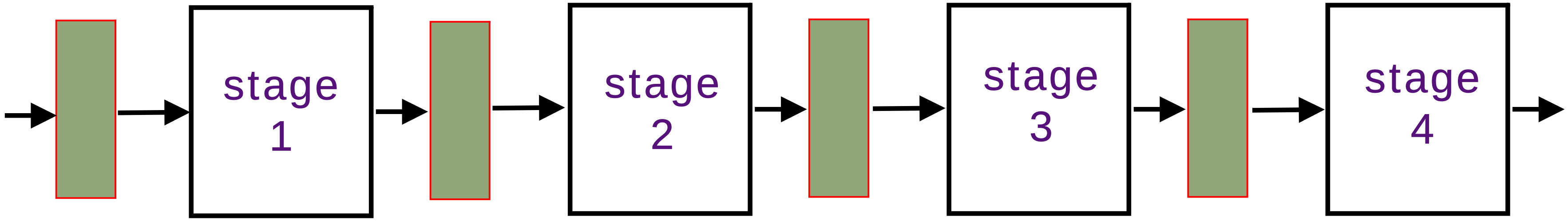
An Ideal Pipeline

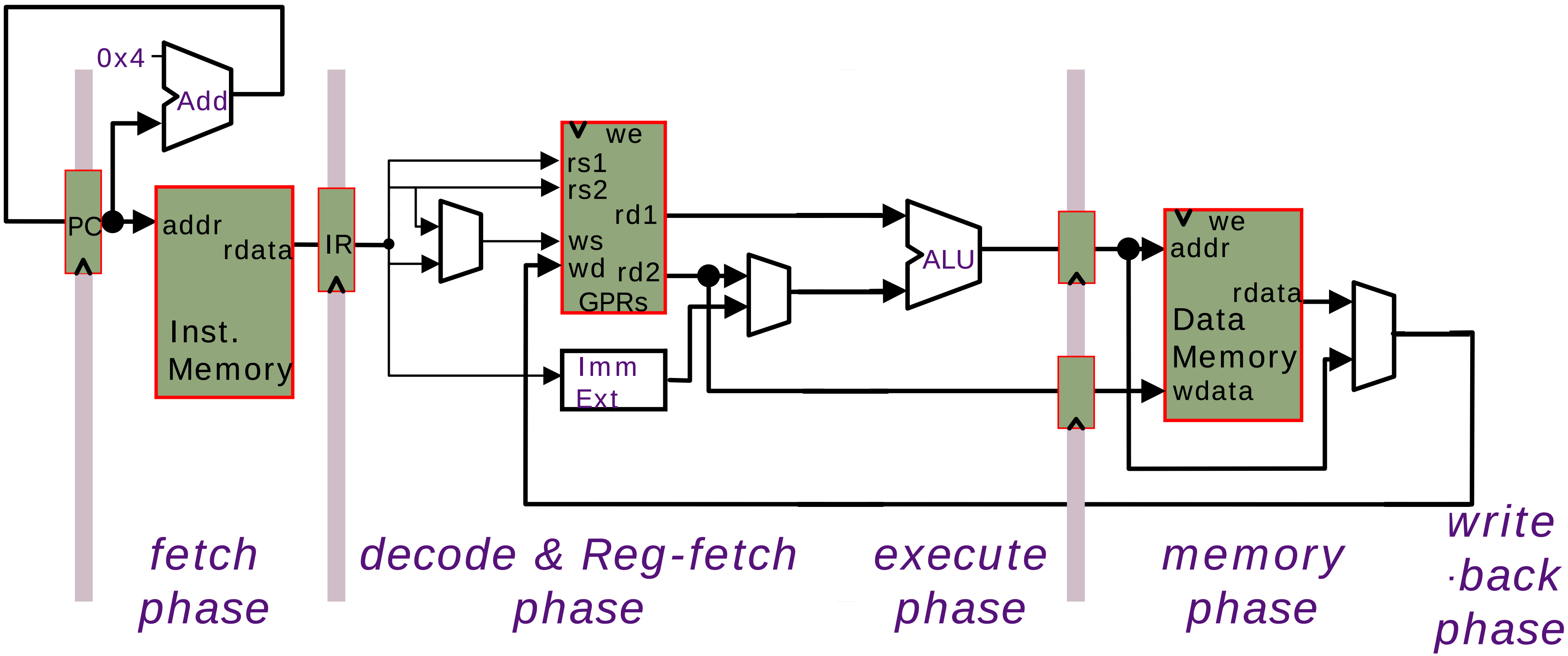


- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- The scheduling of an object entering the pipeline is not affected by the objects in other stages

These conditions generally hold for industrial assembly lines.

But can an instruction pipeline satisfy the last condition?

Alternative Pipelining



$t_C > \max \{t_{IM}, t_{RF}+t_{ALU}, t_{DM}+t_{RW}\} \quad = t_{DM}+ t_{RW}$

$\Rightarrow$ *increase the critical path by 10%*

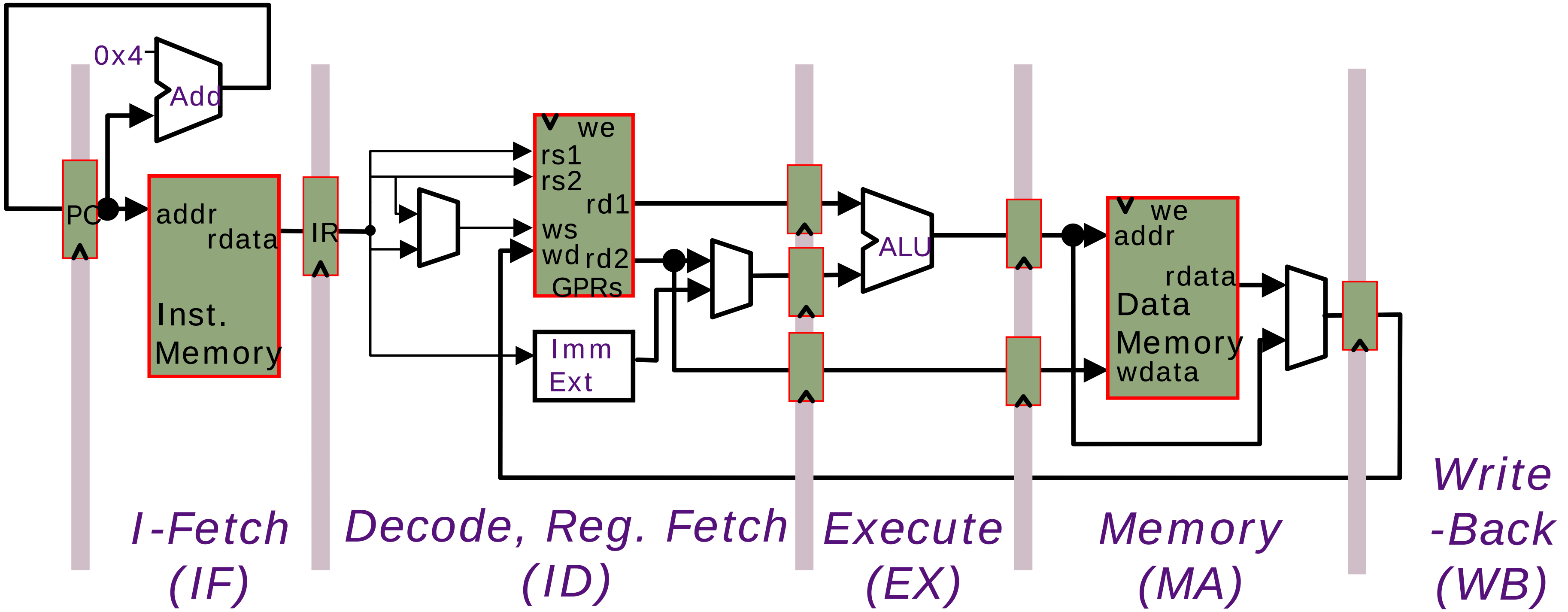
Write-back stage takes much less time than other stages.
Suppose we combined it with the memory phase

Maximum Speedup by Pipelining

Assumptions	Unpipelined t_C	Pipelined t_C	Speedup
1. $t_{IM} = t_{DM} = 10$, $t_{ALU} = 5$, $t_{RF} = t_{RW} = 1$ 4-stage pipeline	27	10	2.7
2. $t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5$ 4-stage pipeline	25	10	2.5
3. $t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5$ 5-stage pipeline	25	5	5.0

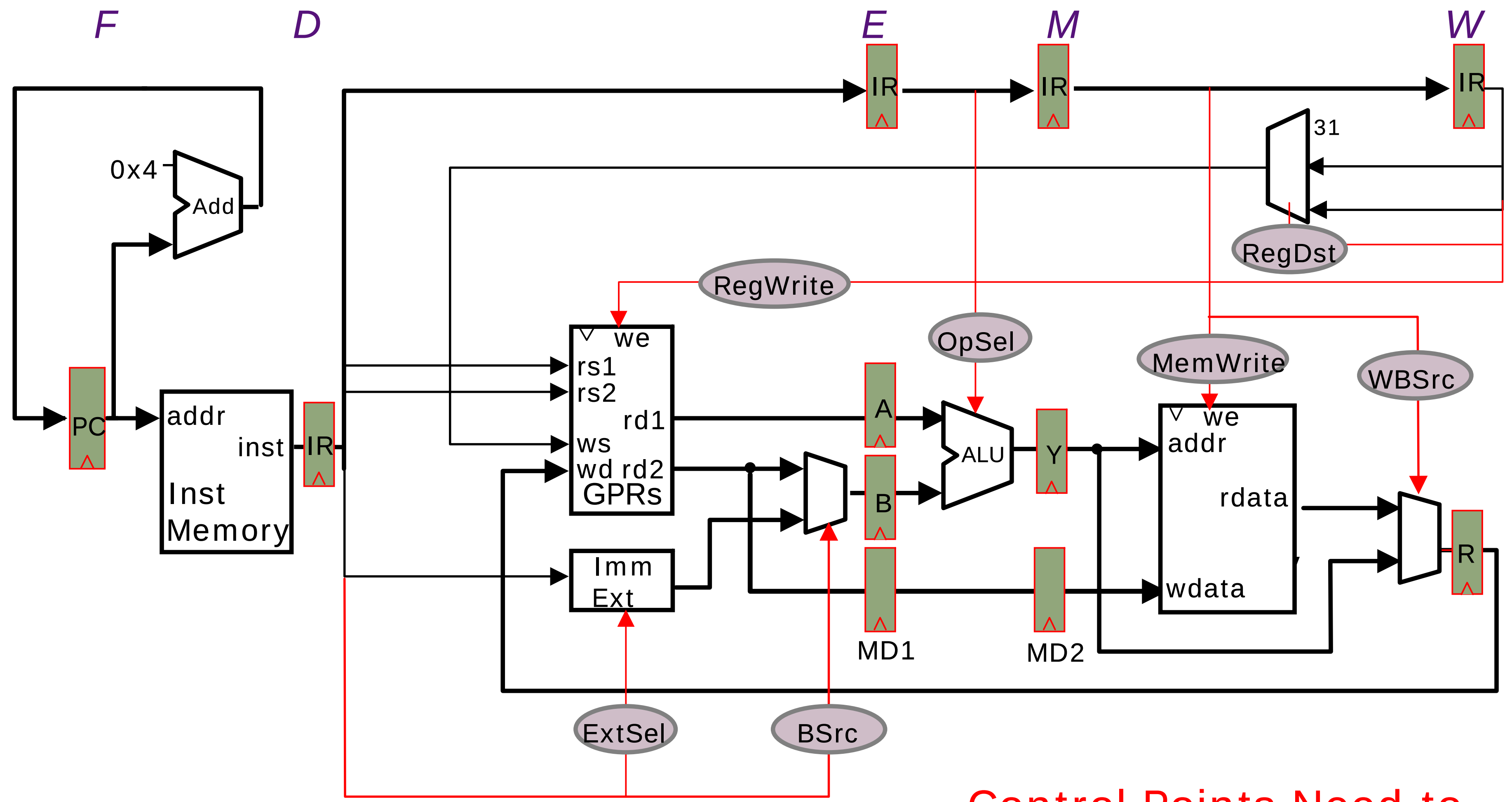
One can achieve higher speedup with more pipeline stages.

5-Stage Pipelined Execution



time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
instruction1	IF_1	ID_1	EX_1	MA_1	WB_1				
instruction2		IF_2	ID_2	EX_2	MA_2	WB_2			
instruction3			IF_3	ID_3	EX_3	MA_3	WB_3		
instruction4				IF_4	ID_4	EX_4	MA_4	WB_4	
instruction5					IF_5	ID_5	EX_5	MA_5	WB_5

Pipelined MIPS Datapath
without jumps



What else is needed?

