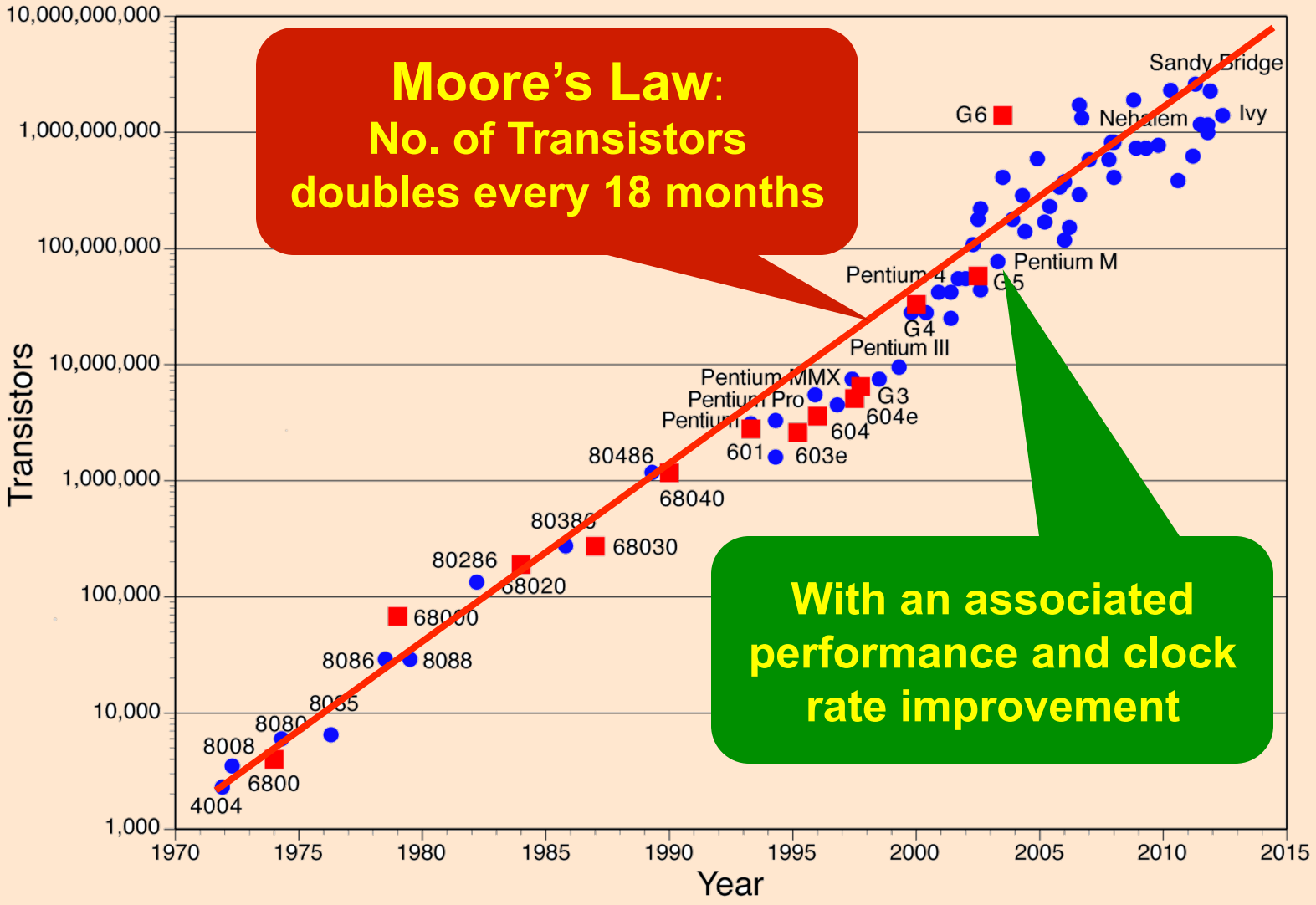

# Instruction-Level Parallelism (ILP) Architectures

R. Govindarajan
CSA & SERC,
IISc, Bangalore
govind@iisc.ac.in

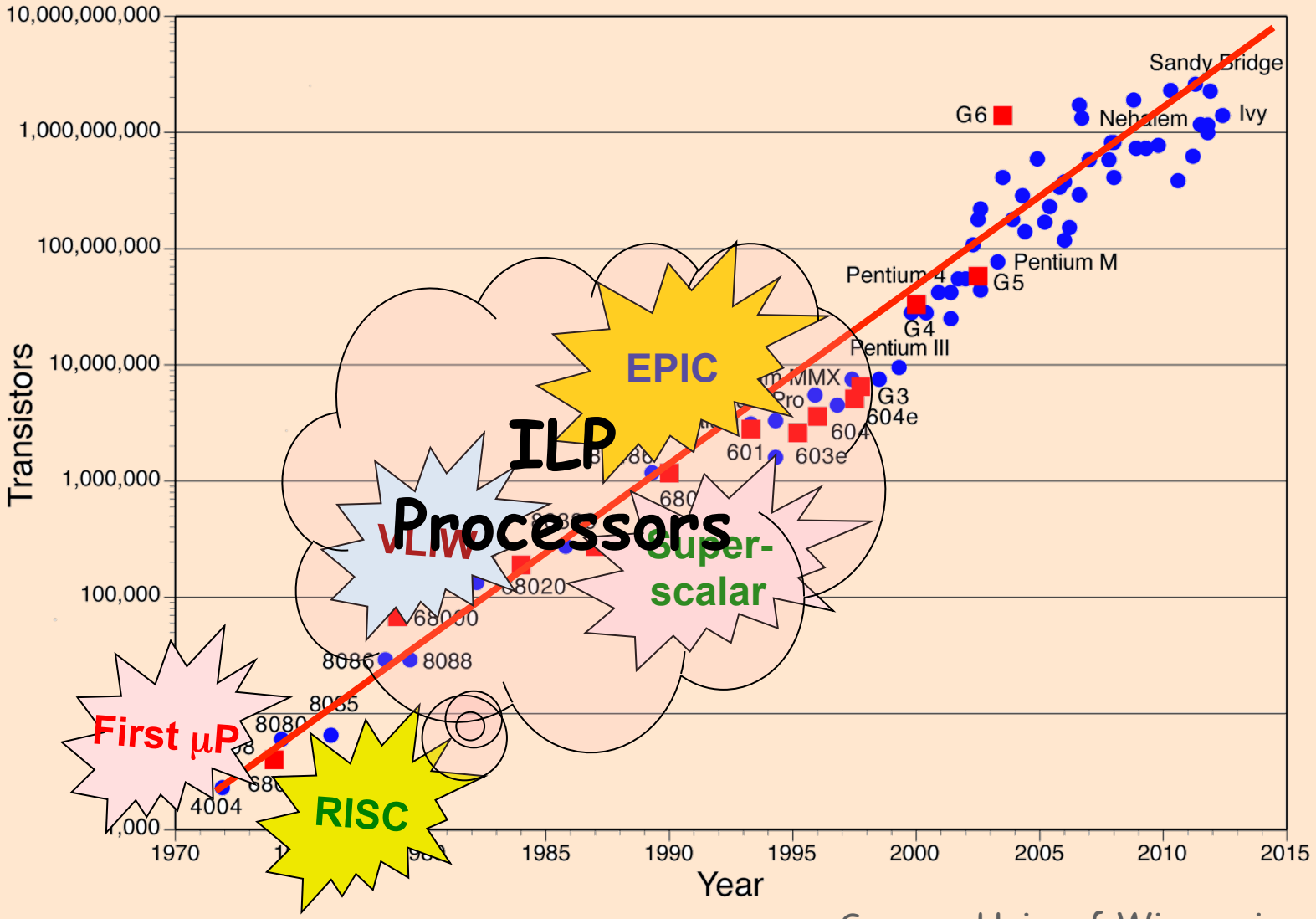
21 Dec. 2020

# Moore's Law : Transistors



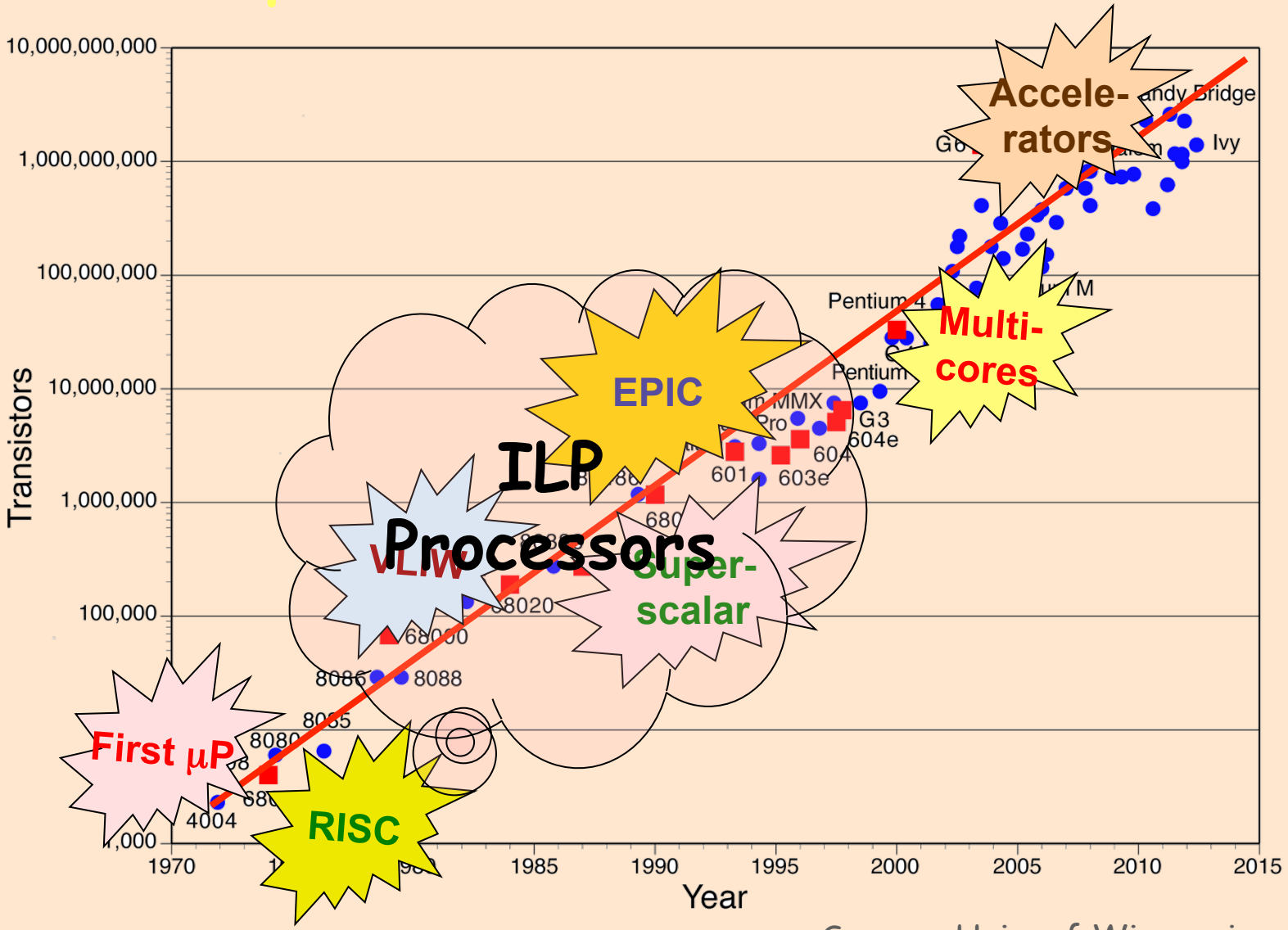

Source: Univ. of Wisconsin

# Processor Architecture Roadmap: Previous Millenium



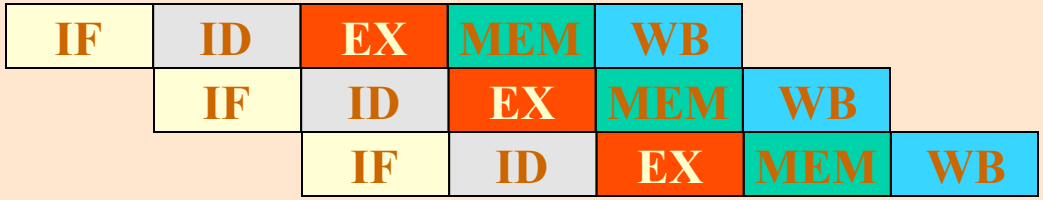

Source: Univ. of Wisconsin

# Processor Architecture Roadmap : Current Millenium




Source: Univ. of Wisconsin

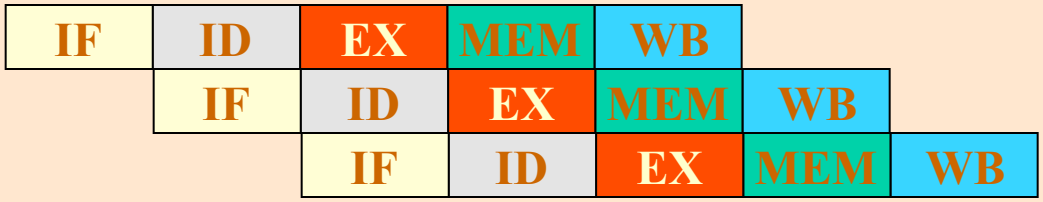
# Towards ILP Architectures



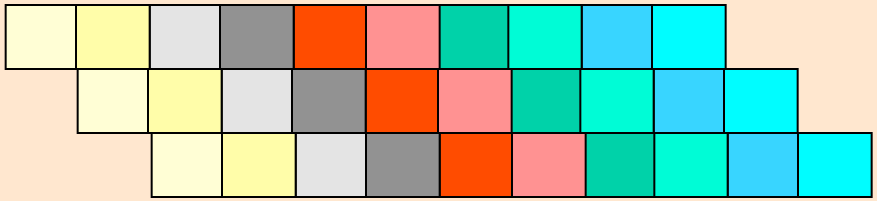

Pipelined

- ILP Architecture
  - Ability to fetch, decode, issue and execute multiple instructions per cycle
  - Why? Improve the instruction execution throughput (Instructions Per Cycle (IPC)).

