# Superscalar Architecture

## Branch Speculation and Predictors

Computer System Architecture

IIT Tirupati

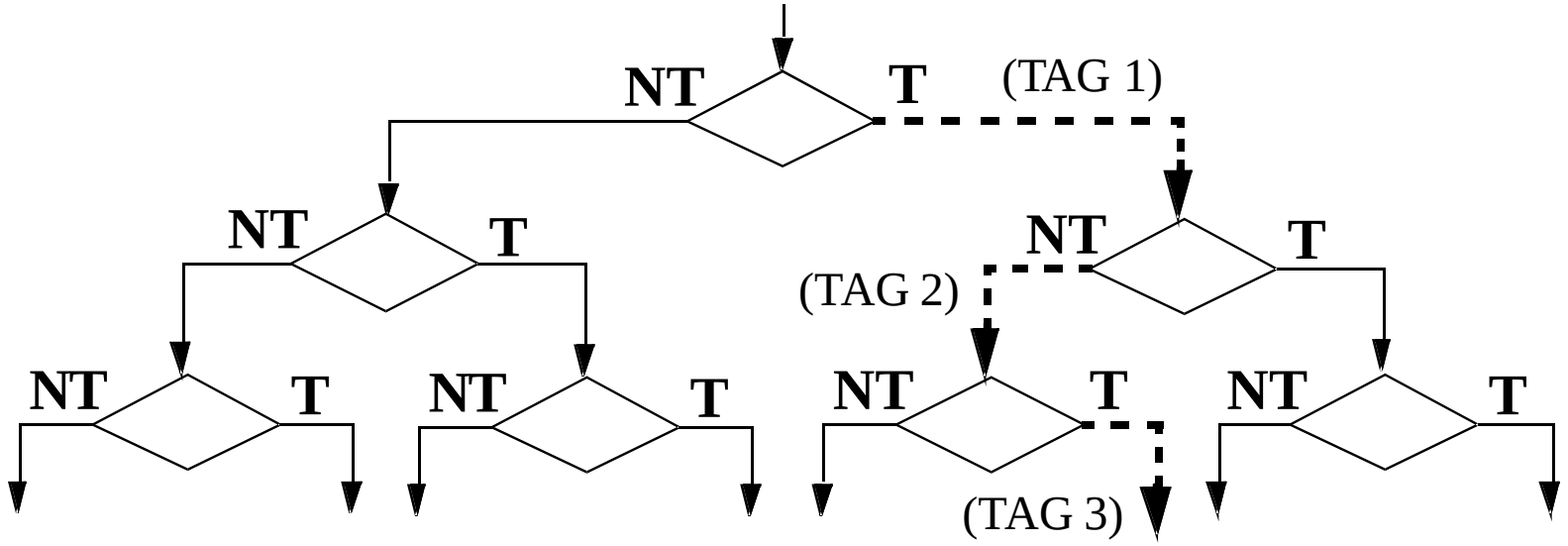
jtt@iittp.ac.in

2nd April, 2020

# Branch Speculation

- Condition resolution | Condition speculation
  - Access register:
    - Condition code register, General purpose register
  - Perform calculation:
    - Comparison of data register(s)
- Target address generation | Target Speculation
  - Access register:
    - PC, General purpose register, Link register
  - Perform calculation:
    - +/- offset, autoincrement, autodecrement

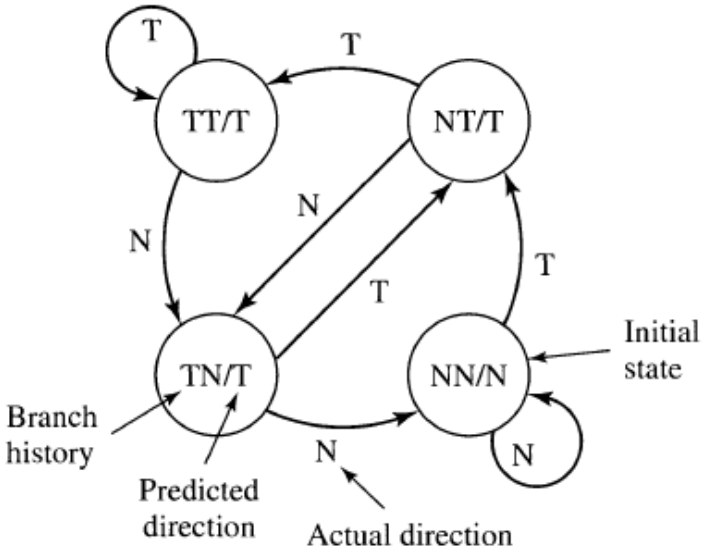
# Branch Speculation



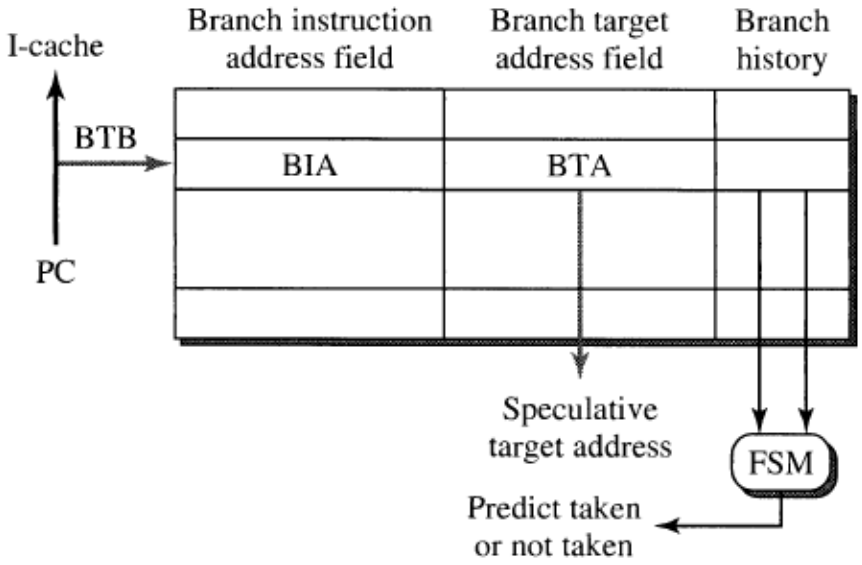

- Leading Speculation
  - Typically done during the Fetch stage
  - Based on potential branch instruction(s) in the current fetch group
- Trailing Confirmation
  - Typically done during the Branch Execute stage
  - Based on the next Branch instruction to finish execution

