR PC[3:2]

1	0	1	1

0000	0
0001	1
0010	0
0011	0
0100	1
0101	0
0110	0
0111	0
1000	0
1001	0
1010	0
1011	0
1100	0
1101	0
1110	0
1111	0

Actually NT

Example

```
        DADDIU R3,R1,#-2
0x04:   BNEZ   R3,L1          ;branch b1 (aa!=2)
        DADD   R1,R0,R0       ;aa=0
L1:     DADDIU R3,R2,#-2
0x10:   BNEZ   R3,L2          ;branch b2 (bb!=2)
        DADD   R2,R0,R0       ;bb=0
L2:     DSUBU  R3,R1,R2       ;R3=aa-bb
0x1C:   BEQZ   R3,L3          ;branch b3 (aa==bb)
```

GHR PC[3:2]

1	0	0	1

0000	0
0001	1
0010	0
0011	0
0100	1
0101	0
0110	0
0111	0
1000	0
1001	0
1010	0
1011	0
1100	0
1101	0
1110	0
1111	0

Predicted NT

Example

```
        DADDIU R3,R1,#-2
0x04:   BNEZ   R3,L1          ;branch b1 (aa!=2)
        DADD   R1,R0,R0       ;aa=0
L1:     DADDIU R3,R2,#-2
0x10:   BNEZ   R3,L2          ;branch b2 (bb!=2)
        DADD   R2,R0,R0       ;bb=0
L2:     DSUBU  R3,R1,R2       ;R3=aa-bb
0x1C:   BEQZ   R3,L3          ;branch b3 (aa==bb)
```

GHR PC[3:2]

0	0	0	1

0000	0
0001	1
0010	0
0011	0
0100	1
0101	0
0110	0
0111	0
1000	0
1001	0
1010	0
1011	0
1100	0
1101	0
1110	0
1111	0

Actually NT

Example

```
        DADDIU R3,R1,#-2
0x04:   BNEZ   R3,L1          ;branch b1 (aa!=2)
        DADD   R1,R0,R0       ;aa=0
L1:     DADDIU R3,R2,#-2
0x10:   BNEZ   R3,L2          ;branch b2 (bb!=2)
        DADD   R2,R0,R0       ;bb=0
L2:     DSUBU  R3,R1,R2       ;R3=aa-bb
0x1C:   BEQZ   R3,L3          ;branch b3 (aa==bb)
```

GHR PC[3:2]

0	0	0	0

0000	0
0001	1
0010	0
0011	0
0100	1
0101	0
0110	0
0111	0
1000	0
1001	0
1010	0
1011	0
1100	0
1101	0
1110	0
1111	0

Predicted NT

Example

```
        DADDIU R3,R1,#-2
0x04:   BNEZ   R3,L1          ;branch b1 (aa!=2)
        DADD   R1,R0,R0       ;aa=0
L1:     DADDIU R3,R2,#-2
0x10:   BNEZ   R3,L2          ;branch b2 (bb!=2)
        DADD   R2,R0,R0       ;bb=0
L2:     DSUBU  R3,R1,R2       ;R3=aa-bb
0x1C:   BEQZ   R3,L3          ;branch b3 (aa==bb)
```

GHR PC[3:2]

0	0	0	0

0000	0
0001	1
0010	0
0011	0
0100	1
0101	0
0110	0
0111	0
1000	0
1001	0
1010	0
1011	0
1100	0
1101	0
1110	0
1111	0

Actually NT

Example

```
        DADDIU R3,R1,#-2
0x04:   BNEZ   R3,L1          ;branch b1 (aa!=2)
        DADD   R1,R0,R0       ;aa=0
L1:     DADDIU R3,R2,#-2
0x10:   BNEZ   R3,L2          ;branch b2 (bb!=2)
        DADD   R2,R0,R0       ;bb=0
L2:     DSUBU  R3,R1,R2       ;R3=aa-bb
0x1C:   BEQZ   R3,L3          ;branch b3 (aa==bb)
```

GHR PC[3:2]

0	0	1	1

0000	0
0001	1
0010	0
0011	0
0100	1
0101	0
0110	0
0111	0
1000	0
1001	0
1010	0
1011	0
1100	0
1101	0
1110	0
1111	0

Predicted NT

Example

```
        DADDIU R3,R1,#-2
0x04:   BNEZ   R3,L1          ;branch b1 (aa!=2)
        DADD   R1,R0,R0       ;aa=0
L1:     DADDIU R3,R2,#-2
0x10:   BNEZ   R3,L2          ;branch b2 (bb!=2)
        DADD   R2,R0,R0       ;bb=0
L2:     DSUBU  R3,R1,R2       ;R3=aa-bb
0x1C:   BEQZ   R3,L3          ;branch b3 (aa==bb)
```

GHR		PC[3:2]	
0	1	1	1

0000	0
0001	1
0010	0
0011	1
0100	1
0101	0
0110	0
0111	0
1000	0
1001	0
1010	0
1011	0
1100	0
1101	0
1110	0
1111	0

Actually T

Limitations of Static Scheduling

The Problem

Simple pipelining uses in-order instruction issue and execution. If an instruction stalls due to a dependency, all later instructions must also stall - even if they have no dependencies.

Example

```
DIV.D F0,F2,F4
ADD.D F10,F0,F8
SUB.D F12,F8,F14
```

The SUB.D must wait for ADD.D, which waits for DIV.D, despite SUB.D having no dependency on DIV.D.

This creates a performance limitation that can be eliminated by allowing instructions to execute out of program order.

Dynamic Scheduling: The Concept

Key Innovations

- Split instruction decode into two parts:
  - checking for structural hazards and
  - waiting for data hazards to clear
- Instructions still issue in-order
- Instructions begin execution as soon as operands are available
- Results in out-of-order execution and completion

Benefits

- Code compiled for one pipeline runs efficiently on different pipelines
- Handles dependencies unknown at compile time
- Tolerates unpredictable delays like cache misses

These advantages come at the cost of significant hardware complexity.

Out-of-Order Execution

- Split the ID (decode) stage into:
  - Issue: Decode instructions, check for structural hazards.
  - Read operands: Wait until no data hazards, then read operands.
- Dynamically scheduled pipeline
  - All instructions pass through the issue stage in order
  - They can be stalled or bypass each other in the second stage (read operands) and thus enter execution out of order

Challenges of Out-of-Order Execution

1

New Hazard Types

Out-of-order execution introduces Write-After-Read (WAR) and Write-After-Write (WAW) hazards that don't exist in simple in-order pipelines.