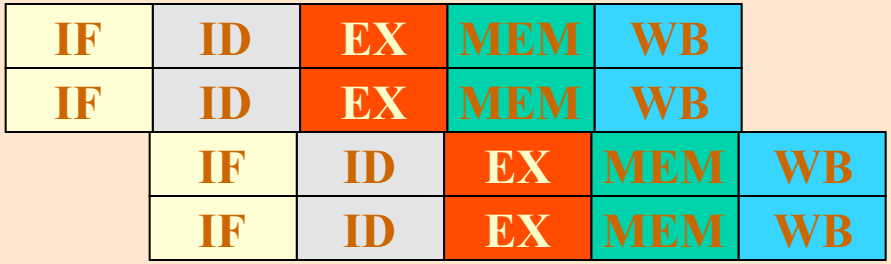
# ILP Architectures



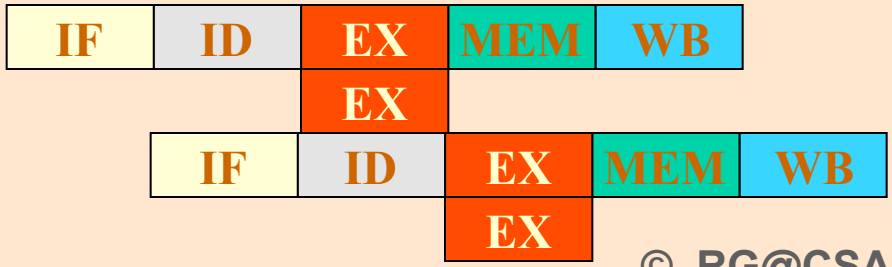

Pipelined



Superpipelined




Superscalar




VLIW

# Instruction-Level Parallelism

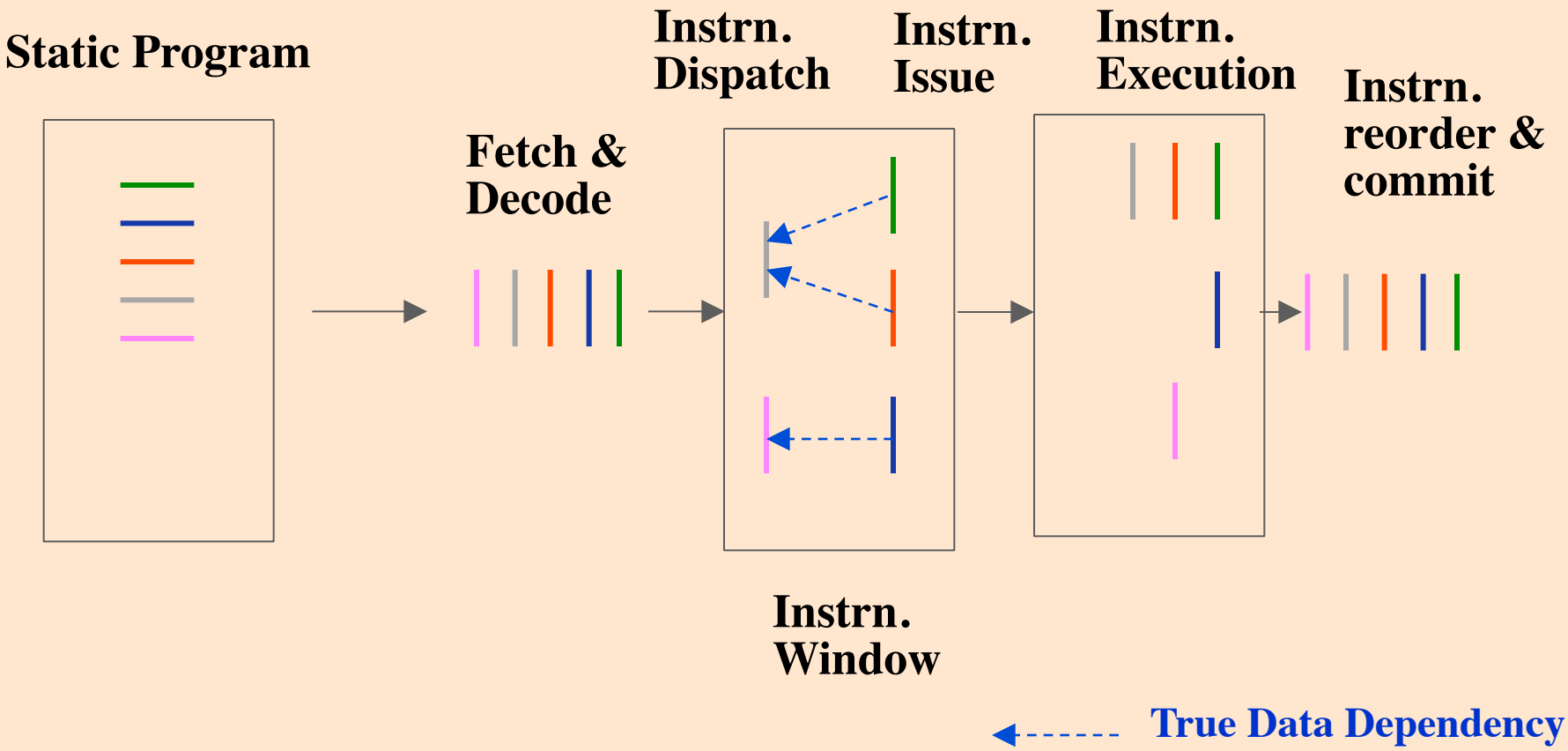

- Removes non-essential sequentiality and achieves parallel execution where possible. However, must maintain sequential specification.
- Superpipelining: stages of the pipeline are further pipelined.
  - Results in deep pipelines and faster clocks.
- Very Long Instruction Word (VLIW) Architecture
  - Multiple operations per instrn. packed at Compile time
  - Compiler/Programmer responsibility to expose ILP
  - Less hardware complexity

  Examples: Cydra, Multi-Flow, TI-C6x, Trimedia.

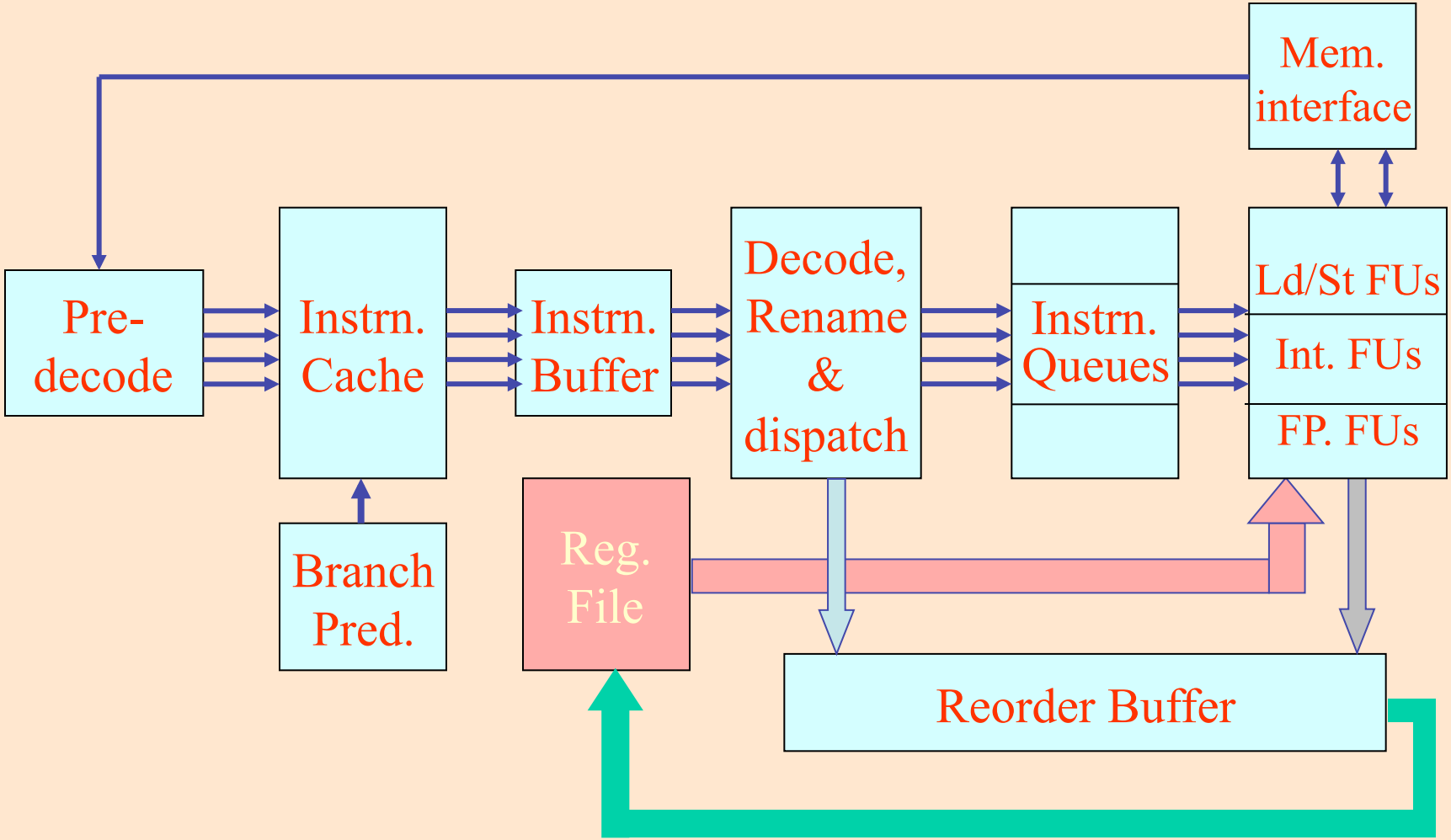
# Superscalar Architecture


- Multiple instructions are fetched, decoded, issued and executed each cycle.
- Hardware detection of which instructions can be executed in parallel in a cycle at run-time.
- *In-order* vs. *Out-of-Order* Issue
- Examples: Most modern processors are superpipelined, superscalar architectures: DEC 21x64, MIPS R-10000, PowerPC, Intel/ AMD x86, UltraSparc II, etc.

