Together with Control Hazards …

In addition to solving the problem of branching within a pipeline, we must solve …

Interrupts: asynchronous event (e.g., I/O)

- ❖ Occurrence of an interrupt checked at every cycle
- ❖ If an interrupt has been raised, don't fetch next instruction, drain the pipe, handle the interrupt

Exceptions (e.g., arithmetic overflow, page fault etc.)

- ❖ Program and data dependent (repeatable), hence "synchronous"

1

Exceptions

Occur "within" an instruction, for example:

- ❖ During IF: page fault
- ❖ During ID: illegal opcode
- ❖ During EX: division by 0
- ❖ During MEM: page fault; protection violation

Handling exceptions

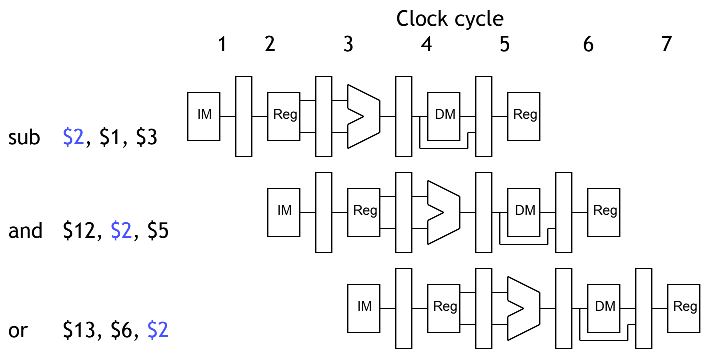
- ❖ A pipeline is *restartable* if the exception can be handled and the program restarted w/o affecting execution

2

Precise exceptions

If exception at instruction *i* then

- ❖ Instructions *i-1, i-2 etc* complete normally (drain the pipe)
- ❖ Instructions *i+1, i+2 etc.* already in the pipeline will be "no-oped" and will be restarted from scratch after the exception has been handled



3

Handling Precise Exceptions

Handling precise exceptions: Basic idea

- ❖ Force a trap instruction on the next IF (i.e., transfer of control to a known location in the O.S.)
- ❖ Turn off writes for all instructions *i* and following
- ❖ When the target of the trap instruction receives control, it saves the PC of the instruction having the exception
- ❖ After the exception has been handled, an instruction "return from trap" will restore the PC

4

Exception Handling

When an exception occurs

- ❖ Address (PC) of offending instruction saved in Exception Program Counter (a register not visible to ISA).
  - In MIPS should save PC – 4
- ❖ Transfer control to OS

OS handling of the exception. Two methods

- ❖ Register the cause of the exception in a status register which is part of the state of the process
- ❖ Transfer to a specific routine tailored for the cause of the exception; this is called vectored interrupts

5

Exception Handling (continued)

OS saves the state of the process (registers etc.)

OS "clears" the exception

- ❖ Can decide to abort the program
- ❖ Can "correct" the exception
- ❖ Can perform useful functions (e.g. I/O interrupt, syscall etc.)

Return to the running process

- ❖ Restores state
- ❖ Restores PC

6

Precise exceptions (continued)

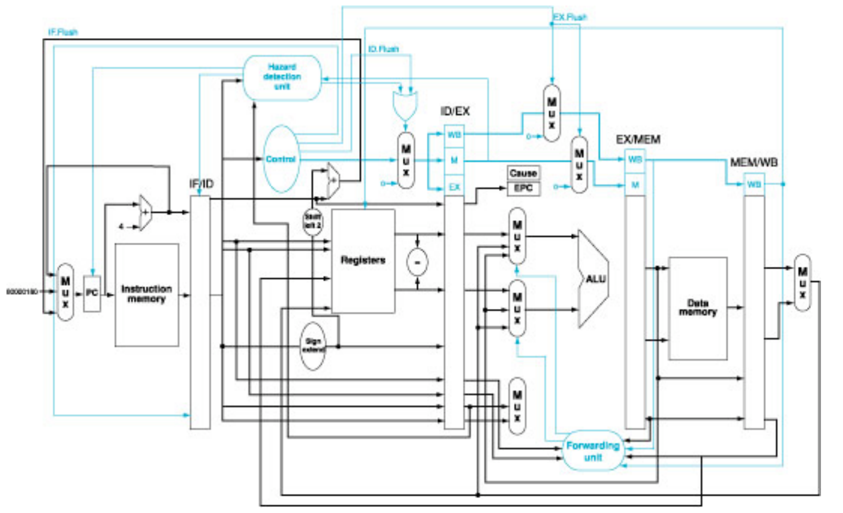
Relatively simple for integer pipeline

- ❖All current machines have precise exceptions for integer and load-store operations