# Branch Instruction Speculation

– Branch target prediction
– Branch direction prediction

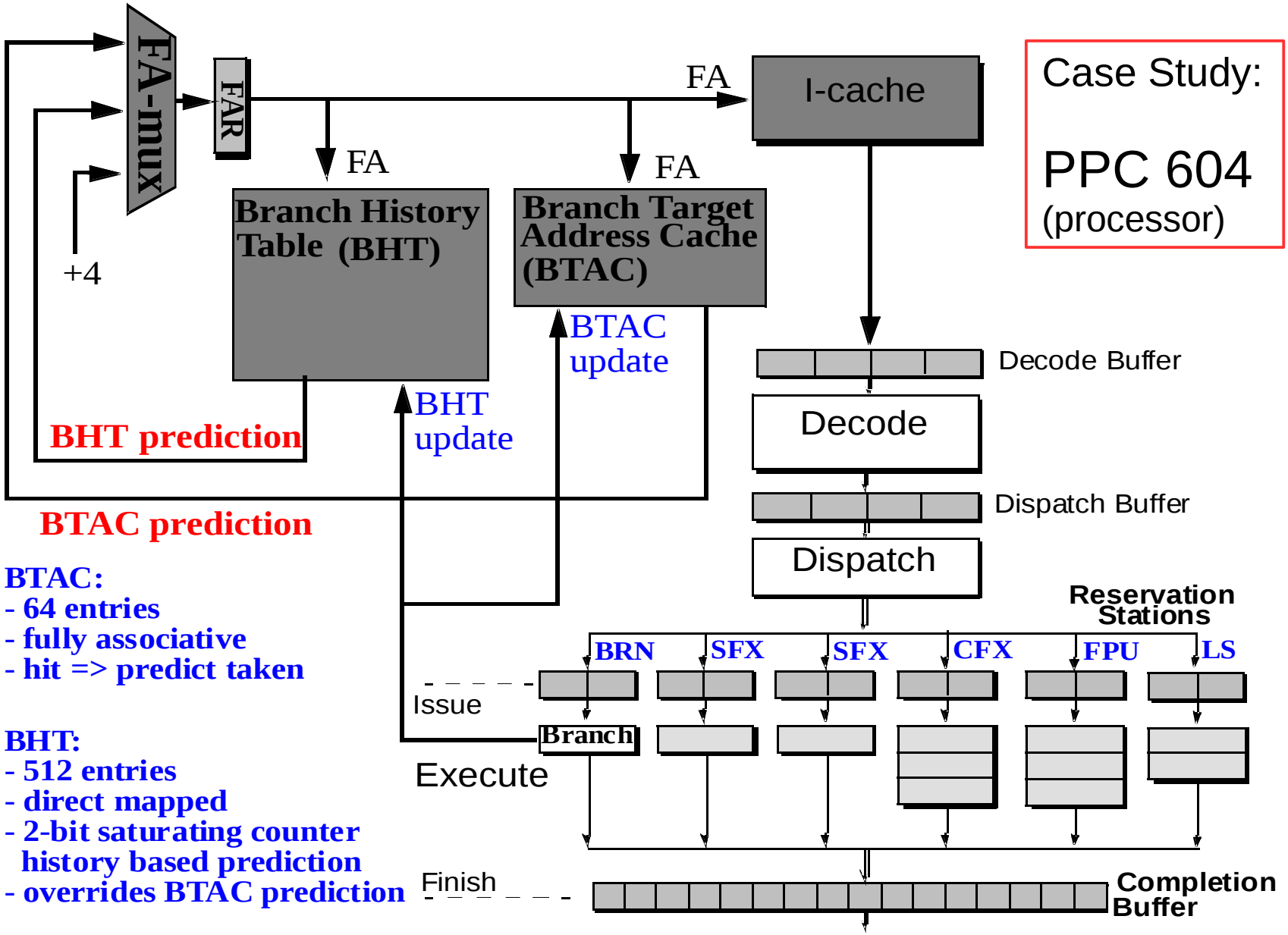



State-transition diagram




Implementation using BTB

# BTAC and BHT Design




Case Study:

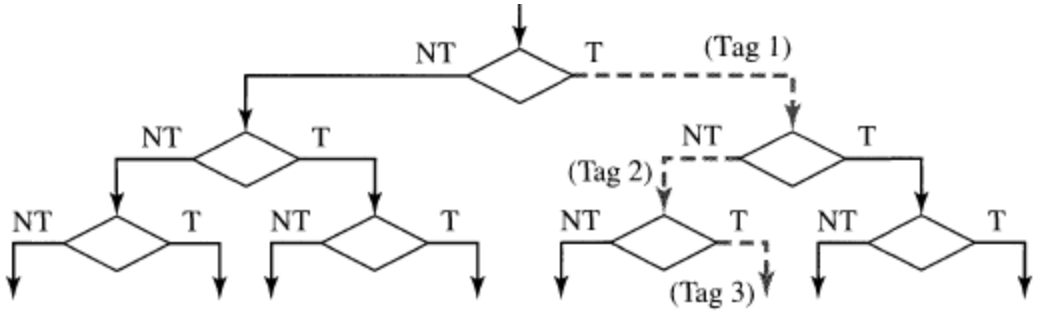
PPC 604
(processor)

**BTAC:**
- 64 entries
- fully associative
- hit => predict taken

**BHT:**
- 512 entries
- direct mapped
- 2-bit saturating counter history based prediction
- overrides BTAC prediction

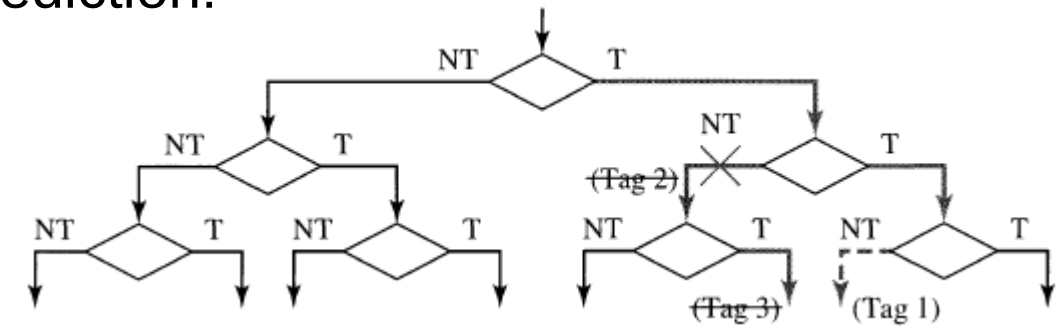
# Recovery from Mispeculation

The predicted path:




The important thing is the TAG

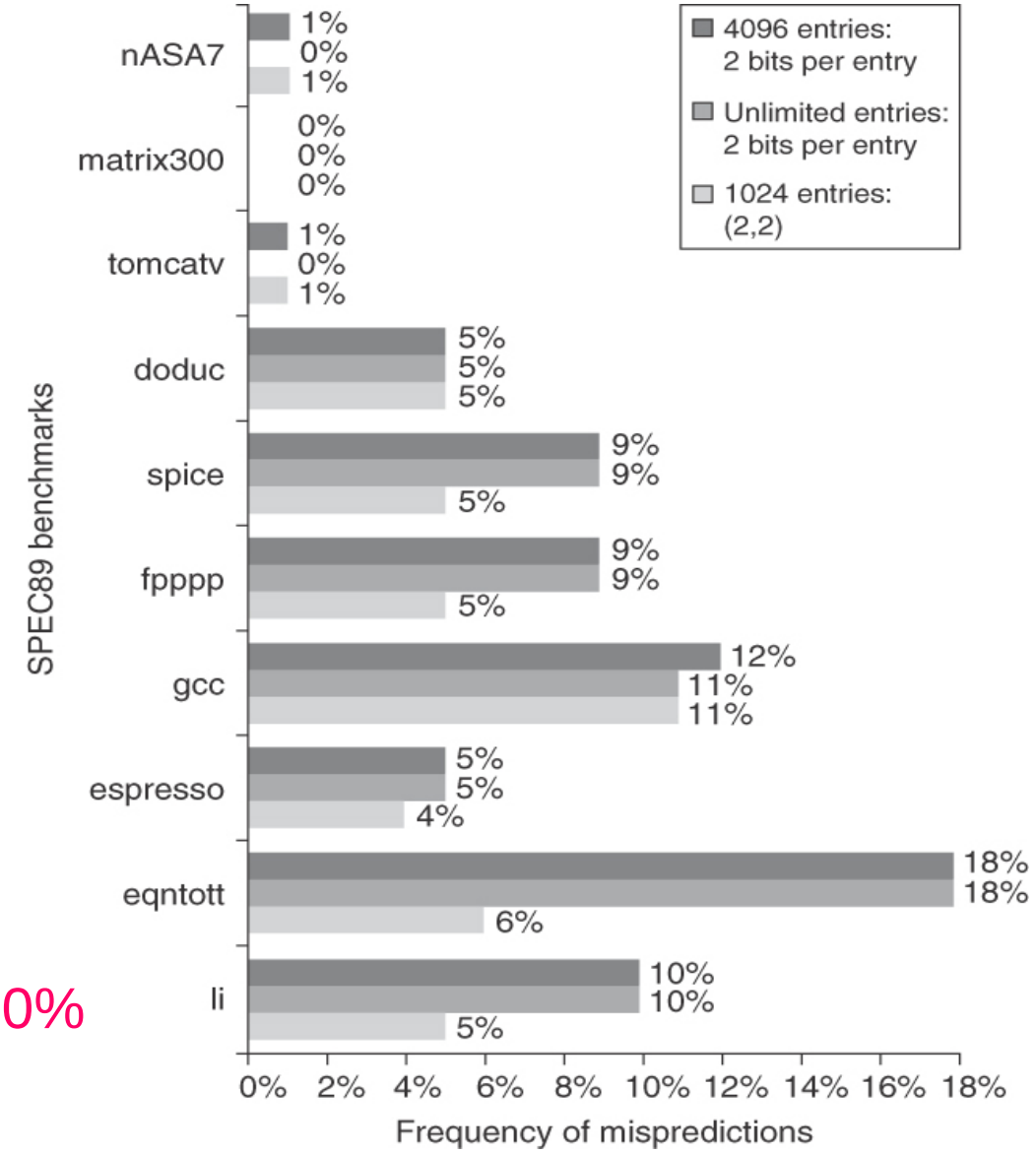
Recovery from wrong prediction:



# Performance of 2-bit Predictor

Prediction accuracy varies:

80% to 95%




Can the accuracy be ~= 100%

# Search for Advanced Predictor

```
If (a == 2) {          Taken

   a = 0 ;

}
If (b == 2){           Taken

   b = 0;

}
If  (a != b) {         NotTaken

      Do something;

}
```

Such branches are co-related.

