Advanced issues in pipelining

Outline

- Handling exceptions
- Supporting multi-cycle operations
- Pipeline evolution
- Examples of real pipelines

Handling exceptions

Exceptions

- In pipelined execution, one instructions completes in N clock cycles
- Clock cycle is the minimum granularity for interrupts
  - So, exceptions can interrupt execution of several instructions!
- We must include **capabilities to "cancel" an instruction** while executing!
- Source of interrupts:
  - Power failure
  - Arithmetic overflow
  - I/O device request
  - OS call
  - Page fault
  - ...

Exception source in pipeline

- Each stage has its sources:
  - **IF**
    - Page fault on instruction fetch
    - Misaligned memory access
    - Memory-protection violation
  - **ID**
    - Undefined or illegal opcode
  - **EX**
    - Arithmetic interrupt
  - **MEM**
    - Page fault on data fetch
    - Misaligned memory access
    - Memory-protection violation

Exception handling

- Complications:
  - Simultaneous exceptions in more than one pipeline stage, e.g.,
    - **Load** with data page fault in MEM stage
    - **Add** with instruction page fault in IF stage
    - **Add** fault will happen BEFORE load fault
- In MIPS, exceptions tend to occur in the EX or MEM stage (e.g., late in the pipe)
- Requirements:
  - Pipeline must be safely shut down
  - PC must be saved so restart point is known
    - If restart is a branch then it will need to be re-executed
      - Which means the condition "code" state must not change

Exception handling (2)

- MIPS/DLX sequence:

  When an exception occurs, the pipeline control does:

  1. Force **`trap`** instruction into pipeline on next IF
  2. "***Squash***" all instructions that follow
     (0's to pipeline registers)
     - *Prevent not-completed instructions from changing state!*
  3. Let all preceding instructions complete if they can
  4. Save the restart PC value
     - If delayed branches are used, save *BDS+1* PCs
  5. OS handle the exception
     (saves PC of faulting instructions for later restart)

Precise exceptions (1)

- Definition:
  - Exceptions are called ***precise*** if they *leave the machine in a state that is consistent with the sequential execution model*
  - Otherwise they are called imprecise (or non-precise)
- Example:
  - A precedes B
  - B finishes before A and modifies state ➡ **imprecise**

Precise exceptions (2)

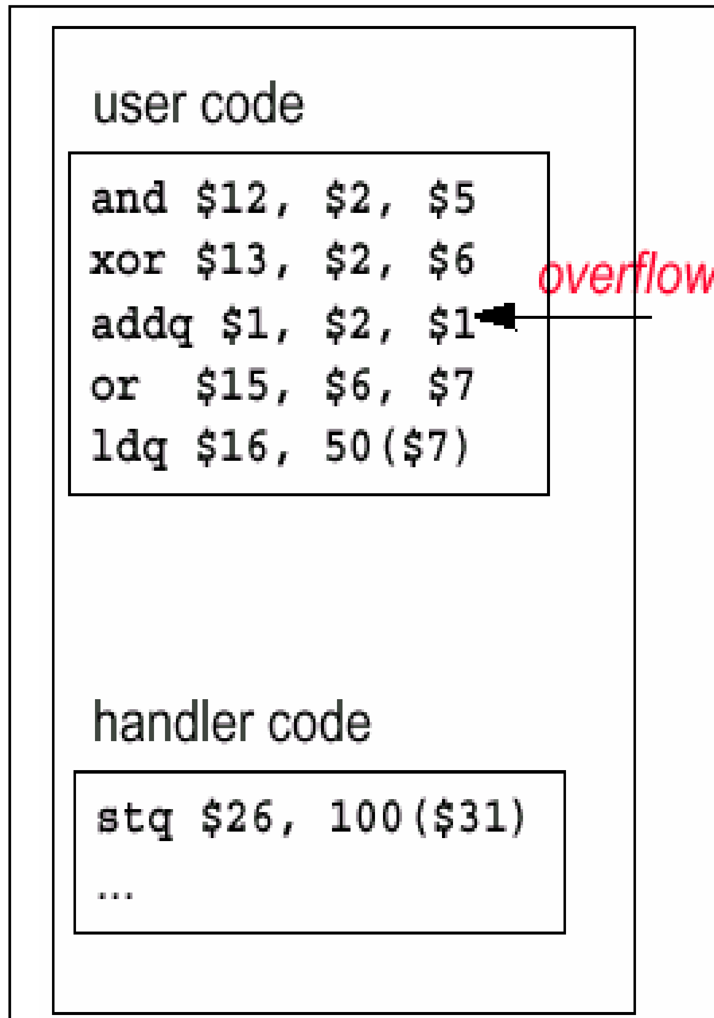
- Condition for preciseness:
  1. All instructions **before** the faulting instruction have completed (and modified machine state)
  2. All instructions **after** the faulting instruction have **not** completed (nor modified machine state)
  3. The faulting instruction may or may not complete, but is either completed or it has to be started
- Precise exceptions are not always possible
  - e.g., FP operations
- Most machines have two operation modes:
  - **Precise** exceptions (slow, less overlapping)
  - **Non-precise** exception (fast)

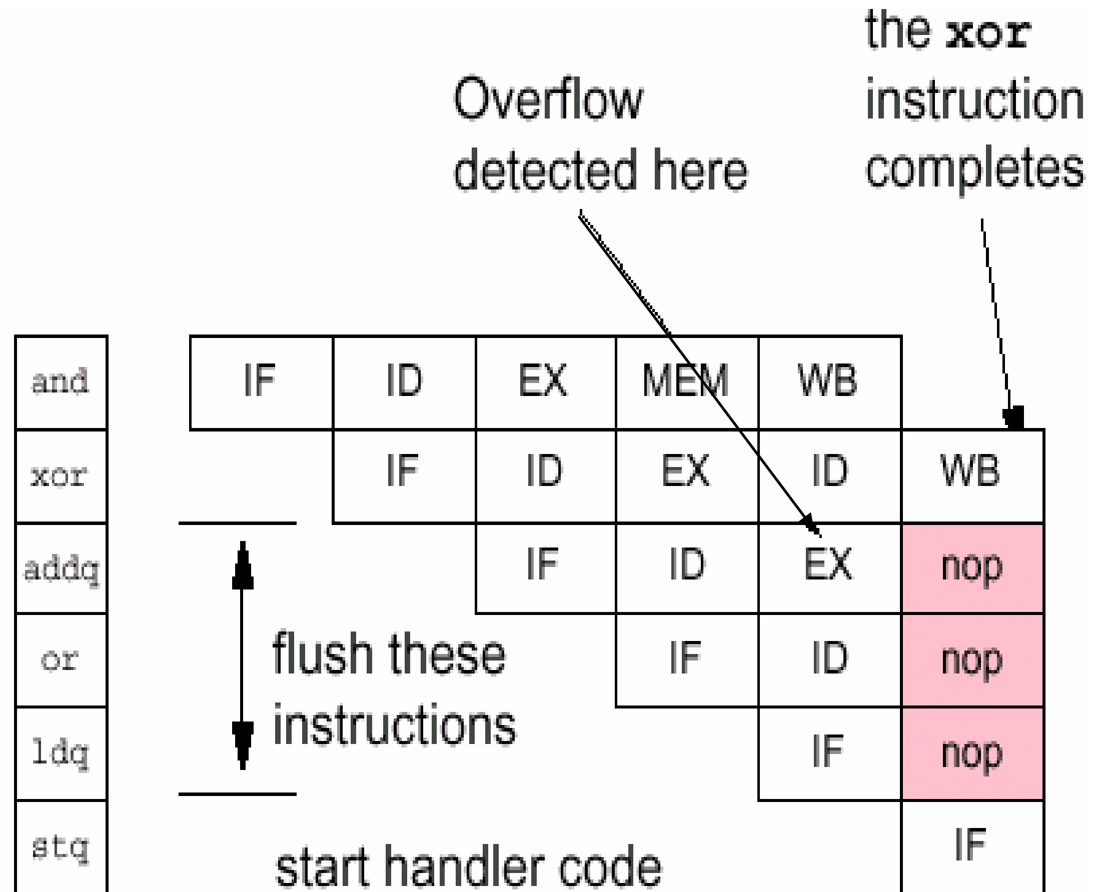
Handling exceptions: example

I	IF	ID	EX	MEM	WB							
I+1		IF	ID	EX	MEM	*WB*						<- Page fault
I+2			IF	ID	EX	*MEM*	*WB*					<- Squash
I+3				IF	ID	*EX*	*MEM*	*WB*				<- Squash
I+4					IF	*ID*	*EX*	*MEM*	*WB*			<- Squash
I+5						IF	ID	EX	MEM	WB		
I+6							IF	ID	EX	MEM	WB	