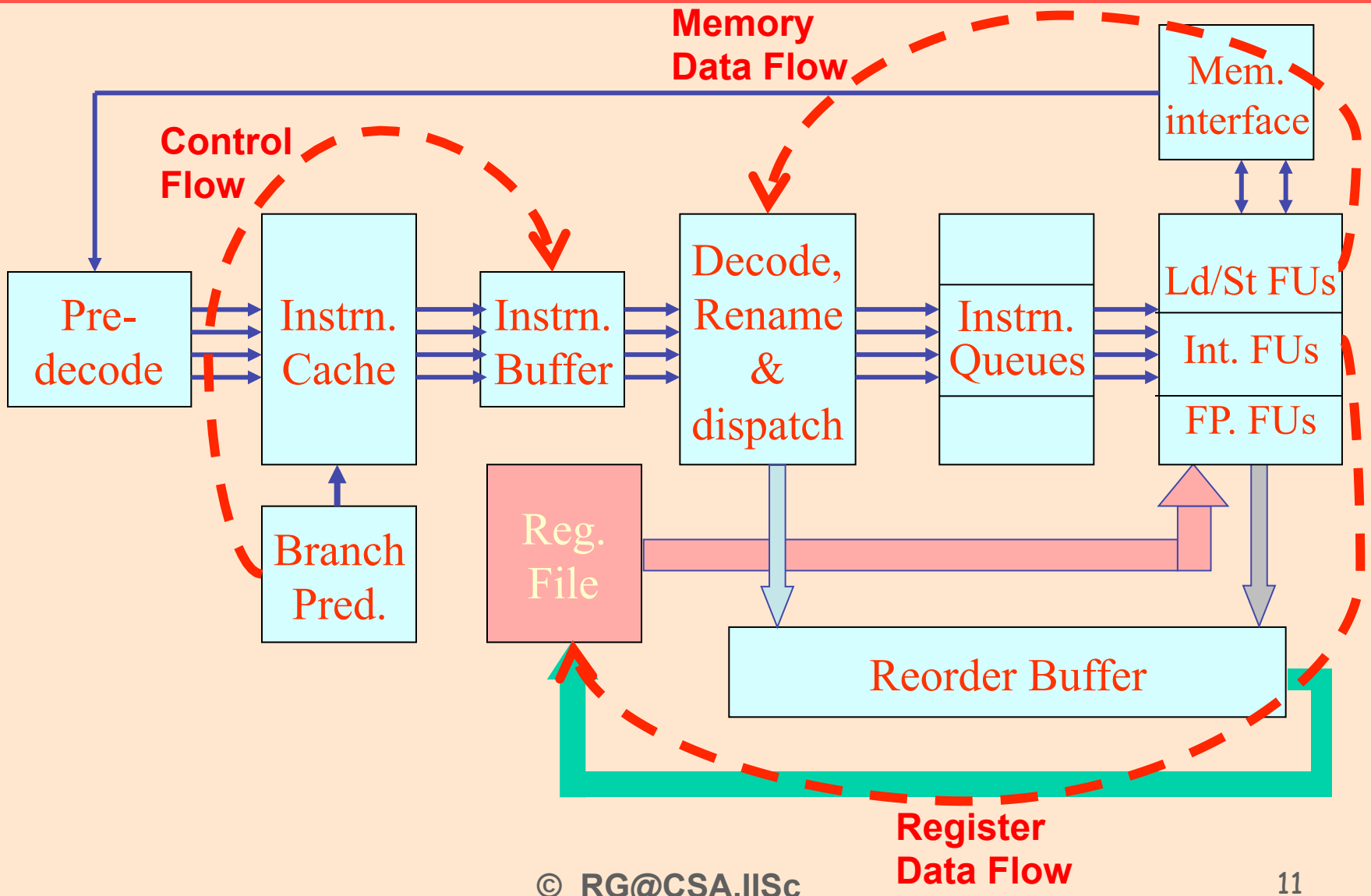
# Superscalar Execution Model



# Microarchitecture Overview



# Impediments to IPC

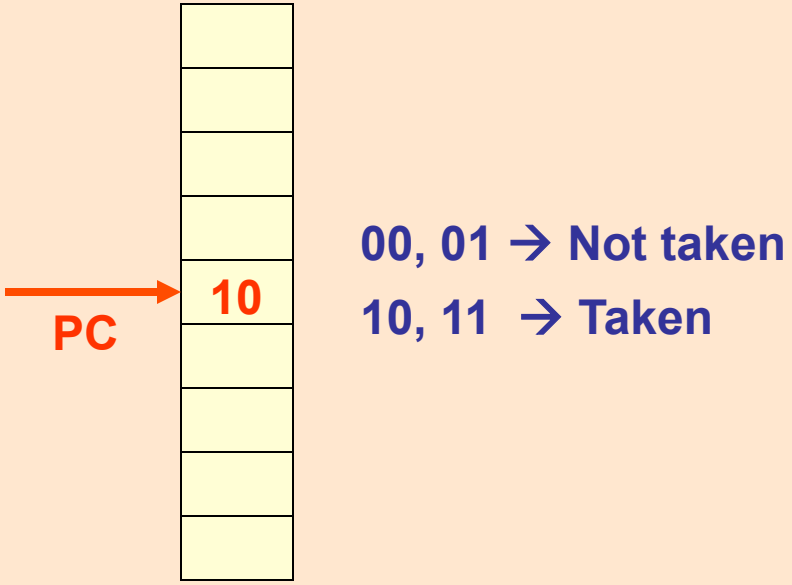


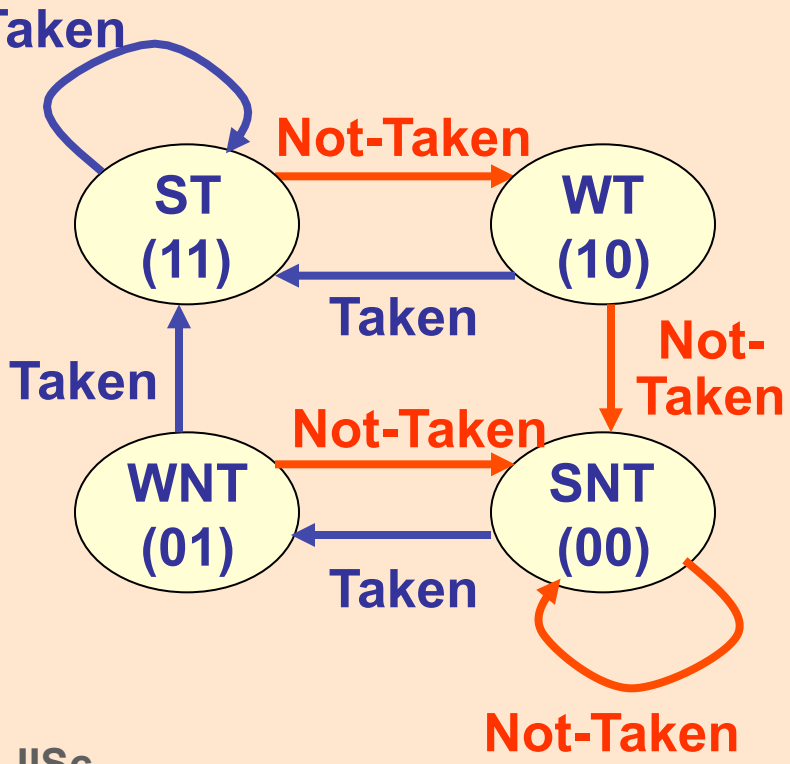
# Reducing Impact of Control Flow


- Conditional branches involve 2 things
  1. resolving the branch (determine whether it is to be taken or not-taken)
  2. computing the branch target address
- Could do both in ID stage to reduce branch stall to 1 cycle
- Branch Prediction
  - Static Prediction (always predict “not-taken”)
  - Dynamic Prediction
    - 1-bit, 2-bit, … predictions
    - Two-level, path-based, perceptron-based predictors, ...
    - Hybrid predictor

# 2-bit Branch Prediction

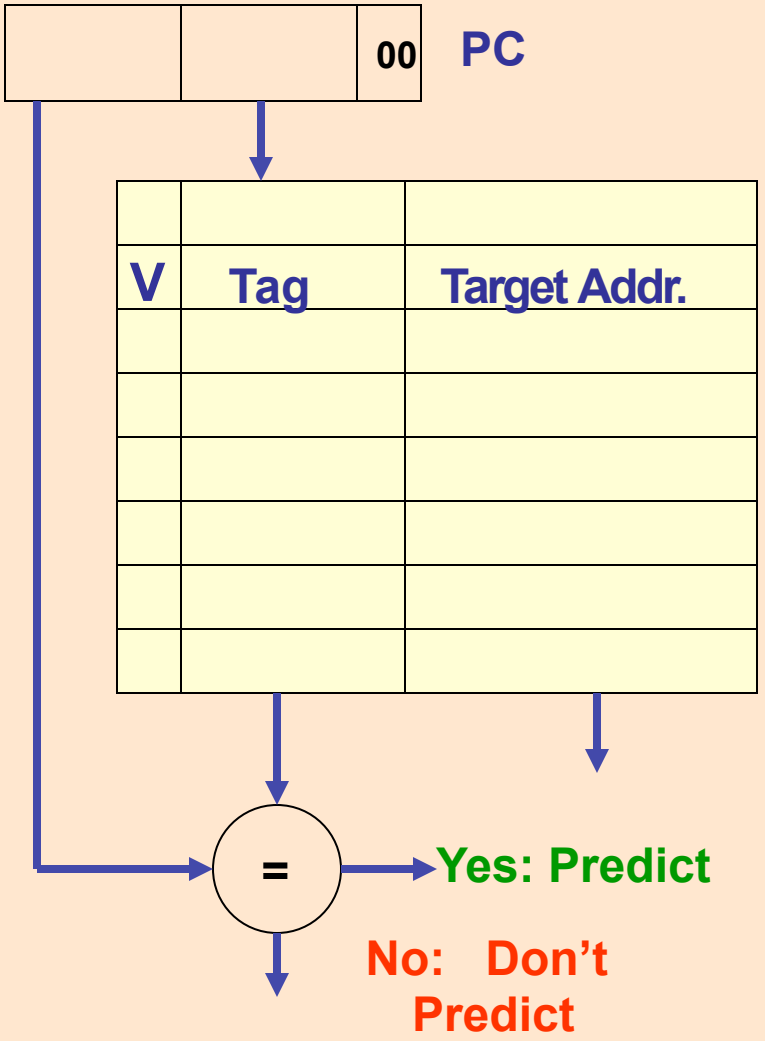
- Store previous history of a branch -- taken or not-taken -- in Branch Prediction Buffer
- State of each static branch represented by 2-bits.
- State-transition on wrong prediction



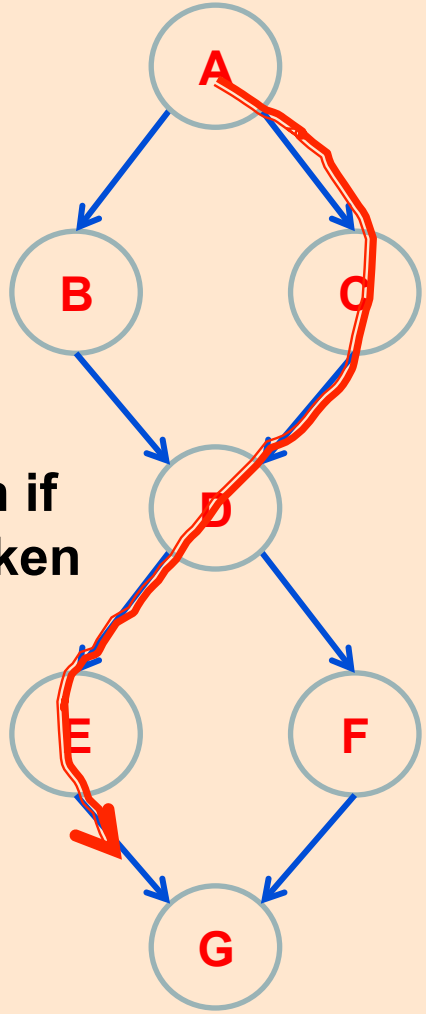
# Branch Target Buffer

- Store branch target address (computed when the static branch instrn. is encountered first) along with prediction bits -- Branch Target Buffer.
- Associate tag bits to avoid other (branch) instrns. influence the prediction.
- When to make the prediction?
- Prediction can be made in IF stage itself! no stall on correct prediction.