- Simplest predictor: Just predict **not taken**
- 1-bit predictor
- 2-bit predictor

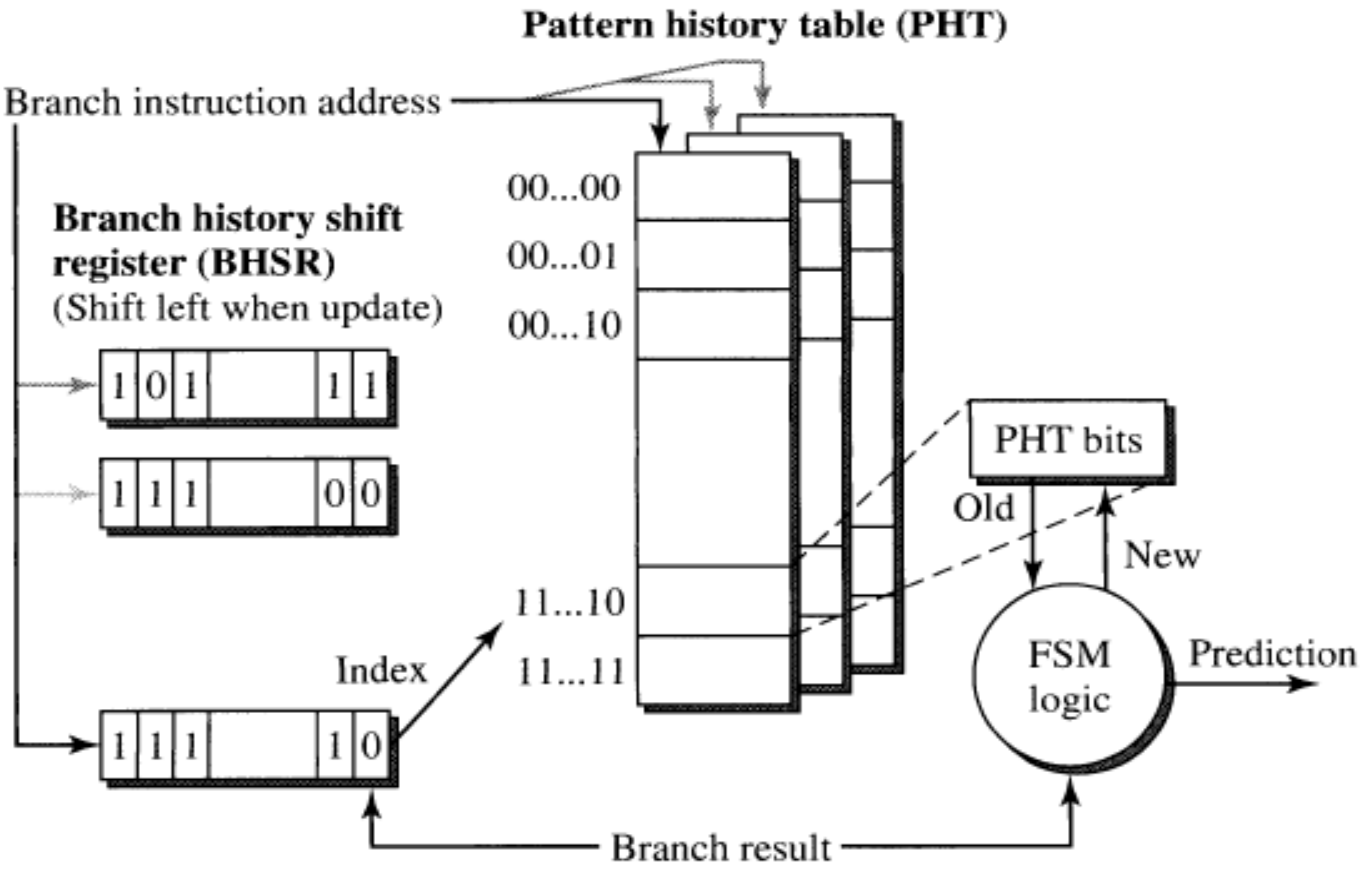
- Advance predictors:
  - Correlating predictor
  - Tournament predictor
  - Tagged Hybrid Predictor

# Correlating Predictor: a Framework

– Need to keep record of recently executed branch instructions!
– History of the branch instruction (for which the prediction is going to be made)
– Intelligent mechanism to make the final prediction out of these history.