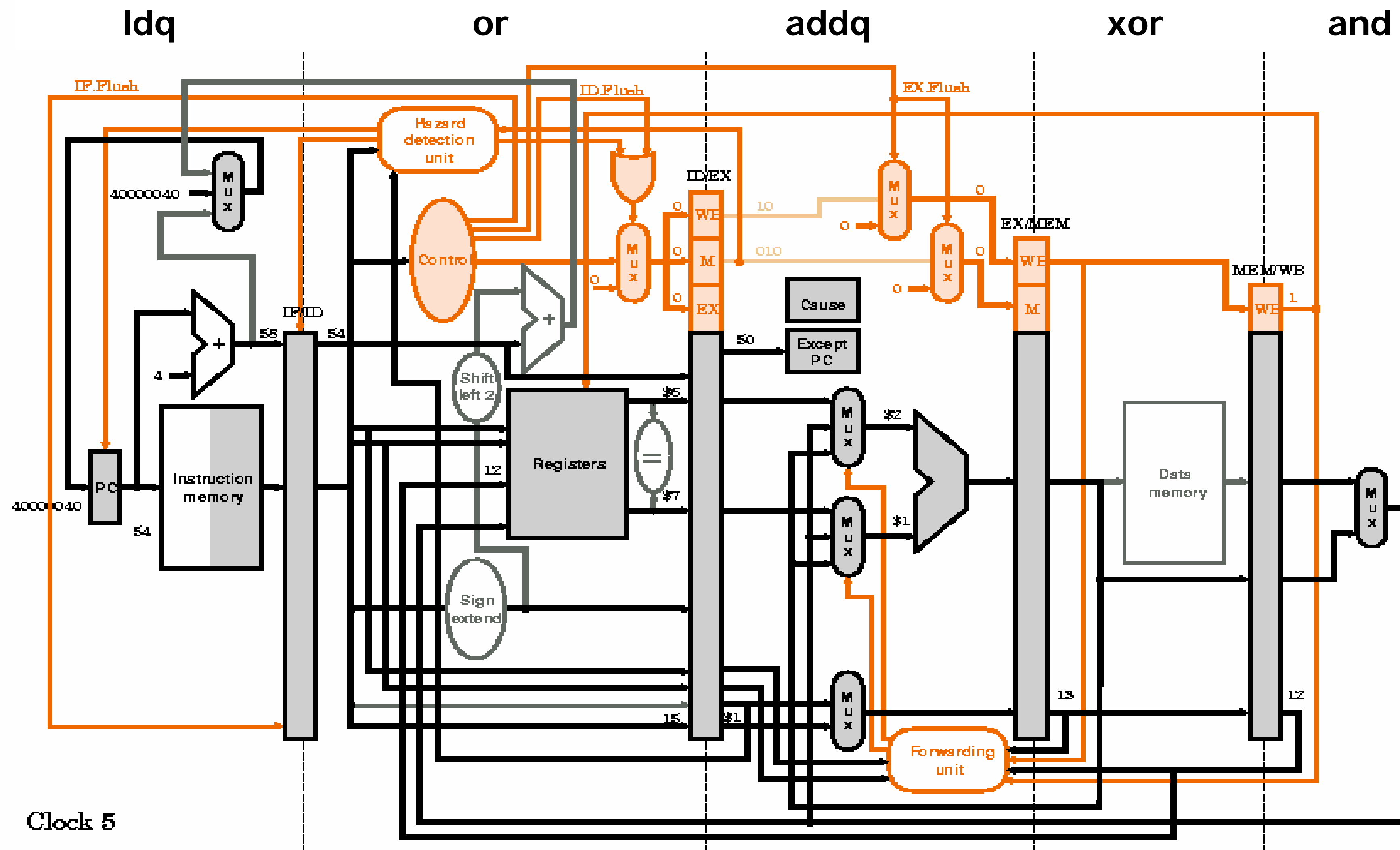
Interrupt handler

Handling exceptions: example (2)

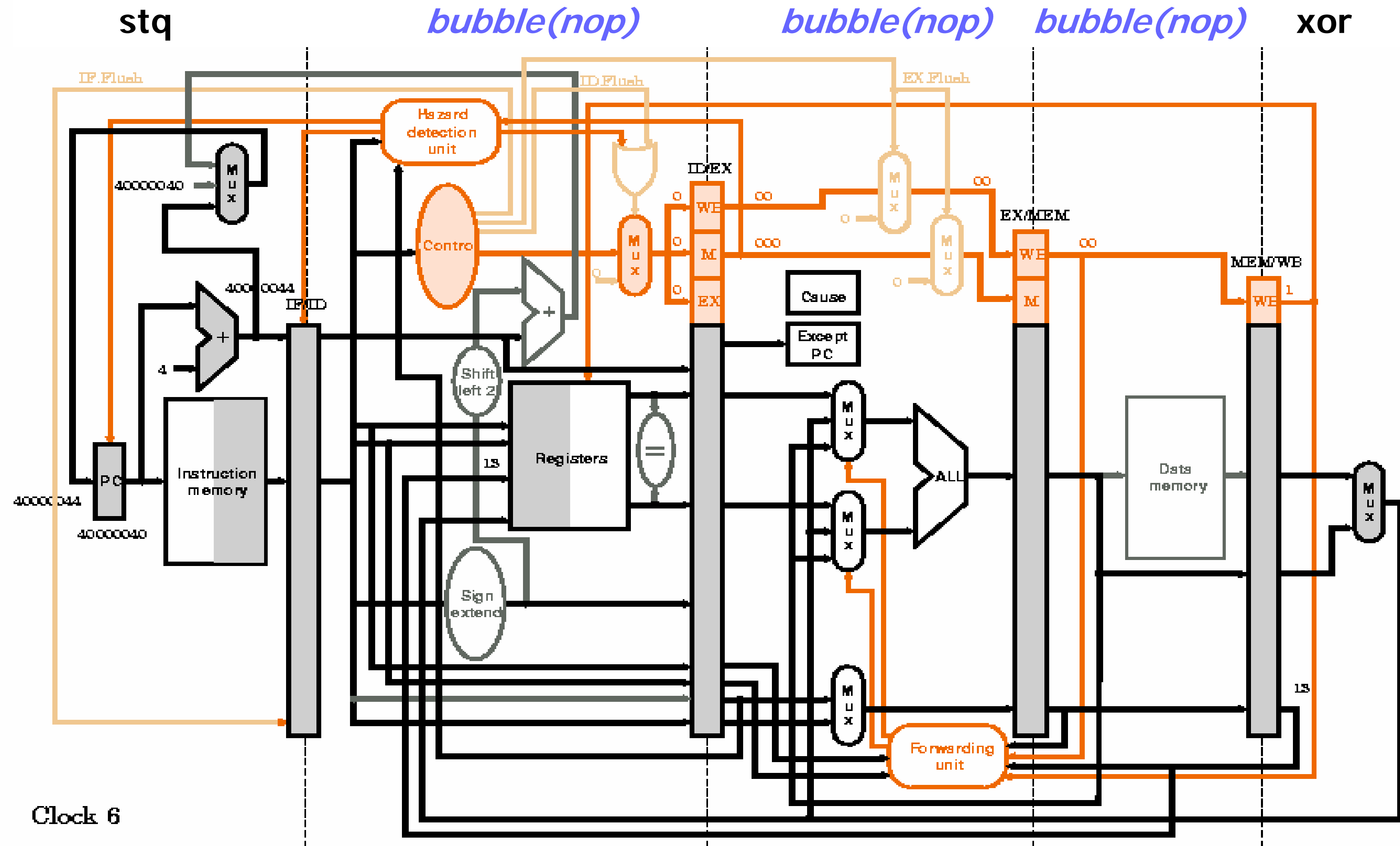




Handling exceptions: example (2)



Handling exceptions: example (2)



