Issues that Pipelines Complicate

Exceptions (or interrupts)

Multi-cycle operations

Classifying Interrupts

Synchronous or asynchronous

User-requested or caused by instruction execution

User maskable or not

Occurs between instructions or within instructions

Can resume execution after the interrupt or the program terminates

(Figure 3.40 is a quick review)

Interrupts

Handling interrupts that occur in the middle of executing an instruction & after which the program is resumed:

- interrupts occur in the middle of an instruction
- must restart interrupting & subsequent instructions

What happens on a ***precise*** interrupt

- ***identify the instruction that caused the interrupt***
- ***instructions before faulting instruction finish***
- disable writes for faulting & subsequent instructions
- force trap instruction into pipeline
- trap routine
  - save the state of the executing program
  - correct the cause of the interrupt
  - restore program state
- ***restart faulting & subsequent instructions***

How Pipelines Complicate Interrupts

Delay slots

```
conditional branch
delay slot instruction:  interrupt!
next instruction
```

- solution: PC/pipe stage

Simultaneous interrupts

- solution: handle them in program order
- still precise

Interrupts out of order wrt sequential instruction execution

- interrupts still must be handled in program order (for precise interrupts)
- interrupt handled before WB (S)

  solution: interrupt recorded in a per-instruction bit vector

  interrupts for *this* instruction handled before any state change, earliest stage first
- interrupt handled immediately

  solution: restart ***all*** instructions

How Pipelines Complicate Interrupts

Some ISAs change state before the stage after which no interrupt can occur

- autoincrement (VAX)
  - solution: history of register values so can back out
- separation of compare (set CCs) & branch
  - solution: CCs/pipe stage
- string copy operations (VAX, IBM 360)
  - solutions:
    - registers for copy addresses & length (IBM 360, VAX, x86)
    - touch pages before do the copy (IBM 360)

Which solutions preserve precise interrupts?

Multicycle Operations

Types of multicycle operations

- **memory accesses that miss in first level cache**
- **floating point operations**
- **integer multiply & divide**

Implementations

- **hardware interlocks**
- **lockup-free caches**
- **separate pipelines with**
  - **multiple functional units**
  - **pipelined functional units**

Consequences (negative)

- more stalls for RAW hazards
- additional forwarding requirements
- additional hazard detection (even structural & WAW)
- imprecise interrupts because operations don't necessarily complete in program-generated order

Problems with Multicycle Operations

Structural hazards

- not fully pipelined functional units
- multiple writes in the same cycle when instructions finish in a different order than their issue order
  - solution: track use of the write port & stall instructions in the decode stage if a conflict
    - set the bit in a shift register that is the number of cycles until the register file will be written
    - if this instruction's bit is set, stall

    adv: check for all structural hazards in 1 place

    disadv: shift register & write conflict logic
  - solution: detect the hazard when it occurs
    - stall before the common final stages

    adv: easier to prioritize access

    disadv: check for structural hazards in 2 places
  - solution: instruction scheduling

Problems with Multicycle Operations

Data hazards

- additional **RAW** hazards in the FP pipeline
- **RAW** hazard between integer & FP instructions
- **WAW** hazard
  - stall second instruction at ID
  - prevent write of first instruction
  - dead assignment elimination
  - this code could be generated in a machine with a branch delay slot

	`bnez r1, target`	
	`divd f0, _, _`	fall through code
`target:`	`lw f0, _`	but branch was taken

  - other real examples come with out-of-order execution
- **WAR** hazard not a problem: why?

Problems with Multicycle Operations

Imprecise interrupts

- cannot restart interrupting & subsequent instructions because some have completed

- problem:
  - caused by out-of-order completion

    instruction completion order different from issue order

divf	FP exception
multf	not done
addf	completed

completed instructions may have changed state

Problems with Multicycle Operations

Imprecise interrupts

- solutions
  - ignore the imprecise interrupt (IBM 360)
  - precise mode/imprecise mode (A)
  - buffer writes until instructions complete

    future file or shadow register; today called register renaming(M)

    history file (VAX)

    costs: additional registers & bypass logic to FUs
  - recreate preciseness in a trap handler (S)

    costs: software complexity
  - stall until no previous instruction will fault (X)

ISA & Pipelines

How instruction sets complicate pipelines:

- variable cycle instructions
  - cause:
    - more complicated hazard detection
    - more forwarding logic
    - more stalling
    - harder to implement precise interrupts
- complex addressing modes
  - cause:
    - WAR hazards from autoincrement
    - same problems as above from multiple memory accesses
- self-modifying code (X)
  - cause pipeline flush on address match
- implicitly set condition codes
  - cause poor code scheduling
- delayed branches

Intel P6 (Pentium Pro)

common fetch & decode pipeline **(fetch unit)**

1 BTB access
2–4 instruction fetch & align for decoding int μops *(later)*
4–6 decode & μop generation *(later)*
7 register renaming
8 write μops to reservation station
possible stalling before issue

integer pipeline

execution unit

9 issue μops to ALUs
10 execute, resolve branch
possible stalling before commit

commit unit

11 write registers &
12 retire

Intel P6

common fetch & decode pipeline (fetch unit)

Stage	
1	BTB access
2	
3	instruction fetch & align for decoding *(later)*
4	
5	decode & μop generation *(later)*
6	
7	register renaming
8	write μops to reservation station
	possible stalling before issue

load pipeline

Unit	Stage	
execution unit	9	issue μops to LD/ST units
	10	calculate effective address
		possible stalling before access
cache access	11	
	12	L1 data cache access
	13	
	14	L2 data cache access
		(bypass if L1 hit)
	15	
		possible stalling before commit
commit unit	16	write registers &
	17	retire

UltraSPARC 1

Integrated integer & (partial) floating point pipeline

Stage	
F	fetch
D	decode
G	group & dispatch; read registers

FP pipeline		integer pipeline	
more decode read registers	R	E	execute calculate effective address
execute	X_1	C	access cache & TLB resolve branches
execute	X_2	N_1	hit/miss?
execute	X_3	N_2	stall for FP pipeline

Stage	
N_3	resolve traps
W	write registers