Control Points Need to Be Connected

How Instructions can Interact with each other in a pipeline

- An instruction in the pipeline may need a resource being used by another instruction in the pipeline → *structural hazard*

- An instruction may depend on something produced by an earlier instruction

  - Dependence may be for a a data calculation
    → data hazard

  - Dependence may be for calculating the next address
    → control hazard (branches, interrrupts)

Resolving Data Hazards

Strategy 1: *Wait for the result to be available by freezing earlier pipeline stages* → *interlocks*

Strategy 2: *Route data as soon as possible after it is calculated to the earlier pipeline stage* → *bypass*

Strategy 3: *Speculate on the dependence*

Two cases:

Guessed correctly → do nothing

Guessed incorrectly → kill and restart

Instruction to Instruction Dependence

- What do we need to calculate next PC:
  - For Jumps
    - Opcode, offset and PC
  - For Jump Register
    - Opcode and Register value
  - For Conditional Branches
    - Opcode, PC, Register (for condition), and offset
  - For all others
    - Opcode and PC

Floating Point ISA

Interaction between the Floating point datapath and the Integer datapath is determined largely by the ISA

MIPS ISA

- separate register files for FP and Integer instructions

  the only interaction is via a set of move instructions (some ISA's don't even permit this)
- separate load/store for FPR's and GPR's but both use GPR's for address calculation
- separate conditions for branches

  FP branches are defined in terms of condition codes

Floating Point Unit

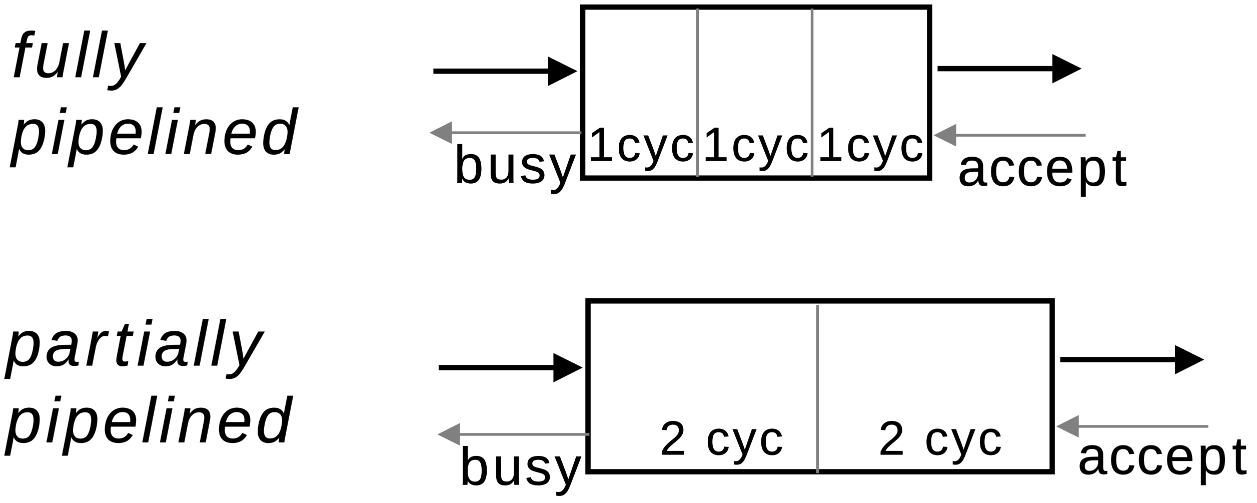
Much more hardware than an integer unit

Single-cycle floating point unit is a bad idea - *why?*

- it is common to have several floating point units
- it is 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 register file needs to have more read and write ports

Function Unit Characteristics



Function units have internal pipeline registers

⇒ operands are latched when an instruction enters a function unit

⇒ inputs to a function unit (e.g., register file) can change during a long latency operation

Realistic Memory Systems

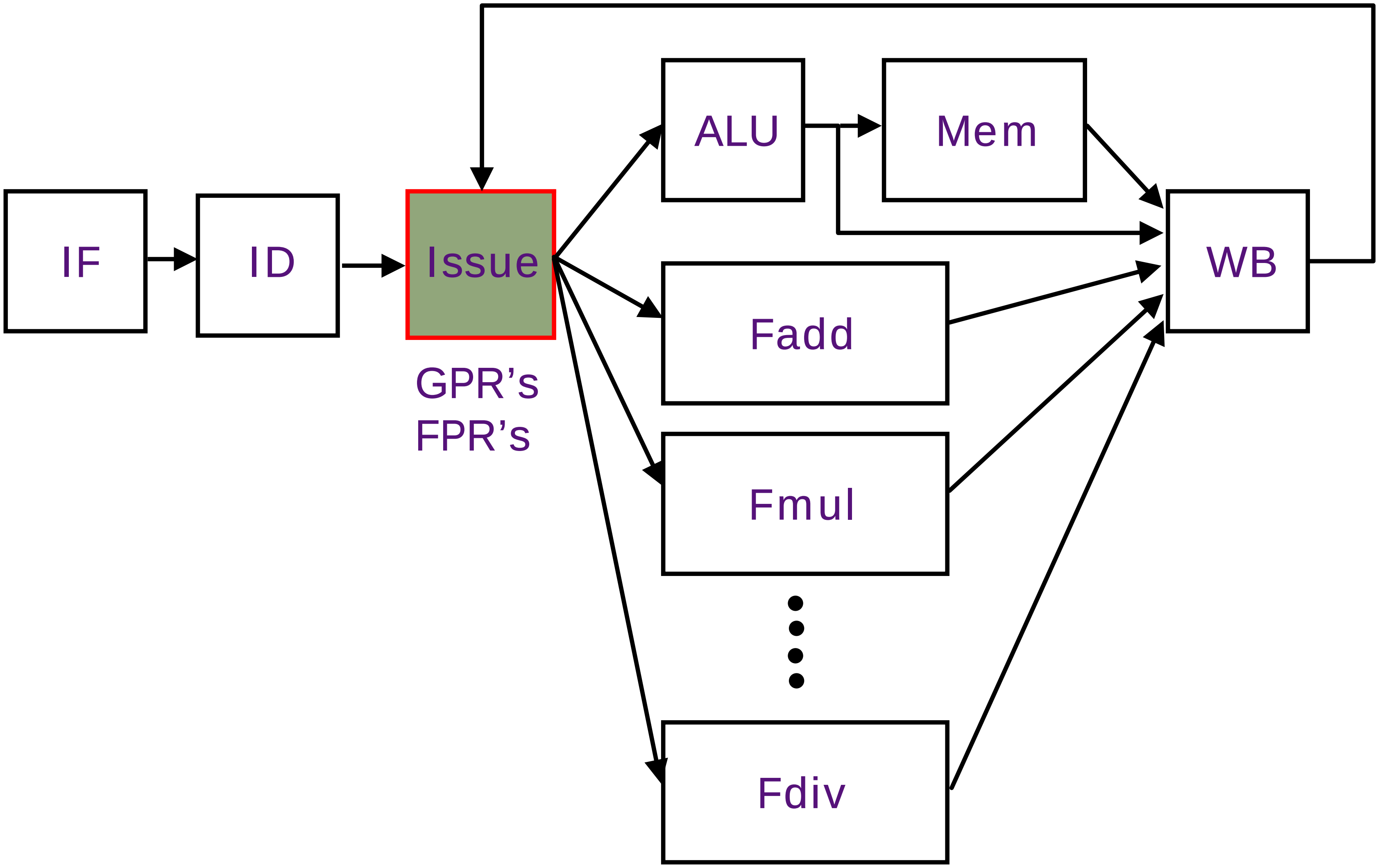
Latency of access to the main memory is usually much greater than one cycle and often unpredictable

Solving this problem is a central issue in computer architecture

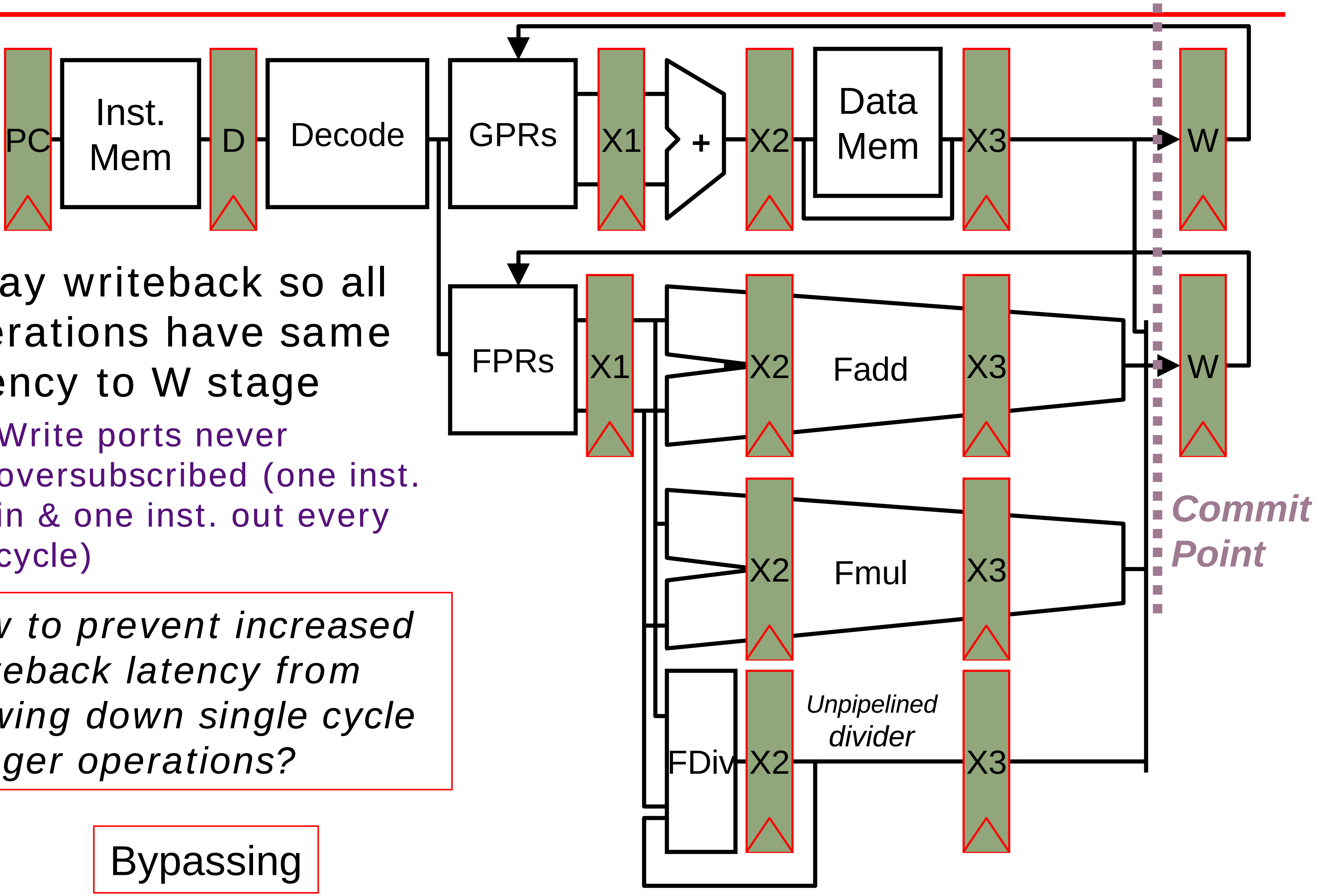
Common approaches to improving memory performance

- separate instruction and data memory ports
  - $\Rightarrow$ *no self-modifying code*
- caches
  - *single cycle except in case of a miss* $\Rightarrow$ *stall*
- interleaved memory
  - *multiple memory accesses* $\Rightarrow$ *bank conflicts*
- split-phase memory operations
  - $\Rightarrow$ *out-of-order responses*

Complex Pipeline Structure



Complex In-Order Pipeline

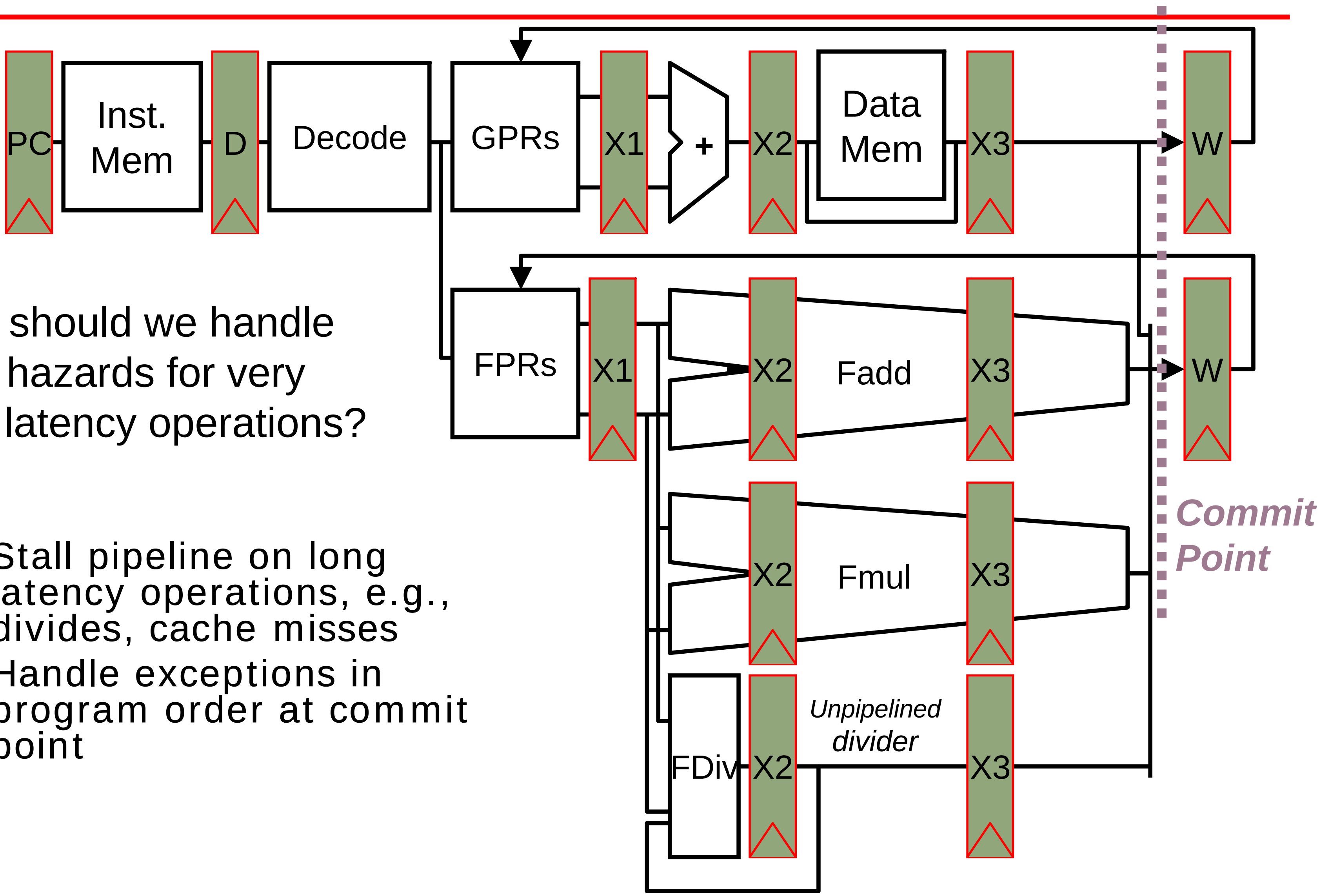


- Delay writeback so all operations have same latency to W stage
  - Write ports never oversubscribed (one inst. in & one inst. out every cycle)