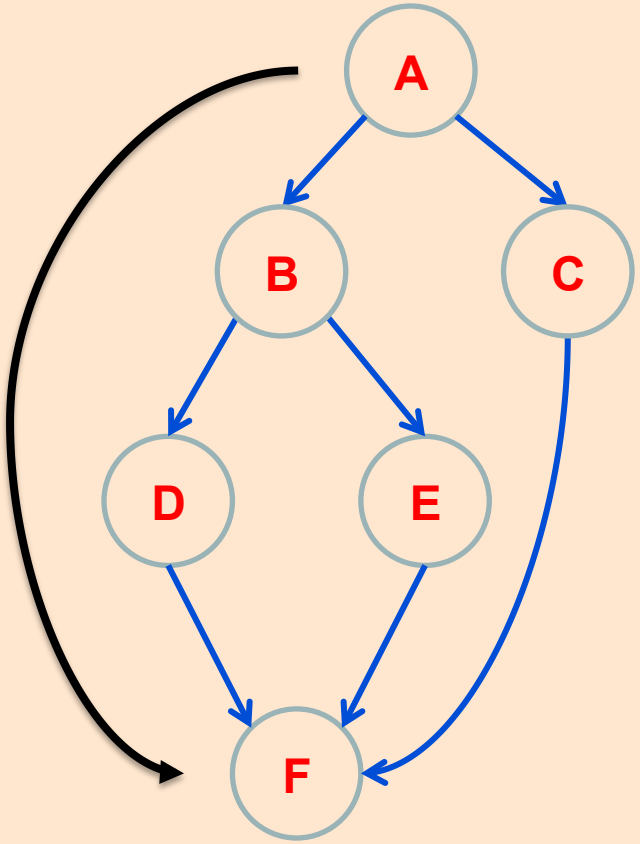
# Other Branch Prediction Schemes


## Correlated Prediction




D is taken if
A is not taken

## Path Prediction

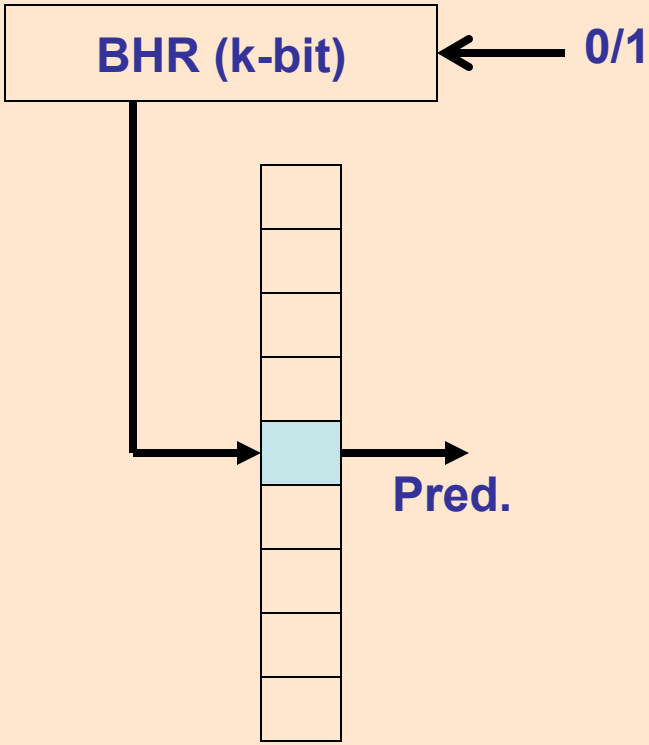


# 2-Level Prediction (contd.)

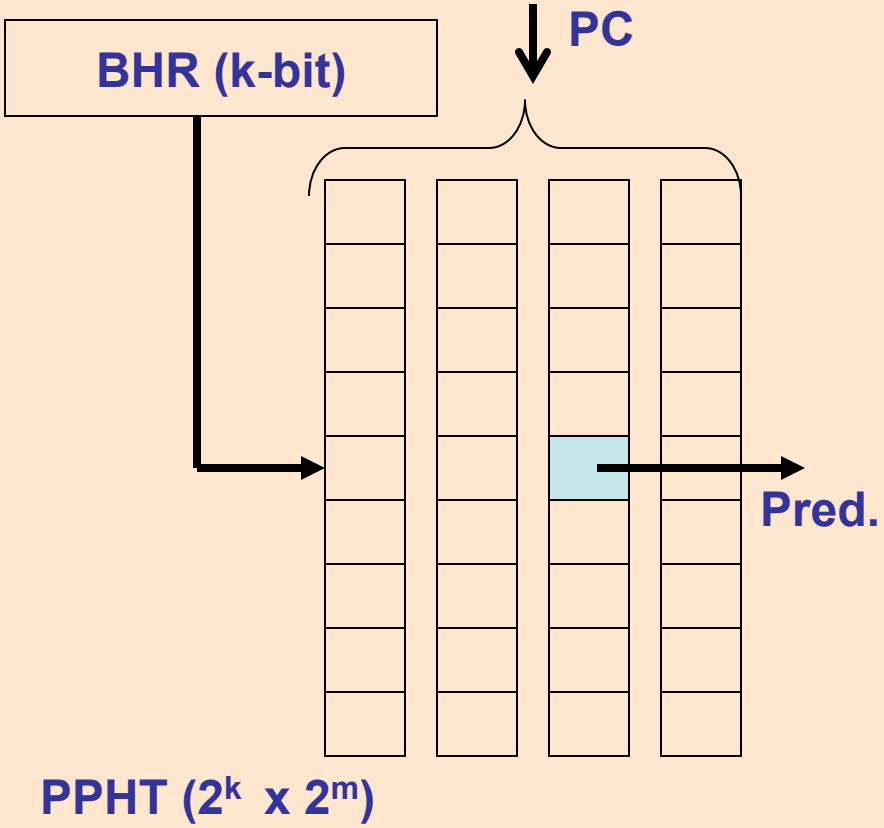

- Branch history register/table to maintain the outcome of previous (dynamic) k branches.
- Pattern History table(s) to record the branch behavior (2-bit FSM).
- Branch history can be global or local
- Pattern history can be global or local.
- Other variations (g-share, p-share)
- Results upto 98% prediction accuracy.

# GAg and GAp Schemes

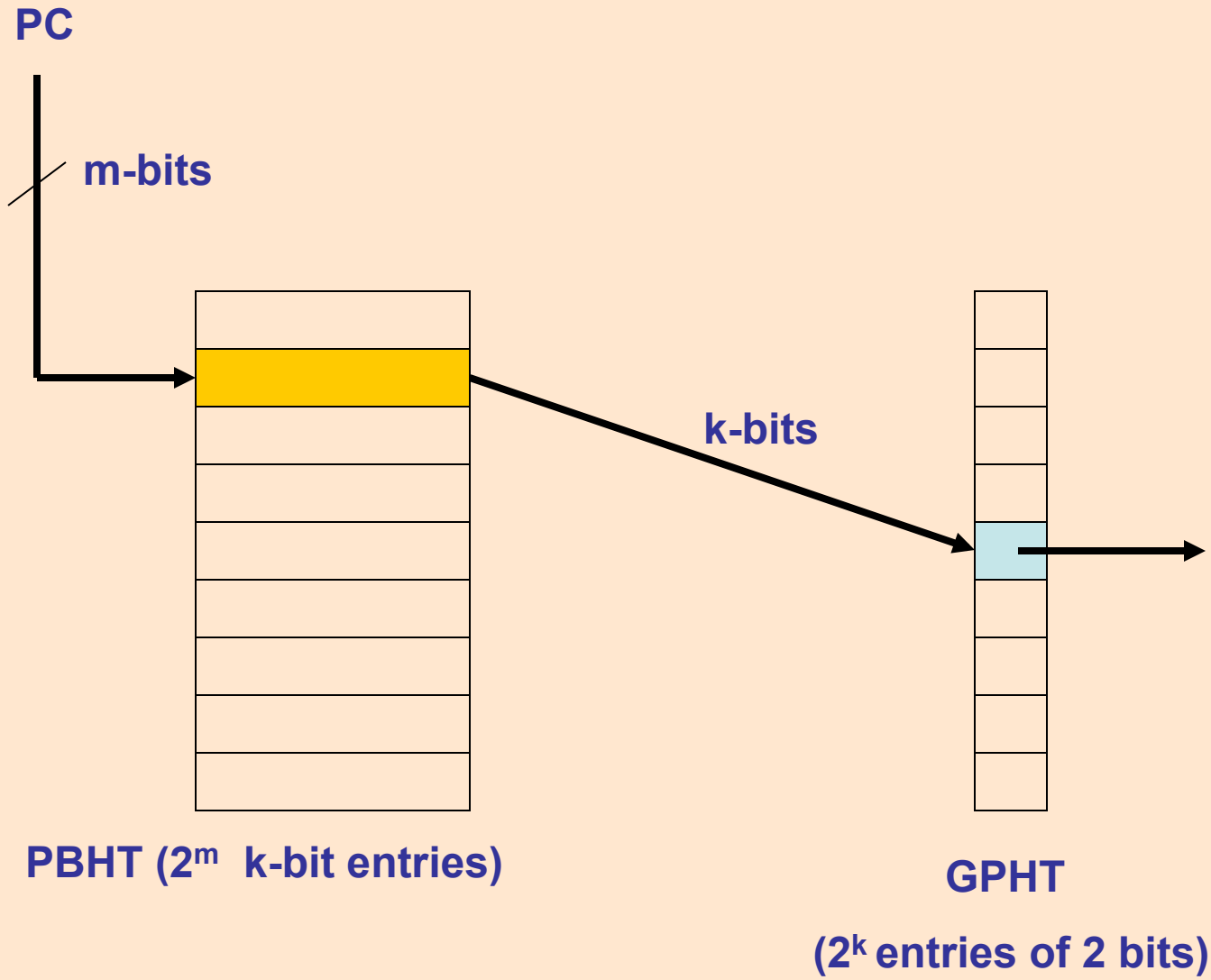

Global branch history register & global pattern history table



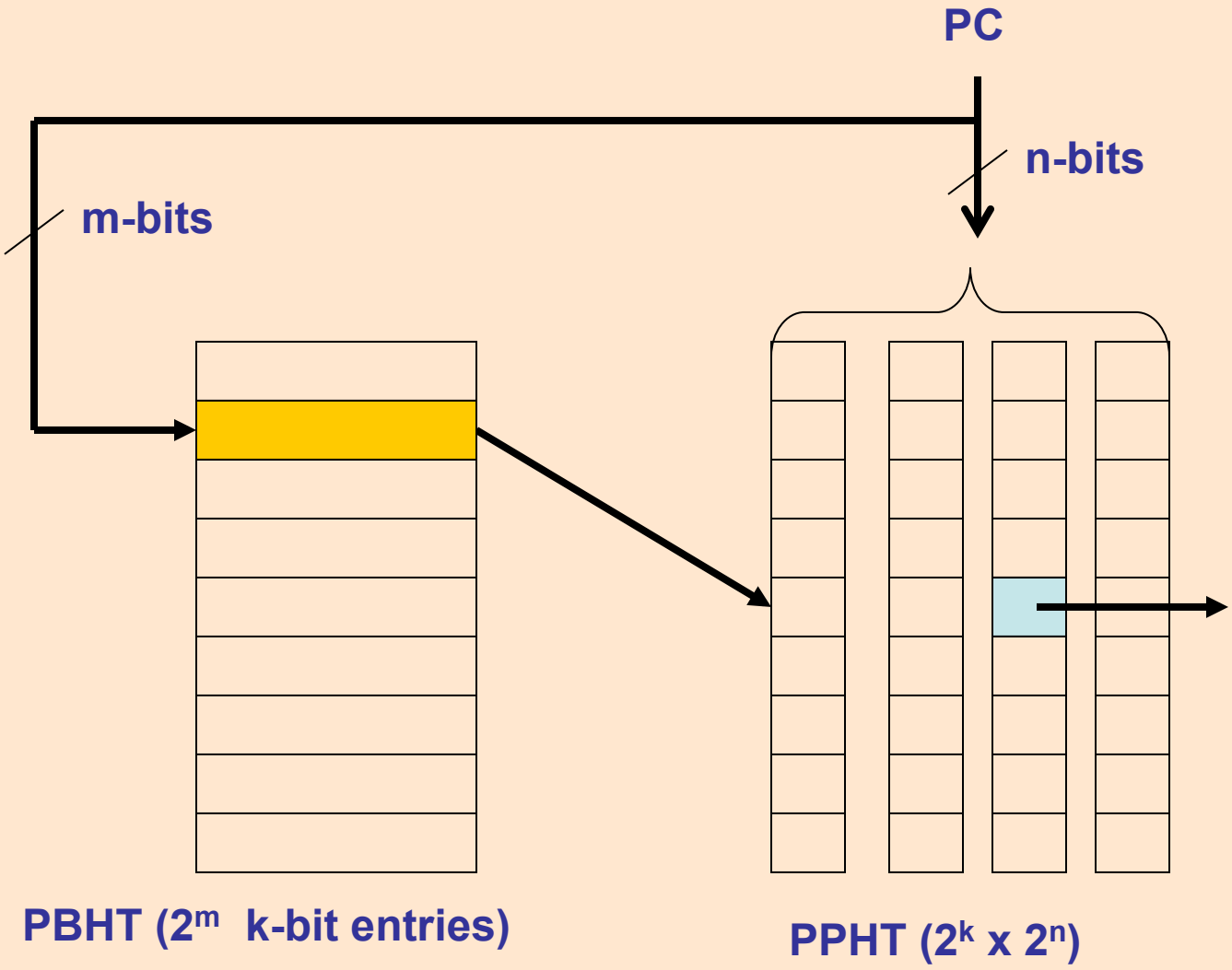

Global branch history reg. and per addr. pattern history table



# PAg Scheme



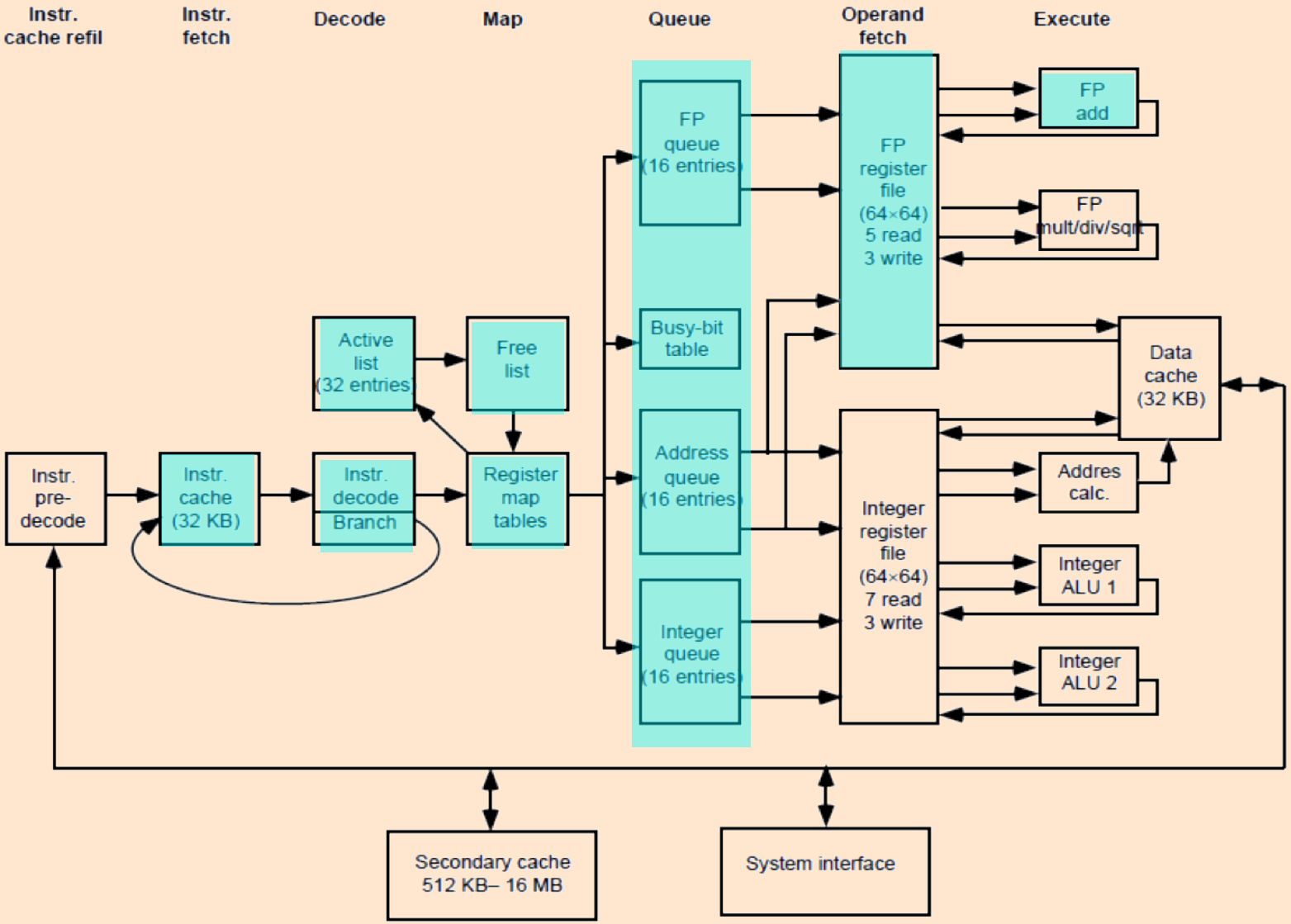
# PAp Scheme



# Branch Predictors: Other Remarks


- Hybrid predictors take advantage of multiple predictors.
- 2-Level predictors have longer warm-up time to build history. Frequent context-switching affects building the history.
- Indirect Branches: Branches or jumps whose target addr. varies at runtime.
  - Function return, jumps for implementing case statements, etc.
  - Target address prediction for function returns using a return address stack of size 8 to 16.

