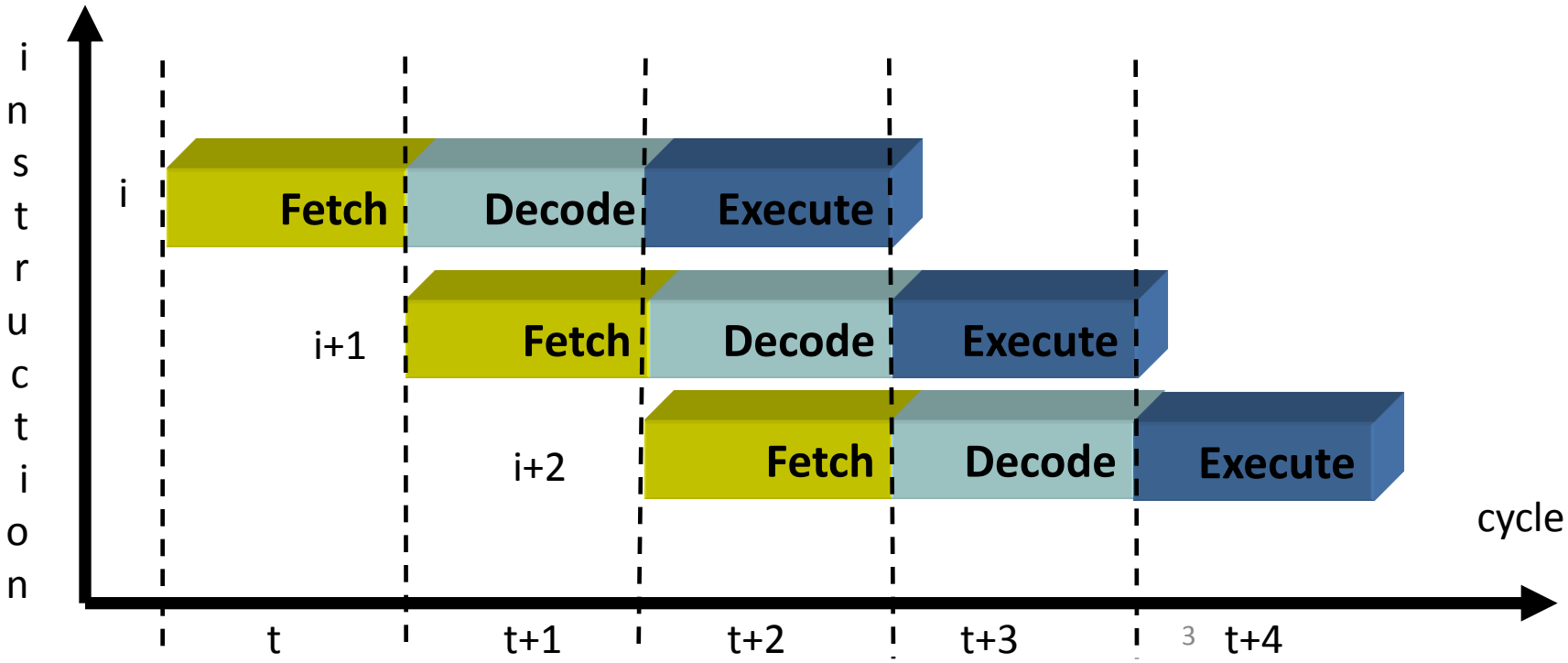
RISC (reduced instruction set computer) & Performance

Recall Key Features of RISC

- Limited and simple instruction set
- Memory access instructions limited to memory <-> registers
- Operations are register to register
- Large number of general purpose registers
  (and use of compiler technology to optimize register use)
- Emphasis on optimising the instruction pipeline
  (& memory management)
- Hardwired for speed (no microcode)

Pipeline Organization

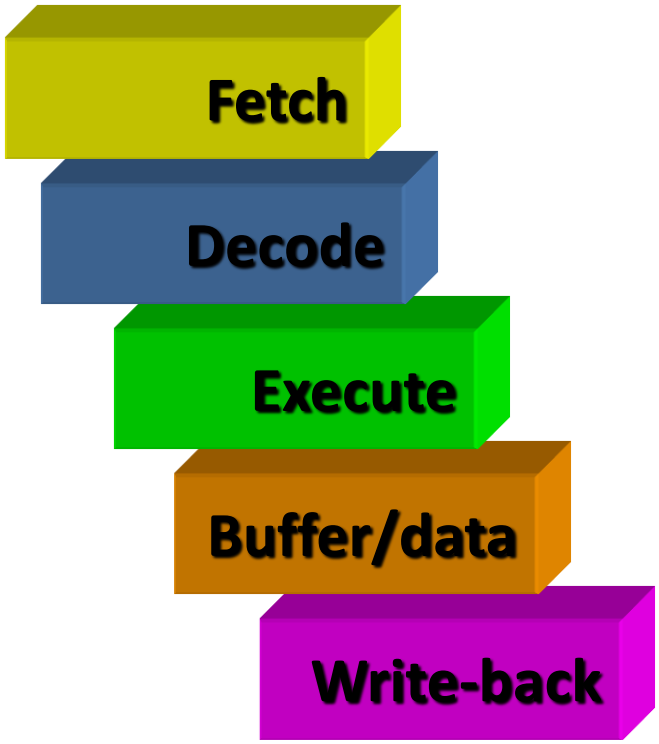
- 3-stage pipeline: Fetch – Decode - Execute
- Three-cycle latency,
  one instruction per cycle throughput