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

Bypassing

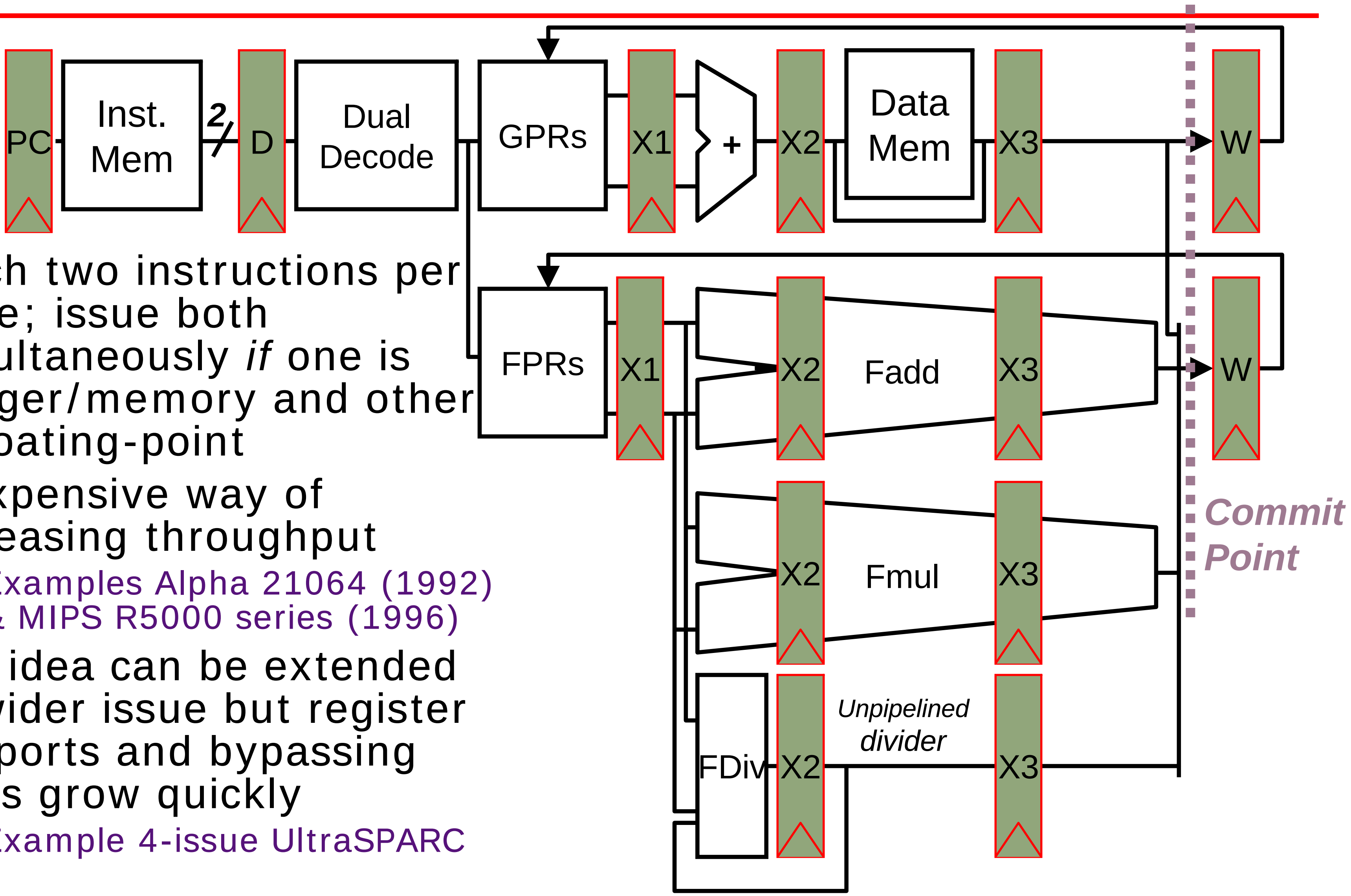
Complex In-Order Pipeline



How should we handle data hazards for very long latency operations?

- Stall pipeline on long latency operations, e.g., divides, cache misses
- Handle exceptions in program order at commit point

Superscalar In-Order Pipeline



- Fetch two instructions per cycle; issue both simultaneously *if* one is integer/memory and other is floating-point
- Inexpensive way of increasing throughput
  - Examples Alpha 21064 (1992) & MIPS R5000 series (1996)
- The idea can be extended to wider issue but register file ports and bypassing costs grow quickly
  - Example 4-issue UltraSPARC

Types of Data Hazards

Consider executing a sequence of

$$r_k \leftarrow (r_i) \ \text{op} \ (r_j)$$

type of instructions

Data-dependence

$$\begin{tikzcd} r_3 \arrow[drr] & \leftarrow & (r_1) & \text{op} & (r_2) \\ r_5 & \leftarrow & (r_3) & \text{op} & (r_4) \end{tikzcd}$$

Read-after-Write (RAW) hazard

Anti-dependence

$$\begin{tikzcd} r_3 & \leftarrow & (r_1) \arrow[dll] & \text{op} & (r_2) \\ r_1 & \leftarrow & (r_4) & \text{op} & (r_5) \end{tikzcd}$$

Write-after-Read (WAR) hazard

Output-dependence

$$\begin{tikzcd} r_3 \arrow[d, bend right] & \leftarrow & (r_1) & \text{op} & (r_2) \\ r_3 & \leftarrow & (r_6) & \text{op} & (r_7) \end{tikzcd}$$

Write-after-Write (WAW) hazard

Detecting Data Hazards

Range and Domain of instruction i

R(i) = Registers (or other storage) modified by instruction i

D(i) = Registers (or other storage) read by instruction i

Suppose instruction j follows instruction i in the program order. Executing instruction j before the effect of instruction i has taken place can cause a

RAW hazard if $R(i) \cap D(j) \neq \emptyset$

WAR hazard if $D(i) \cap R(j) \neq \emptyset$

WAW hazard if $R(i) \cap R(j) \neq \emptyset$

Register vs. Memory Data Dependence

- Data hazards due to register operands can be determined at the decode stage *but*

- Data hazards due to memory operands can be determined only after computing the effective address

store	$M[(r1) + disp1] \leftarrow (r2)$
load	$r3 \leftarrow M[(r4) + disp2]$

Does $(r1 + disp1) = (r4 + disp2)$?

When is it Safe to Issue an Instruction?

- Suppose a data structure keeps track of all the instructions in all the functional units

- The following checks need to be made before the Issue stage can dispatch an instruction
  - Is the required function unit available?
  - Is the input data available? $\Rightarrow$ RAW?
  - Is it safe to write the destination? $\Rightarrow$ WAR? WAW?
  - Is there a structural conflict at the WB stage?

A Data Structure for Correct Issues

Keeps track of the status of Functional Units

Name	*Busy*	*Op*	*Dest*	*Src1*	*Src2*
Int					
Mem					
Add1					
Add2					
Add3					
Mult1					
Mult2					
Div					

The instruction i at the Issue stage consults this table

FU available?	check the busy column
RAW?	search the dest column for i's sources
WAR?	search the source columns for i's destination
WAW?	search the dest column for i's destination

An entry is added to the table if no hazard is detected;
An entry is removed from the table after Write-Back

Simplifying the Data Structure Assuming In-order Issue

- Suppose the instruction is not dispatched by the Issue stage
  - If a RAW hazard exists
  - or if the required FU is busy,
  - and if operands are latched by functional unit on issue

Can the dispatched instruction cause a

WAR hazard ?

NO: Operands read at issue

WAW hazard ?

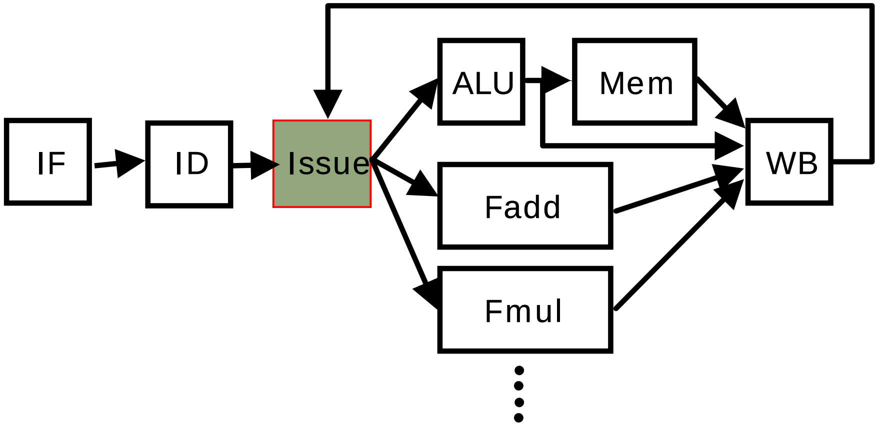
YES: Out-of-order completion

Simplifying the Data Structure ...

- No WAR hazard
  - $\Rightarrow$ no need to keep *src1* and *src2*
- The Issue stage does not dispatch an instruction in case of a WAW hazard
  - $\Rightarrow$ a register name can occur at most once in the *dest* column
- WP[reg#] : a bit-vector to record the registers for which writes are pending
  - *These bits are set to true by the Issue stage and set to false by the WB stage*
  - $\Rightarrow$Each pipeline stage in the FU's must carry the *dest* field and a flag to indicate if it is valid

    "*the (we, ws) pair*"

Out-of-Order Issue



- Issue stage buffer holds multiple instructions waiting to issue.
- Decode adds next instruction to buffer if there is space and the instruction does not cause a WAR or WAW hazard.
- Any instruction in buffer whose RAW hazards are satisfied can be issued *(for now at most one dispatch per cycle)*. On a write back (WB), new instructions may get enabled.

How many Instructions can be in the pipeline

Which features of an ISA limit the number of instructions in the pipeline?

Number of Registers

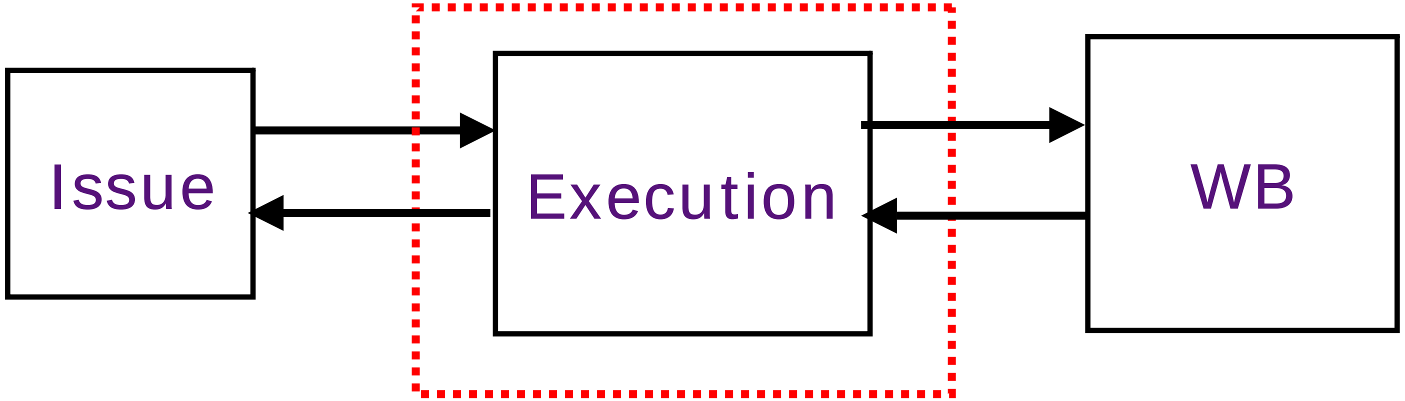
Which features of a program limit the number of instructions in the pipeline?

Control transfers

Out-of-order dispatch by itself does not provide any significant performance improvement !

Little’s Law

Throughput (T) = Number in Flight (N) / Latency (L)



Example:

--- 4 floating point registers
--- 8 cycles per floating point operation

$\Rightarrow$ *½ issues per cycle!*

Overcoming the Lack of Register Names

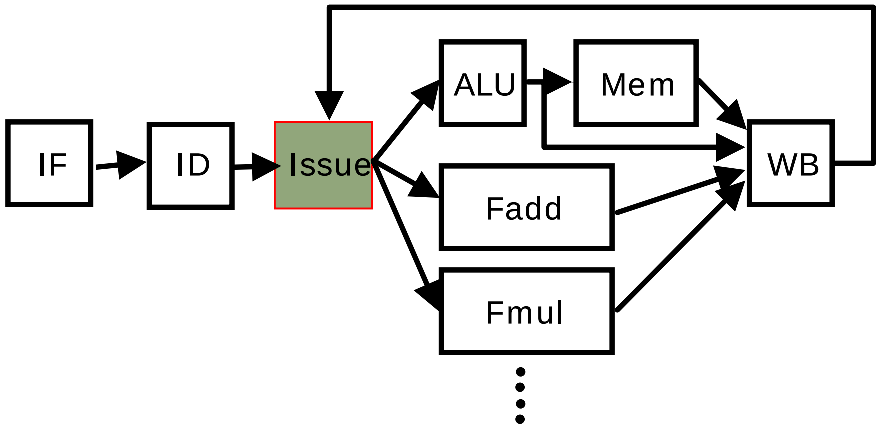
Floating Point pipelines often cannot be kept filled with small number of registers.