# MIPS R10000 Processor



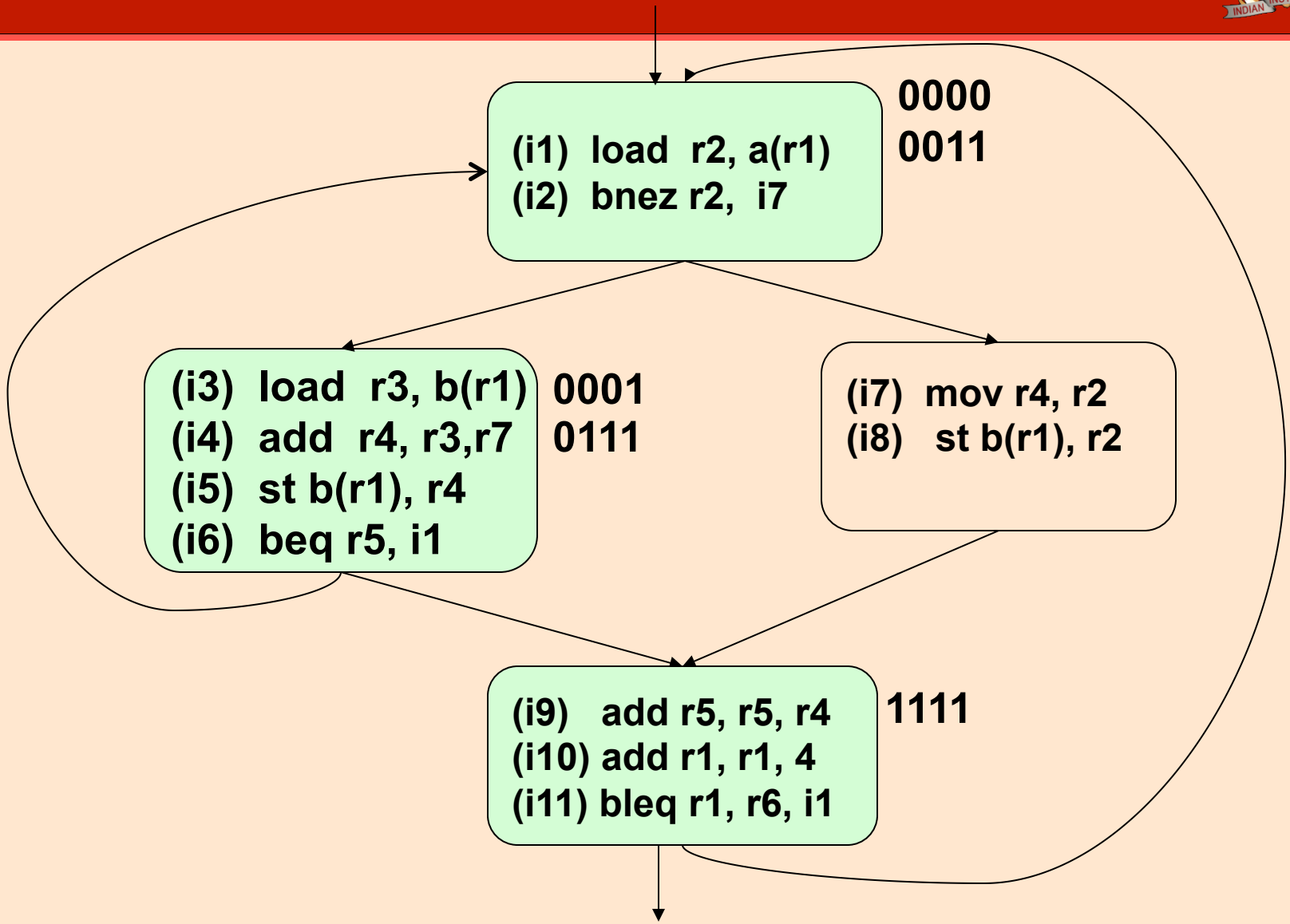
# Instruction Fetch


- Multiple instructions are fetched from I-Cache into Instrn. Buffer.
  - E.G., R10K fetches 4 instrns. (word aligned) within a 16-word I-Cache line.
- Fetch bandwidth might be lower due to
  - Branch instrn. in the fetched instrns.
  - Instrn. fetch starting from the middle of a cache line
  - I-Cache miss

# Branch Prediction


- Branch prediction
  - Predecode logic to identify branch instrns. early
  - Branch prediction (through Branch History Table) using PC value (and previous pattern history)
  - R10K has 512-entry 2-bit BHT to support speculative execution upto 4 pending branches.
- Speculatively execute past multiple pending branches.
- Branch stack to save alternate branch addr. and Int. and FP reg. remap tables.
- 4-bit branch mask with speculative instrn.
- On misprediction, all instrns. with corpg. branch mask bit set are squashed.

# Branch Prediction

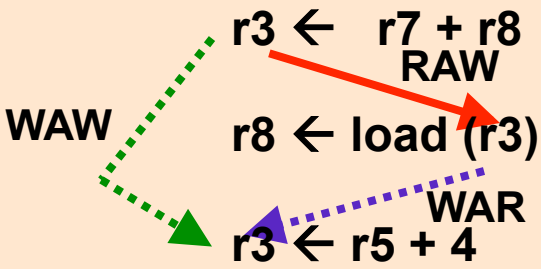


# Instruction Decode


- Decodes instructions from Instrn. Buffer.
- Detection of RAW hazards and elimination of WAR and WAR hazards through ***Dynamic Register Renaming***

# Register Data Dependences


- Program data dependences cause hazards
  - True dependences (RAW)
  - Antidependences (WAR)
  - Output dependences (WAW)




- RAW dependency must for correctness
- WAR and WAW dependencies – false dependencies – can be eliminated : register renaming

# Instruction Decode


- Decodes instructions from Instrn. Buffer.
- Detection of RAW hazards and elimination of WAR and WAR hazards through ***Dynamic Register Renaming***
  - WAR and WAW hazards occur due to reuse of registers.
  - Limited ***logical*** registers - e.g., 32 Int. and 32 FP Registers.
  - Available ***Physical*** registers/Storage is higher.
  - Replace the logical destination register with a *free* Physical register; any subsequent instrn. which use the value uses the Physical register/storage location.

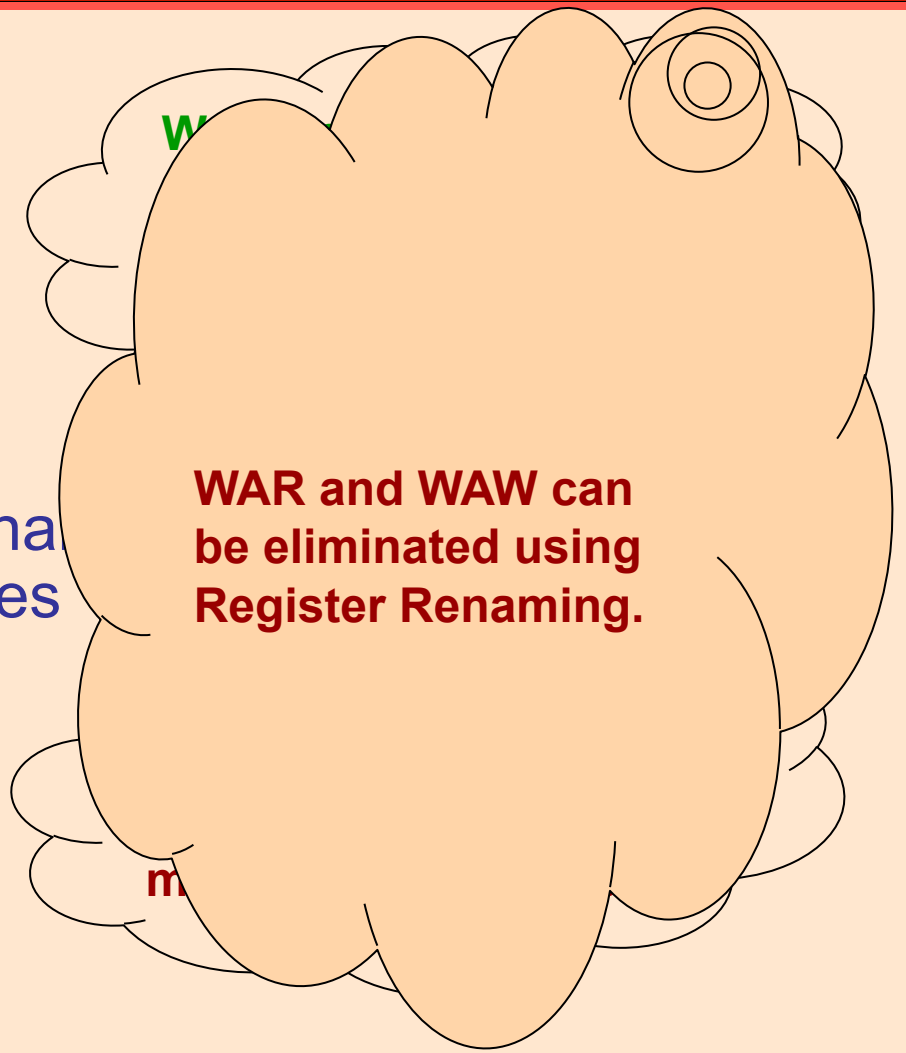
# Register Renaming

- Example:

  mult r2 ← r4, r5
  add r8 ← r2, r4
  sub r4 ← r1, r6
  div r2 ← r19, r20

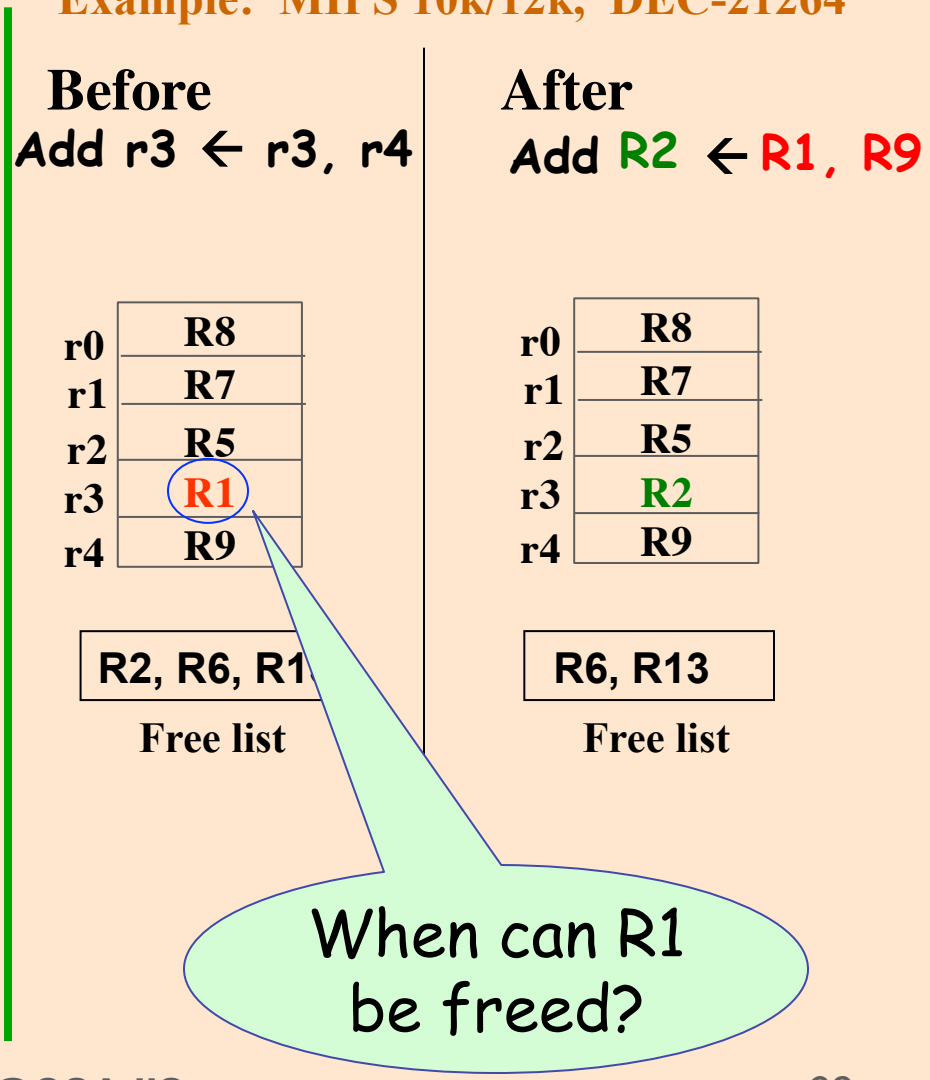
- Using Physical Registers rena
  WAR and WAW dependences
  (name dependences)

  mult R2 ← R4, R5
  add R8 ← R2, R4
  sub R14 ← R1, R6
  div R25 ← R19, R20



# Register Renaming


- Using Physical Registers (size equals twice the no. of logical registers)
  - A register map table which maintains the association betn. logical and physical registers
  - Free list maintains available free regs.
  - Each source operand reg. is replaced by corpg. Physical reg.
  - Destn. reg. (logical) is assigned a free physical register.
  - Physical reg. is freed after all uses of the value -- when a subseq. instrn. with r3 as destn. commits!