Individual BHSR for every branch instruction

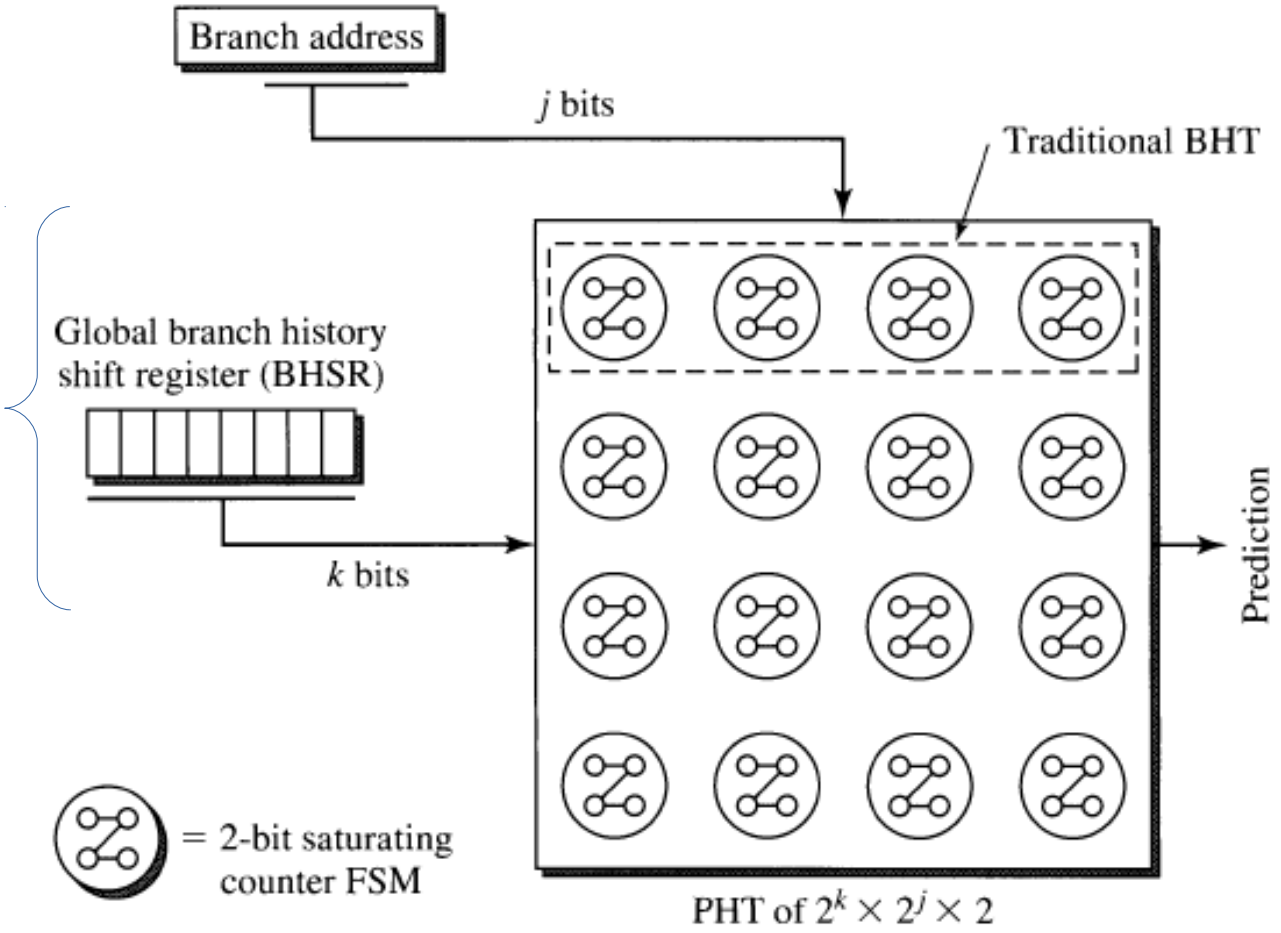
Left shift is performed on prediction




This is called as two level adaptive branch prediction

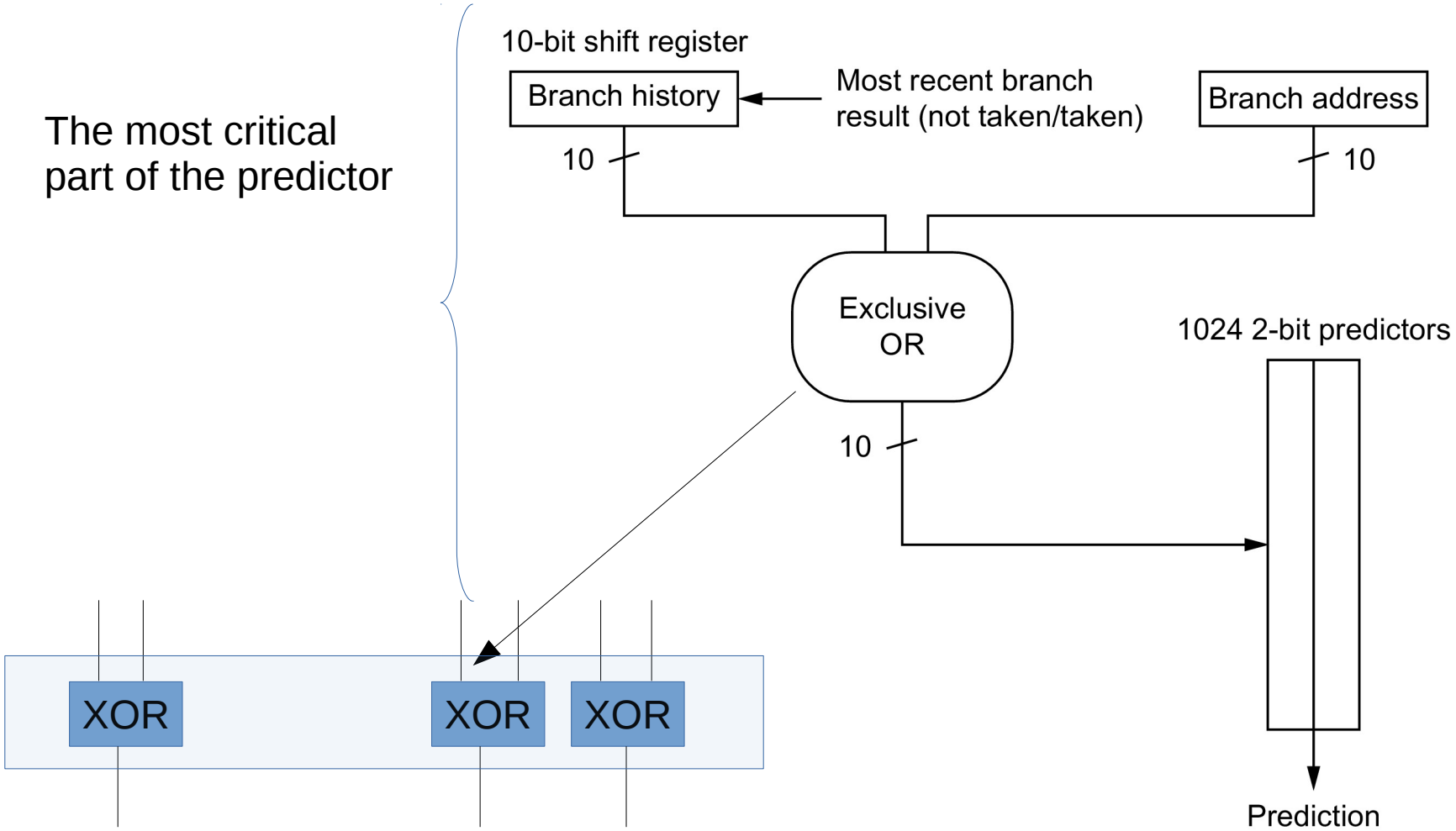
# Two-level Prediction: Global BHSR

Common for all the branch instructions



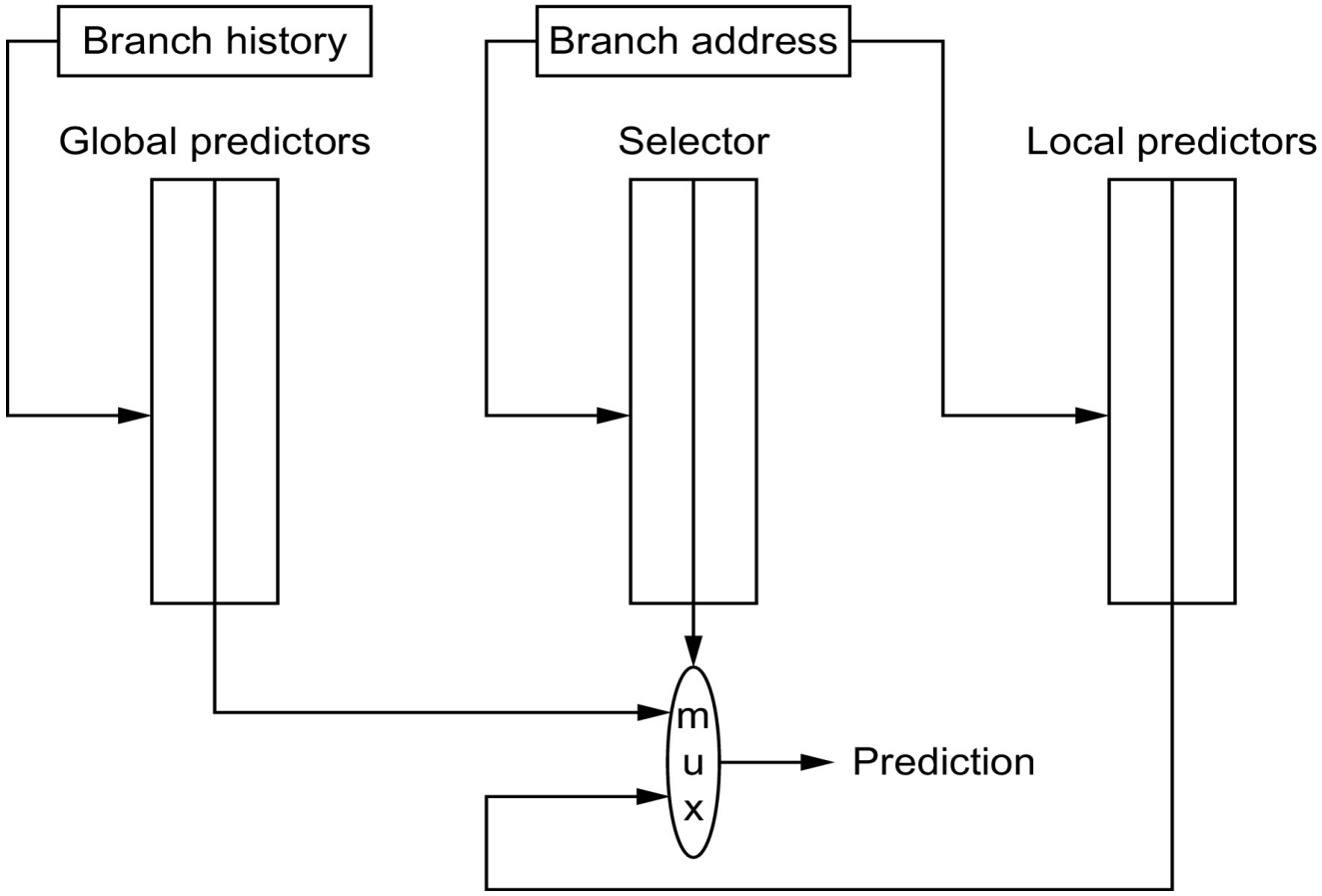
# gshare Predictor

The most critical part of the predictor

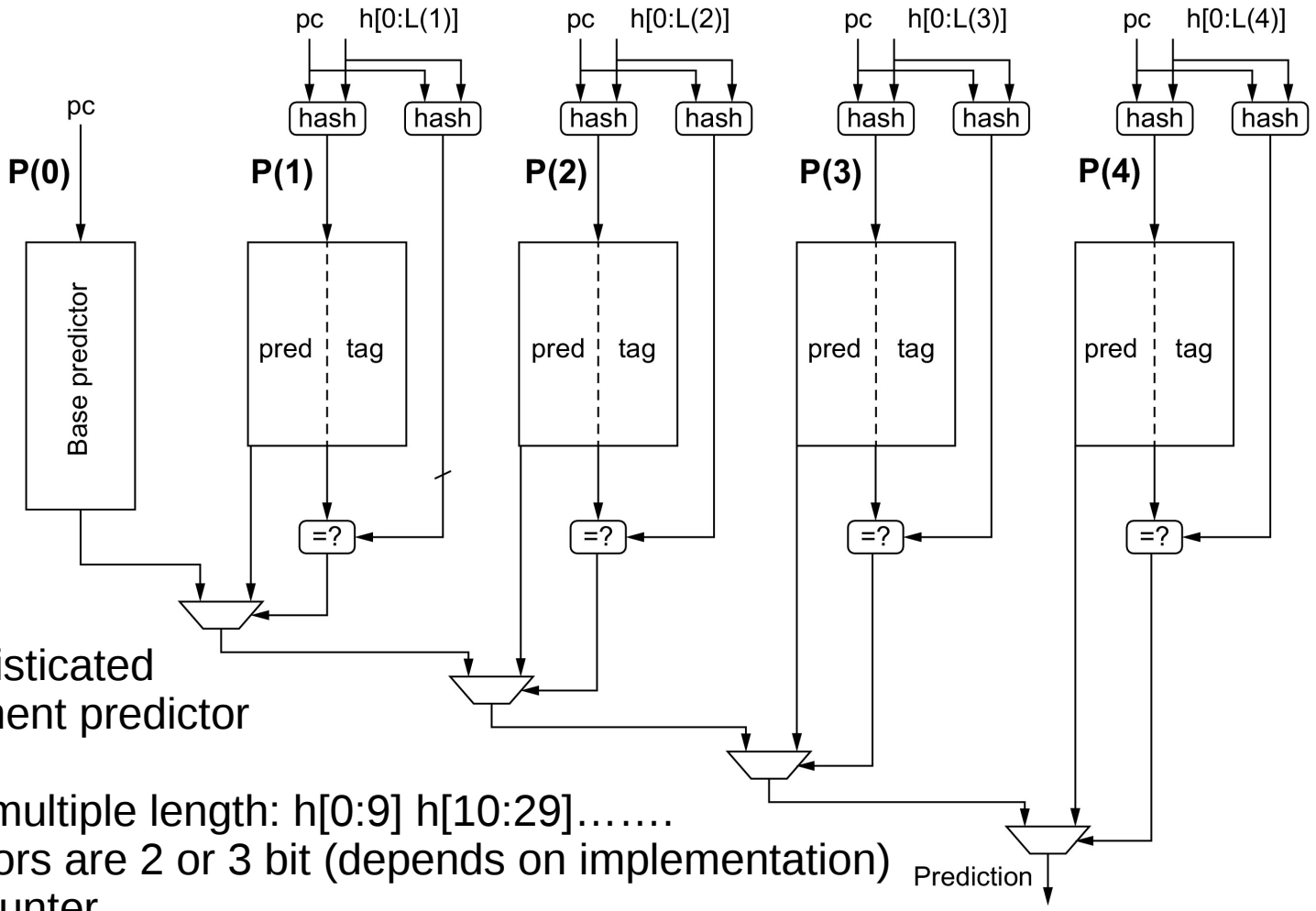


# Tournament Predictor

Intelligently chose the one from global and local predictors



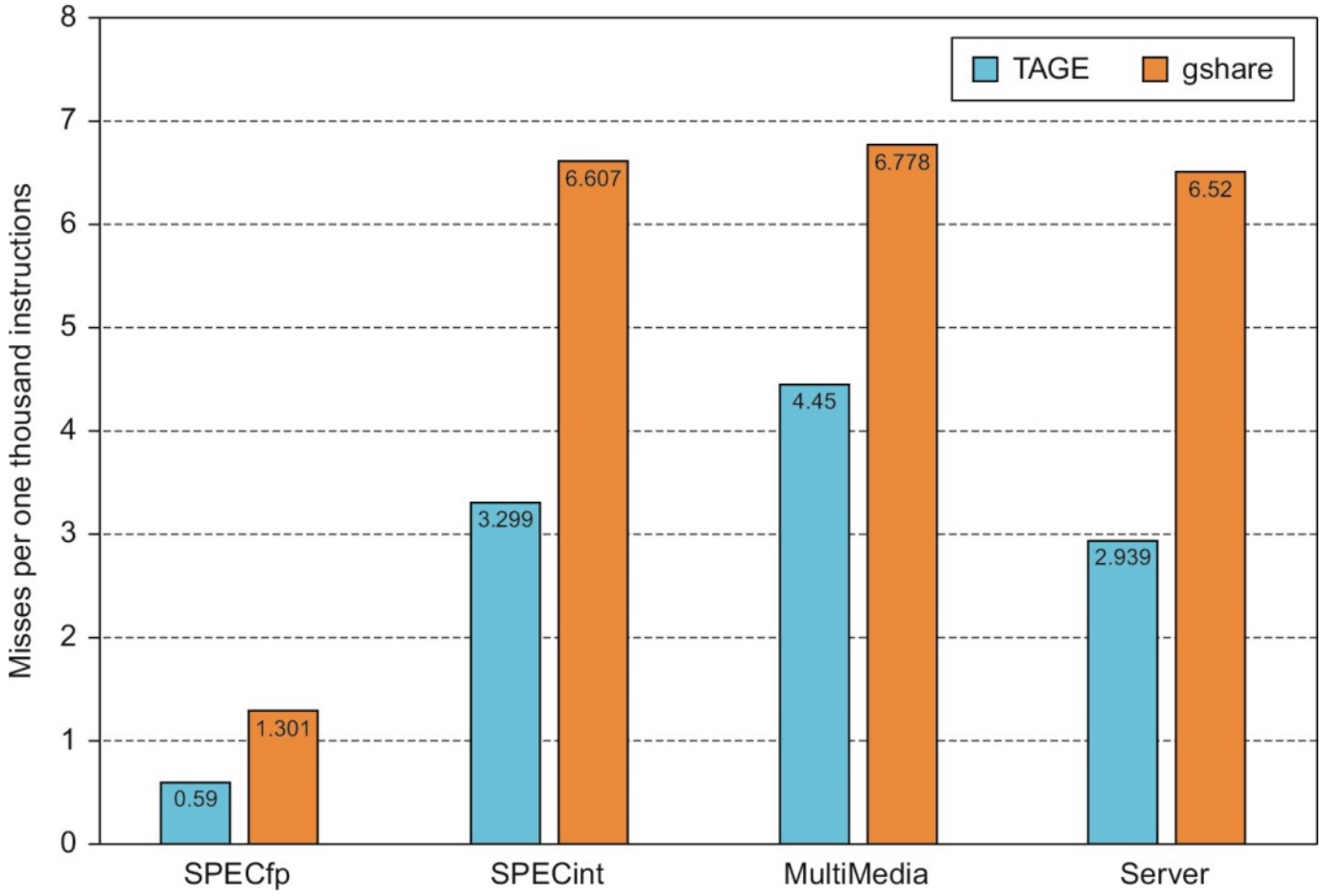
# Tagged Hybrid Predictor




– A more sophisticated than tournament predictor

– History with multiple length: h[0:9] h[10:29]…….
– Each predictors are 2 or 3 bit (depends on implementation) saturating counter.
– A complex structure, takes time for prediction, but highly accurate

# Tagged Vs gshare



# References

1) Advance branch prediction technique, Chapter 5, Shen and Lipasti

2) Reducing branch cost with advanced branch prediction, Chapter 3, Computer Architecture: Quantitative Approach.

3) A PPM-like, tag-based branch predictor, Pierre Michaud, Journal of Instruction Level Parallelism, Vol 7, 2005.

# Next Lecture

Dealing with data-flow (register and memory)

Dynamically resolving
WAR, WAW, and RAW

# Register Renaming

R1 ← R2 + R3
R1 ← R3 * R4

R5 ← R5 + R6
R6 ← R4 + R7

R1 ← R2 + R3
**RR1** ← R3 * R4

R5 ← R5 + R6
**RR6** ← R4 + R7

How this can be done in hardware?