2

Exception Handling

Must preserve exception behavior as if instructions executed in strict program order, despite out-of-order completion.

3

Register Conflicts

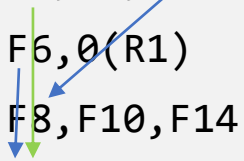
Limited register sets (like IBM 360's four FP registers) create artificial dependencies that limit performance.

These challenges are addressed through register renaming and careful exception management.

Register Renaming: Eliminating False Dependencies

Original Code with Dependencies

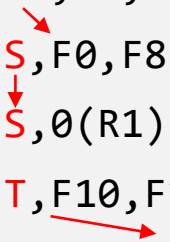
```
DIV.D F0,F2,F4
ADD.D F6,F0,F8
S.D   F6,0(R1)
SUB.D F8,F10,F14
MUL.D F6,F10,F8
```



Contains WAR hazards between ADD.D/SUB.D and S.D/MUL.D, plus WAW hazard between ADD.D/MUL.D.

After Register Renaming

```
DIV.D F0,F2,F4
ADD.D S,F0,F8
S.D   S,0(R1)
SUB.D T,F10,F14
MUL.D F6,F10,T
```

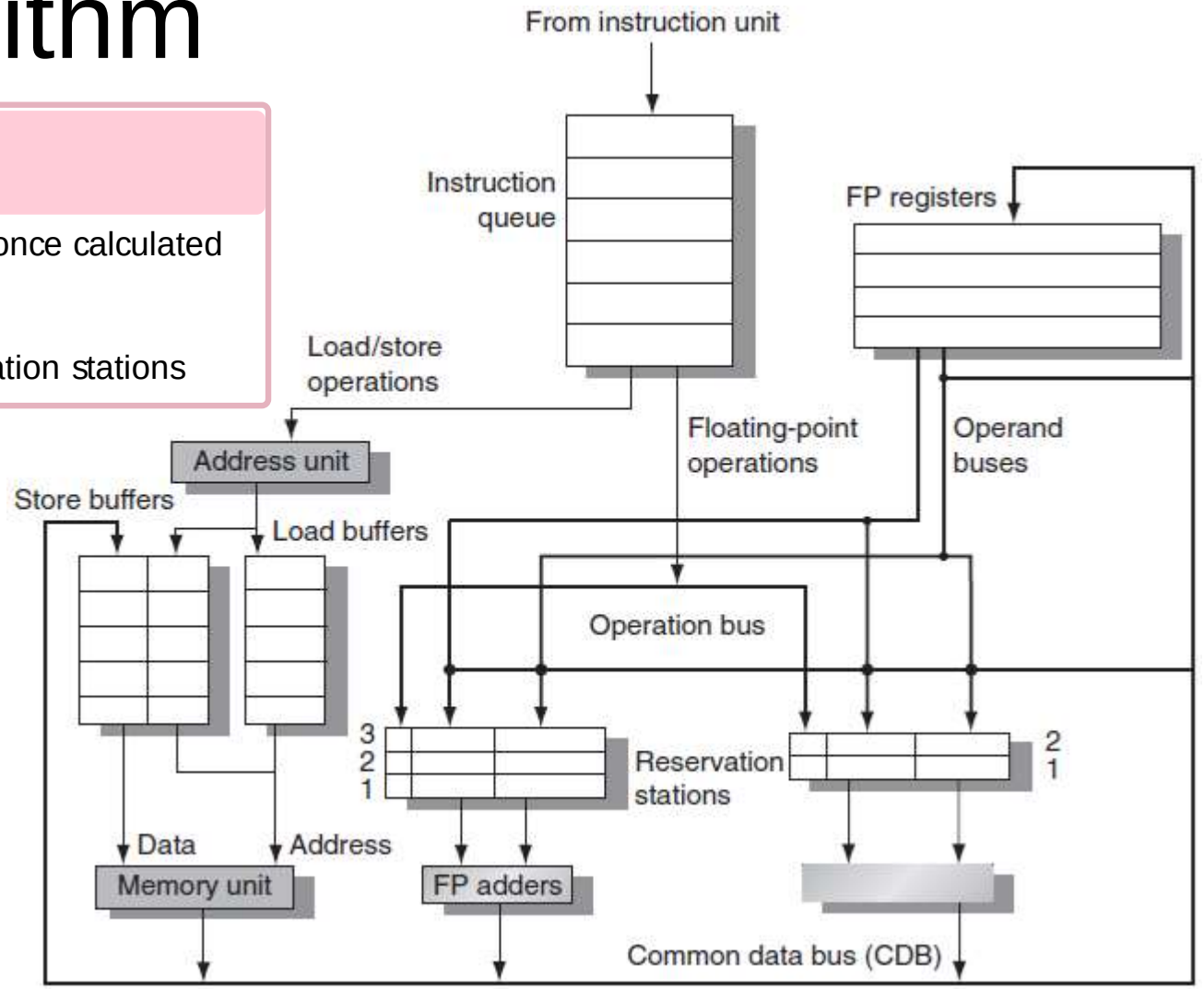


Using temporary registers S and T eliminates all name dependencies, allowing parallel execution.

Floating-Point Unit using Tomasulo's Algorithm

Load/Store Buffers

- **A**: Effective address once calculated
- Functions like reservation stations



Register File

- **Qi**: Reservation station computing result for this register

Reservation Stations

- **Op**: Operation to perform
- **Qj, Qk**: Stations producing source operands
- **Vj, Vk**: Source operand values
- **A**: Address information for loads/stores
- **Busy**: Station occupancy status

Instruction Lifecycle in Tomasulo's Algorithm

Issue

- Get next instruction from queue (FIFO order)
- If empty reservation station exists, issue instruction
- Record which functional units will produce needed operands
- Rename registers to eliminate WAR/WAW hazards

Execute

- Monitor common data bus for needed operands
- When all operands available, execute at functional unit
- Loads/stores require two-step execution process
- Delay until preceding branches complete (for exception handling)

Write Result

- Broadcast result on common data bus
- Update registers and any waiting reservation stations
- Store results wait in buffer until address and value available

Example

Instruction Status

Instruction		Iss.	Exe.	Wr.
L.D	F6,32(R2)			
L.D	F2,44(R3)			
MUL.D	F0,F2,F4			
SUB.D	F8,F2,F6			
DIV.D	F10,F0,F6			
ADD.D	F6,F8,F2			