Pipeline Organization

- 5-stage pipeline:
  - Reduces work per cycle => allows higher clock frequency
  - Separates data and instruction memory => reduction of CPI (average number of clock Cycles Per Instruction)

- Stages:



Pipelining Review

Pipelining:

- Break instruction cycle into *n* phases (one stage per phase)
  - e.g. Fetch, Decode, ReadOPs, Execute1, Execute2, WriteBack
- Fetch a new instruction each phase
- Maximum speed gain is *n*
- Hazards reduce the ability to achieve a gain of *n*
  - Types of Hazards
    - Resource
      - » Hazard occurs when instruction needs a resource being used by another instruction
    - Data
      - » RAW (hazard if read can occur before write has finished)
      - » WAR (hazard if write can occur before read is finished)
      - » WAW (hazard if writes occur in the unintended order)
    - Control
      - » Hazard occurs when a wrong fetch decision at a branch results in an extra instruction fetch and a pipeline flush
- Stalling can always “fix” a hazard

Data Hazards

- Read after Write (RAW) – true dependency
  - A Hazard occurs if the Read occurs before the Write is complete
    - e.g. Reg 1 ← Reg 1 + Reg 2 {write occurs after execution}
      Reg 3 ← reg 1 – Reg 3 {read occurs before execution}

- Write after Read (WAR) – anti-dependency
  - A Hazard occurs if the Write occurs before the Read happens
    - e.g. Reg ← M(ptr) {2 memory accesses – long read} **{M(ptr) & M(pc) are same loc}**
      M(pc) ← Reg {1 memory access – short write}

- Write after Write (WAW) – output dependency
  - A Hazard occurs if the two Writes occur in the reverse order than intended
    - e.g. Reg A ← M(PTR) {2 memory accesses – long write}
      Reg A ← Reg B {0 memory accesses – short write}

Control Hazard

Control Hazards occur when a wrong fetch decision results in a new instruction fetch and the pipeline being flushed

Solutions include:

- Multiple Pipeline streams
- Prefetching the branch target
- Using a Loop Buffer
- Branch Prediction
- Delayed Branch
- Reordering of Instructions
- Multiple Copies of Registers (backups)

RISC Pipelining

- Most instructions are register to register
- Arithmetic/logic instruction:
  - I: Instruction fetch
  - E: Execute (ALU operation with register input and output)
- Load/store instruction:
  - I: Instruction fetch
  - E: Execute (calculate memory address)
  - D: Memory (register to memory or memory to register operation)

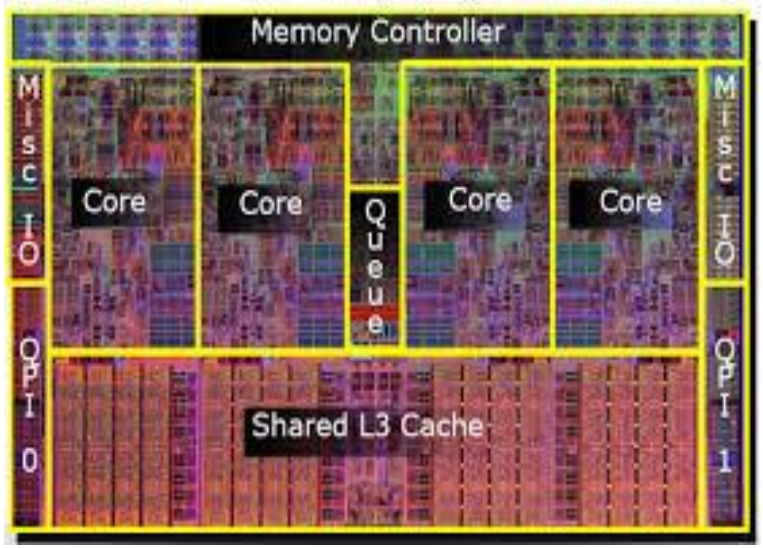
Delay Slots in the Pipeline

Sequential	1	2	3	4	5	6	7	8	9	10	11
LOAD rA, m1	I	E	D								
LOAD rB, m2				I	E	D					
ADD rC, rA, rB							I	E			
STORE m3, rC									I	E	D