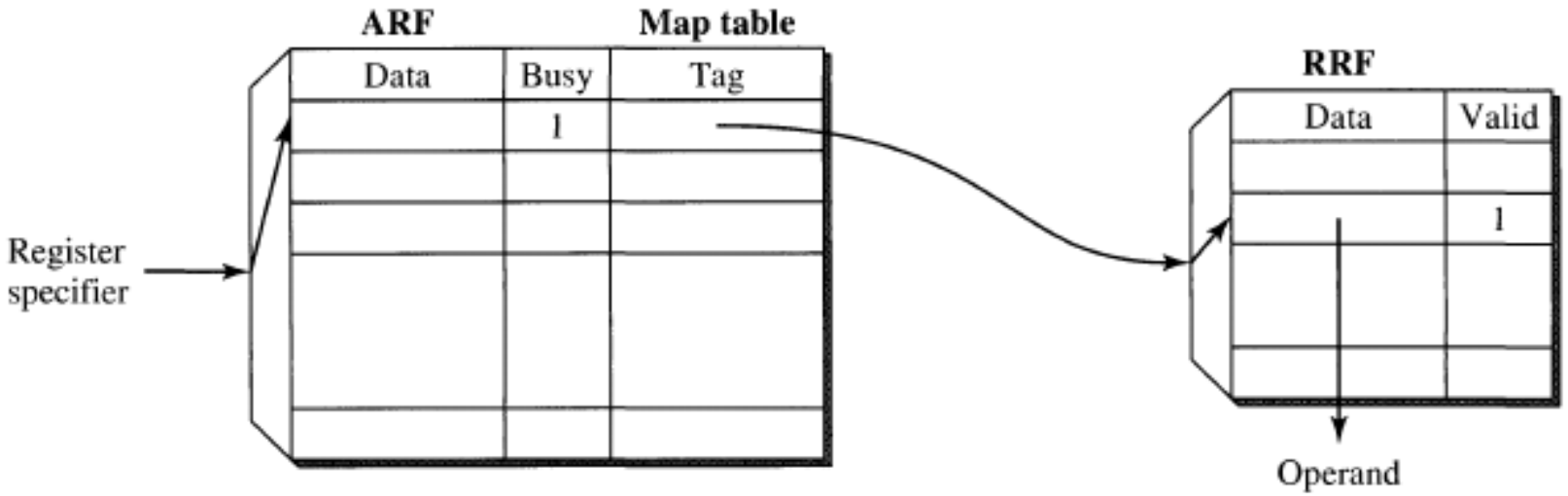
# Register Renaming

R1 ← R2 + R3
R1 ← R3 * R4

R5 ← R5 + R6
R6 ← R4 + R7

R1 ← R2 + R3
**RR1** ← R3 * R4

R5 ← R5 + R6
**RR6** ← R4 + R7



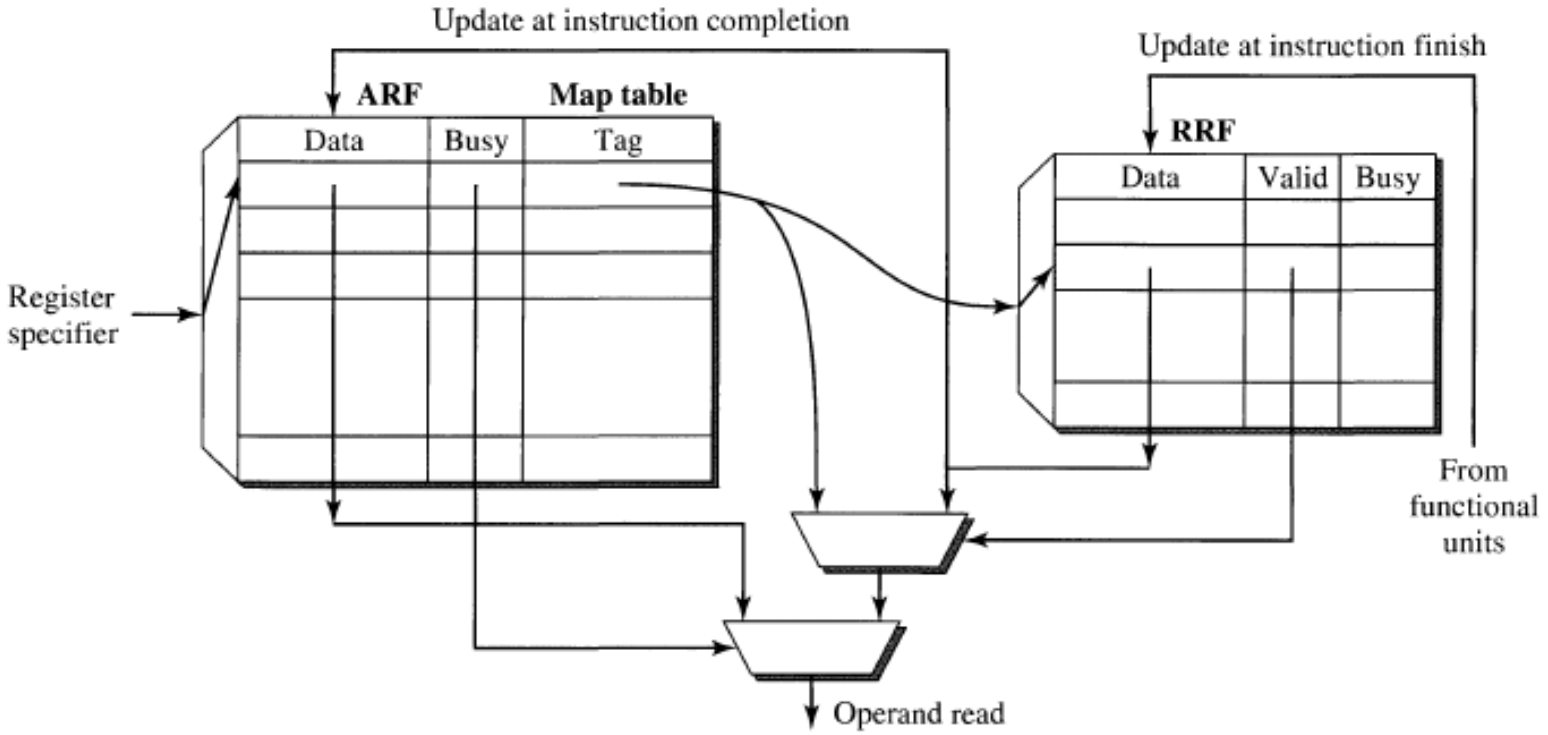

ARF – Architectural Register File
RRF – Renamed Register File

# Register Renaming

Updating the value in RRF and ARF
at finish and complete

R1 ← R2 + R3
R1 ← R3 * R4

R5 ← R5 + R6
R6 ← R4 + R7



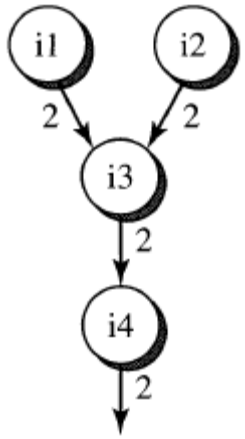
# True Data Dependency

Read after Write (RAW): one of the challenge for parallel execution

```
i1:  f2 ← load,4(r2)
i2:  f0 ← load,4(r5)
i3:  f0 ← fadd,f2,f0
i4:  4(r6) ← store,f0
i5:  f14 ← laod,8(r7)
i6:  f6 ← load,0(r2)
i7:  f5 ← load,0(r3)
i8:  f5 ← fsub,f6,f5
i9:  f4 ← fmul,f14,f5
i10:  f15 ← load,12(r7)
i11:  f7 ← load,4(r2)
i12:  f8 ← load,4(r3)
i13:  f8 ← fsub,f7,f8
i14:  f8 ← fmul,f15,f8
i15:  f8 ← fsub,f4,f8
i16:  0(r8) ← store,f8
```

Analyse latency and *data flow limit*

Let ADD, SUB, LOAD takes 2 cycles
MUL and DIV takes 4 cycles



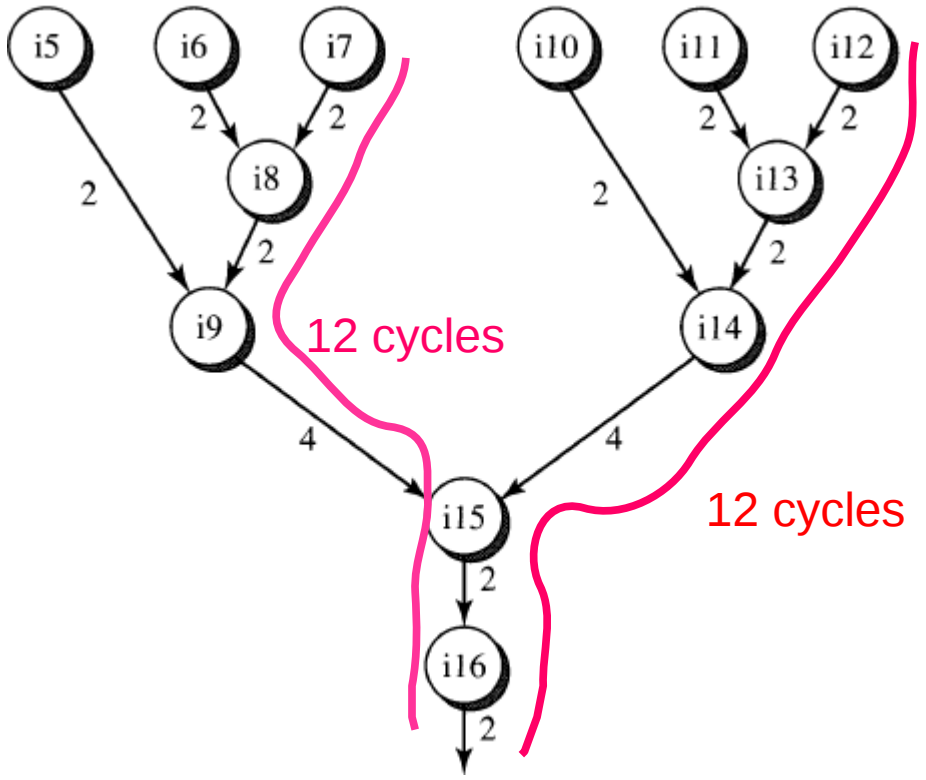

Data flow graph

# True Data Dependency

Read after Write (RAW): one of the challenge for parallel execution

```
i1:  f2 ← load,4(r2)
i2:  f0 ← load,4(r5)
i3:  f0 ← fadd,f2,f0
i4:  4(r6) ← store,f0
i5:  f14 ← laod,8(r7)
i6:  f6 ← load,0(r2)
i7:  f5 ← load,0(r3)
i8:  f5 ← fsub,f6,f5
i9:  f4 ← fmul,f14,f5
i10:  f15 ← load,12(r7)
i11:  f7 ← load,4(r2)
i12:  f8 ← load,4(r3)
i13:  f8 ← fsub,f7,f8
i14:  f8 ← fmul,f15,f8
i15:  f8 ← fsub,f4,f8
i16:  0(r8) ← store,f8
```