Reservation Station

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1							
Load2							
Add1							
Add2							
Add3							
Mult1							
Mult2							

Register Status

Field	F0	F2	F4	F6	F8	F10	F12
Qi							

Example

Instruction Status

Instruction		Iss.	Exe.	Wr.
L.D	F6,32(R2)	√		
L.D	F2,44(R3)			
MUL.D	F0,F2,F4			
SUB.D	F8,F2,F6			
DIV.D	F10,F0,F6			
ADD.D	F6,F8,F2			

Reservation Station

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	Y	Load					32+R[2]
Load2							
Add1							
Add2							
Add3							
Mult1							
Mult2							

Register Status

Field	F0	F2	F4	F6	F8	F10	F12
Qi				Load1			

Example

Instruction Status

Instruction		Iss.	Exe.	Wr.
L.D	F6,32(R2)	√	√	
L.D	F2,44(R3)	√		
MUL.D	F0,F2,F4			
SUB.D	F8,F2,F6			
DIV.D	F10,F0,F6			
ADD.D	F6,F8,F2			

Reservation Station

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	Y	Load					32+R[2]
Load2	Y	Load					44+R[3]
Add1							
Add2							
Add3							
Mult1							
Mult2							

Register Status

Field	F0	F2	F4	F6	F8	F10	F12
Qi		Load2		Load1			

Example

Instruction Status

Instruction		Iss.	Exe.	Wr.
L.D	F6,32(R2)	√	√	
L.D	F2,44(R3)	√	√	
MUL.D	F0,F2,F4	√		
SUB.D	F8,F2,F6			
DIV.D	F10,F0,F6			
ADD.D	F6,F8,F2			

Reservation Station

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	Y	Load					32+R[2]
Load2	Y	Load					44+R[3]
Add1							
Add2							
Add3							
Mult1	Y	MUL		F[4]	Load2		
Mult2							

Register Status

Field	F0	F2	F4	F6	F8	F10	F12
Qi	Mult1	Load2		Load1			

Example

Instruction Status

Instruction		Iss.	Exe.	Wr.
L.D	F6,32(R2)	√	√	
L.D	F2,44(R3)	√	√	
MUL.D	F0,F2,F4	√		
SUB.D	F8,F2,F6	√		
DIV.D	F10,F0,F6			
ADD.D	F6,F8,F2			

Reservation Station

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	Y	Load					32+R[2]
Load2	Y	Load					44+R[3]
Add1	Y	SUB			Load2	Load1	
Add2							
Add3							
Mult1	Y	MUL		F[4]	Load2		
Mult2							

Register Status

Field	F0	F2	F4	F6	F8	F10	F12
Qi	Mult1	Load2		Load1	Add1		

Example

Instruction Status

Instruction		Iss.	Exe.	Wr.
L.D	F6,32(R2)	√	√	
L.D	F2,44(R3)	√	√	
MUL.D	F0,F2,F4	√		
SUB.D	F8,F2,F6	√		
DIV.D	F10,F0,F6	√		
ADD.D	F6,F8,F2			

Reservation Station

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	Y	Load					32+R[2]
Load2	Y	Load					44+R[3]
Add1	Y	SUB			Load2	Load1	
Add2							
Add3							
Mult1	Y	MUL		F[4]	Load2		
Mult2	Y	DIV			Multi1	Load1	

Register Status

Field	F0	F2	F4	F6	F8	F10	F12
Qi	Mult1	Load2		Load1	Add1	Mult2	

Example

Instruction Status

Instruction		Iss.	Exe.	Wr.
L.D	F6,32(R2)	√	√	
L.D	F2,44(R3)	√	√	
MUL.D	F0,F2,F4	√		
SUB.D	F8,F2,F6	√		
DIV.D	F10,F0,F6	√		
ADD.D	F6,F8,F2	√		

Reservation Station

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	Y	Load					32+R[2]
Load2	Y	Load					44+R[3]
Add1	Y	SUB			Load2	Load1	
Add2	Y	ADD			Add1	Load2	
Add3							
Mult1	Y	MUL		F[4]	Load2		
Mult2	Y	DIV			Multi1	Load1	

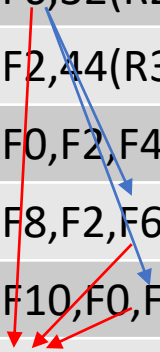
Register Status

Field	F0	F2	F4	F6	F8	F10	F12
Qi	Mult1	Load2		Add2	Add1	Mult2	

Example

Instruction Status

Instruction		Iss.	Exe.	Wr.
L.D	F6,32(R2)	√	√	
L.D	F2,44(R3)	√	√	
MUL.D	F0,F2,F4	√		
SUB.D	F8,F2,F6	√		
DIV.D	F10,F0,F6	√		
ADD.D	F6,F8,F2	√		



WAW and WAR hazards are broken

Real dependencies are preserved

Reservation Station

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	Y	Load					32+R[2]
Load2	Y	Load					44+R[3]
Add1	Y	SUB			Load2	Load1	
Add2	Y	ADD			Add1	Load2	
Add3							
Mult1	Y	MUL		F[4]	Load2		
Mult2	Y	DIV			Multi1	Load1	

Register Status

Field	F0	F2	F4	F6	F8	F10	F12
Qi	Mult1	Load2		Add2	Add1	Mult2	

Register renaming

Example

Instruction Status

Instruction		Iss.	Exe.	Wr.
L.D	F6,32(R2)	√	√	√
L.D	F2,44(R3)	√	√	
MUL.D	F0,F2,F4	√		
SUB.D	F8,F2,F6	√		
DIV.D	F10,F0,F6	√		
ADD.D	F6,F8,F2	√		

Reservation Station

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1							
Load2	Y	Load					44+R[3]
Add1	Y	SUB		M[32+R[2]]	Load2		
Add2	Y	ADD			Add1	Load2	
Add3							
Mult1	Y	MUL		F[4]	Load2		
Mult2	Y	DIV		M[32+R[2]]	Multi1		

Register Status

Field	F0	F2	F4	F6	F8	F10	F12
Qi	Mult1	Load2		Add2	Add1	Mult2	

Example

Instruction Status

Instruction		Iss.	Exe.	Wr.
L.D	F6,32(R2)	√	√	√
L.D	F2,44(R3)	√	√	√
MUL.D	F0,F2,F4	√	√	
SUB.D	F8,F2,F6	√	√	
DIV.D	F10,F0,F6	√		
ADD.D	F6,F8,F2	√		