Pipelined	1	2	3	4	5	6	7
LOAD rA, m1	I	E	D				
LOAD rB, m2		I	E	D			
ADD rC, rA, rB			I		E		
STORE m3, rC				I		E	D

Cache memory

- L1 – on chip (very close to the CPU), size – 32-128 bytes, cycles per operation 3-4
- L2 – on chip (close to the CPU), size – 256-512 bytes, cycles per operation 10-11
- L3 – on chip shared memory, size – 3-8 Mbytes, cycles per operation 30-40



Instruction format

- Is a definition
  - how instructions are represented in binary code
- Instructions typically consist of
  - Opcode (Operation Code)
    - defines the operation (e.g. addition)
  - Operands
    - what's being operated on
- Instructions typically have 0, 1, or 2 operands

Alignment

- Often multiple bytes can be fetched from memory
- Alignment specifies how the (beginning) address of a multiple-byte data is determined.
  - data must be aligned in some way. For example
    - 4-byte words starting at addresses 0,4,8, …
    - 8-byte words starting at addresses 0, 8, 16, …
- Alignment makes memory data accessing more efficient

Example

- A hardware design that has data fetched from memory every 4 bytes

- Fetching an unaligned data (as shown) means to access memory twice.

Performance

Computer Performance: TIME, TIME, TIME

- Response Time (latency)
  - How long does it take for my job to run?
  - How long does it take to execute a job?
  - How long must I wait for the database query?
- Throughput
  - How many jobs can the machine run at once?
  - What is the average execution rate?
  - How much work is getting done?

- *If we upgrade a machine with a new processor what do we increase?*
- *If we add a new machine to the lab what do we increase?*

Execution Time

- Elapsed Time
  - counts everything *(disk and memory accesses, I/O , etc.)*
  - a useful number, but often not good for comparison purposes
- CPU time
  - doesn't count I/O or time spent running other programs
  - can be broken up into system time, and user time

- Our focus: user CPU time
  - time spent executing the lines of code that are "in" our program

Book's Definition of Performance

- For some program running on machine X,

  $$\text{Performance}_X = 1 / \text{Execution time}_X$$

- "X is n times faster than Y"

  $$\text{Performance}_X \ / \ \text{Performance}_Y = n$$

- Example:
  - machine A runs a program in 20 seconds
  - machine B runs the same program in 25 seconds

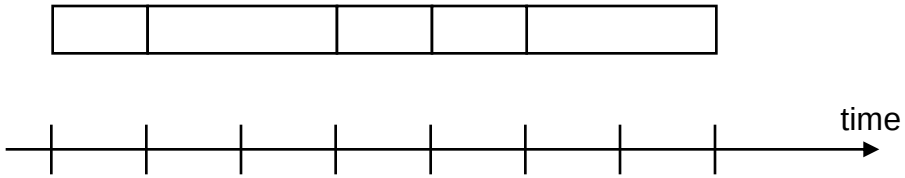
Principal Design Metrics: CPI and Cycle Time

$$Performance = \frac{1}{ExecutionTime}$$

$$Performance = \frac{1}{totalCPI \times CycleTime}$$

$$Performance = \frac{1}{\sum \frac{Cycles}{Instructions} \times \frac{Seconds}{Cycle}} = \frac{Instructions}{Seconds}$$

Different numbers of cycles for different instructions



- Multiplication takes more time than addition
- Floating point operations take longer than integer ones
- Accessing memory takes more time than accessing registers

- *Important point: changing the cycle time often changes the number of cycles required for various instructions (more later)*

Example

- Our favorite program runs in 10 seconds on computer A, which has a 400 Mhz. clock. We are trying to help a computer designer build a new machine B, that will run this program in 6 seconds. The designer can use new (or perhaps more expensive) technology to substantially increase the clock rate, but has informed us that this increase will affect the rest of the CPU design, causing machine B to require 1.2 times as many clock cycles as machine A for the same program. What clock rate should we tell the designer to target?"