# Register Renaming


**Before**

Add r3 ← r3, r4

| | |
|---|---|
| r0 | R8 |
| r1 | R7 |
| r2 | R5 |
| r3 | R1 |
| r4 | R9 |

R2, R6, R1

Free list

**After**

Add R2 ← R1, R9

| | |
|---|---|
| r0 | R8 |
| r1 | R7 |
| r2 | R5 |
| r3 | R2 |
| r4 | R9 |

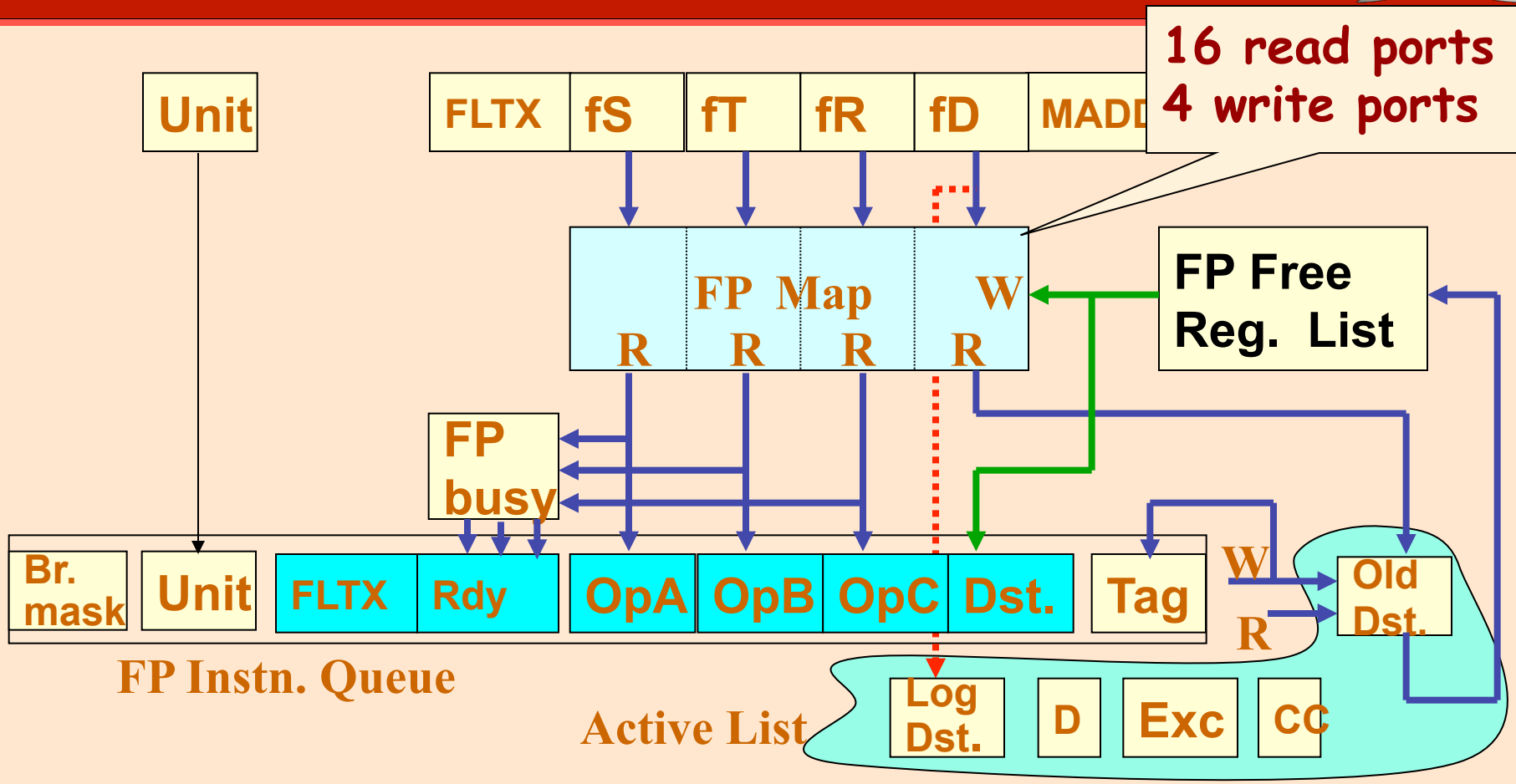
R6, R13

Free list

When can R1 be freed?

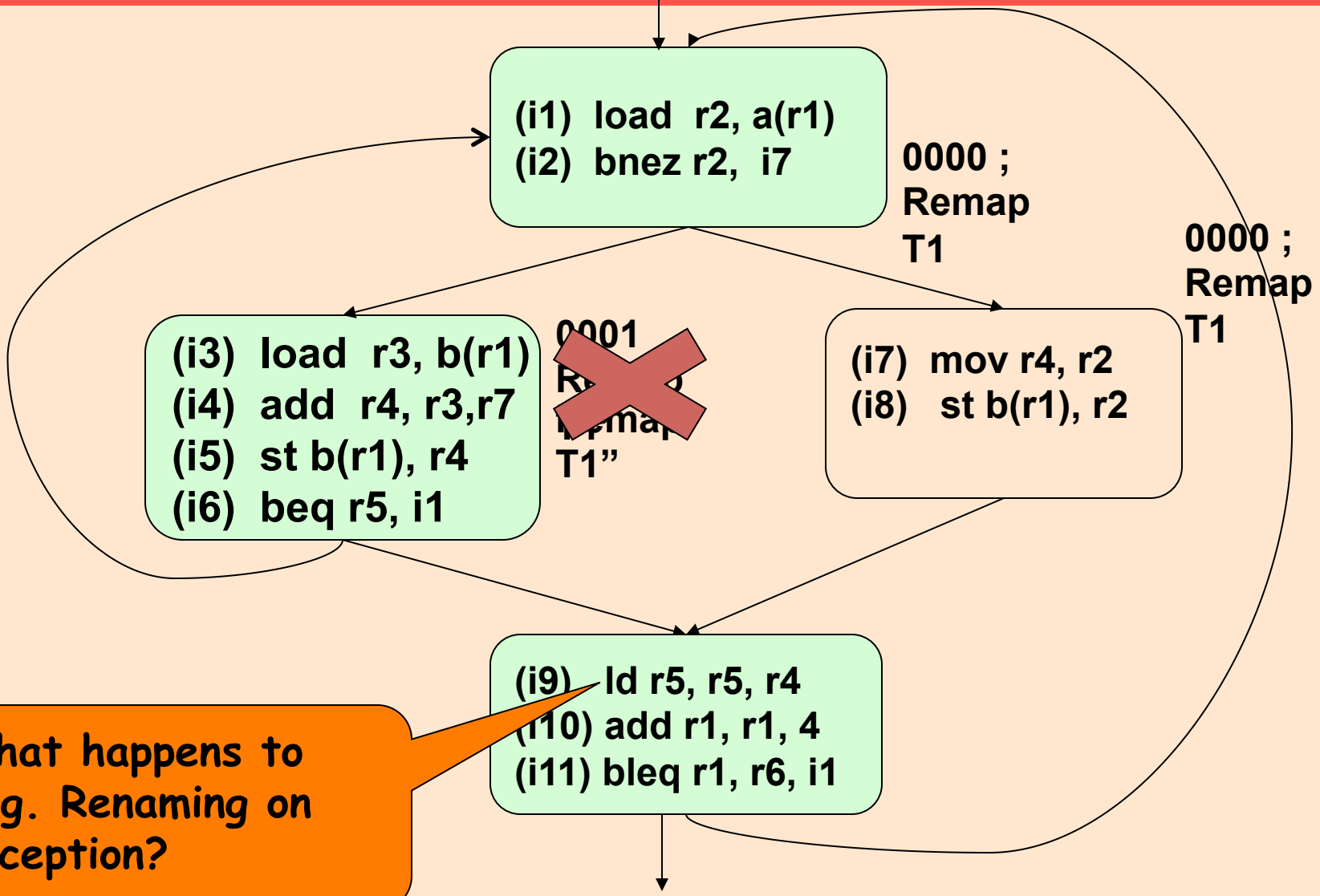
```
i1: add r3 ← r3, r4        i1:  add R2 ← R1, R9
i2: sub  r5 ← r3, r1       i2: sub  R6 ← R2, R7
...                        ...

   ##  instrns.with r3
   as source register;
   but not as destn.
   register

...                        ...

i10: mult r3 ← r8, r9      i10: mult R_ ← R_, R_
i11: ld r10 ←  M(r3)       i11:  ld R_  ← M(R_ )
```

# Register Renaming in R10000



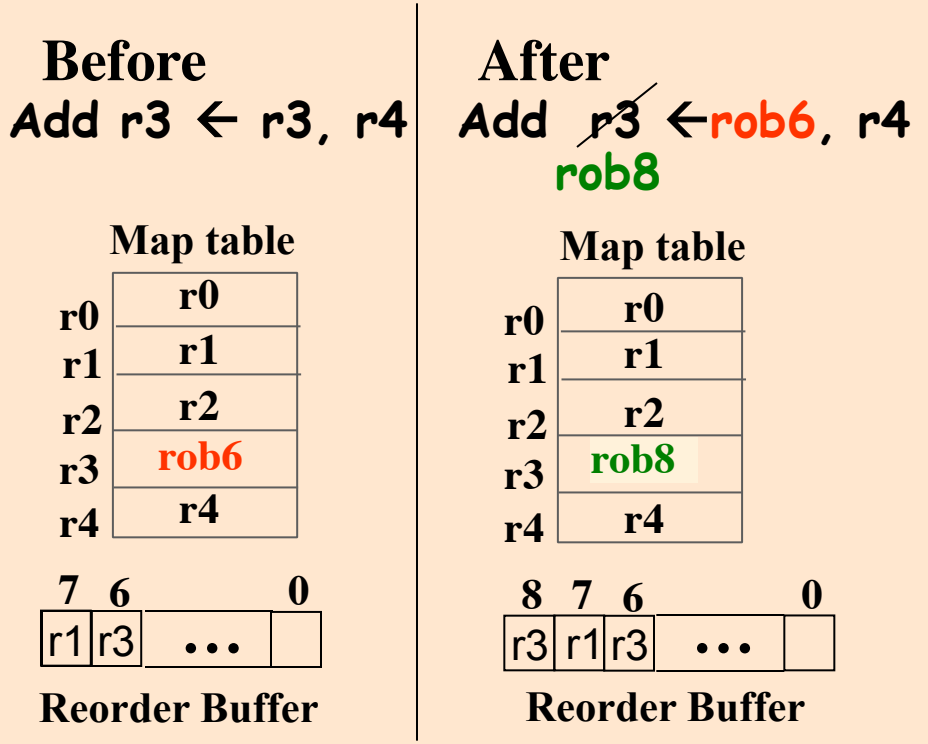

- Renaming complexity proportional to IW
- Renaming delay proportional to IW.

# Recovering Renaming on Branch MisPrediction



# Alternative Register Renaming


- Using *Reorder Buffer* as temporary storage for speculated values.
- No. of Physical Regs. = No. of Logical Regs.
  - Each destn. reg. is assigned entries in the tail of the ROB.
  - Upon instrn. execution, result values are written in ROB.
  - When the instrn. reaches the head of ROB, the value is written in physical (same as logical) reg.
  - Source operands are read from ROB entry or register as indicated by the register map table.




Example: Pentium Pro, HP-PA8000, Power-PC 604, SPARC64

# In-order vs. Out-of-Order


## In-order Issue

load R2 ← M(R3)

RAW

mult R6 ← R4, R2

Stalls

sub R8 ← R6, R1

Stalls

store M(R9) ← R8

Stalls

add R3 ← R3, 4

Stalls

add R9 ← R9, 4

## Out-of-Order Issue

load R2 ← M(R3)

mult R6 ← R4, R2

Stalls

sub R8 ← R6, R1

Stalls

store M(R9) ← R8

add R3 ← R3, 4

add R9 ← R9, 4

proceeds to FU for exec.