Reservation Station

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1							
Load2							
Add1	Y	SUB	M[44+R[3]]	M[32+R[2]]			
Add2	Y	ADD		M[44+R[3]]	Add1		
Add3							
Mult1	Y	MUL	M[44+R[3]]	F[4]			
Mult2	Y	DIV		M[32+R[2]]	Multi1		

Register Status

Field	F0	F2	F4	F6	F8	F10	F12
Qi	Mult1			Add2	Add1	Mult2	

Example

Instruction Status

Instruction		Iss.	Exe.	Wr.
L.D	F6,32(R2)	√	√	√
L.D	F2,44(R3)	√	√	√
MUL.D	F0,F2,F4	√	√	
SUB.D	F8,F2,F6	√	√	√
DIV.D	F10,F0,F6	√		
ADD.D	F6,F8,F2	√	√	

Out of order execution

Reservation Station

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1							
Load2							
Add1							
Add2	Y	ADD	F[8]	M[44+R[3]]			
Add3							
Mult1	Y	MUL	M[44+R[3]]	F[4]			
Mult2	Y	DIV		M[32+R[2]]	Multi1		

Register Status

Field	F0	F2	F4	F6	F8	F10	F12
Qi	Mult1			Add2		Mult2	

Example

Instruction Status

Instruction		Iss.	Exe.	Wr.
L.D	F6,32(R2)	√	√	√
L.D	F2,44(R3)	√	√	√
MUL.D	F0,F2,F4	√	√	√
SUB.D	F8,F2,F6	√	√	√
DIV.D	F10,F0,F6	√	√	
ADD.D	F6,F8,F2	√	√	~~√~~

If more than one instructions finish at the same time, only one of them can access the common data bus and complete

Reservation Station

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1							
Load2							
Add1							
Add2	Y	ADD	F[8]	M[44+R[3]]			
Add3							
Mult1							
Mult2	Y	DIV	F[0]	M[32+R[2]]			

Register Status

Field	F0	F2	F4	F6	F8	F10	F12
Qi				Add2		Mult2	

Example

Instruction Status

Instruction		Iss.	Exe.	Wr.
L.D	F6,32(R2)	√	√	√
L.D	F2,44(R3)	√	√	√
MUL.D	F0,F2,F4	√	√	√
SUB.D	F8,F2,F6	√	√	√
DIV.D	F10,F0,F6	√	√	
ADD.D	F6,F8,F2	√	√	√

Reservation Station

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1							
Load2							
Add1							
Add2							
Add3							
Mult1							
Mult2	Y	DIV	F[0]	M[32+R[2]]			

Register Status

Field	F0	F2	F4	F6	F8	F10	F12
Qi						Mult2	

Example

Instruction Status

Instruction		Iss.	Exe.	Wr.
L.D	F6,32(R2)	√	√	√
L.D	F2,44(R3)	√	√	√
MUL.D	F0,F2,F4	√	√	√
SUB.D	F8,F2,F6	√	√	√
DIV.D	F10,F0,F6	√	√	√
ADD.D	F6,F8,F2	√	√	√

Reservation Station

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1							
Load2							
Add1							
Add2							
Add3							
Mult1							
Mult2							

Register Status

Field	F0	F2	F4	F6	F8	F10	F12
Qi							

Hardware-Based Speculation

As we exploit more instruction-level parallelism, control dependencies become increasingly limiting. Branch prediction alone may not generate sufficient ILP for wide-issue processors that need to execute a branch every cycle.

Hardware speculation overcomes these limitations by executing instructions as if branch predictions were always correct, with mechanisms to handle incorrect speculation.

1 **Dynamic Branch Prediction**

Chooses which instructions to execute

2 **Speculative Execution**

Allows execution before control dependencies are resolved, with ability to undo incorrect speculation

3 **Dynamic Scheduling**

Handles scheduling of different basic block combinations

This approach essentially implements data flow execution: operations execute as soon as their operands are available.

Extending Tomasulo's Algorithm for Speculation

- To support speculation, we must separate result bypassing (needed for speculative execution) from instruction completion. This separation allows an instruction to execute and bypass results without performing irreversible updates until we know the instruction is no longer speculative.
- The key insight: allow instructions to execute out of order but force them to commit in order, preventing any irrevocable action (state updates or exceptions) until an instruction commits.
- This requires an additional hardware buffer—the reorder buffer (ROB)—to hold results between execution completion and instruction commit.

Reorder Buffer (ROB)

1

Purpose

Holds instruction results between execution completion and commit

Serves as a source of operands for instructions, similar to reservation stations

Integrates the function of the store buffer from Tomasulo's algorithm

2

Structure

Each ROB entry contains four fields:

- Instruction type (branch, store, or register operation)
- Destination field (register number or memory address)
- Value field (holds result until commit)
- Ready field (indicates completion status)

Unlike Tomasulo's algorithm where register files are updated immediately, with speculation, registers are only updated when instructions commit (when we know definitively they should execute).

Instruction Execution Steps with Speculation

Issue

Get instruction from queue if there's an empty reservation station and ROB slot

Send available operands from registers or ROB to reservation station

Update control entries and allocate ROB entry for the result

Execute

Monitor CDB for unavailable operands (checking for RAW hazards)

When both operands are available, execute the operation

For stores, calculate effective address at this stage

Write Result

Write result to CDB (with ROB tag) and from CDB into ROB

Update any waiting reservation stations

For stores, write value to be stored into ROB entry if available

Commit

Update register/memory with result when instruction reaches ROB head

For incorrect branch predictions, flush ROB and restart at correct path

Remove instruction from ROB after commit

Precise Exceptions with Speculation

Key Difference from Tomasulo's Algorithm

With speculation, no instruction after the earliest uncompleted instruction is allowed to complete. This enables precise exceptions.

If an instruction causes an exception, we wait until it reaches the ROB head and then take the interrupt, flushing any pending instructions.

Handling Exceptions

- Exceptions are recorded in the ROB but not recognized until commit
- If a speculated instruction raises an exception but shouldn't have executed (due to branch misprediction), the exception is flushed
- Only when an instruction reaches the ROB head do we know it's no longer speculative and the exception should be taken

This approach maintains a precise interrupt model while allowing dynamic execution—a significant advantage over basic Tomasulo's algorithm where instructions might complete out of order.