Further complications in pipelines

- Complex Addressing Modes and Instructions
  - Address modes:
    - Autoincrement causes register change during instruction execution
    - Interrupts? Need to restore register state
    - Adds WAR and WAW hazards since writes are no longer the last stage.
  - Memory-Memory Move Instructions
    - Must be able to handle multiple page faults
    - Long-lived instructions: partial state save on interrupt
  - Condition Codes
    - Ex., PSW bits

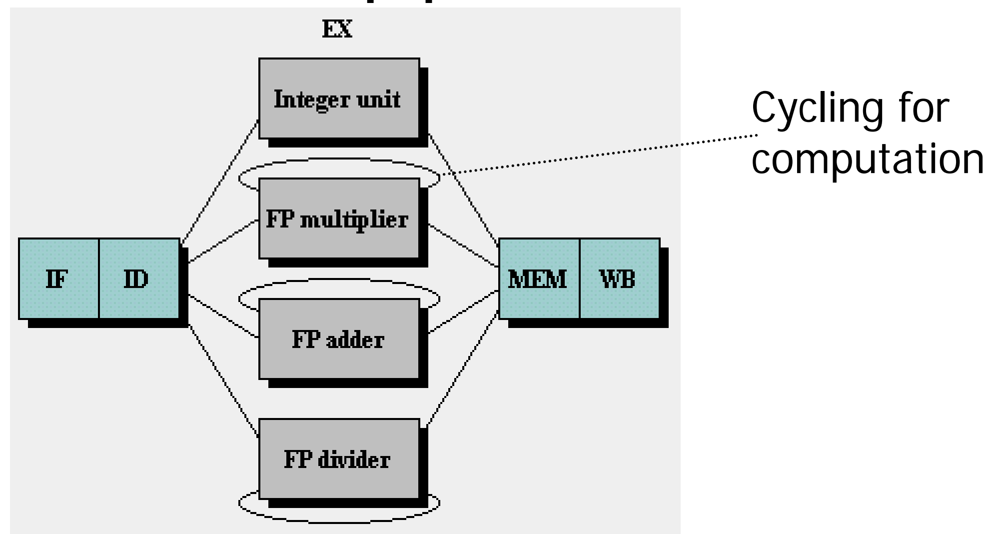
Multi-cycle operations

Multi-cycle Operations

- It is impractical to require that all operations complete in one clock cycle
- What about FP operations?
  - Recall that clock cycle is bounded by slowest pipeline stage
  - Doing FP ops in one cycle would mean to use a slow clock
- Use a modified pipeline:
  - The **EX cycle is repeated** as many times as needed to complete the operation
  - There may be multiple floating-point functional units.

Multi-cycle Operations (2)

- Four separate functional units:
  - The main integer unit
  - FP and integer multiplier
  - FP adder (handles FP add, subtract, and conversion)
  - FP and integer divider
- If we assume that the execution stages of these functional units are not pipelined



Cycling for computation

Multi-Cycle Operations (3)

- Floating point gives long execution time.
- It's possible to pipeline the FP execution unit so it can initiate new instructions without waiting full latency.
  - Can also have multiple FP units.
- Typical values:

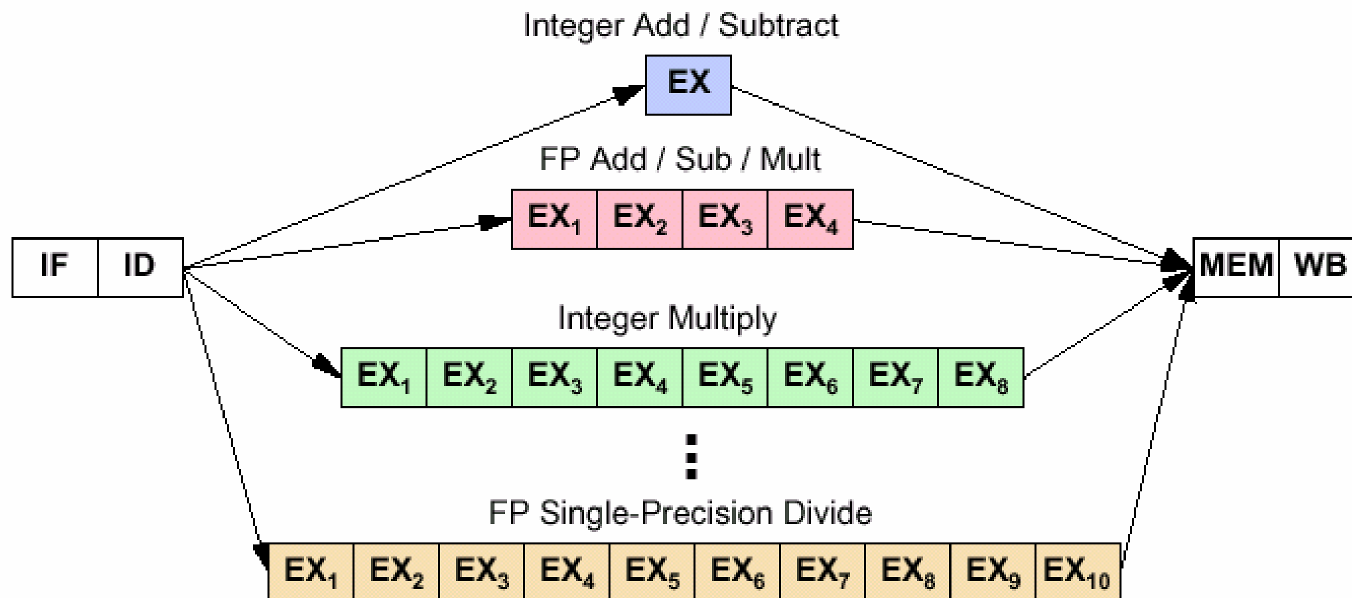
FP Instruction	Latency
Add, Subtract	4
Multiply	8
Divide	36
Square root	112
Negate	2
Absolute value	2
FP compare	3

→ # of stages after EX an Instruction produces a result

Initiation interval = time between two successive ops (= latency-1)

Multi-Cycle Operations (4)

- To reduce initiation interval, some “long” operations are pipelined
  - Some may not be pipelined
    - e.g, sometimes division not pipelined
- Implementation:



Multi-Cycle Operations (5)

- Example (independent operations):

MULTD	IF	ID	*M1*	*M2*	*M3*	*M4*	*M5*	*M6*	*M7*	MEM	WB
ADDD		IF	ID	*A1*	*A2*	*A3*	*A4*	MEM	WB		
LD			IF	ID	EX	MEM	WB				
SD				IF	ID	EX	MEM	WB			

Multi-Cycle Operations: problems

- Longer latencies increase the chance of RAW hazards
  - Ex: instruction needing result of an FP ADD must wait 4 cycles
- Non-pipelined operations increase the chance of structural hazards
- WAW are now possible (but no WAR)
  - Instructions do not reach WB in order
- Instruction complete in different order than issued
  - Problems with exceptions
- Different latencies may require multiple write to registers