IBM 360 had only 4 Floating Point Registers

Can a microarchitecture use more registers than specified by the ISA without loss of ISA compatibility ?

Robert Tomasulo of IBM suggested an ingenious solution in 1967 based on on-the-fly *register renaming*

Register Renaming



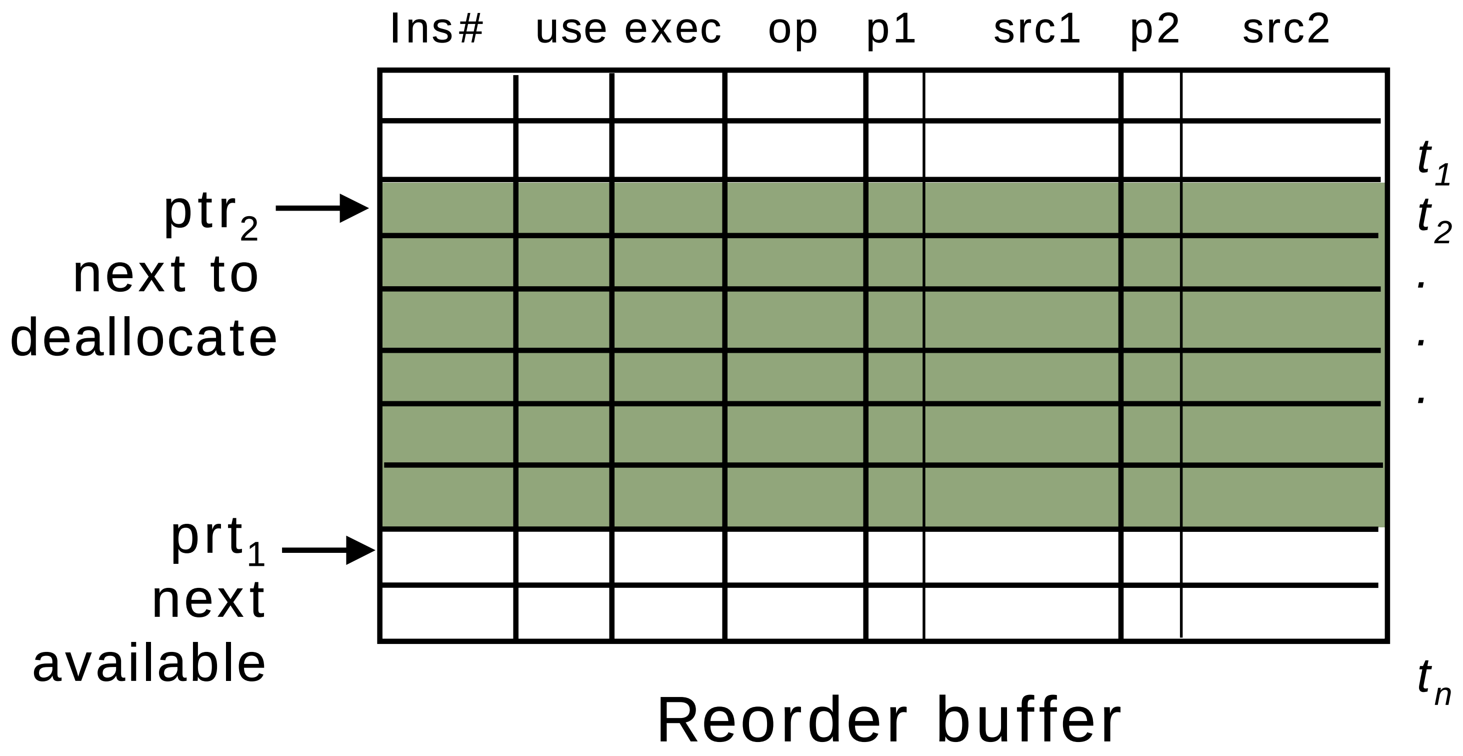
- Decode does register renaming and adds instructions to the issue stage reorder buffer (ROB)

    $\Rightarrow$ renaming makes WAR or WAW hazards impossible

- Any instruction in ROB whose RAW hazards have been satisfied can be dispatched.

    $\Rightarrow$ Out-of-order or dataflow execution

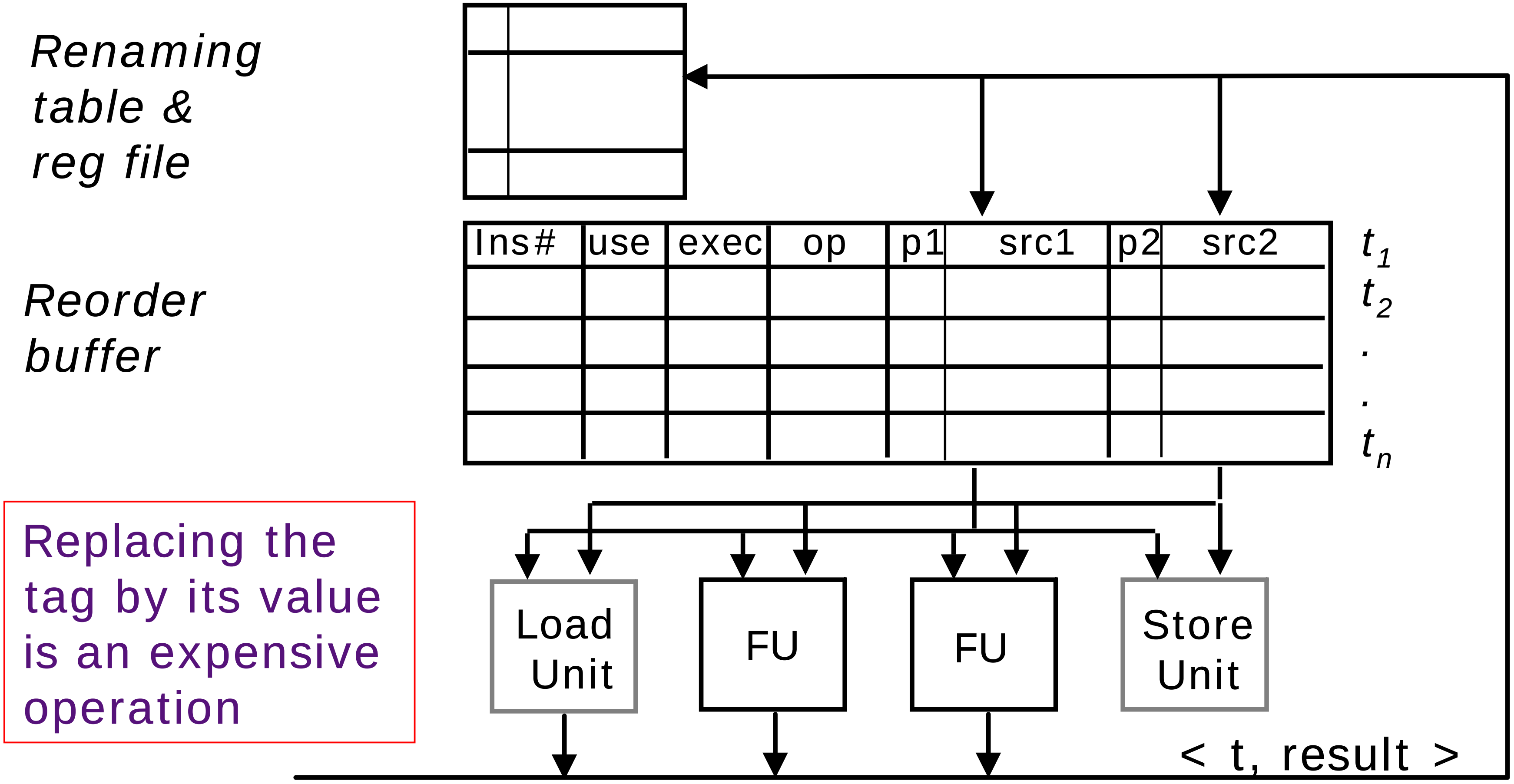
Dataflow execution



Instruction slot is candidate for execution when:

- It holds a valid instruction (“use” bit is set)
- It has not already started execution (“exec” bit is clear)
- Both operands are available (p1 and p2 are set)

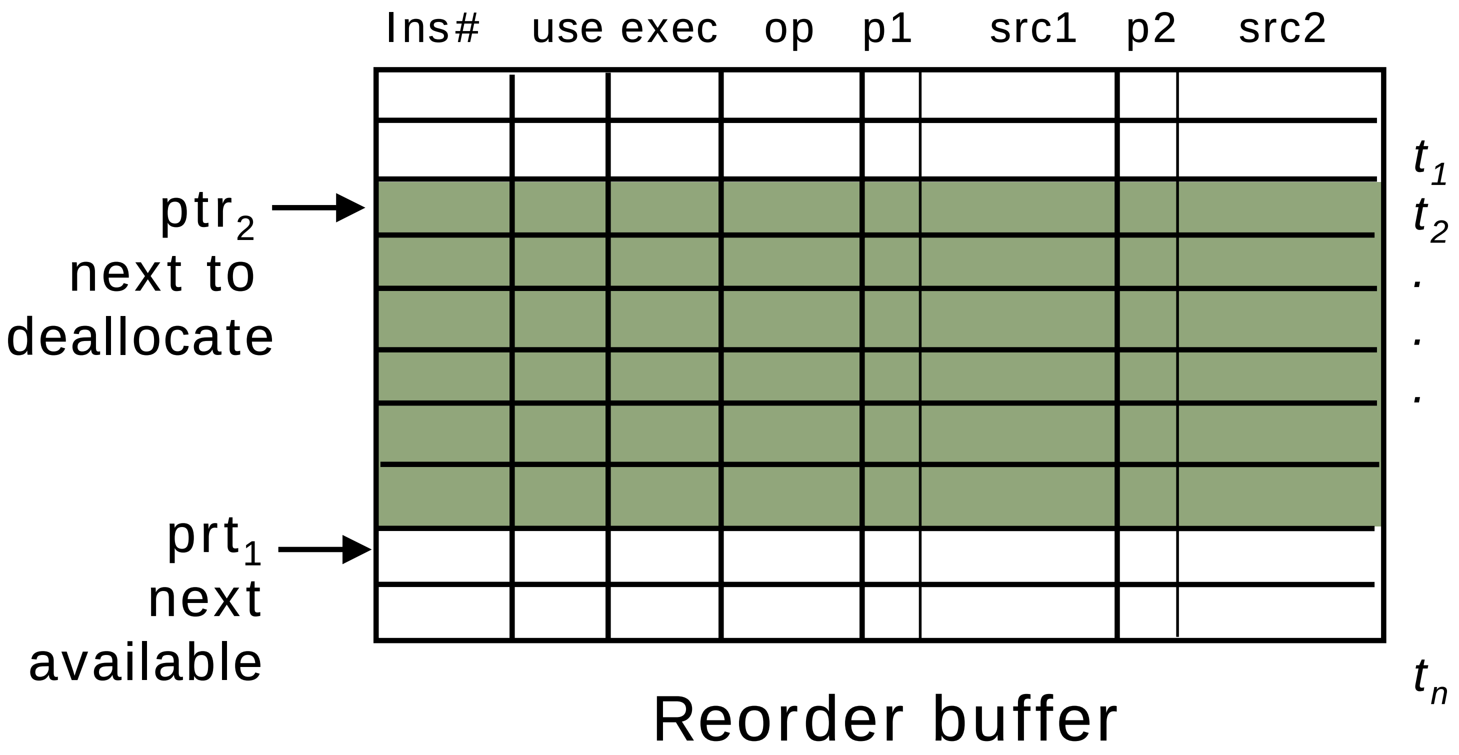
Data-Driven Execution



Replacing the tag by its value is an expensive operation

- Instruction template (i.e., tag t) is allocated by the Decode stage, which also stores the tag in the reg file
- When an instruction completes, its tag is deallocated

Simplifying Allocation/Deallocation



Instruction buffer is managed circularly

- “exec” bit is set when instruction begins execution
- When an instruction completes its “use” bit is marked free
- ptr_2 is incremented only if the “use” bit is marked free

Effectiveness?

Renaming and Out-of-order execution was first implemented in 1969 in IBM 360/91 but did not show up in the subsequent models until mid-Nineties.

Why ?