Memory Hazard Handling with Speculation

Store Instruction Handling

Unlike in Tomasulo's algorithm, stores update memory only when they reach the ROB head, ensuring memory isn't updated until instructions are no longer speculative.

WAW and WAR Hazards

Eliminated through memory because actual memory updates occur in order when a store reaches the ROB head.

RAW Hazards

Maintained by two restrictions:

1. Not allowing loads to execute if any active store has a matching destination
2. Maintaining program order for effective address computation

Some processors bypass values directly from store to load when RAW hazards occur.

Example

Reorder Buffer

#	Busy	Instruction		State	Dest.	Value
1		L.D	F6,32(R2)	Commit	F6	M[32+R[2]]
2		L.D	F2,44(R3)	Commit	F2	M[44+R[3]]
3	Y	MUL.D	F0,F2,F4	Execute	F0	
4	Y	SUB.D	F8,F2,F6	Write result	F8	#2 - #1
5	Y	DIV.D	F10,F0,F6	Execute	F10	
6	Y	ADD.D	F6,F8,F2	Write result	F6	#4 + #2

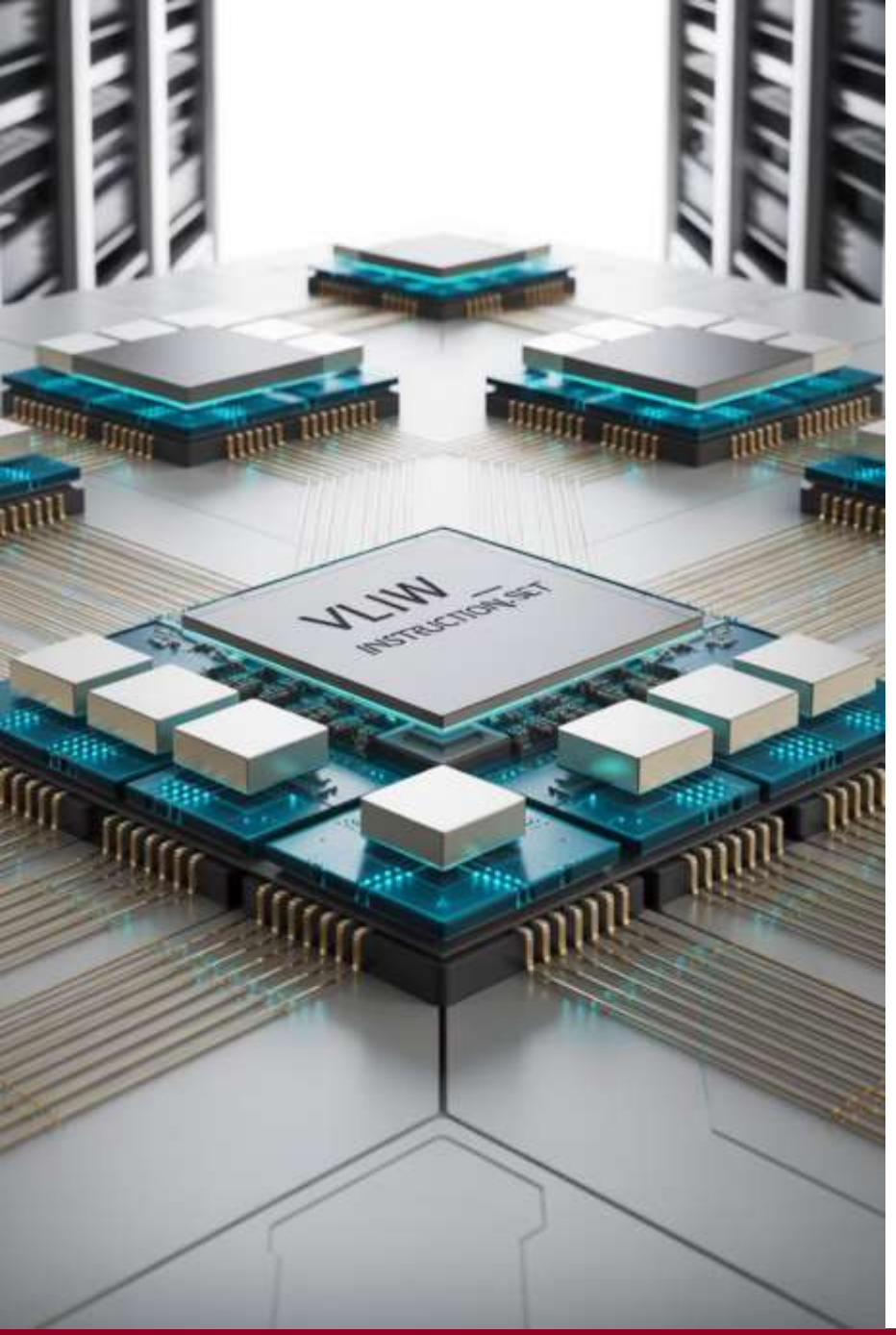
Reservation Station

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest.	A
Load1								
Load2								
Add1								
Add2								
Add3								
Mult1	Y	MUL	M[44+R[3]]	F[4]			#3	
Mult2	Y	DIV		M[32+R[2]]	#3		#5	

Multiple-Issue Approaches

Approach	Issue Structure	Hazard Detection	Scheduling	Examples
Superscalar (static)	Dynamic	Hardware	Static	MIPS, ARM Cortex-A8
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Intel Core i3/i5/i7, AMD Phenom, IBM Power 7
VLIW/LIW	Static	Primarily software	Static	TI C6x (signal processing)

Each approach represents different trade-offs between hardware complexity and compiler responsibility for extracting parallelism.



VLIW Architecture Fundamentals

Key Characteristics

- Multiple independent functional units
- Operations packaged into one very long instruction
- Typical width: 5+ operations per instruction
- 80-120 bits per instruction
- Itanium: 6 operations per instruction packet

Parallelism Extraction

- Loop unrolling to generate straight-line code
- Local scheduling within basic blocks
- Global scheduling across branches (more complex)
- Trace scheduling (specialized for VLIWs)

Example

```
Loop: L.D    F0,0(R1)    ;F0=array element
      ADD.D  F4,F0,F2    ;add scalar in F2
      S.D    F4,0(R1)    ;store result
      DADDUI R1,R1,#-8   ;decrement pointer
      BNE    R1,R2,Loop  ;branch R1!=R2
```

Unroll the loop

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)				BNE R1,R2,Loop

Two memory references
Two FP units
One integer operation

Per cycle

Multiple-Issue vs. Vector Processing

- When parallelism comes from simple FP loop unrolling, vector processors may be equally or more efficient than multiple-issue processors.
- Multiple-issue processors excel at:
  - Extracting parallelism from less structured code
  - Easily caching all forms of data
- For these reasons, multiple-issue approaches have become the primary method for exploiting instruction-level parallelism, with vector capabilities often added as extensions.