Multi-Cycle Operations: problems

- Example: RAW stalls

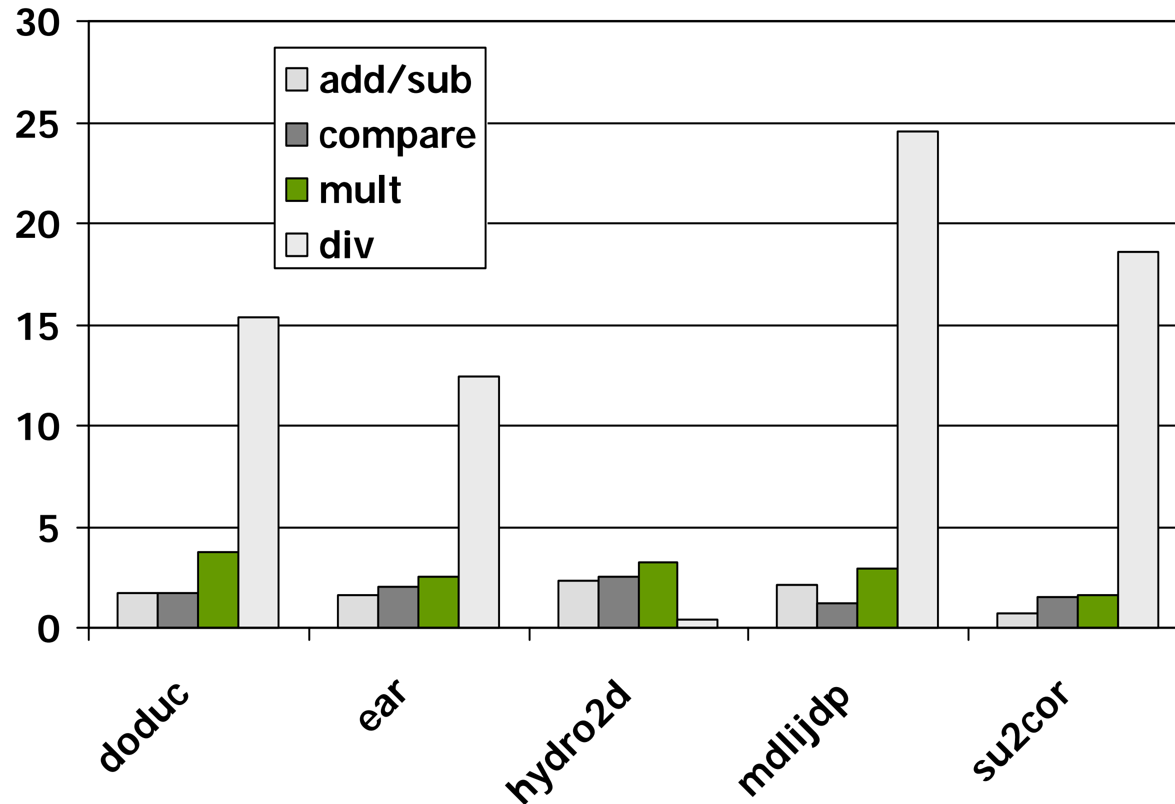
LD **F4**,0(R2)	IF	ID	EX	MEM	WB											
MULTD **F0**,**F4**,F,6		IF	ID	*stall*	M1	M2	M3	M4	M5	M6	M7	MEM	WB			
ADDD F2,**F0**,F8			IF	ID	stall	stall	stall	stall	stall	stall	stall	A1	A2	A3	A4	MEM
SD F2,0(R2)				IF	stall	stall	stall	stall	stall	stall	stall	stall	stall	stall	stall	MEM

- Example: multiple writes to register

MULTD F0,F4,F,6	IF	ID	*M1*	*M2*	*M3*	M4	M5	M6	M7	MEM	**WB**
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
ADDD F2, F2, F6				IF	ID	A1	A2	A3	A4	MEM	**WB**
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
LD F8, 0(R2)							IF	ID	EX	MEM	**WB**

Multi-Cycle Operations

- Performance impact:
  - Average # of stalls for operation type



Pipeline evolution

Outline

- Out-of-order execution
- Superscalar pipelines
- Superpipelines

Out-of-order execution

- Assumptions so far:
  - **In-order instruction execution**:
    - *If an instruction is stalled in the pipeline, later instructions cannot proceed*
  - **Static scheduling**
    - The hardware *minimizes* the impact of hazards
    - It is up to the compiler to *schedule* dependent instructions so as to avoid hazards
      - Static = No exploitation of run-time scheduling opportunities
- Improvements:
  - Out-of-order (OOO) execution
  - Dynamic scheduling
    - Done by the hardware!

OOO execution

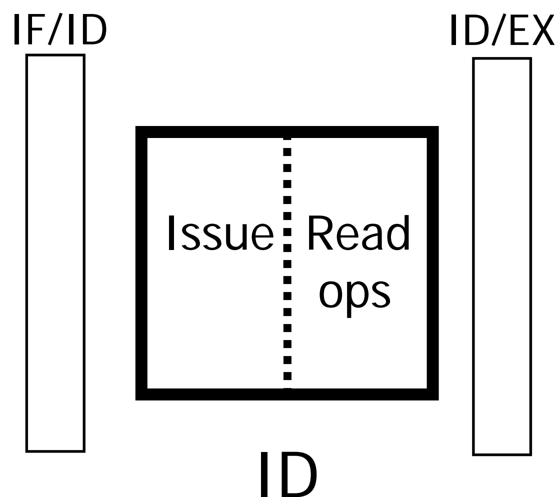
- In traditional pipelines, a hazards also stalls instructions that are not affected by it
  - Example

```
DIVD     $0, $2, $4
ADDD     $10, $0, $8
SUBD     $12, $8, $4
```

  - The SUB is independent of the rest, yet it is stopped (for several cycles)
    - Particularly critical for multi-cycle instructions
  - Why not let SUB complete?
    - Need extra hardware to do this
    - Need to separate:
      - **Hazard detection phase**
      - **Instruction issue phase**

OOO execution (2)

- Hazard detection and instruction issue done in the ID stage
  - Split ID conceptually into two stages:
    - **Issue**
      - Decode instructions and check for hazards
    - **Read operands**
      - Wait until no data hazards, then read operands



OOO execution (3)

- Definitions:
  - Three phases of instruction processing:
    1. Instruction **issue** (or *dispatch*)
    2. Instruction **execution**:
       - *There will be* multiple instructions in execution at the same time
         (thanks to availability of several independent functional units)
    3. Instruction **commit** (or completion)
       - When results are written
  - Typically:
    - Issue: **in-order**
    - Execution: **out-of-order**
    - Commit: *may be either one*

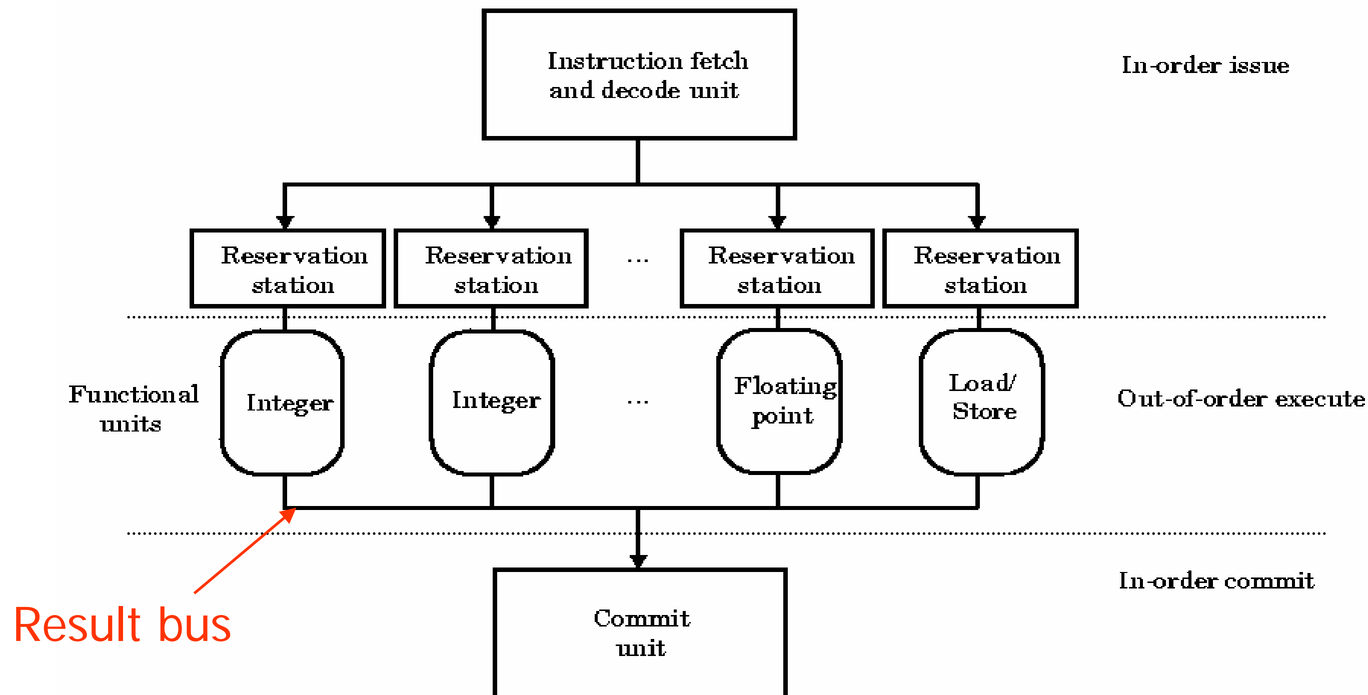
Dynamic scheduling

- OOO *execution* = *instruction* issue is still in order!
  - WAR hazards now possible!
    - Example:

```
DIVD    $0, $2, $4
ADDD    $10, $0, $8
SUBD    $8, $8, $4
```