1. Effective on a very small class of programs
2. Made exceptions imprecise
3. Did not address the memory latency problem which turned out be a much bigger issue than FU latency

One more problem needed to be solved

Control transfers

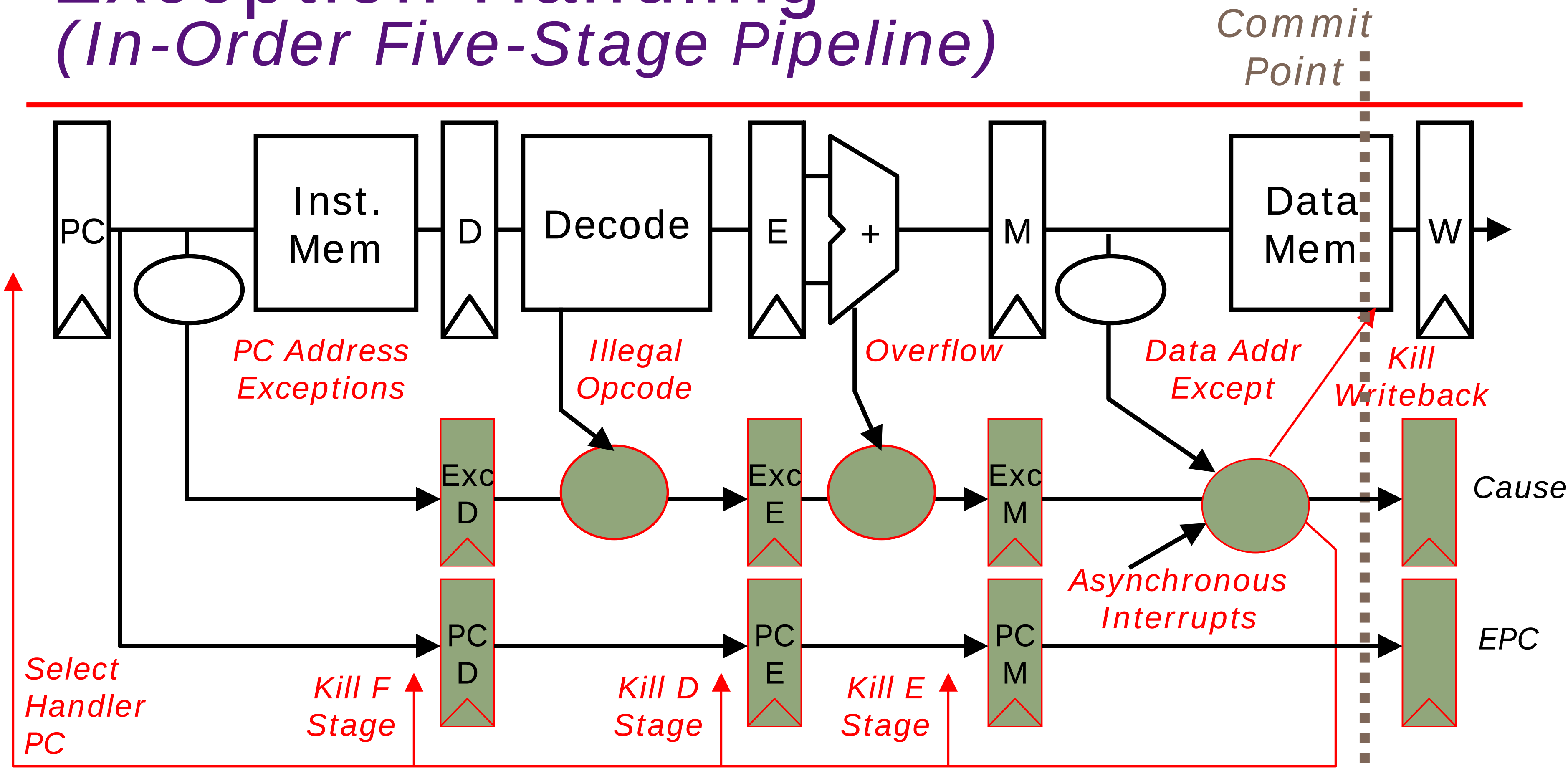
Precise Interrupts

It must appear as if an interrupt is taken between two instructions (say I_i and I_{i+1})

- the effect of all instructions up to and including I_i is totally complete
- no effect of any instruction after I_i has taken place

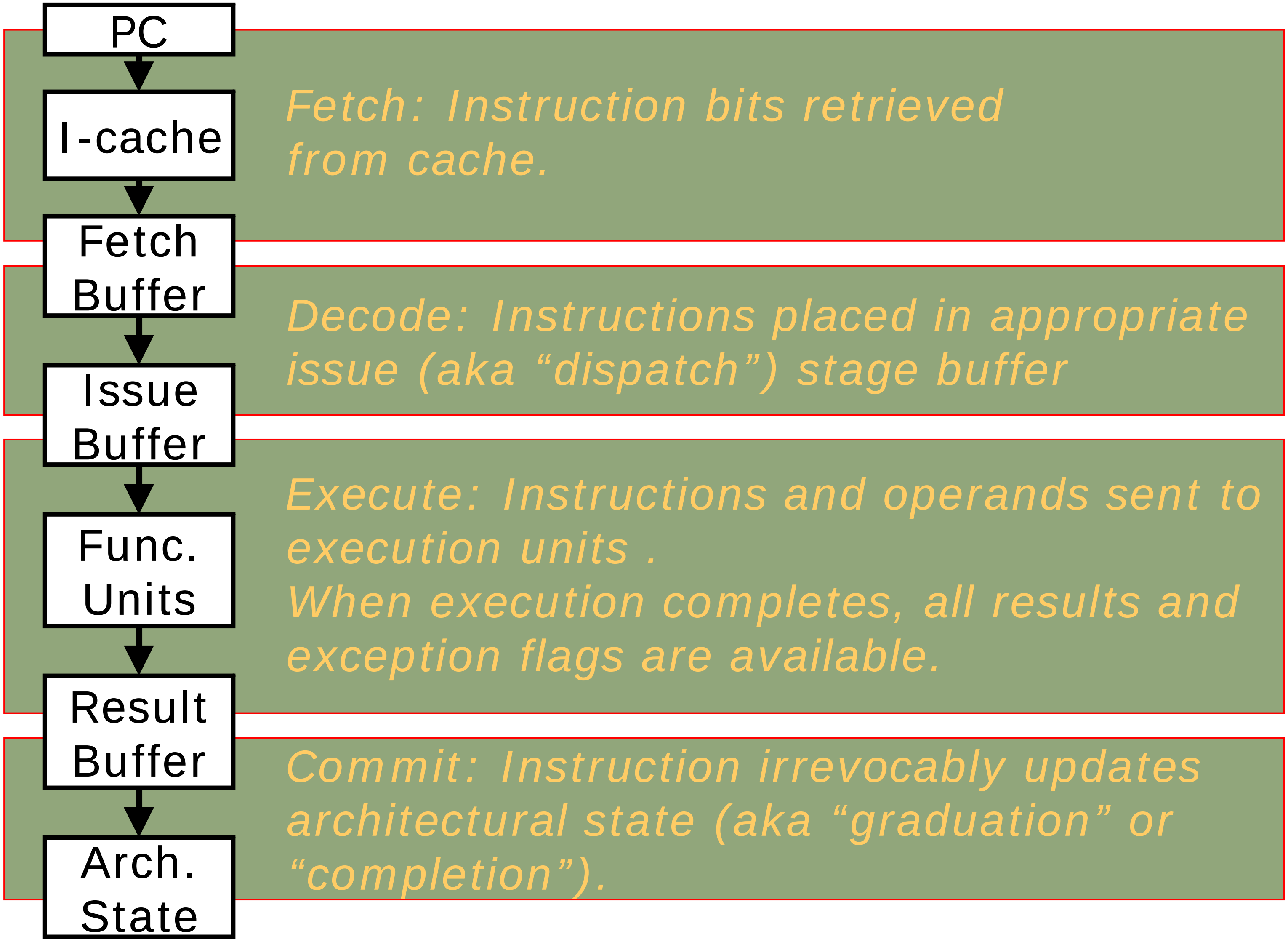
The interrupt handler either aborts the program or restarts it at I_{i+1} .

Exception Handling
(In-Order Five-Stage Pipeline)



- Hold exception flags in pipeline until commit point (M stage)
- Exceptions in earlier pipe stages override later exceptions
- Inject external interrupts at commit point (override others)
- If exception at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage

Phases of Instruction Execution



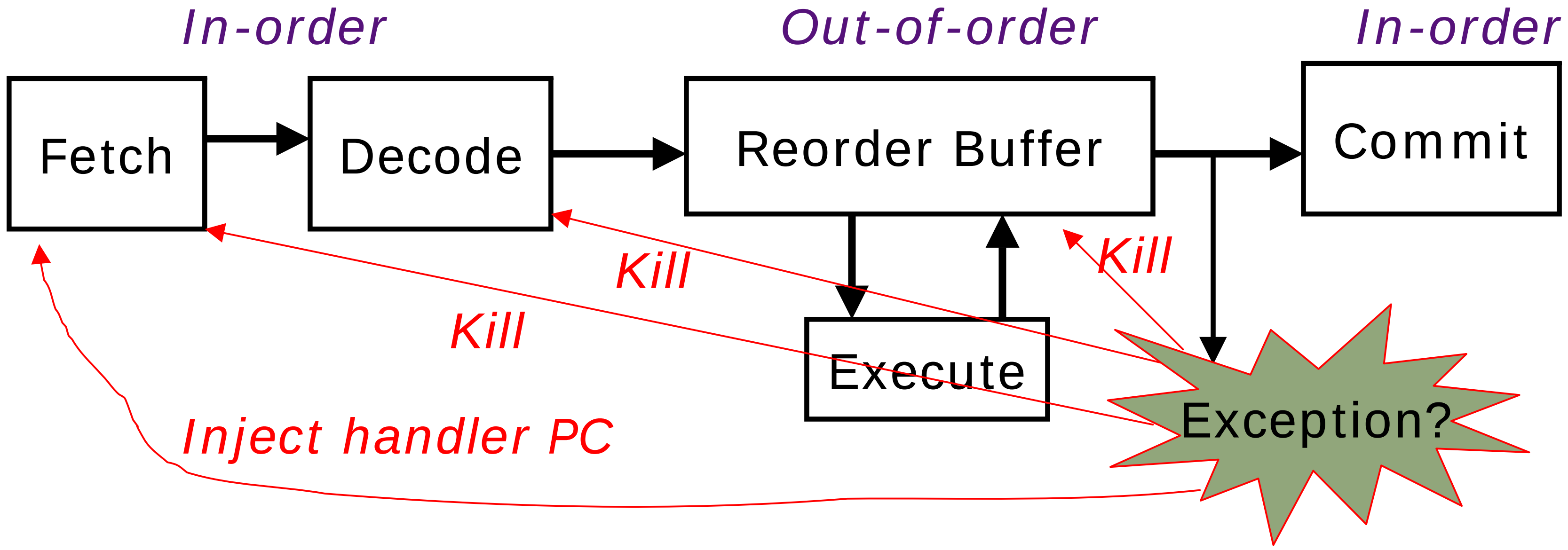
Fetch: Instruction bits retrieved from cache.

Decode: Instructions placed in appropriate issue (aka “dispatch”) stage buffer

Execute: Instructions and operands sent to execution units .
When execution completes, all results and exception flags are available.

Commit: Instruction irrevocably updates architectural state (aka “graduation” or “completion”).

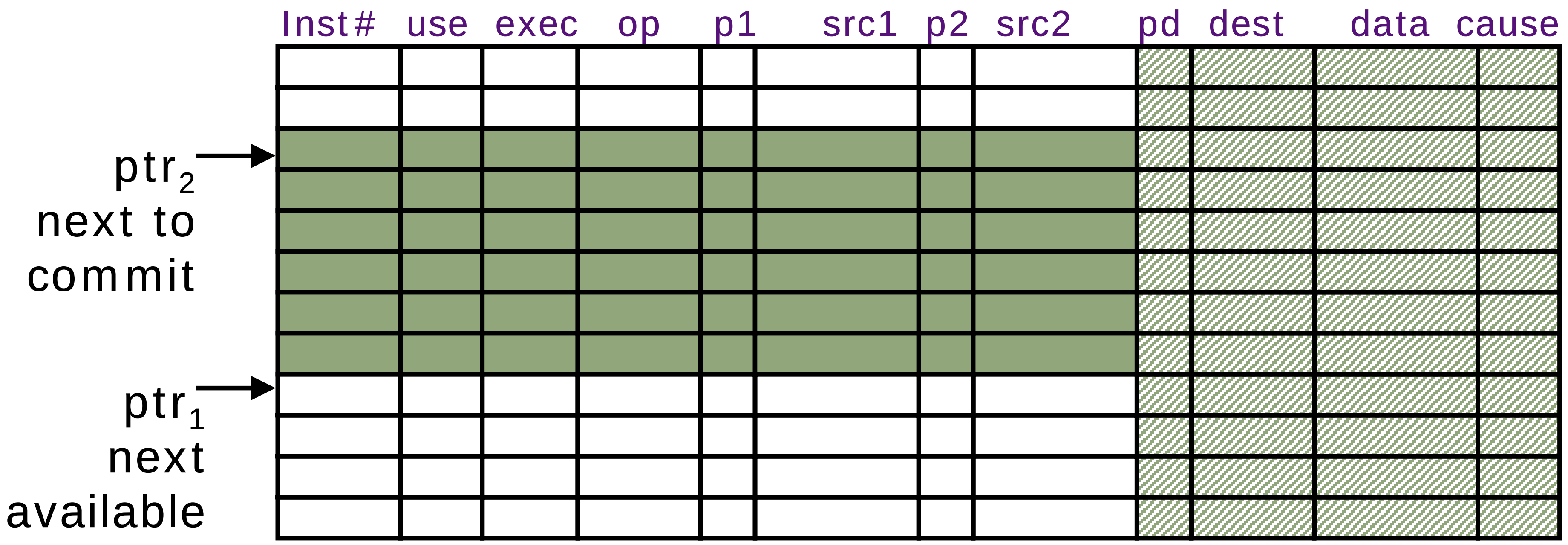
In-Order Commit for Precise Exceptions



- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order ($\Rightarrow$ out-of-order completion)
- *Commit* (write-back to architectural state, i.e., regfile & memory, is in-order

Temporary storage needed to hold results before commit (shadow registers and store buffers)