Can lead to loss of performance for pipes with multiple cycles execution stage

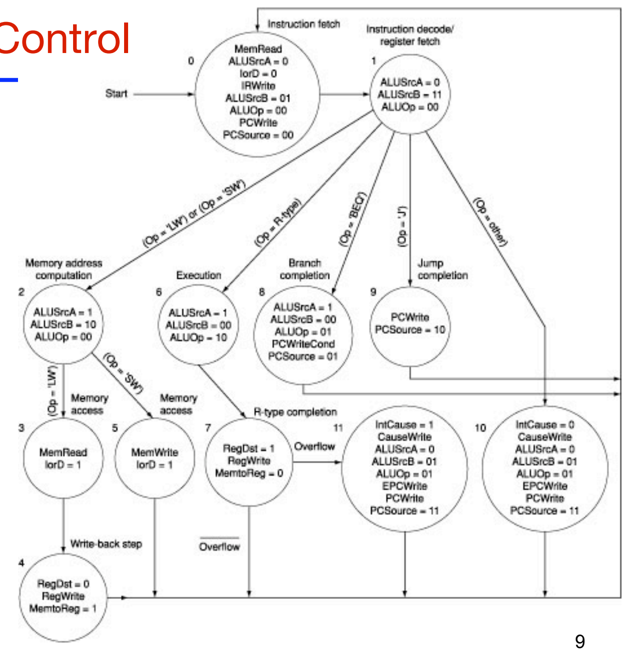
7

Exception Support



8

Exception Control



9

Integer pipeline precise exceptions

Recall that exceptions can occur in all stages but WB

Exceptions must be treated in *instruction order*

- ❖Instruction *i* starts at time *t*
- ❖Exception in MEM stage at time *t* + *3* (treat it first)
- ❖Instruction *i* + 1 starts at time *t* +*1*
- ❖Exception in IF stage at time *t* + *1* (occurs earlier but treat it 2nd)

10

Treating exceptions in order

Use pipeline registers

- ❖ Status vector of possible exceptions carried along with the instruction
- ❖ Once an exception is posted, no writing (no change of state; easy in integer pipeline -- just prevent store in memory)
- ❖ When an instruction leaves MEM stage, check for exception

11

Difficulties in less RISCy environments

Due to instruction set ("long" instructions")

- ❖ String instructions (but use of general registers to keep state)
- ❖ Instructions that change state before last stage (e.g., autoincrement mode in Vax and *update addressing* in Power PC) and these changes are needed to complete inst. (require ability to back up)

Condition codes (another way to handle branches)

- ❖ Must remember when last changed

12

CPI

The average number of clock cycles per instruction, or CPI, is a function of the machine and program.

- ❖ The CPI depends on the actual instructions appearing in the program—a floating-point intensive application might have a higher CPI than an integer-based program
- ❖ It also depends on the CPU implementation. For example, a Pentium can execute the same instructions as an older 80486, but faster

We often assume each instruction takes one cycle, so we assume CPI = 1.

- ❖ The CPI can be >1 due to memory stalls and slow instructions
- ❖ The CPI can be <1 on machines that execute more than 1 instruction per cycle (superscalar)

13

Recall Clock Cycle Facts

One “cycle” is the minimum time it takes the CPU to do any work.

- ❖ The clock cycle time or clock period is just the length of a cycle
- ❖ The clock rate, or frequency, is the reciprocal of the cycle time

Generally, a higher frequency is better.

Some examples illustrate some typical frequencies

- ❖ A 500MHz processor has a cycle time of 2ns.
- ❖ A 2GHz (2000MHz) CPU has a cycle time of just 0.5ns (500ps)

14

Execution time, again

CPU $\text{time}_{X,P}$ = Instructions executed_P * $\text{CPI}_{X,P}$ * Clock cycle time_X

The easiest way to remember this is match up units:

$$\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Clock cycles}}{\text{Instructions}} * \frac{\text{Seconds}}{\text{Clock cycle}}$$

Make things faster by making any component smaller!!

	Program	Compiler	ISA	Organization	Technology
Instruction Executed					
CPI					
Clock Cycle TIme					

Often easy to reduce one component by increasing another

15

Example 1: ISA-compatible processors

Let's compare the performances two 8086-based processors.