If SUBD is executed first, it will write a new $8 before it is read by ADDD

OOO execution: organization



- Reservation station (a.k.a. ***issue queues***)
  - Buffers where instructions wait for the desired unit
  - Realize the "issue" phase

OOO execution: organization (2)

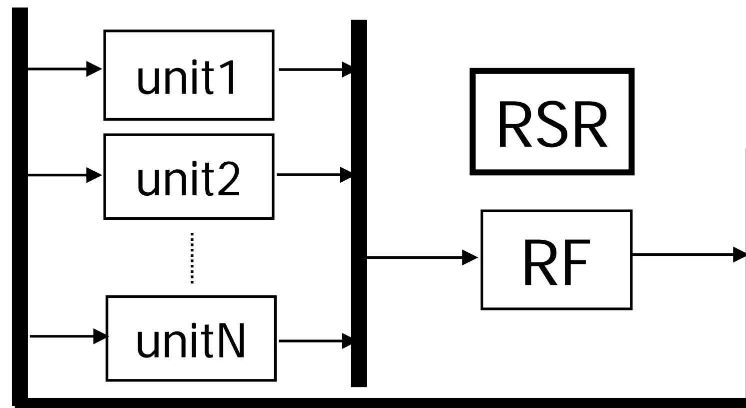
- Traditional MEM stage has disappeared
  - MEM accesses managed by a proper LOAD/STORE unit
- EX stage consists of multiple units with different latencies
- There is a single **result bus** for transferring results of the various units to register file

OOO execution schemes

- Three solutions:
  - **Commit-in-order**:
    - Instructions are committed according to the architectural order
  - **Reorder buffer**:
    - Instructions are committed out of order, but they are forced to modify the machine state according to the architectural order
  - **History buffer**:
    - Instructions are committed out of order, and machine state is modified in any order, but it is always possible to **restore a consistent state** in case of hazards

OOO execution: architecture

- Single result bus requires a bus reservation scheme
  - Implemented through the Result Shift Register (RSR)
    - Position *j* of RSR is reserved when instruction I (taking *j* cycles for EX) is issued
    - If taken by an earlier instruction, I is kept waiting in reservation stations for 1 cycle (after that, retry)
    - Contents of RSR shifted at each cycle



RSR structure

- Information contained in the RSR:
  - Identifier of unit to be used
  - Destination register
  - Validity bit (actual reservation or not)
  - PC (for state restoration)
- Example:

```
100  multd f12,f8,f10 (10 cycles)
104  addd f18,f14,f16 (2 cycles)
```

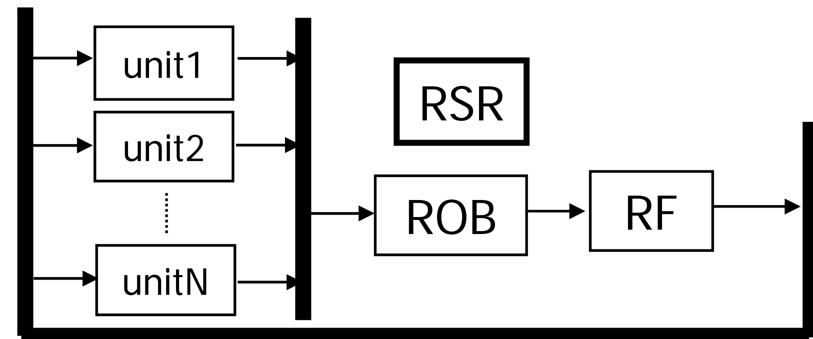
	Unit	Rd	V	PC
1				
2	Add	F18	1	104
…				
10	Mult	F12	1	100
11				
…				

Commit-in-order

- RSR is reserved so that commit is in the same order as the program
- Implementation
  - Instruction I when reserving RSR slot *j* also reserves slots *0,…,j-1* that are not yet reserved
    - Done by using some bits in an RSR entry

Reorder buffer (1)

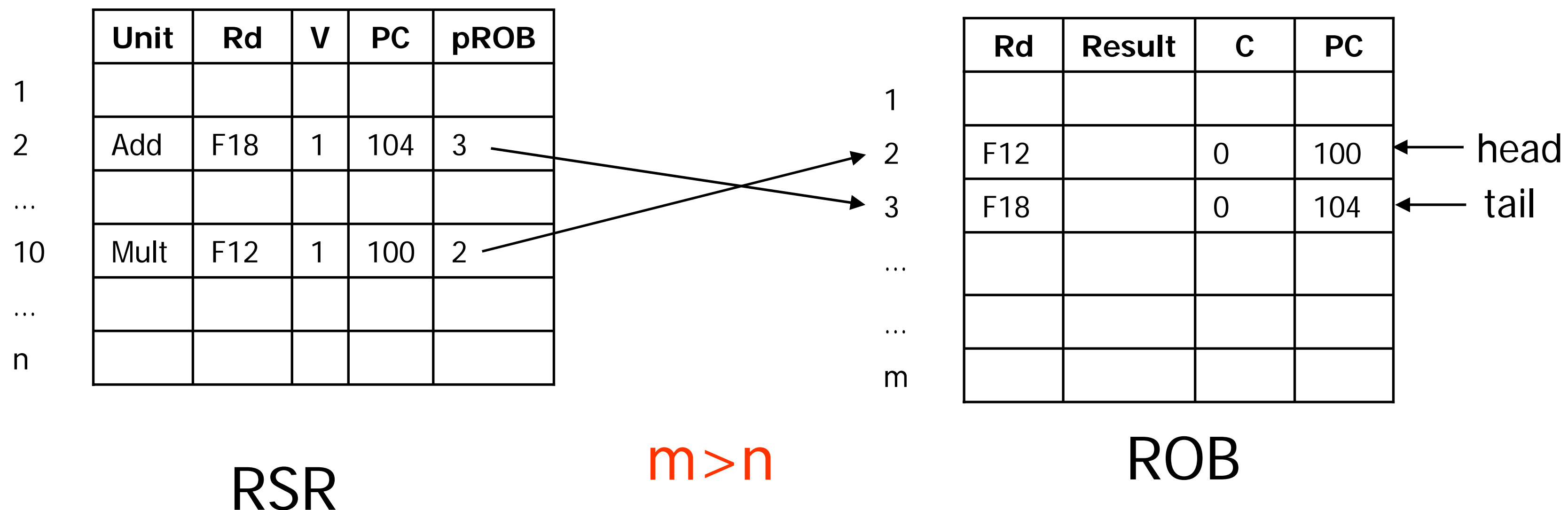
- Results are fed to the Reorder buffer (ROB)
  - ROB works as a circular queue
    - Head = instruction that when completed will modify first the state
    - Tail = where issued instructions are inserted
    - Entries contain:
      - Destination register
      - Instruction result
      - Commit bit (instruction is committed)
      - PC
  - RSR now contains a pointer to position in ROB



Reorder buffer (2)

- Completed instructions will leave RSR but not ROB
  - Instructions leave ROB **in order!**
    - Entries that leave RSR *write result* in ROB entry!
- Example:

```
100   multd f12,f8,f10   (10 cycles)
104   addd f18,f14,f16   (2 cycles)
```



RSR

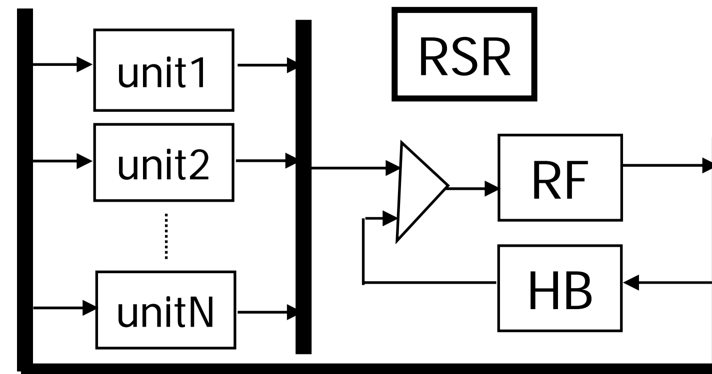
	Unit	Rd	V	PC	pROB
1					
2	Add	F18	1	104	3
...					
10	Mult	F12	1	100	2
...					
n					