Extensions for Precise Exceptions

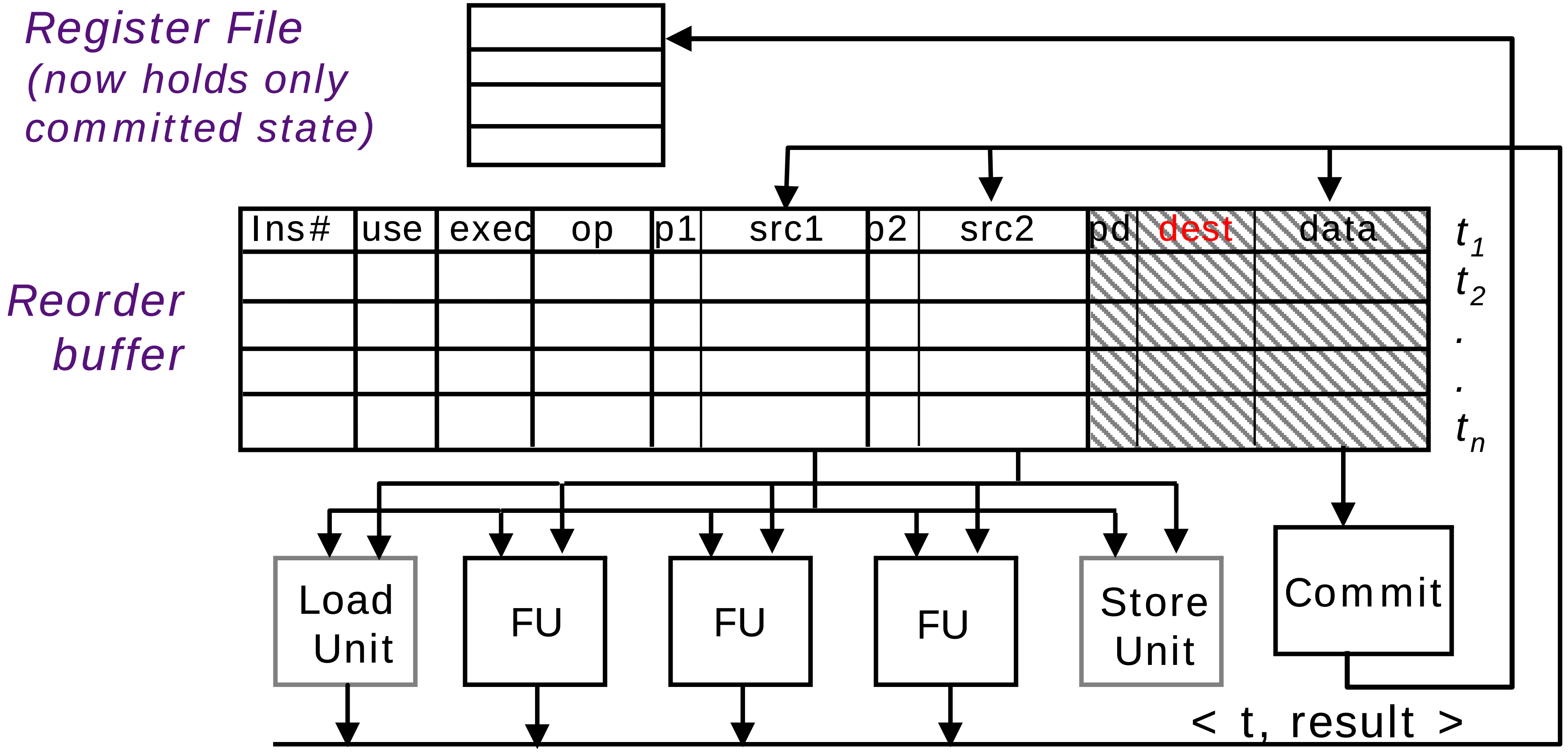


Reorder buffer

- add <pd, dest, data, cause> fields in the instruction template
- commit instructions to reg file and memory in program order ⇒ buffers can be maintained circularly
- on exception, clear reorder buffer by resetting ptr$_1$=ptr$_2$

  (stores must wait for commit before updating memory)

Rollback and Renaming

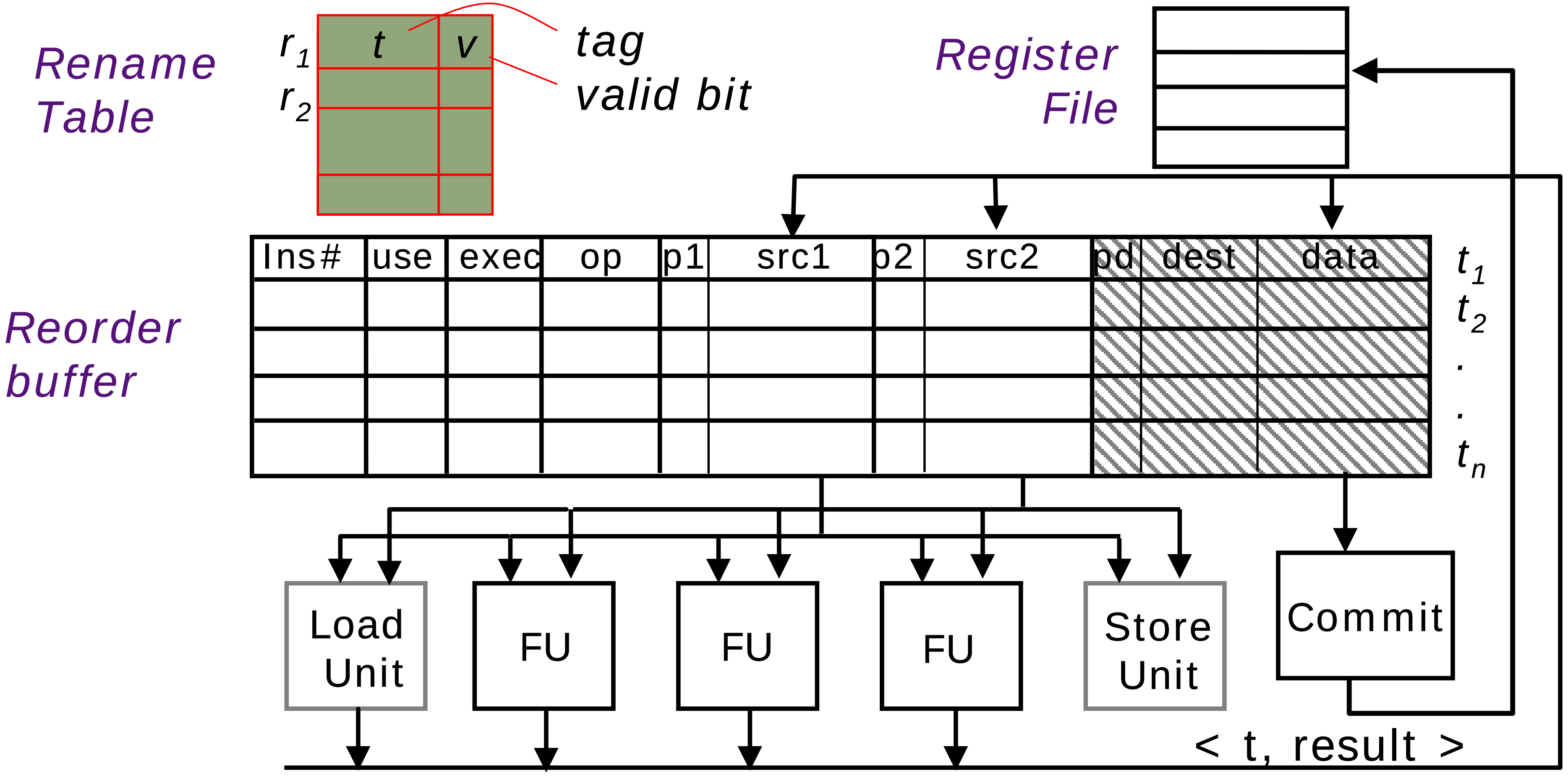


Register file does not contain renaming tags any more.

How does the decode stage find the tag of a source register?

Search the "dest" field in the reorder buffer

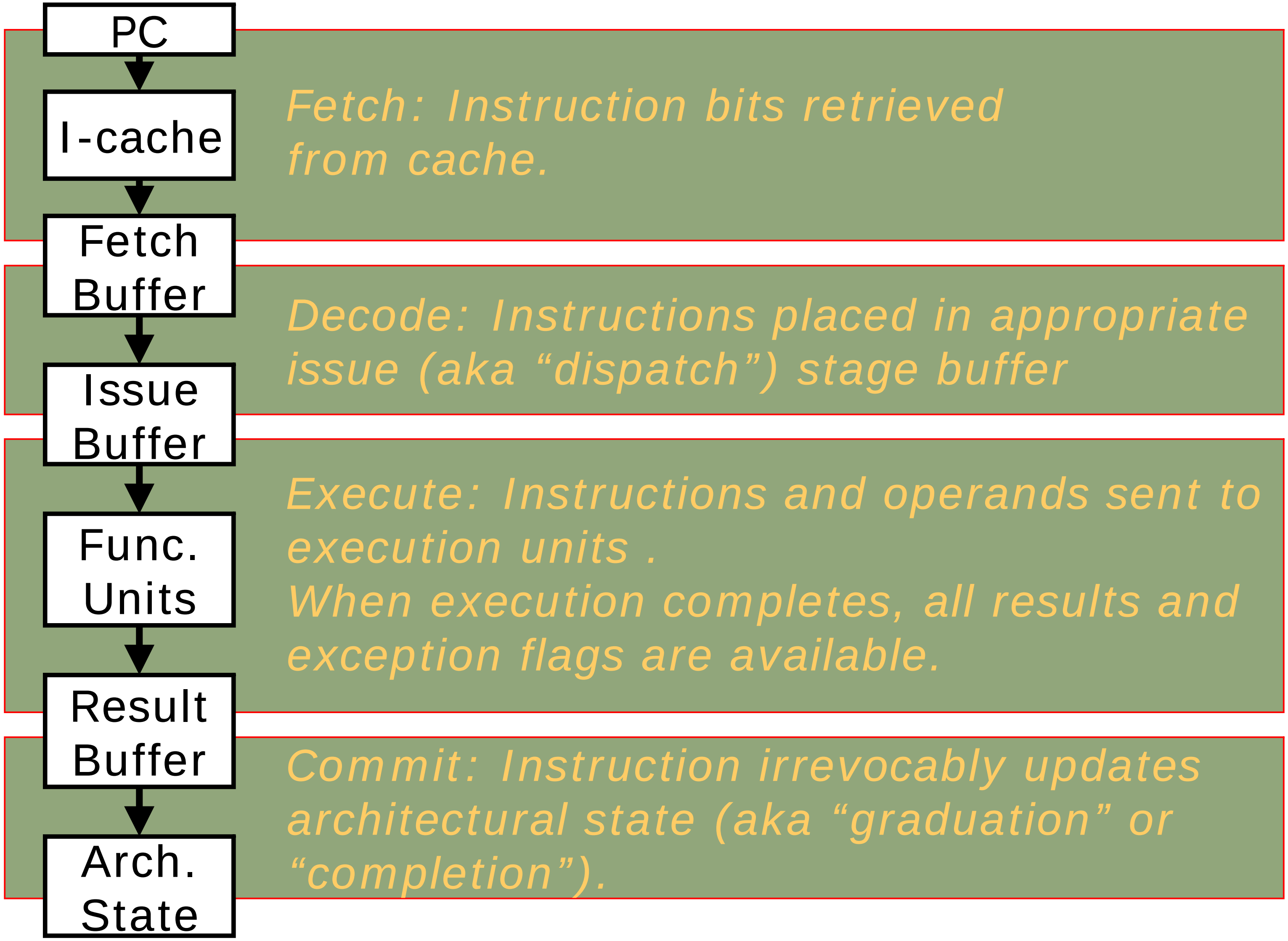
Renaming Table



Renaming table is a cache to speed up register name look up. It needs to be cleared after each exception taken.

When else are valid bits cleared? *Control transfers*

Phases of Instruction Execution



Fetch: Instruction bits retrieved from cache.

Decode: Instructions placed in appropriate issue (aka “dispatch”) stage buffer

Execute: Instructions and operands sent to execution units .
When execution completes, all results and exception flags are available.

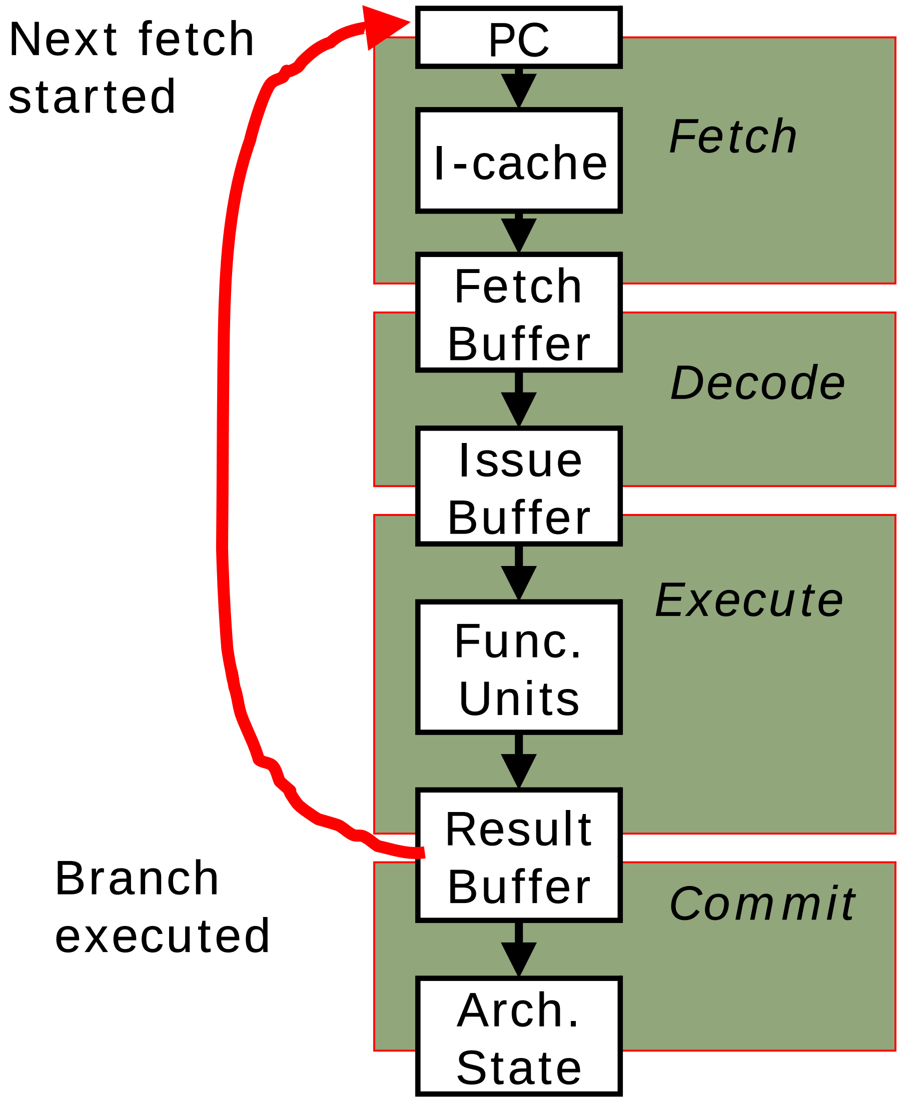
Commit: Instruction irrevocably updates architectural state (aka “graduation” or “completion”).