Example: MIPS-8000, SPARC, DEC 21064, 21164

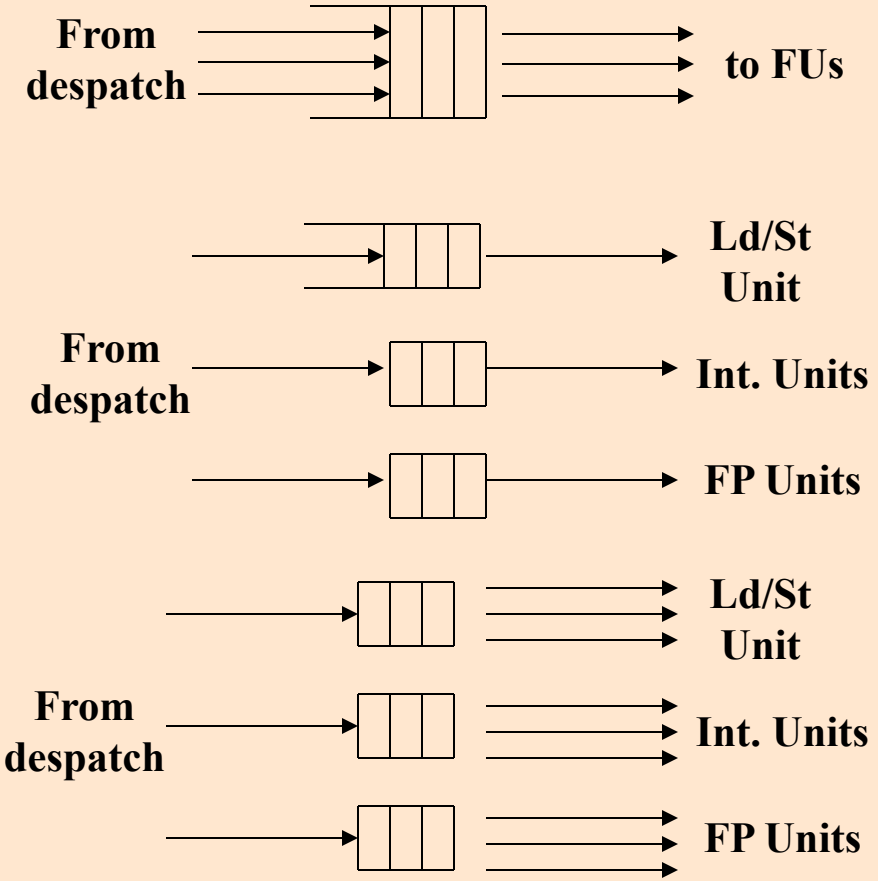
Example: MIPS-10000, PowerPC620, DEC-21264

# Instruction Dispatch & Issue


- Decoded and renamed instns. are dispatched to
  - Instruction Queues
    - Single instrn. queue for all instrn. types (typically in *in-order* issue processors!)
    - Different queues for each instrn. type (e.g., Integer, FP, and Load/Store) -- typically in *out-of-order* issue processors!

  or

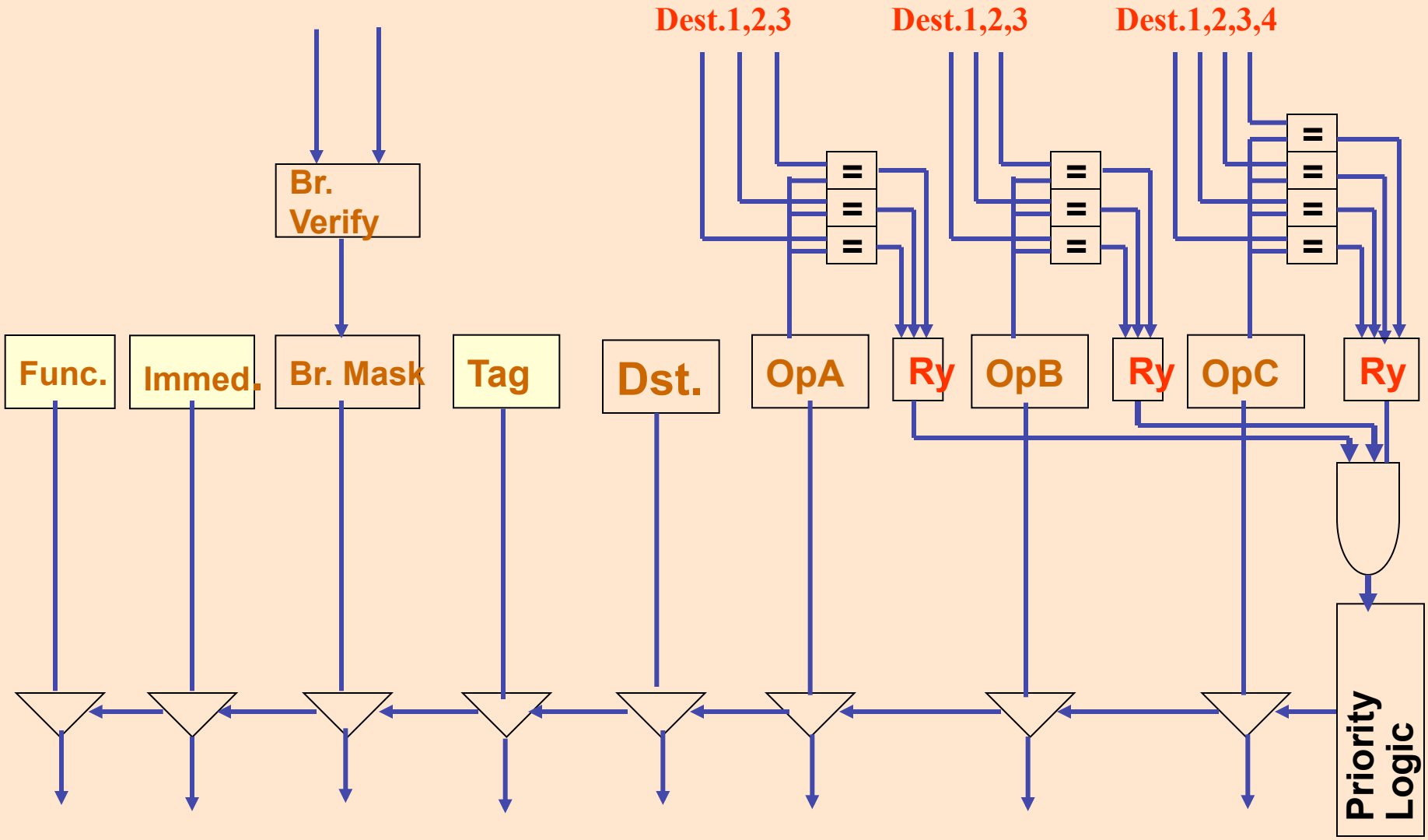
  - Reservation stations for each each FU or FU type (typically for out-of-order issue processors)



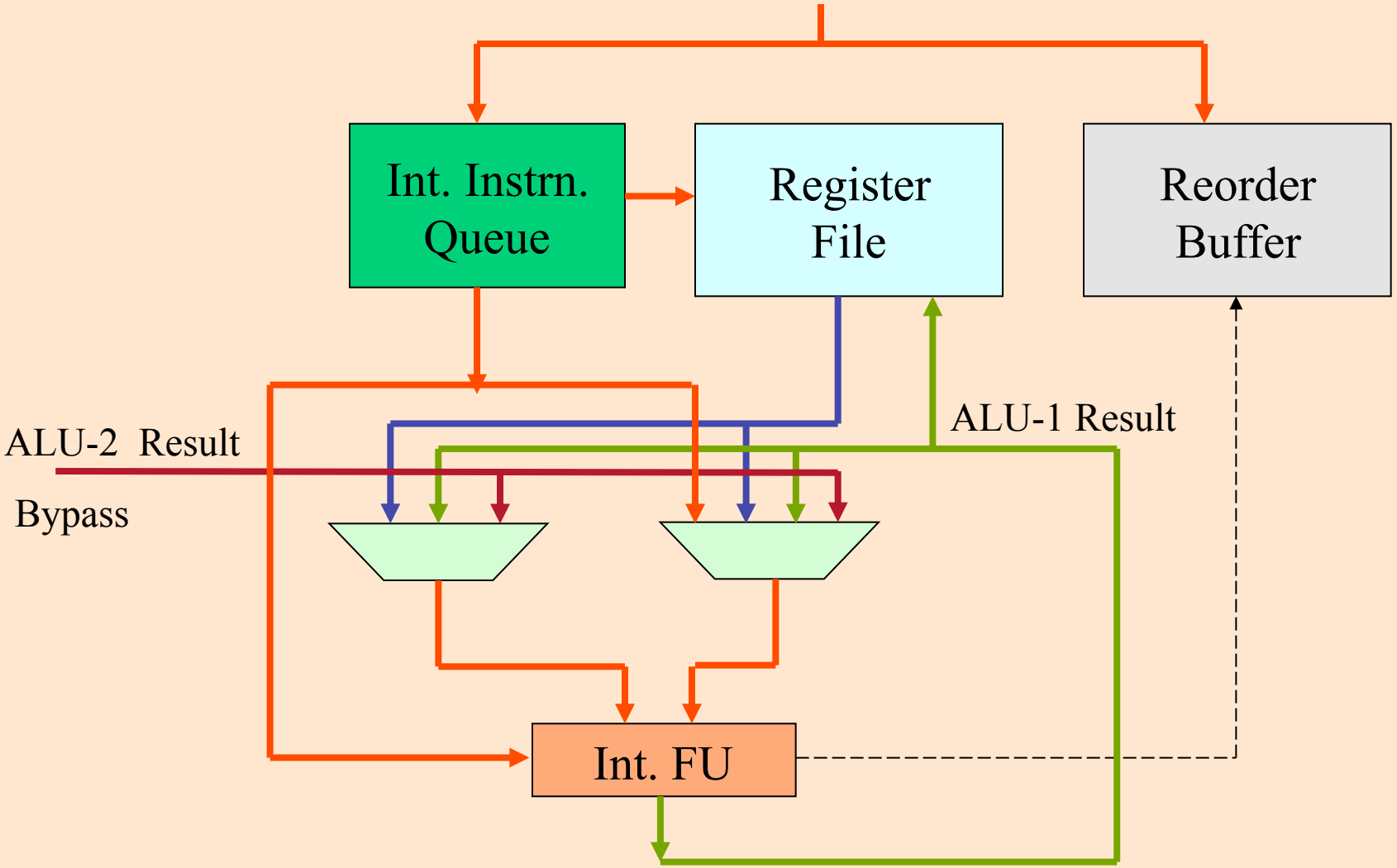
# Instruction Issue: Wakeup & Select


- Instrn. Wakeup: waits in instrn. queue/ reservation station for
  - data dependences to be satisfied (check for ready bits in the Instrn. Queue/Res. Station entry).
  - resource (Functional Unit) to be available.
  - Issue logic complexity proportional to Instrn. Window size and Issue width.
- Instruction Select:
  - Among the ready instrns. a subset is selected (based on ***priority*** ) to be issued to the FUs.
  - Selection logic complexity proportional to WS

# Wakeup & Select in R10000



# Instn. Issue & Execution



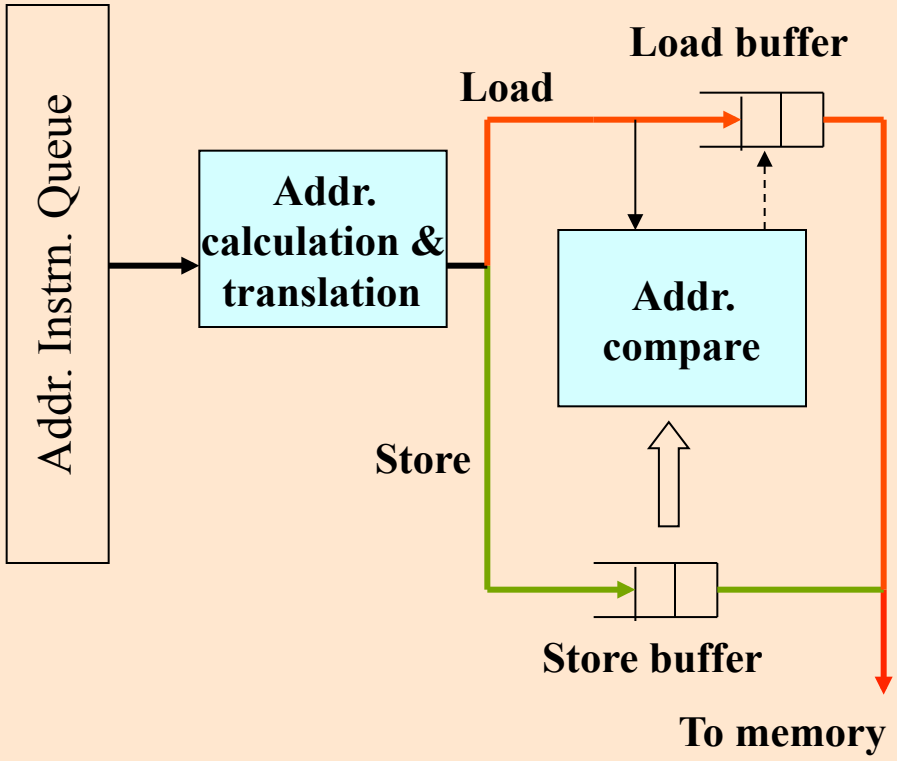
# Load/Store Queue


- Memory ops. involve address *calculation* and *translation*
  - use of TLB for translation.
- Load/Store queue is typically FIFO to ensure load/stores dependences in out-of-order execution.

Store M(r8) ← r3

load r6 ← M (r16)

**Dependent on store or not?**




- Memory renaming (unlike reg. renaming) is difficult.