ROB

	Rd	Result	C	PC	
1					
2	F12		0	100	← head
3	F18		0	104	← tail
...					
...					
m					

History buffer (1)

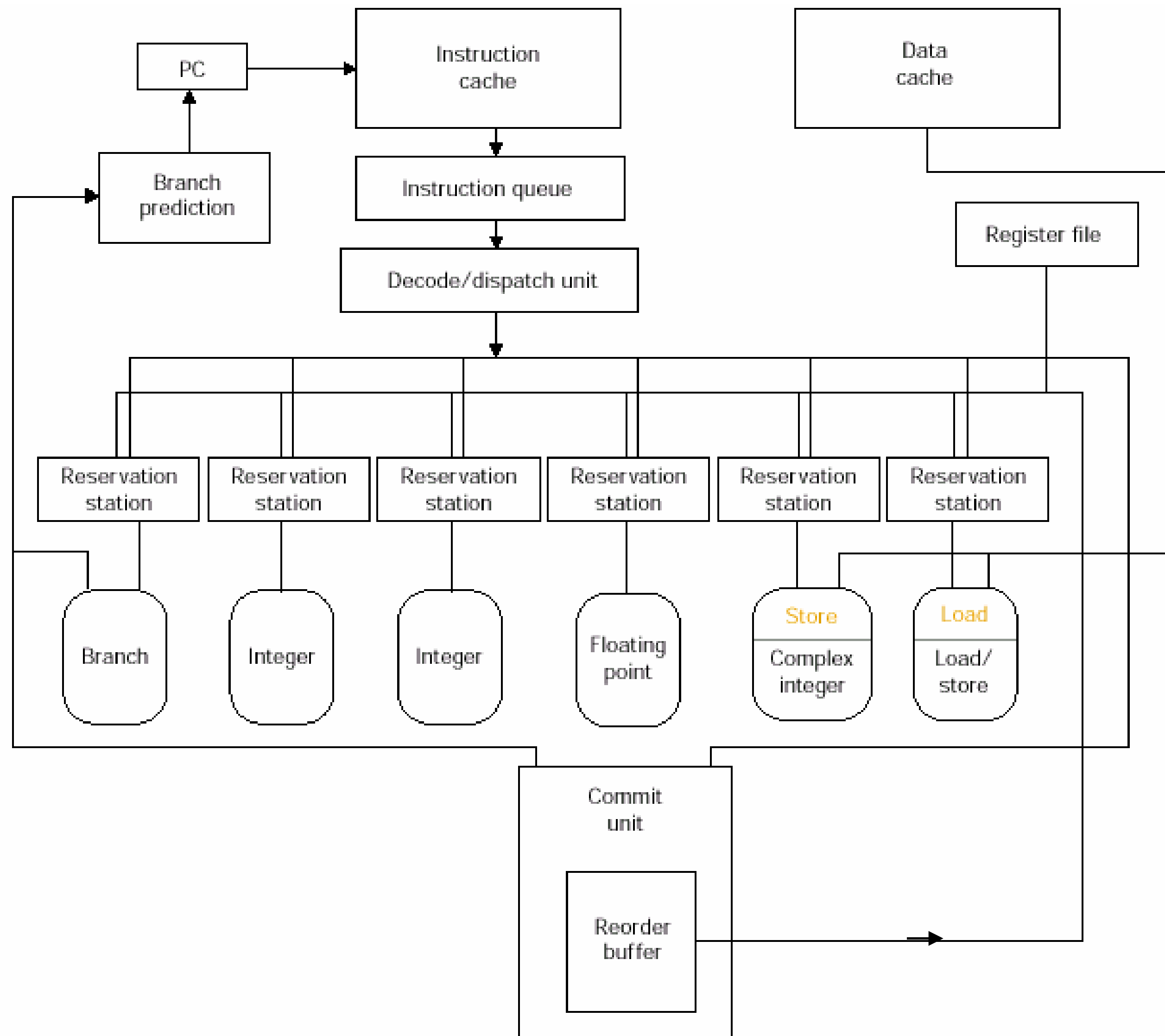
- Improves ROB
  - Similar to ROB
    - Circular queue
    - Entries contain:
      - Destination register
      - **Old value of destination register**
      - Commit bit
      - PC
  - RSR contains a pointer to HB



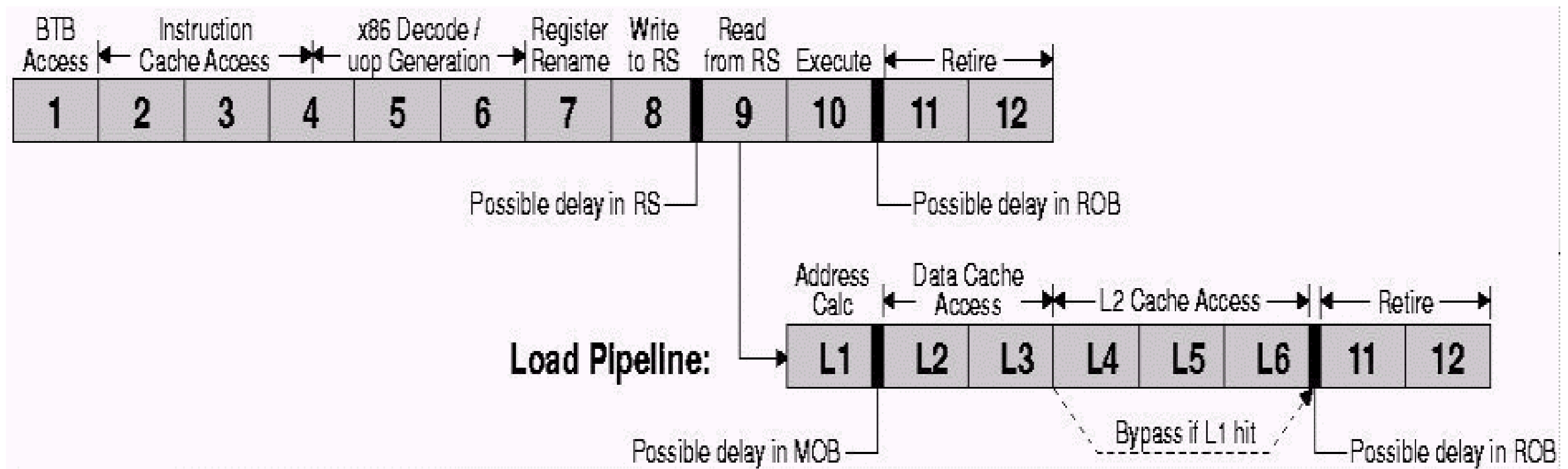
History buffer (2)

- Results are written in RF as soon as available
- If there is no need of restoring the previous state, the (committed) instruction at the top of HB leaves HB
- Note:
  - Instructions can be issued only if dependencies are solved; if not, they are queued
  - Consequence: state must be restored only because of
    - Interrupts
    - Mispredicted branches