Data Flow Graph (DFG)

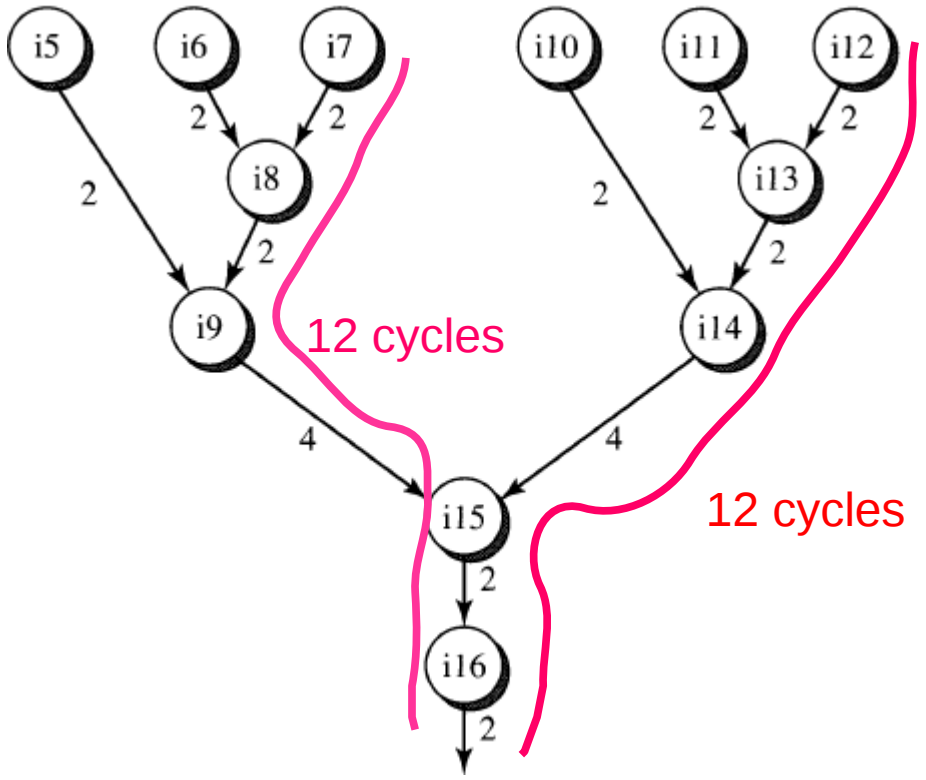


# True Data Dependency

Read after Write (RAW): one of the challenge for parallel execution

```
i1:  f2 ← load,4(r2)
i2:  f0 ← load,4(r5)
i3:  f0 ← fadd,f2,f0
i4:  4(r6) ← store,f0
i5:  f14 ← laod,8(r7)
i6:  f6 ← load,0(r2)
i7:  f5 ← load,0(r3)
i8:  f5 ← fsub,f6,f5
i9:  f4 ← fmul,f14,f5
i10:  f15 ← load,12(r7)
i11:  f7 ← load,4(r2)
i12:  f8 ← load,4(r3)
i13:  f8 ← fsub,f7,f8
i14:  f8 ← fmul,f15,f8
i15:  f8 ← fsub,f4,f8
i16:  0(r8) ← store,f8
```

Data Flow Graph (DFG)

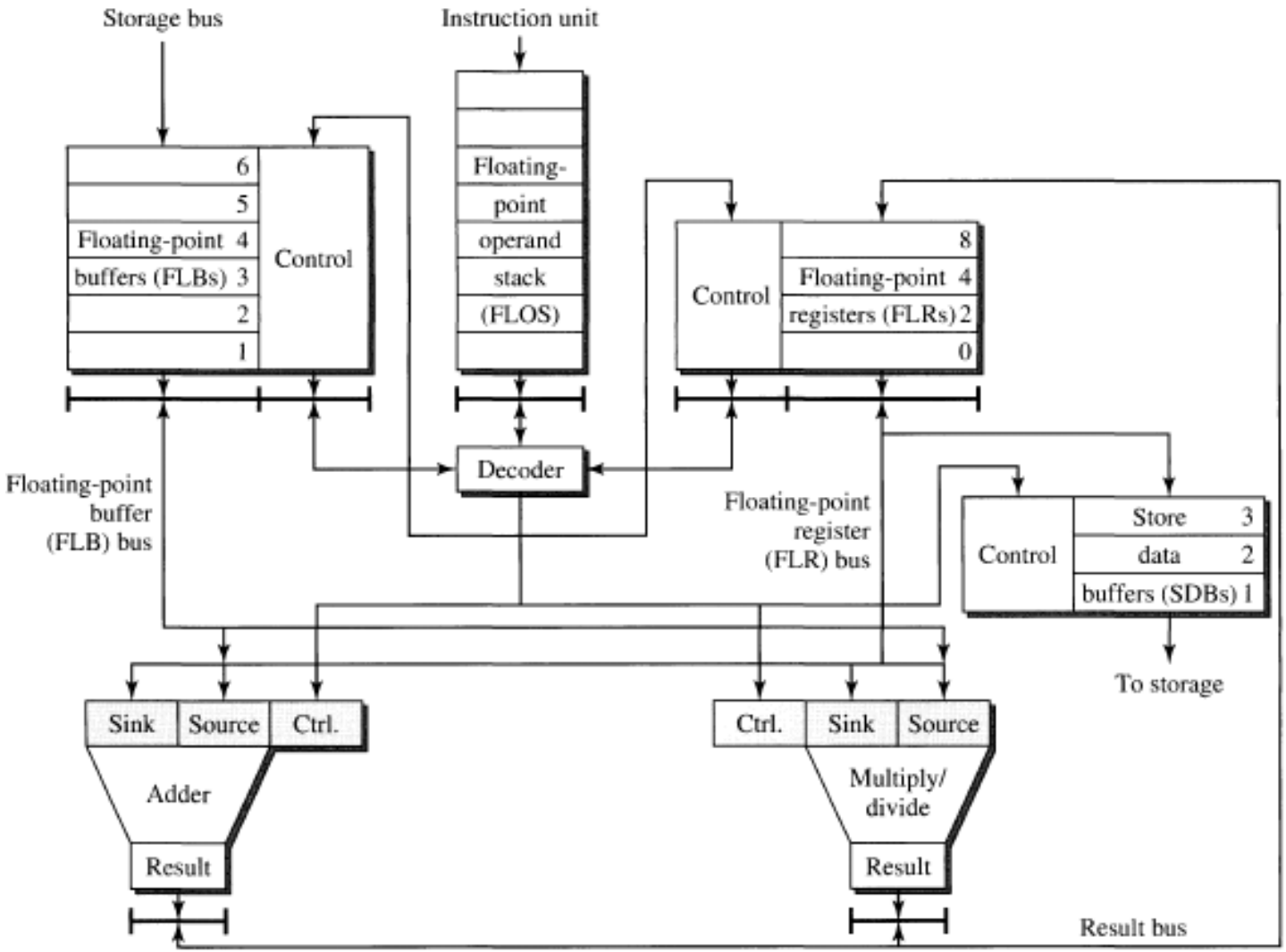


# How to Ensure Data Flow

Hardware implementation for data-flow techniques:

IBM FP unit processor

(without tomasulo)