Control Flow Penalty

Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution !

How much work is lost if pipeline doesn't follow correct instruction flow?

~ Loop length x pipeline width



MIPS Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

1) Is the preceding instruction a taken branch?

2) If so, what is the target address?

Instruction	*Taken known?*	*Target known?*
J	After Inst. Decode	After Inst. Decode
JR	After Inst. Decode	After Reg. Fetch
BEQZ/BNEZ	After Reg. Fetch*	After Inst. Decode

Reducing Control Flow Penalty

Software solutions

- *Eliminate branches - loop unrolling*
  Increases the run length
- *Reduce resolution time - instruction scheduling*
  Compute the branch condition as early as possible (of limited value)

Hardware solutions

- Find something else to do - *delay slots*
  Replaces pipeline bubbles with useful work (requires software cooperation)
- *Speculate - branch prediction*
  Speculative execution of instructions beyond the branch

Branch Prediction

Motivation:

Branch penalties limit performance of deeply pipelined processors

Modern branch predictors have high accuracy (>95%) and can reduce branch penalties significantly

Required hardware support:

Prediction structures:

- Branch history tables, branch target buffers, etc.

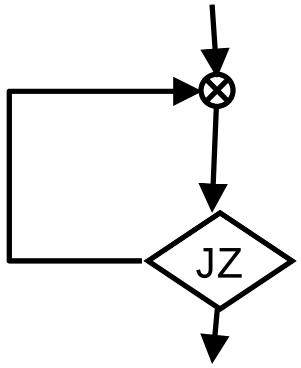
Mispredict recovery mechanisms:

- *Keep result computation separate from commit*
- Kill instructions following branch in pipeline
- Restore state to state following branch

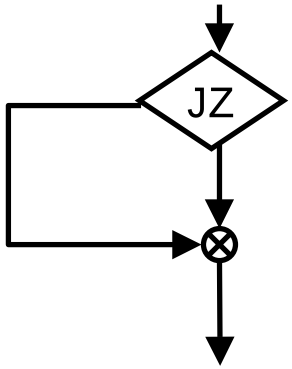
Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:

backward
90%



forward
50%



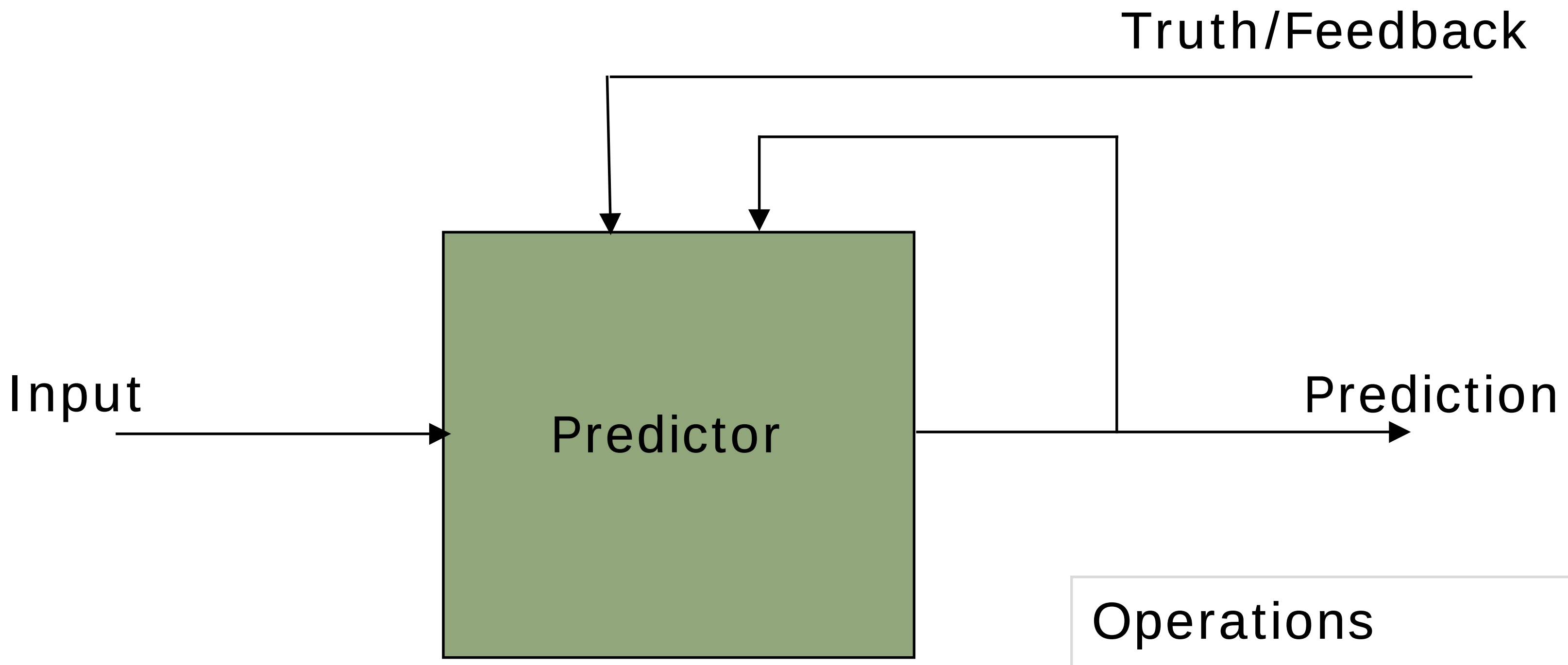
ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110

bne0 *(preferred taken)* beq0 *(not taken)*

ISA can allow arbitrary choice of statically predicted direction, e.g., HP PA-RISC, Intel IA-64

typically reported as ~80% accurate

Dynamic Prediction



Operations

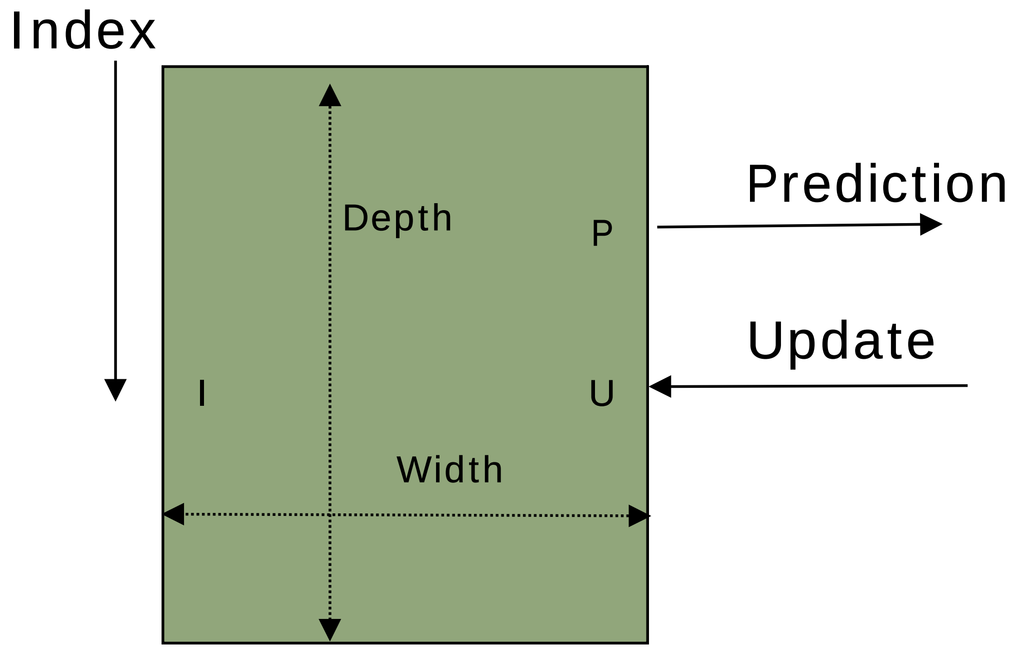
- Predict
- Update

Prediction as a feedback control process

Predictor Primitive

- Indexed table holding values

- Operations
  - Predict
  - Update



- Algebraic notation

Prediction = P[Width, Depth](Index; Update)

Dynamic Branch Prediction

learning based on past behavior

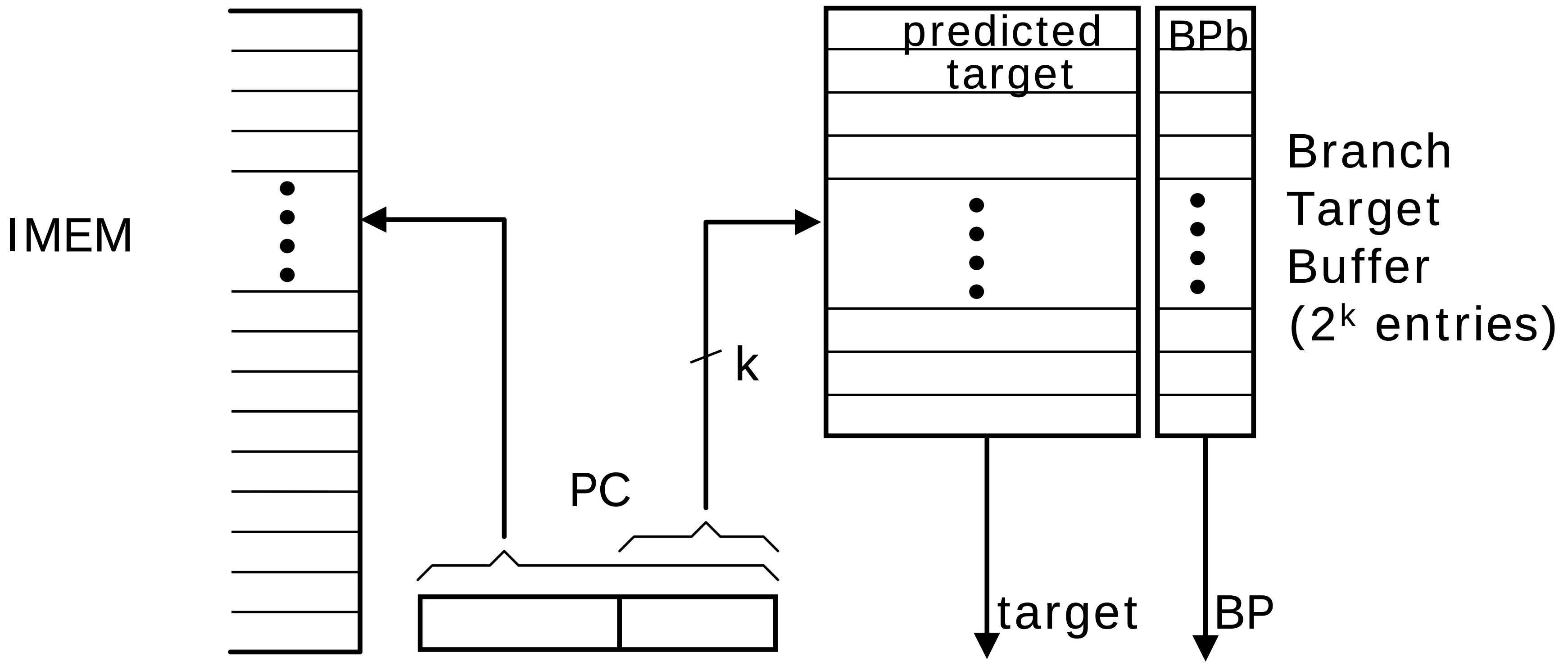
Temporal correlation

The way a branch resolves may be a good predictor of the way it will resolve at the next execution

Spatial correlation

Several branches may resolve in a highly correlated manner *(a preferred path of execution)*

Branch Target Buffer



BP bits are stored with the predicted target address.