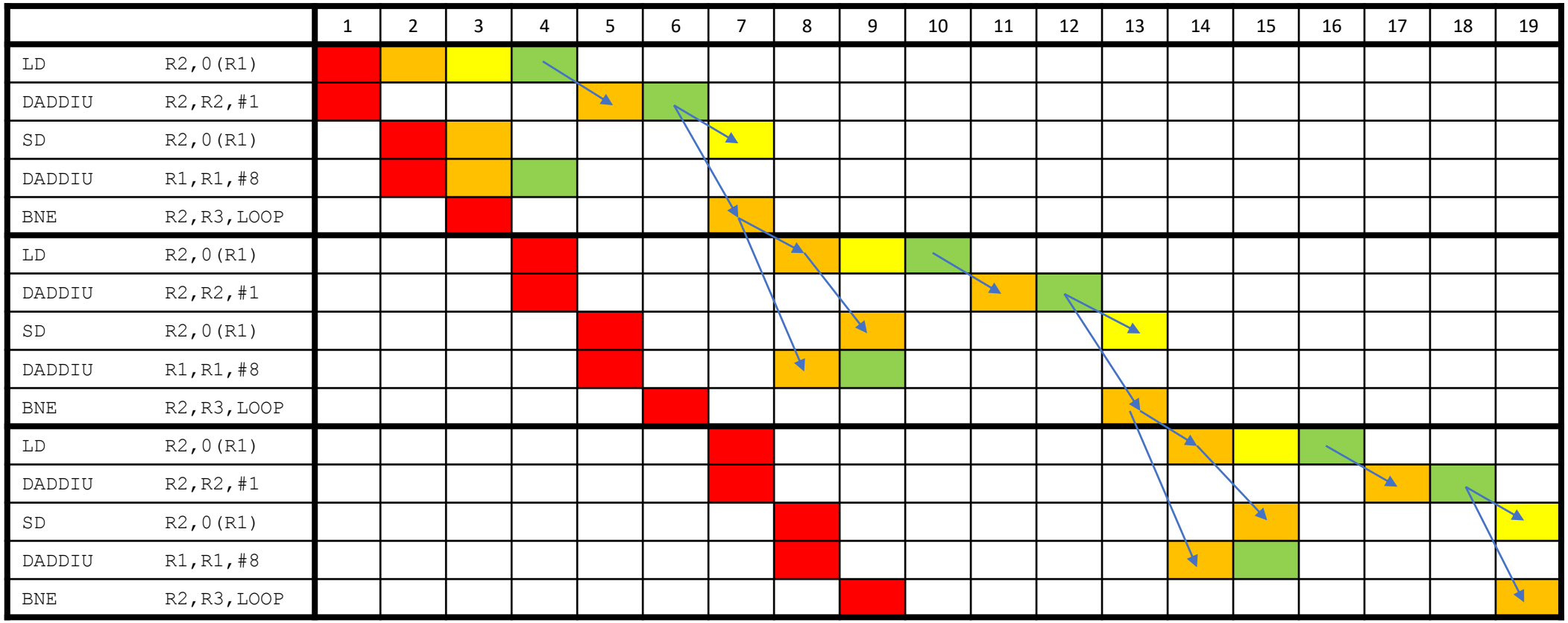
Putting It All Together

- When we combine dynamic scheduling, multiple issue, and speculation, we create a microarchitecture similar to those in modern processors.
- This integration allows for significant performance improvements by executing multiple instructions simultaneously while handling dependencies and branch predictions.

Example

```
Loop:   LD      R2,0(R1)        ;R2=array element
        DADDIU  R2,R2,#1        ;increment R2
        SD      R2,0(R1)        ;store result
        DADDIU  R1,R1,#8        ;increment pointer
        BNE     R2,R3,LOOP      ;branch if not last element
```

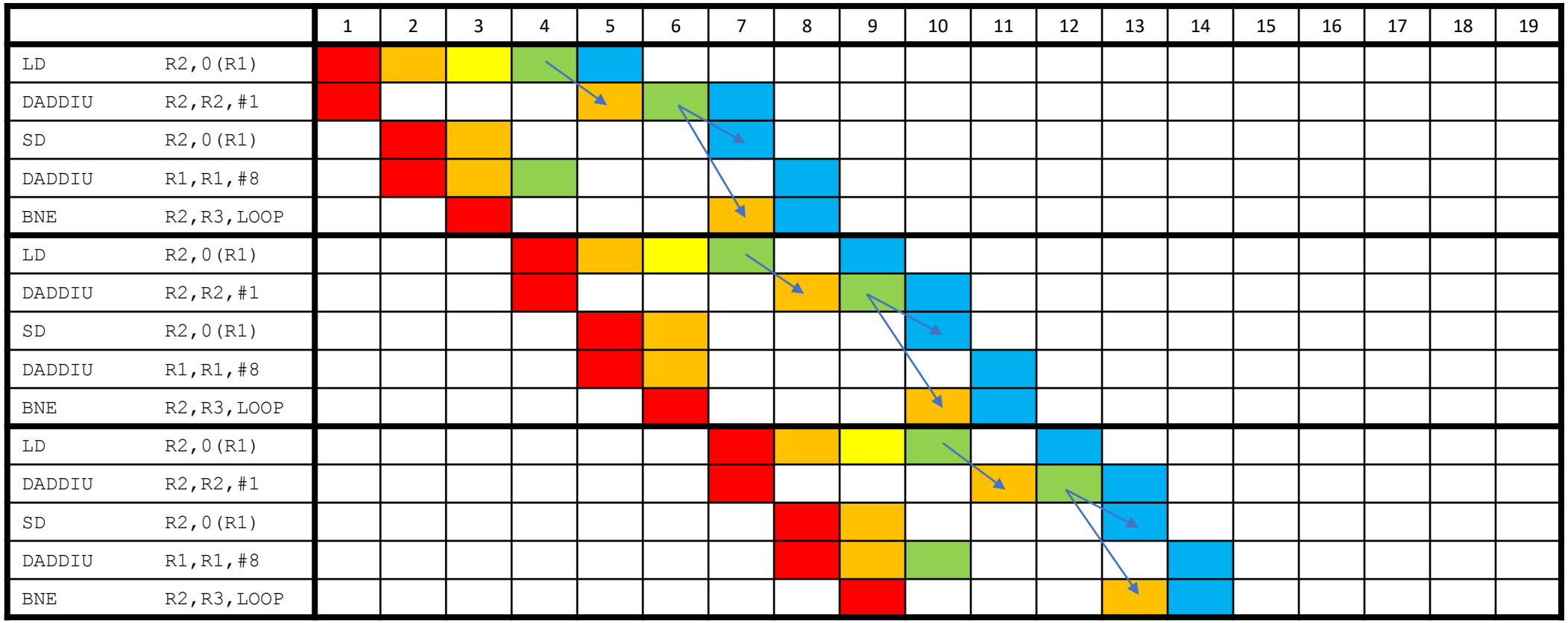
One address calculation
One ALU operation
One branch evaluation
Two instruction committed
Per cycle