- ❖ An 800MHz AMD Duron, w/ a CPI of 1.2 for MP3 compress
- ❖ A 1GHz Pentium III with a CPI of 1.5 for the same program

Compatible processors implement identical instruction sets and will use the same executable files, with the same number of instructions

But they implement the ISA differently, which leads to different CPIs

CPU $\text{time}_{AMD,P}$= Instructions_P * $\text{CPI}_{AMD,P}$ * Cycle time_{AMD}

=

CPU $\text{time}_{P3,P}$ = Instructions_P * $\text{CPI}_{P3,P}$ * Cycle time_{P3}

=

16

Example 2: Comparing across ISAs

Intel's Itanium (IA-64) ISA is designed to facilitate executing multiple instructions per cycle. If an Itanium processor achieves an average CPI of .3 (3 instructions per cycle), how much faster is it than a Pentium4 (which uses the x86 ISA) with an average CPI of 1?

a) Itanium is three times faster
b) Itanium is one third as fast
c) Not enough information

17

Improving CPI

Many processor design techniques improve CPI

❖ Often they only improve CPI for certain types of instructions

$$\text{CPI} = \sum_{i=1}^{n} \text{CPI}_i \times F_i \qquad \text{where } F_i = \frac{I_i}{\text{Instruction Count}}$$

F_i = Fraction of instructions of type i

- First Law of Performance:

Make the common case fast

18

Example: CPI improvements

Base Machine:

Op Type	Freq (fi)	Cycles	CPIi
ALU	50%	3	
Load	20%	5	
Store	10%	3	
Branch	20%	2	

How much faster would the machine be if:

- ❖ we added a cache to reduce average load time to 3 cycles?
- ❖ we added a branch predictor to reduce branch time by 1 cycle?
- ❖ we could do two ALU operations in parallel?

19

Amdahl's Law

Amdahl's Law states that optimizations are limited in their effectiveness.

$$\text{Execution time after improvement} = \frac{\text{Time affected by improvement}}{\text{Amount of improvement}} + \text{Time unaffected by improvement}$$

For example, doubling the speed of floating-point operations sounds like a great idea. But if only 10% of the program execution time T involves floating-point code, then the overall performance improves by just 5%.

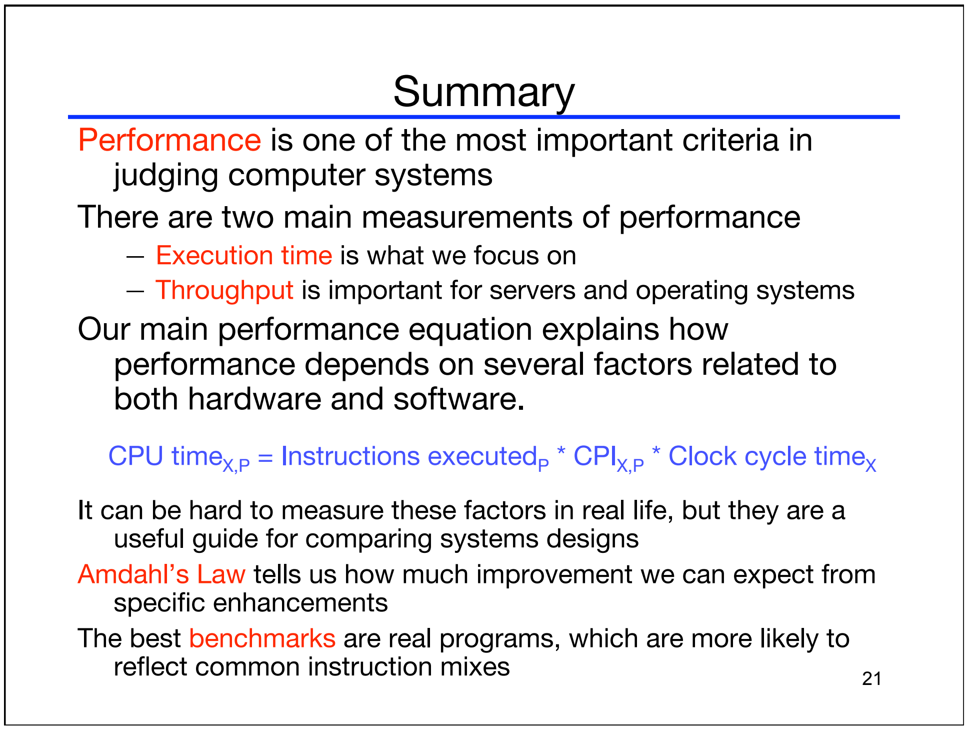
$$\text{Execution time after improvement} = \frac{0.10\,T}{2} + 0.90\,T = 0.95\,T$$

- Second Law of Performance:

Make the fast case common

What's the max speedup from improving floating point?

20



Summary

Performance is one of the most important criteria in judging computer systems

There are two main measurements of performance

- Execution time is what we focus on
- Throughput is important for servers and operating systems

Our main performance equation explains how performance depends on several factors related to both hardware and software.

$$\text{CPU time}_{X,P} = \text{Instructions executed}_{P} * \text{CPI}_{X,P} * \text{Clock cycle time}_{X}$$

It can be hard to measure these factors in real life, but they are a useful guide for comparing systems designs

Amdahl's Law tells us how much improvement we can expect from specific enhancements

The best benchmarks are real programs, which are more likely to reflect common instruction mixes

21