P6 architecture



P6 Pipeline



P6 pipeline

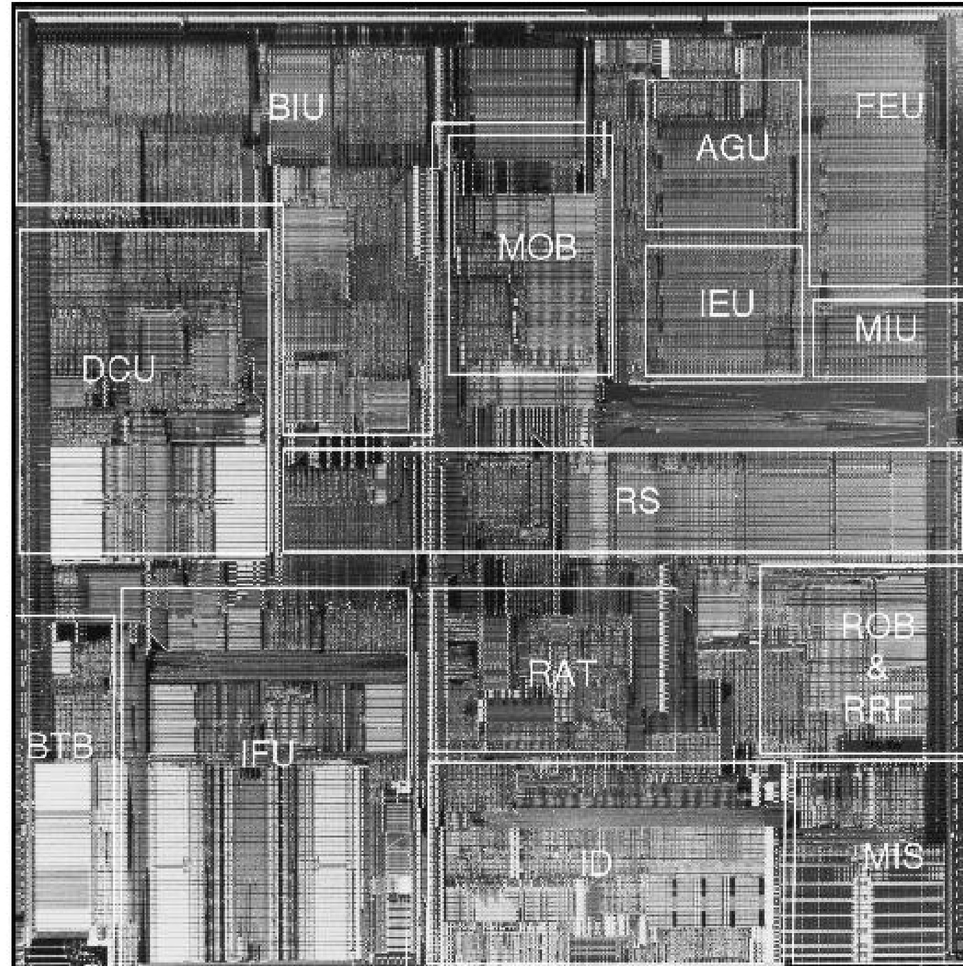


Figure 4. The P6 CPU measures 17.5 × 17.5 mm when fabricated in a 0.5-micron four-layer-metal BiCMOS process.

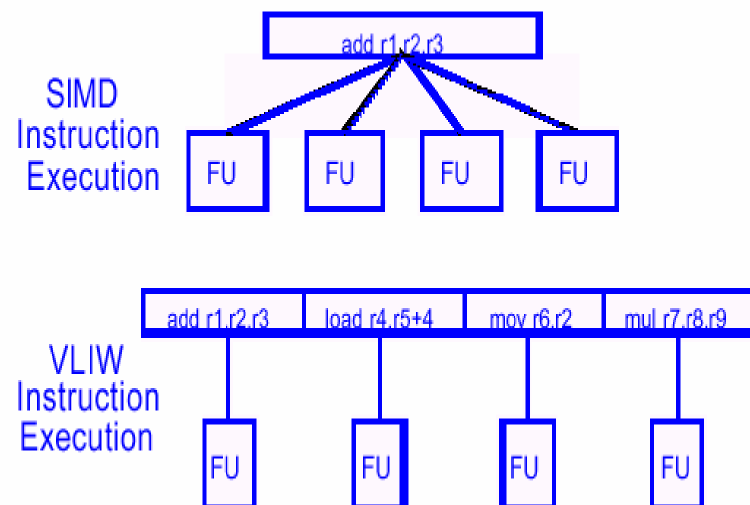
Advanced architectures

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Two variants:
  - **Superscalar**:
    - Varying no. instructions/cycle (1 to 8), scheduled by compiler or by HW
    - IBM PowerPC, Sun UltraSparc, DEC Alpha, HP 8000
  - **Very Long Instruction Words** (**VLIW**):
    - Fixed number of instructions (4-16) *scheduled by the compiler*
    - Put operations into wide templates
- Instructions Per Clock cycle (IPC) vs. CPI

VLIW

- Statically scheduled ILP architecture.
  - Wide instructions specify many independent simple operations
  - Multiple functional units execute all of the operations in an instruction concurrently
  - Instructions directly control the hardware with minimal decoding
- A powerful compiler is responsible for locating and extracting ILP from the program and for scheduling operations to exploit the available parallel resources
- VLIW Instruction
  - 100 - 1000 bits



EPIC

- EPIC (Explicitly Parallel Instruction Computing)
  - An ISA philosophy/approach (IA-64 - Itanium)
- Very closely related to but not the same as VLIW
- **Similarities**:
  - Compiler generated wide instructions
  - Static detection of dependencies
  - ILP encoded in the binary (a group)
  - Large number of architected registers
- **Differences**:
  - Instructions in a **bundle** can have dependencies
    - Hardware interlock between dependent instructions
  - Allows dynamic scheduling and functional unit binding
  - Accommodates varying number of functional units and latencies

Superscalar v. VLIW

Attributes	Superscalar	VLIW
Multiple instructions/cycle	yes	yes
Multiple operations/instruction	no	yes
Instruction stream parsing	yes	no
Run-time analysis of register dependencies	yes	no
Run-time analysis of memory dependencies	maybe	occasionally
Runtime instruction reordering	maybe (Resv. Stations)	no
Runtime register allocation	maybe (renaming)	maybe (iteration frames)

Intel/HP "Explicitly Parallel Instruction Computer (EPIC)"

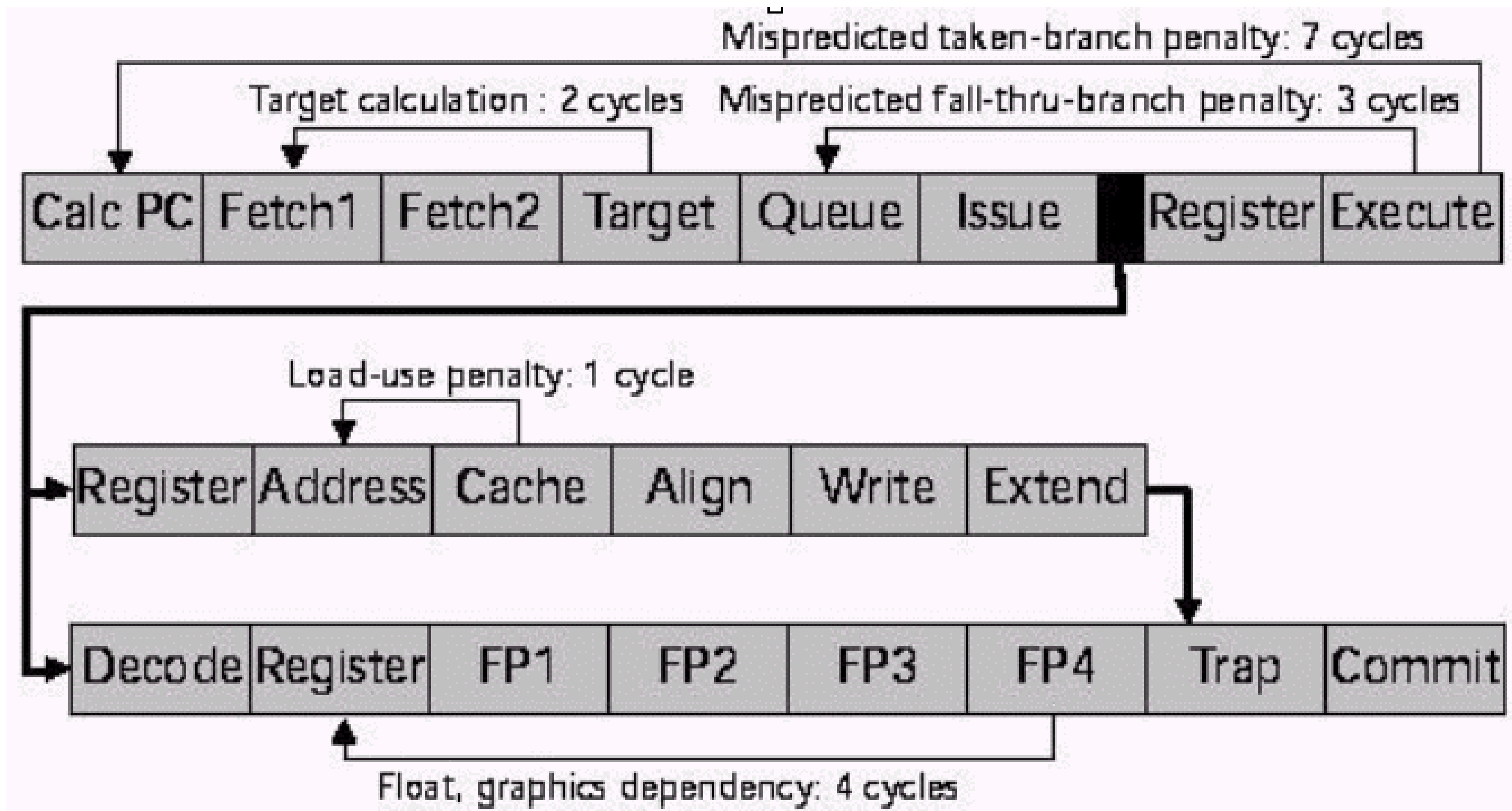
- 3 Instructions in 128 bit "groups"
- Field determines if instructions dependent or independent
  - Smaller code size than old VLIW, larger than x86/RISC
  - Groups can be linked to show independence > 3 instr
- 64 integer registers + 64 floating point registers
  - Not separate files per funcitonal unit as in old VLIW
- Hardware checks dependencies
  (interlocks => binary compatibility over time)
- Predicated execution (select 1 out of 64 1-bit flags)
  => 40% fewer mispredictions?
- IA-64 : name of instruction set architecture; EPIC is type
  - Merced is name of first implementation (1999/2000?)
  - LIW = EPIC?