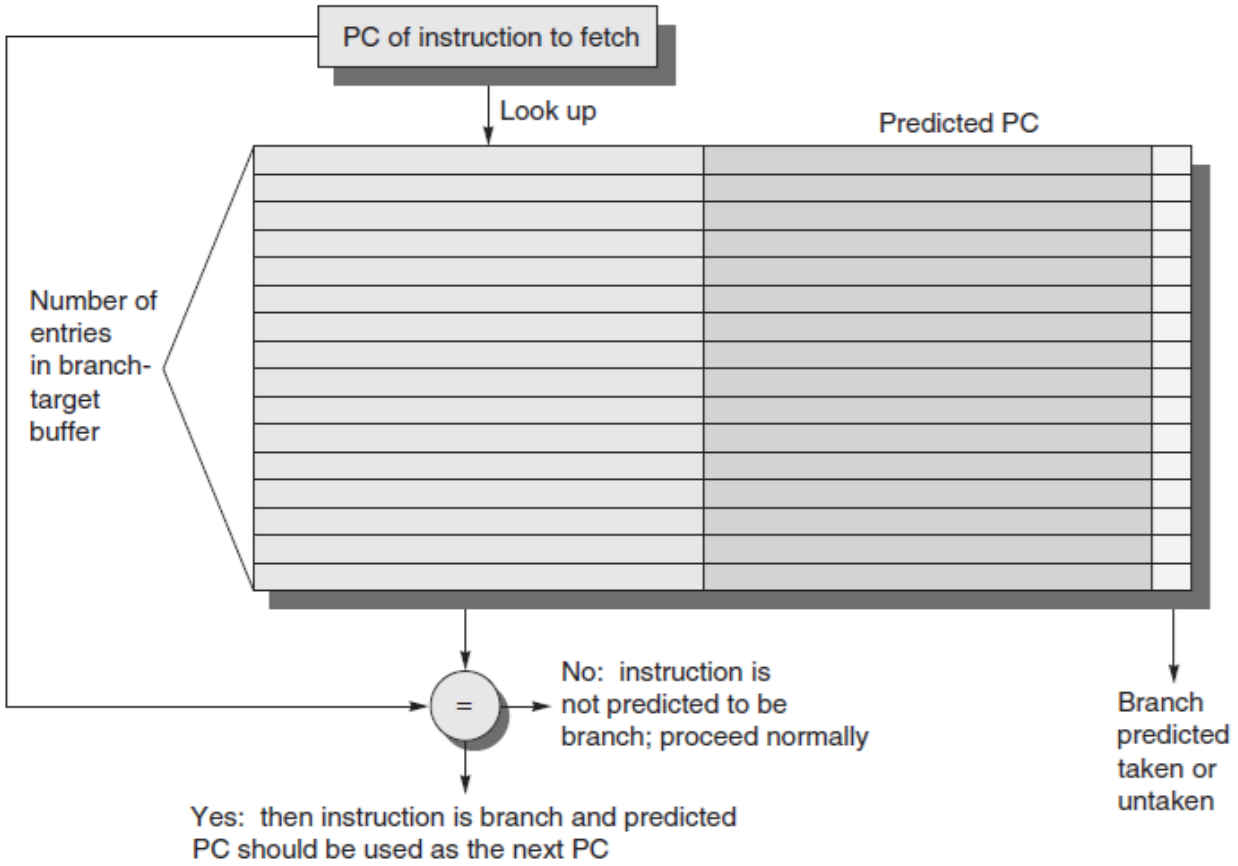
Example with Speculation

```
Loop:   LD      R2,0(R1)        ;R2=array element
        DADDIU  R2,R2,#1        ;increment R2
        SD      R2,0(R1)        ;store result
        DADDIU  R1,R1,#8        ;increment pointer
        BNE     R2,R3,LOOP      ;branch if not last element
```

One address calculation
One ALU operation
One branch evaluation
Two instruction committed
Per cycle