# Commit Stage

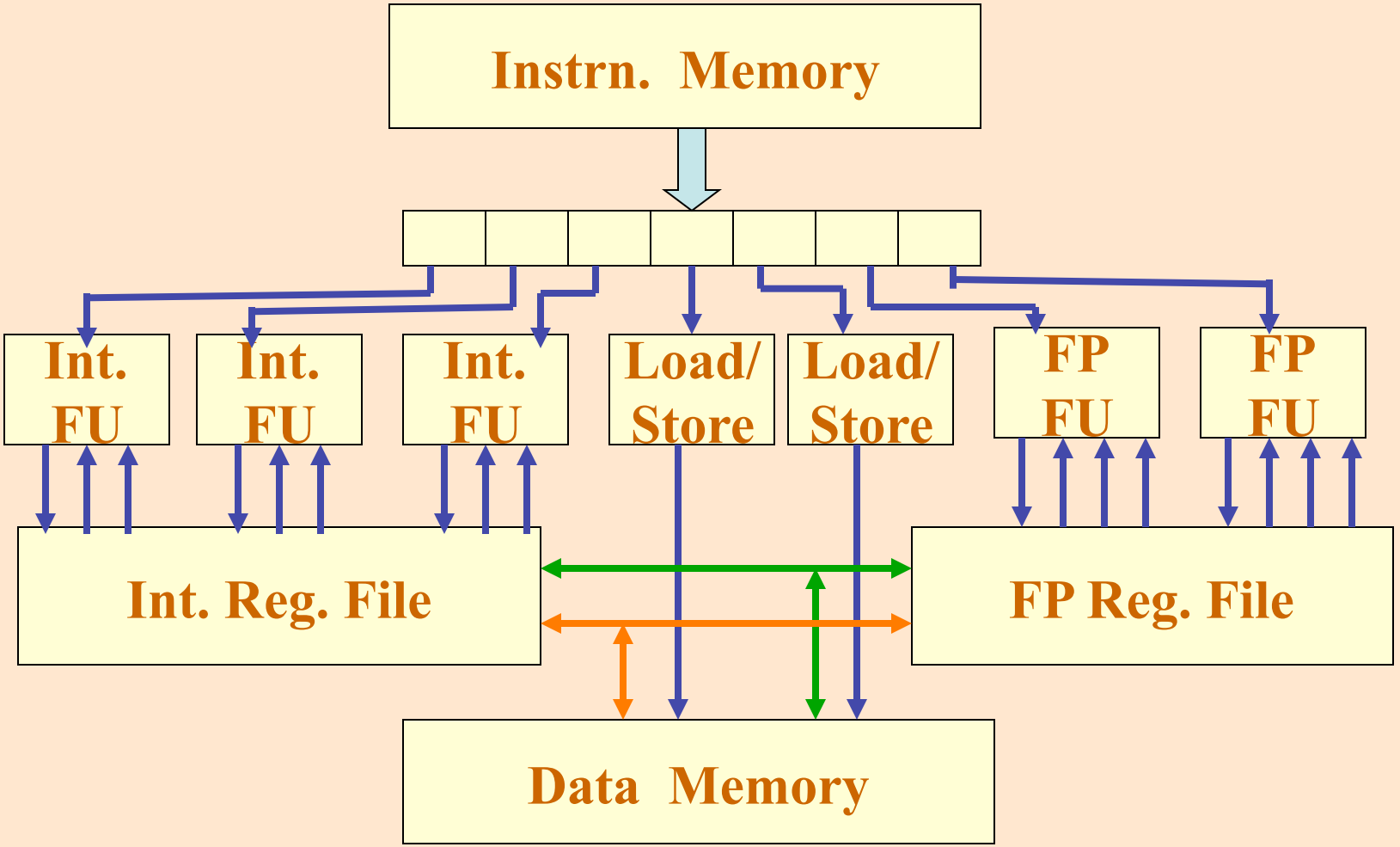

- To maintain appearance of sequential exec. in ***speculative*** and ***out-of-order*** execution.
- To maintain ***precise interrupts.***
- Typically, instructions are committed in program order (from Reorder Buffer)
- In case of branch misprediction, speculative executed instrns. and their effects are annulled by
  - reloading the register map table (saved on branch prediction)
  - Restoring register map table suffices.
  - squashing a part of the Reorder Buffer

# VLIW Architecture: Motivation


- Reduce the hardware needed for dynamic instrn. scheduling.
- Fast, simple decoding and instrn. issue logic that can issue multiple operations in a cycle.
- Compiler to pack to multiple ops. in a single instrn. (inspired by *horizontal micro-programming*)
- No hazard detection (interlocking) -- compiler to ensure correct semantics.

Smart compiler and a Dump (but fast) processor!

# VLIW Processor Organization



# An Example Program


- Consider

```
for (i=0; i < 100; i++)
        a[i] = a[i] + s;
```

- Assembly code

```
L: LD   F0, 0(R1)          ; 1 stall cycle
   ADDD  F4,F2,F0          ; 2 stall cycle
   ST    0(R1), F4
   ADD   R1, R1, #8
   Sub R2,R2, #1
   Bnez R2, L              ; 1 branch stall cycle
```

- VLIW with 1 Mem., 1 Int., 1FP and 1 Branch per instrn. For 5-times unrolled loop.

# An Example VLIW Program


| Mem. Op | FP Op. | Int. Op. | Branch |
|---|---|---|---|
| LD F0, 0(R1) | -- | Sub R2, R2 #1 | -- |
| -- | -- | -- | -- |
| -- | ADDD F4,F2, F0 | Add R1, R1, #8 | -- |
| -- | -- | -- | -- |
| -- | -- | -- | Bnez R2, L |
| ST -8 (R1), F4 | -- | -- | -- |

1 cycle

2 cycle

- 6 Cycles even on the VLIW archiitecture!
- Unroll the loop a few times and schedule?

# An Example VLIW Program


| Mem. Op | FP Op. | Int. Op. | Branch |
|---|---|---|---|
| LD F0, 0(R1) | -- | -- | -- |
| LD F6, 8(R1) | -- | -- | -- |
| LD F10, 16(R1) | ADDD F4,F2, F0 | -- | -- |
| LD F14, 24(R1) | ADDD F8,F2, F6 | -- | -- |
| LD F18, 32(R1) | ADDD F12,F2,F10 | -- | -- |
| ST 0(R1), F4 | ADDD F16,F2,F14 | -- | -- |
| ST 8(R1), F8 | ADDD F20,F2,F18 | Sub R2,R2, #5 | -- |
| ST 16(R1), F1 | -- | Add R1,R1,#40 | -- |
| ST -16(R1),F16 | -- | -- | Bnez R2, L |
| ST -8(R1), F20 | -- | -- | -- |

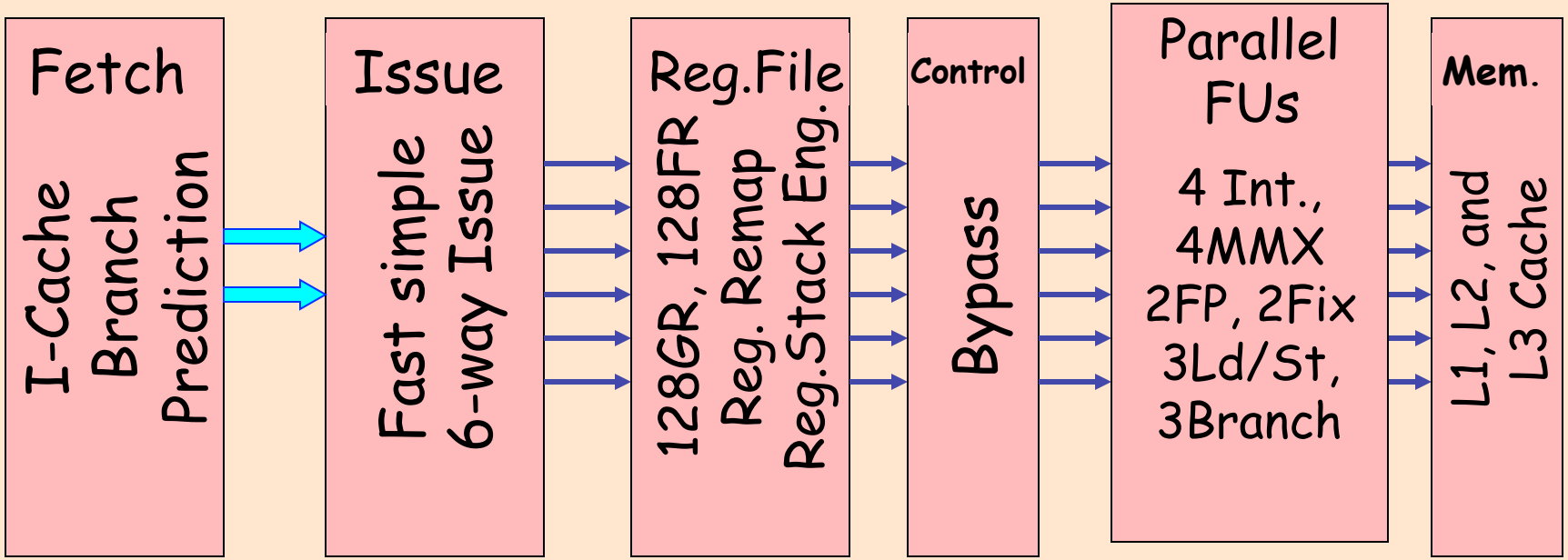


Avg. no. of operations/instrn. = 18/10 = 1.8

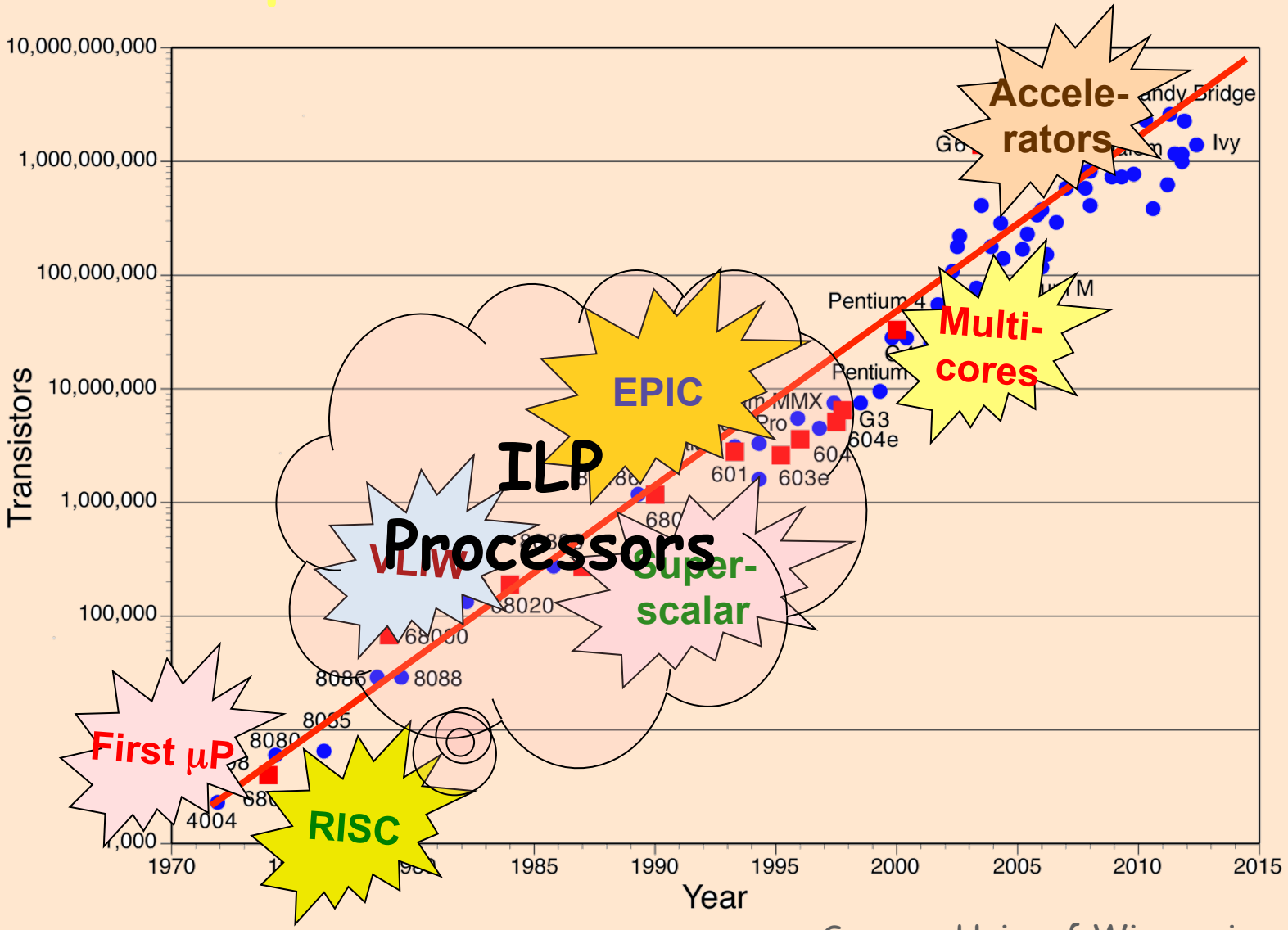
# EPIC - IA64 Architecture


- Explicitly Parallel Instruction Computing:
  - Compiler packs independent instrns. in a 128-bit bundle consisting three 41-bit instrn. and a 5-bit template (for easy decoding).
  - Stop or No-Stop at the end of the bundle.
  - Instrn. independence explicitly conveyed.
  - Instrn. packing based on 12 basic template.
  - Multiple bundles can be issued in a cycle.

# Itanium Microarchitecture



# Processor Architecture Roadmap : Current Millenium




Source: Univ. of Wisconsin

# References



- David Patterson and John L. Hennessy, “Computer Architecture: A Quantitative Approach”, Morgan Kaufmann.
- Smruti Ranjan Sarangi, “Computer Organisation and Architecture” McGraw Hill
- J.E. Smith and G.S. Sohi. Microarchitecture of Superscalar Processors. Proceedings of the IEEE, 83(12), 1609-1624.
- K.C. Yeager. The MIPS R10000 Superscalar Processor. IEEE MICRO, 28-40, April 1996.


# Thank You !!

23 Sep. 2020