Architectural Trends

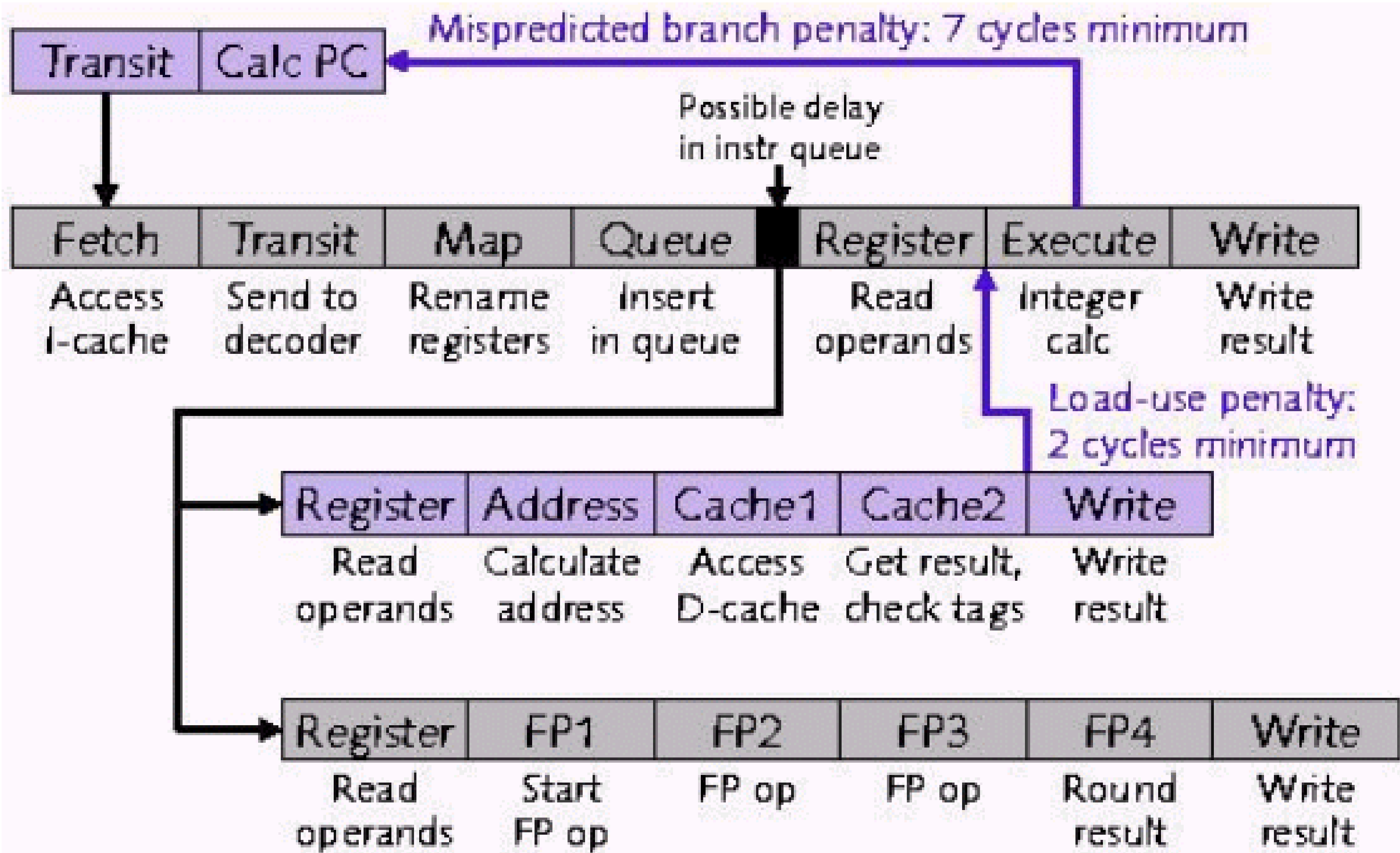
- Simultaneous Multi-Threading (SMT) processors
  - Thread-level parallelism (TLP) as opposed to instruction-level one (ILP)
- Memory-centric processors (I-RAM)
  - Vector-processor ISA organized around a large amount of on-chip memory (not only caches)
- Super-speculative processors
  - Superscalar architectures bringing speculation and OOO to the extreme
- Trace processors
  - Execute pre-fetched sequences of instructions viewed as a trace

Examples of real pipelines

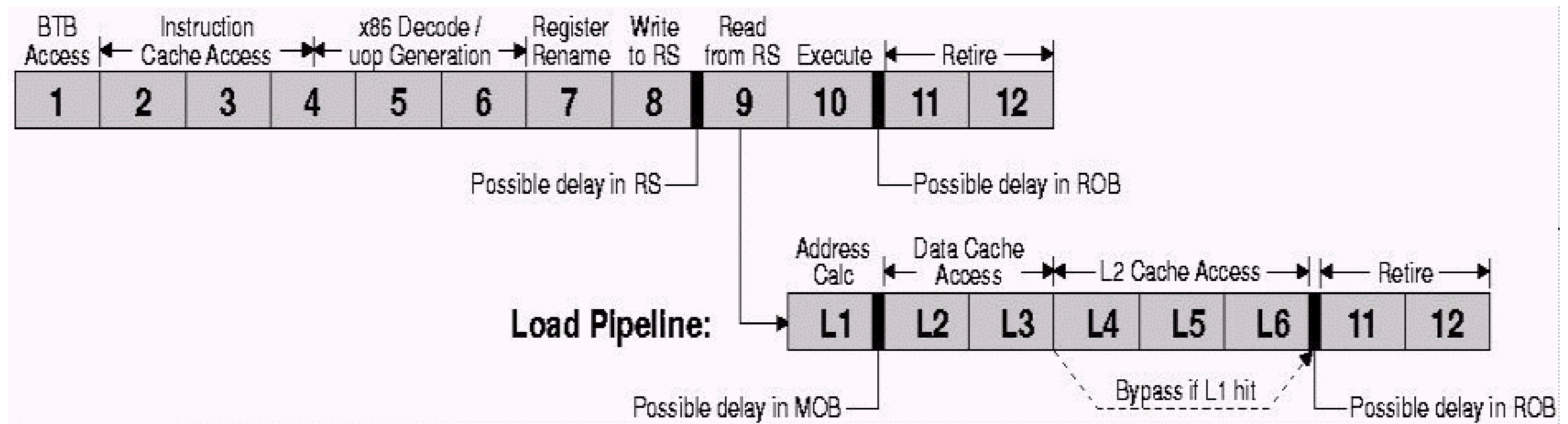
UltraSPARC pipeline



Alpha pipeline



PentiumPro pipeline



Summary

- Interrupts, Instruction Set, FP makes pipelining harder
- Hardware can improve that