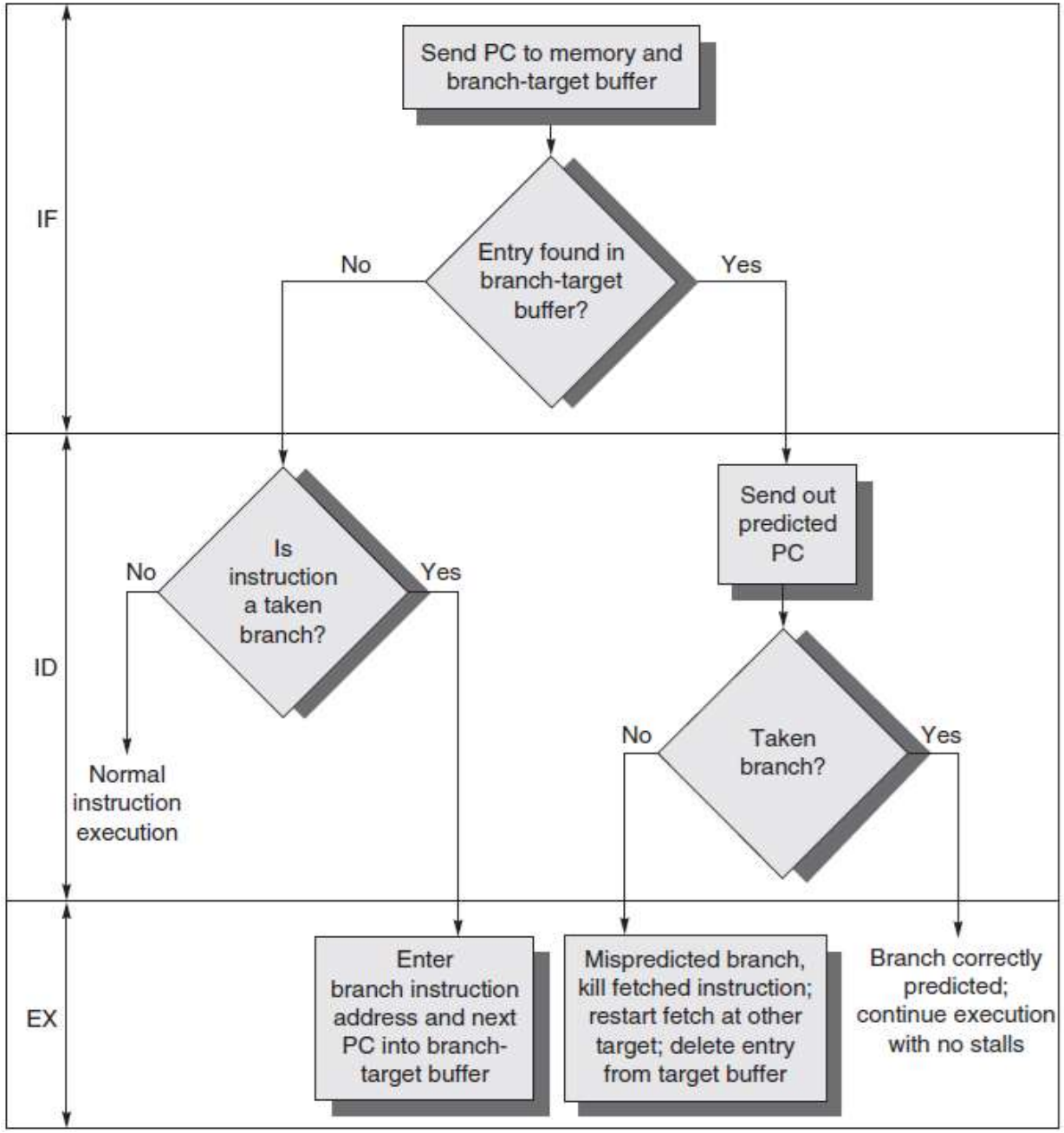


Branch Target Buffer



- If the instruction is a branch and we know what the next PC should be, we can have a branch penalty of zero.
- A branch-prediction cache that stores the predicted address for the next instruction after a branch is called a branch-target buffer or branch-target cache.

Steps using the Branch Target Buffer



Branch Prediction & Branch Target Buffer

Instruction in buffer	Prediction	Actual branch	Penalty cycles
Yes	Taken	Taken	0
Yes	Taken	Not taken	2
No		Taken	2
No		Not taken	0

- There is no branch penalty if everything is correctly predicted and the branch is found in the target buffer.
- If the branch is not correctly predicted, the penalty is equal to one clock cycle to update the buffer with the correct information (during which an instruction cannot be fetched) and one clock cycle, if needed, to restart fetching the next correct instruction for the branch.
- If the branch is not found and taken, a two-cycle penalty is encountered, during which time the buffer is updated.

Example

Instruction in buffer	Prediction	Actual branch	Penalty cycles
Yes	Taken	Taken	0
Yes	Taken	Not taken	2
No		Taken	2
No		Not taken	0

- Determine the total branch penalty for a branch-target buffer.
  - Prediction accuracy is 90% (for instructions in the buffer).
  - Hit rate in the buffer is 90% (for branches predicted taken).
- Answer
  - Probability (branch in buffer, but actually not taken)
    = Percent buffer hit rate × Percent incorrect predictions
    = 90% × 10% = 0.09
  - Probability (branch not in buffer, but actually taken) = 10%
  - Branch penalty = (0.09 + 0.10) × 2 = 0.38

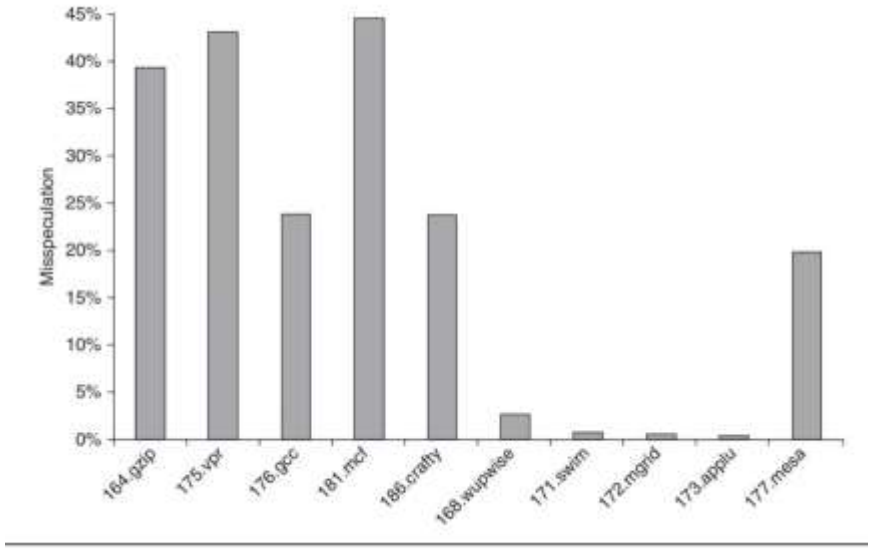
Speculation Costs and Benefits

Advantages

- Uncovers events that would stall the pipeline early (e.g., cache misses)
- Increases available instruction-level parallelism
- Allows execution to proceed past control dependencies

Costs

- Takes time and energy to execute speculative instructions
- Recovery from incorrect speculation reduces performance
- Requires additional processor resources (silicon area and power)
- Exceptional events during speculation can cause significant performance loss



The graph shows the fraction of instructions executed due to misspeculation - typically much higher for integer programs (~30%) than for FP programs, making speculation less energy efficient for integer applications.