IF stage: *If (BP=taken) then nPC=target else nPC=PC+4*
later: *check prediction, if wrong then kill the instruction and update BTB & BPb else update BPb*

BTB is only for Control Instructions

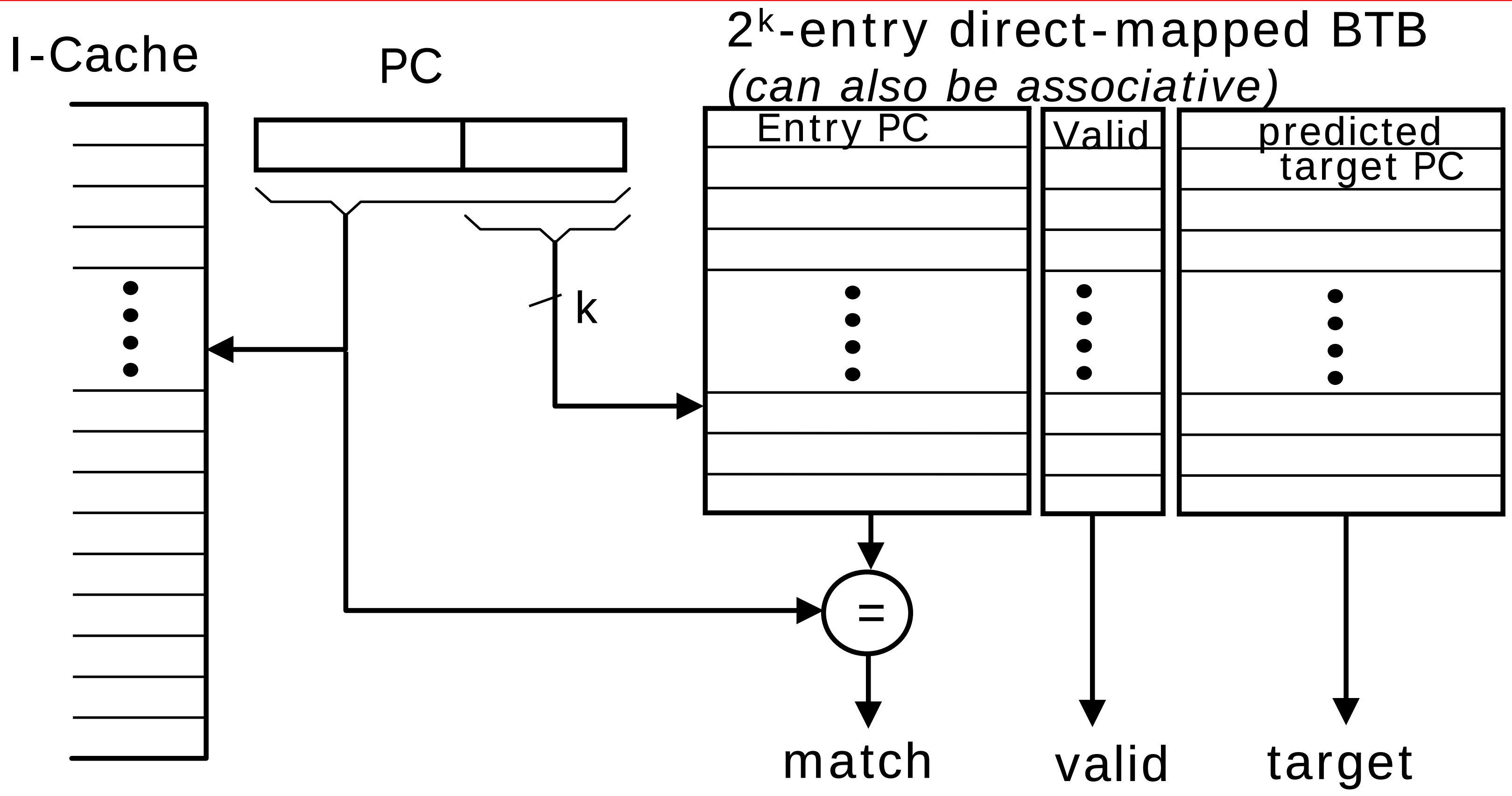
BTB contains useful information for branch and jump instructions only

$\Rightarrow$ Do not update it for other instructions

For all other instructions the next PC is (PC)+4 !

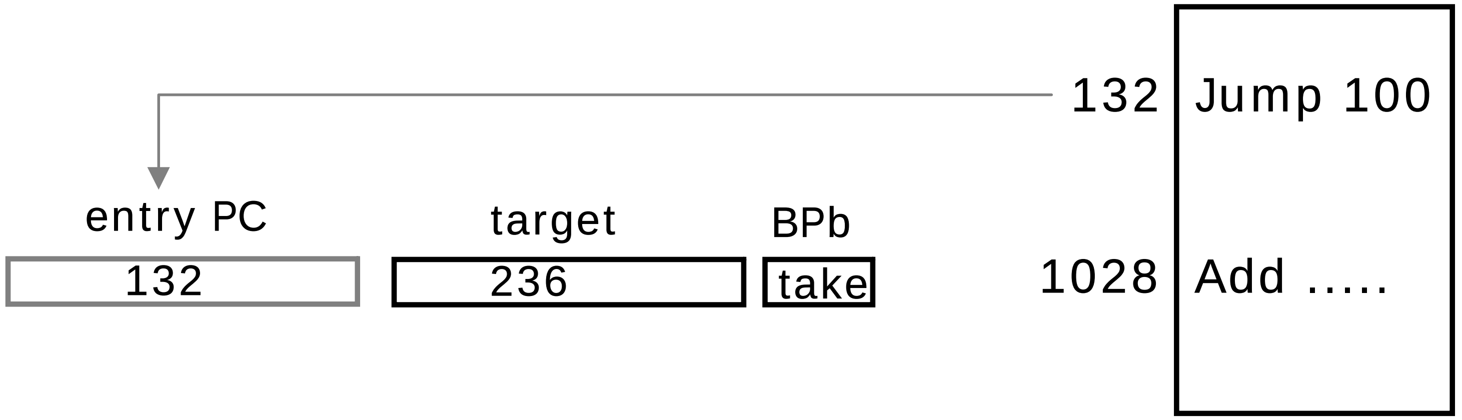
How to achieve this effect without decoding the instruction?

Branch Target Buffer (BTB)



- Keep both the branch PC and target PC in the BTB
- PC+4 is fetched if match fails
- Only *taken* branches and jumps held in BTB
- Next PC determined *before* branch fetched and decoded

Consulting BTB Before Decoding



- The match for PC=1028 fails and 1028+4 is fetched
  $\Rightarrow$ *eliminates false predictions after ALU instructions*

- BTB contains entries only for control transfer instructions
  $\Rightarrow$ *more room to store branch targets*

Uses of Jump Register (JR)

- Switch statements (jump to address of matching case)

  BTB works well if same case used repeatedly

- Dynamic function call (jump to run-time function address)

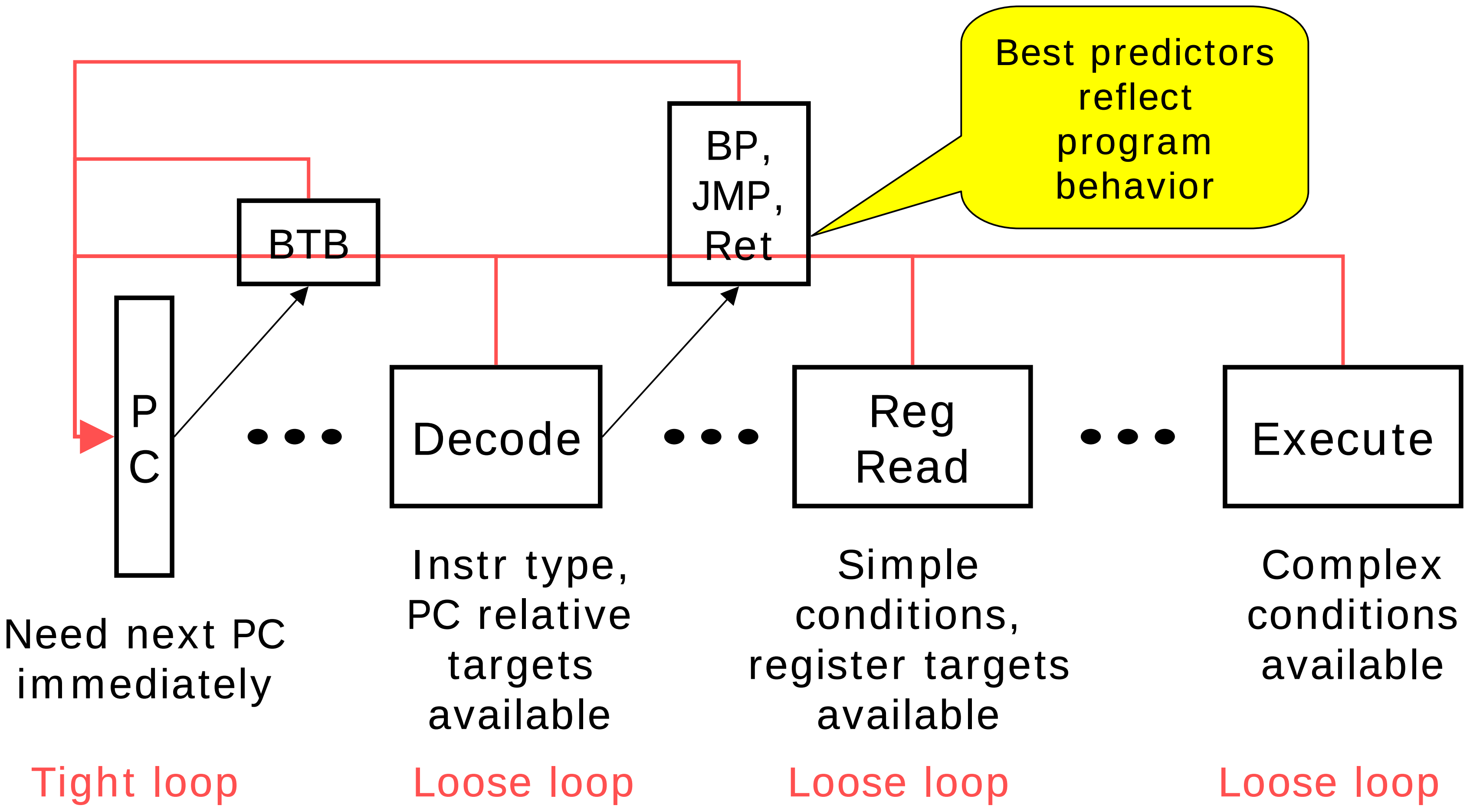
  BTB works well if same function usually called, (e.g., in C++ programming, when objects have same type in virtual function call)

- Subroutine returns (jump to return address)

  BTB works well if usually return to the same place

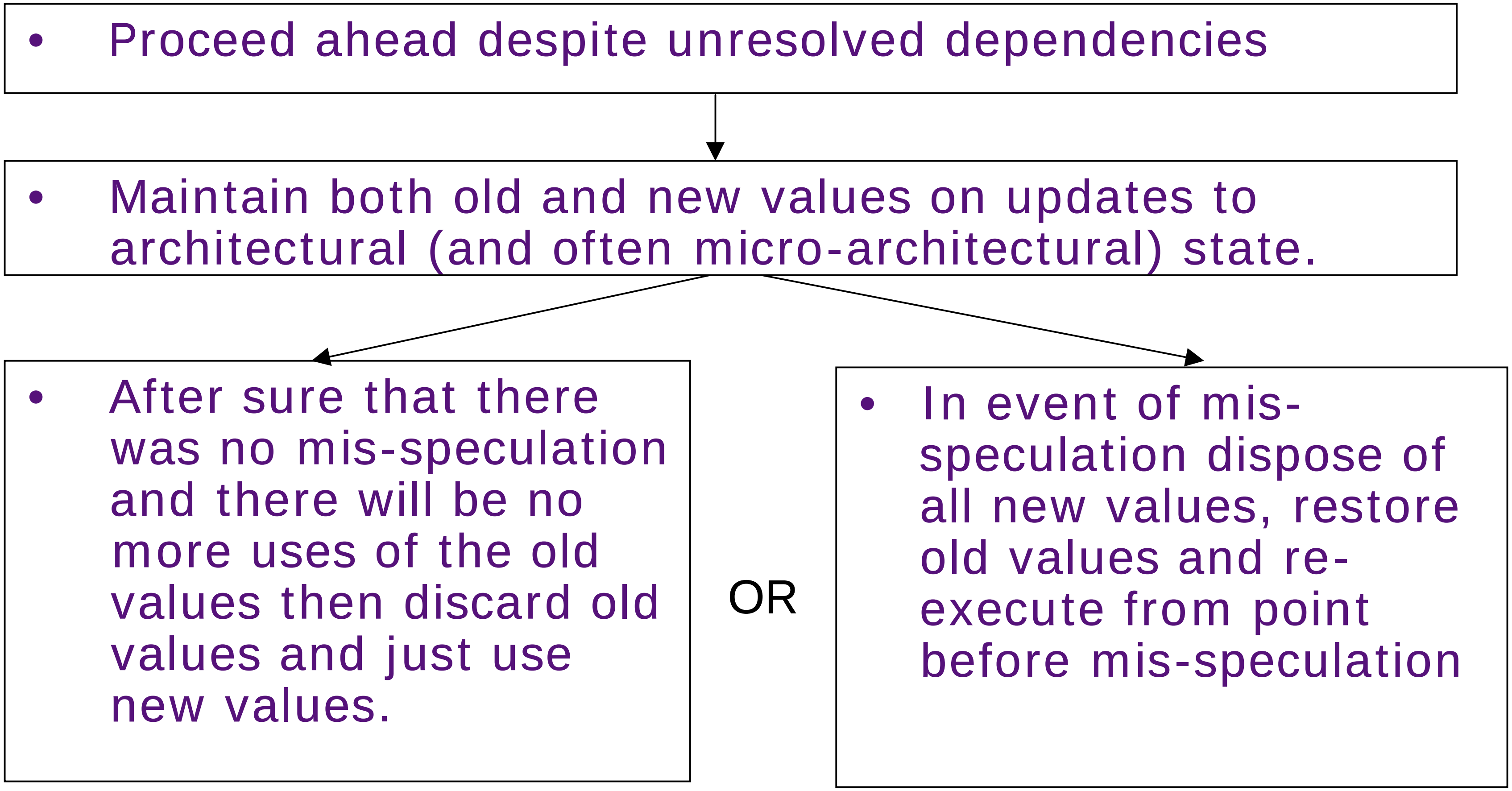
  $\Rightarrow$ *Often one function called from many distinct call sites!*

Overview of branch prediction


Best predictors reflect program behavior
BP, JMP, Ret
BTB
P C
Decode
Reg Read
Execute
Need next PC immediately
Instr type, PC relative targets available
Simple conditions, register targets available
Complex conditions available
Tight loop
Loose loop
Loose loop
Loose loop

Must speculation check always be correct? **No…**

Speculative Execution Recipe



- Proceed ahead despite unresolved dependencies

- Maintain both old and new values on updates to architectural (and often micro-architectural) state.

- After sure that there was no mis-speculation and there will be no more uses of the old values then discard old values and just use new values.

OR

- In event of mis-speculation dispose of all new values, restore old values and re-execute from point before mis-speculation

Why might one use old values? O-O-O WAR hazards

Value Management Strategies

Greedy Update:

– Update value in place, and

– Maintain a log of old values to use for recovery.

Lazy Update:

– Buffer new value leaving old value in place.

– Replace old value only at ‘commit’ time.

Why leave an old value in place?

Old value might be used even after new value is generated

Speculating Both Directions

An alternative to branch prediction is to execute both directions of a branch *speculatively*

- resource requirement is proportional to the number of concurrent speculative executions
- only half the resources engage in useful work when both directions of a branch are executed speculatively
- branch prediction takes less resources than speculative execution of both paths

With accurate branch prediction, it is more cost effective to dedicate all resources to the predicted direction