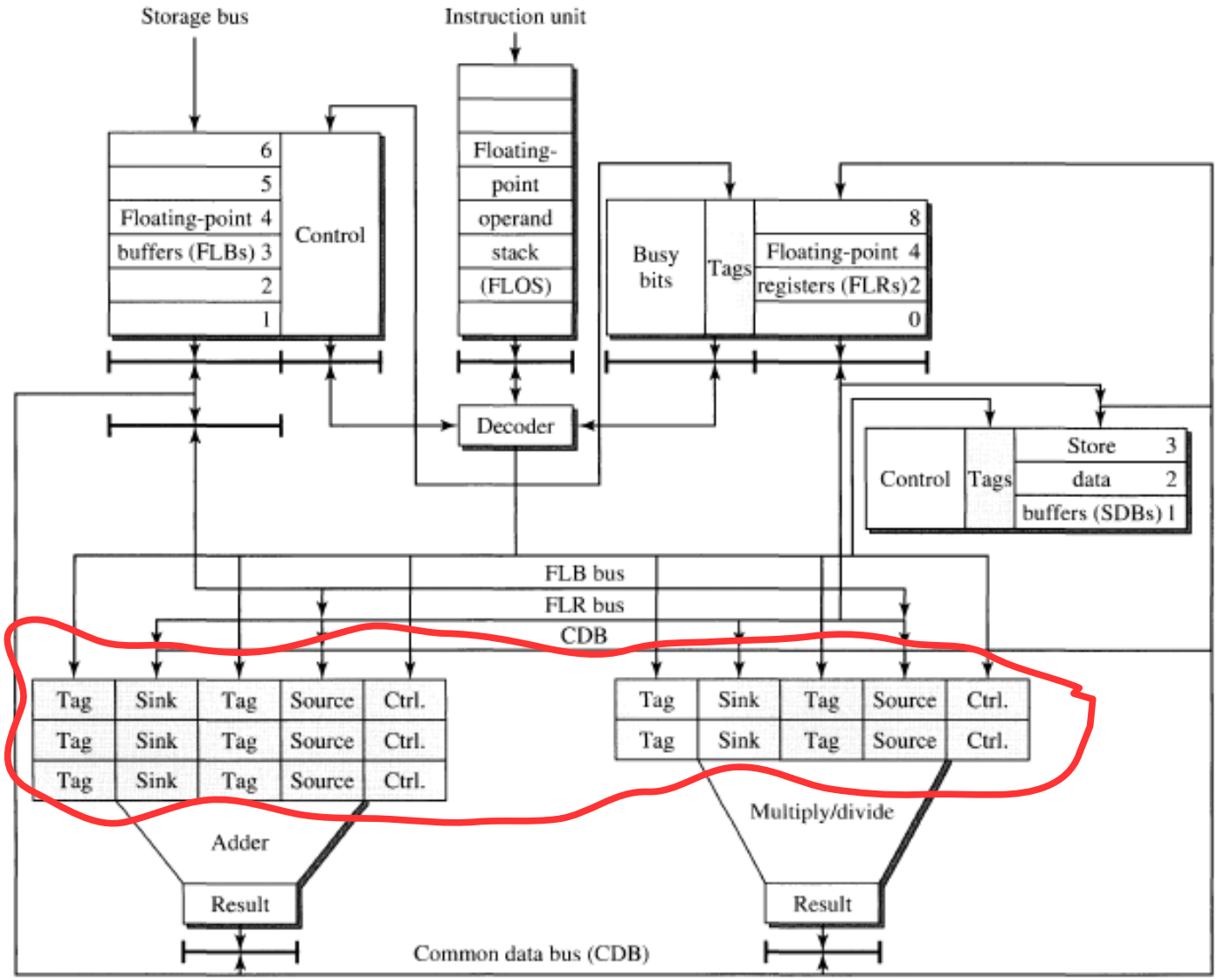
# How to Ensure Data Flow

Hardware implementation for data-flow techniques:

With Tomasulo Algorithm

(IBM machine),

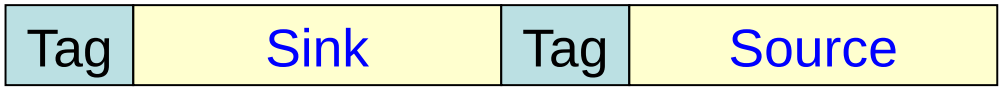
Almost all the cpu uses this



# Tomasulo Algorithm

Working of Tomasulo's Algorithm:

RS




(Reservation Station)

FLR


(Floating Point Registers)

SDB


(Store Data Buffer)

# Tomasulo Algorithm

Working of Tomasulo's Algorithm:

w: R4 ← R0 + R8

x: R2 ← R0 * R4

y: R4 ← R4 + R8

z: R8 ← R4 * R2

# Tomasulo Algorithm

Cycle 1: Dispatched instructions: w, x (in order)

w: R4 ← R0 + R8

x: R2 ← R0 * R4

y: R4 ← R4 + R8

z: R8 ← R4 * R2

RS

| | | Tag | Sink | Tag | Source |
|---|---|---|---|---|---|
| w | 1 | 0 | 6.0 | 0 | 7.8 |
| | 2 | | | | |
| | 3 | | | | |

w Adder

RS

| | | Tag | Sink | Tag | Source |
|---|---|---|---|---|---|
| x | 4 | 0 | 6.0 | 1 | --- |
| | 5 | | | | |

Mult/Div

FLR

| | Busy | Tag | Data |
|---|---|---|---|
| 0 | | | 6.0 |
| 2 | yes | 4 | 3.5 |
| 4 | yes | 1 | 10.0 |
| 8 | | | 7.8 |

# Tomasulo Algorithm

Cycle 2: Dispatched instructions: y, z (in order)

w: R4 ← R0 + R8

x: R2 ← R0 * R4

y: R4 ← R4 + R8

z: R8 ← R4 * R2

RS

| | | Tag | Sink | Tag | Source |
|---|---|---|---|---|---|
| w | 1 | 0 | 6.0 | 0 | 7.8 |
| y | 2 | 1 | --- | 0 | 7.8 |
| | 3 | | | | |

w Adder

RS

| | | Tag | Sink | Tag | Source |
|---|---|---|---|---|---|
| x | 4 | 0 | 6.0 | 1 | --- |
| z | 5 | 2 | --- | 4 | --- |

Mult/Div

FLR

| | Busy | Tag | Data |
|---|---|---|---|
| 0 | | | 6.0 |
| 2 | yes | 4 | 3.5 |
| 4 | yes | 2 | 10.0 |
| 8 | yes | 5 | 7.8 |

# Tomasulo Algorithm

Cycle 3: Dispatched instructions: _____

w: R4 ← R0 + R8

x: R2 ← R0 * R4

y: R4 ← R4 + R8

z: R8 ← R4 * R2

RS

| | | Tag | Sink | Tag | Source |
|---|---|---|---|---|---|
| | 1 | | | | |
| y | 2 | 0 | 13.8 | 0 | 7.8 |
| | 3 | | | | |

y Adder

RS

| | | Tag | Sink | Tag | Source |
|---|---|---|---|---|---|
| x | 4 | 0 | 6.0 | 0 | 13.8 |
| z | 5 | 2 | --- | 4 | --- |

x Mult/Div

FLR

| | Busy | Tag | Data |
|---|---|---|---|
| 0 | | | 6.0 |
| 2 | yes | 4 | 3.5 |
| 4 | yes | 2 | 10.0 |
| 8 | yes | 5 | 7.8 |

# Tomasulo Algorithm

Cycle 4: Dispatched instructions:_____

w: R4 ← R0 + R8

x: R2 ← R0 * R4

y: R4 ← R4 + R8

z: R8 ← R4 * R2

RS

| | | Tag | Sink | Tag | Source |
|---|---|---|---|---|---|
| | 1 | | | | |
| y | 2 | 0 | 13.8 | 0 | 7.8 |
| | 3 | | | | |

y Adder

RS

| | | Tag | Sink | Tag | Source |
|---|---|---|---|---|---|
| x | 4 | 0 | 6.0 | 0 | 13.8 |
| | 5 | 2 | --- | 4 | --- |

x Mult/Div

FLR

| | Busy | Tag | Data |
|---|---|---|---|
| 0 | | | 6.0 |
| 2 | yes | 4 | 3.5 |
| 4 | yes | 2 | 10.0 |
| 8 | yes | 5 | 7.8 |

# Tomasulo Algorithm

Cycle 5: Dispatched instructions:_____

w: R4 ← R0 + R8

x: R2 ← R0 * R4

y: R4 ← R4 + R8

z: R8 ← R4 * R2

RS

| | Tag | Sink | Tag | Source |
|---|---|---|---|---|
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |

Adder

RS

| | | Tag | Sink | Tag | Source |
|---|---|---|---|---|---|
| x | 4 | 0 | 6.0 | 0 | 13.8 |
| z | 5 | 0 | 21.6 | 4 | --- |

Mult/Div

FLR

| | Busy | Tag | Data |
|---|---|---|---|
| 0 | | | 6.0 |
| 2 | yes | 4 | 3.5 |
| 4 | | | 21.6 |
| 8 | yes | 5 | 7.8 |

# Tomasulo Algorithm

Cycle 6: Dispatched instructions:_____

w: R4 ← R0 + R8

x: R2 ← R0 * R4

y: R4 ← R4 + R8

z: R8 ← R4 * R2

RS

| | Tag | Sink | Tag | Source |
|---|---|---|---|---|
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |

Adder

RS

| | Tag | Sink | Tag | Source |
|---|---|---|---|---|
| 4 | | | | |
| z 5 | 0 | 21.6 | 0 | 82.8 |

(z) Mult/Div

FLR

| | Busy | Tag | Data |
|---|---|---|---|
| 0 | | | 6.0 |
| 2 | | | 82.8 |
| 4 | | | 21.6 |
| 8 | yes | 5 | 7.8 |

# Next Lecture

Memory Data Flow

- Load bypassing
- Load forwarding