- Don't Panic, can easily work this out from basic principles

Now that we understand cycles

- A given program will require
  - some number of instructions (machine instructions)
  - some number of cycles
  - some number of seconds
- We have a vocabulary that relates these quantities:
  - cycle time (seconds per cycle)
  - clock rate (cycles per second)
  - CPI (cycles per instruction)

    a floating point intensive application might have a higher CPI
  - MIPS (millions of instructions per second)

    this would be higher for a program using simple instructions

Performance

- Performance is determined by execution time
- Do any of the other variables equal performance?
  - number of cycles to execute program?
  - number of instructions in program?
  - number of cycles per second?
  - average number of cycles per instruction?
  - average number of instructions per second?

- Common pitfall: thinking one of the variables is indicative of performance when it really isn't.

CPI Example

- Suppose we have two implementations of the same instruction set architecture (ISA).

  For some program,

  Machine A has a clock cycle time of 10 ns. and a CPI of 2.0
  Machine B has a clock cycle time of 20 ns. and a CPI of 1.2

  What machine is faster for this program, and by how much?

of Instructions Example

- A compiler designer is trying to decide between two code sequences for a particular machine. Based on the hardware implementation, there are three different classes of instructions: Class A, Class B, and Class C, and they require 1, 2, and 3 CPI (respectively).

  The first code sequence has 5 instructions: 2 of A, 1 of B, and 2 of C
  The second sequence has 6 instructions: 4 of A, 1 of B, and 1 of C.

  Which sequence will be faster? How much?
  What is the CPI for each sequence?

MIPS example

- Two different compilers are being tested for a 100 MHz. machine with three different classes of instructions: Class A, Class B, and Class C, which require 1, 2, and 3 CPI (respectively). Both compilers are used to produce code for a large piece of software.

  The first compiler's code uses 5 million Class A instructions, 1 million Class B instructions, and 1 million Class C instructions.

  The second compiler's code uses 10 million Class A instructions, 1 million Class B instructions, and 1 million Class C instructions.

- Which sequence will be faster according to MIPS?
- Which sequence will be faster according to execution time?

Benchmarks

- Performance best determined by running a real application
  - Use programs typical of expected workload
  - Or, typical of expected class of applications
    e.g., compilers/editors, scientific applications, graphics, etc.
- Small benchmarks
  - nice for architects and designers
  - easy to standardize
  - can be abused
- SPEC (System Performance Evaluation Cooperative)
  - companies have agreed on a set of real program and inputs
  - can still be abused (Intel's "other" bug)
  - valuable indicator of performance (and compiler technology)

Amdahl's Law

Execution Time After Improvement =

Execution Time Unaffected +(Execution Time Affected / Amount of Improvement)

- Example:

  "Suppose a program runs in 100 seconds on a machine, with
  multiply responsible for 80 seconds of this time. How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?"

  How about making it 5 times faster?

- *Principle: Make the common case fast*

Example

- Suppose we enhance a machine making all floating-point instructions run five times faster. If the execution time of some benchmark before the floating-point enhancement is 10 seconds, what will the speedup be if half of the 10 seconds is spent executing floating-point instructions?

- We are looking for a benchmark to show off the new floating-point unit described above, and want the overall benchmark to show a speedup of 3. One benchmark we are considering runs for 100 seconds with the old floating-point hardware. How much of the execution time would floating-point instructions have to account for in this program in order to yield our desired speedup on this benchmark?

Remember

- Performance is specific to a particular program/s
  - Total execution time is a consistent summary of performance

- For a given architecture performance increases come from:
  - increases in clock rate (without adverse CPI affects)
  - improvements in processor organization that lower CPI
  - compiler enhancements that lower CPI and/or instruction count

- Pitfall: expecting improvement in one aspect of a machine's performance to affect the total performance

- You should not always believe everything you read! Read carefully! (see newspaper articles, e.g., Exercise 2.37)

Floating Point (a brief look)

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g., 3.15576×10^9
- Representation:
  - sign, exponent, significand: $(-1)^{\text{sign}} \times \text{significand} \times 2^{\text{exponent}}$
  - more bits for significand gives more accuracy
  - more bits for exponent increases range
- IEEE 754 floating point standard:
  - single precision: 1 sign bit, 8 bit exponent, 23 bit significand
  - double precision: 1 sign bit, 11 bit exponent, 52 bit significand

IEEE 754 floating-point standard

- Leading “1” bit of significand is implicit
- Exponent is “biased” to make sorting easier
  - all 0s is smallest exponent all 1s is largest
  - bias of 127 for single precision and 1023 for double precision
  - summary: $(-1)^{\text{sign}} \times (1.\text{significand}) \times 2^{\text{exponent} - \text{bias}}$
- Example:
  - decimal: -.75 = -3/4 = $-3/2^2$
  - binary: -.11 = -1.1×2^{-1}
  - floating point: exponent = 126 = 01111110
  - IEEE single precision: `1 01111110 10000000000000000000000`