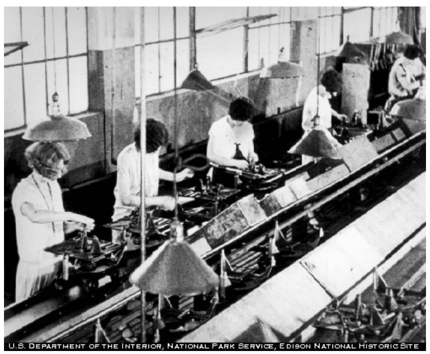
Chapter

MIPS Pipe Line



Pipelining

To complete an instruction a computer needs to perform a number of actions.

These actions may use different parts of the CPU.

Pipelining is when the parts run simultaneously on different instructions.

It is a vital technique in the quest for more powerful computers.

Clock rate is technology
Pipelining is the clever use of that technology.

Assembly line:

different stages are completing different steps on different objects.

Each stage is a **pipe stage** or **segment**
The pipeline connects them all.

Pipelining does not increase the speed at which the first instruction completes.

Pipelining increases the number of instructions which finish per second (in the steady state)

Pipelining predates the retreat from speed by Intel and AMD

Pipeline

Design overview

Pipeline is the core of the design
Only use hardware, where there are net performance gains.

Principles:

Simple instructions and few addressing modes:
CISC includes many ways to address memory. May need several parameters, uses microcode and may need several cycles just to calculate an address. RISC has simple address modes. Every instruction needs only one cycle per pipeline stage.

Direct, pointers, offset

Register-Register (Load/Store)design:
The only way registers and memory interact is via a load or store operation. All other operations involve only registers. CISC supports arithmetico-logic operations on memory

Pipeline

4 Principles

Principles:

Pipelining:
Multistage pipeline which allows the CPU to perform more than one instruction at a time. The predictability (and similarity) of the time for all instructions aids in creating an efficient pipeline.

Hardware control no (or a little) microcode:
No micro-coded ROM to execute complex instructions. All instructions directly in hardware for speed (and simplicity)

Not true of VAX, nor Inte

Reliance on optimising compilers:
Optimising Compilers don't just create low level instructions to implement the high level constructs.
Reorder instructions, use of the registers to minimise memory accesses. Simplicity of opcodes, consistency of timings and absence of complex addressing modes.

All ease problem of compiler writing

Pipeline

Principles:

High performance Memory Hierarchy:
Need to keep pace with CPU. Introduce memory/cache hierarchy including large number of registers
Fast static RAM split cache.

D-cache data cache
I-cache instruction cache

Write buffers – on chip memory management.

Notes

ALU connects to the buses and thence to the register file of 32 GPRs

Only load/store connect registers with the D-Cache

Instruction fetch: fetches a single instruction per cycle from the I-Cache at an address given by the PC.

Instruction fetch is controlled by the pipeline decode and control unit.

The PC is incremented by 4 after each fetch. (Byte addressable and 32 bit words). PC can be loaded by a jump or branch target address.

Aim: 1 instruction per clock cycle.
Achievement depends on cache and pipelining.

CPI measure of performance.

Clock

Assembly line all items pass onto the next stage at the same time.

Pipeline all instructions pass onto the next stage at the same time.

The time that each stage takes is a
Processor cycle

The cycle must leave time for the slowest stage to complete.

Need to balance the work done on each cycle.

Processor cycle is usually one *clock cycle* of the machine, sometimes two.

In a perfectly balanced pipeline the instruction throughput is just *p* times the unpipelined machine. Where p is the number of stages.

Pipeline overhead

Decrease in average time per completed instruction.

Often measure as **Clock Cycles per Instruction**

Needs no input from the programmer to work

Not more on RISC machines

The value will fill 32 or 64 bits.

RISC architecture

Not comprehensive … review

- Data Operations only on registers
- Only load and store operations on memory half and double word options
- Few instructions all 1 word
- 32 Integer **G**eneral **P**urpose **R**egisters (GPR)

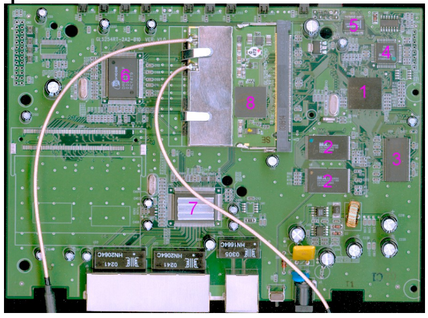
Instruction Set

ALU : Two registers to a third
Register & signed extended immediate
Ops include ADD, SUB, AND, OR.
Immediate versions of the ops
Signed and unsigned arithmetic ADDU

Load/ Store Register source (*base register)* and an immediate field (*offset).*
Sum makes the *effective address.*
Second register is the source or sink.

Branch/ Jumps Conditional transfer of control
Unconditional transfer.
Destination is a signed extended offset added to the **PC**

Immediate
Actual value #3



Simple Execution Cycle

1. Instruction Fetch (**IF**). Send the Program Counter (**PC**) to memory and fetch next instruction. Update PC by adding 4.

Instructions are 4 bytes

1. Instruction Decode (**ID**) / Register Fetch
    a. Decode the instruction
    b. Read the registers
    c. Equality test on registers as read
    d. Sign extend the offset field
    e. Compute possible branch target address by adding offset to PC

Things that can be done without penalty

In case required

In case required

Decode in parallel with Register, because the register specifiers are in a fixed place in the word ***fixed-field decoding***

Internal parallel

May not need it, but it takes no extra time.
Also calculate sign extended immediate.

1. Execution/effective address cycle (EX)
ALU operates on operands prepared in 2
    a. Memory ref: Base register + offset to give effective address
    b. Register-Register execute op code
    c. Register-Immediate execute op code

Branch: 2 cycles
Store: 4 cycles
Others; 5 cycles

1. Memory Access (MEM): Load, read using effective address, write the data from the second register using the first effective address from the first register

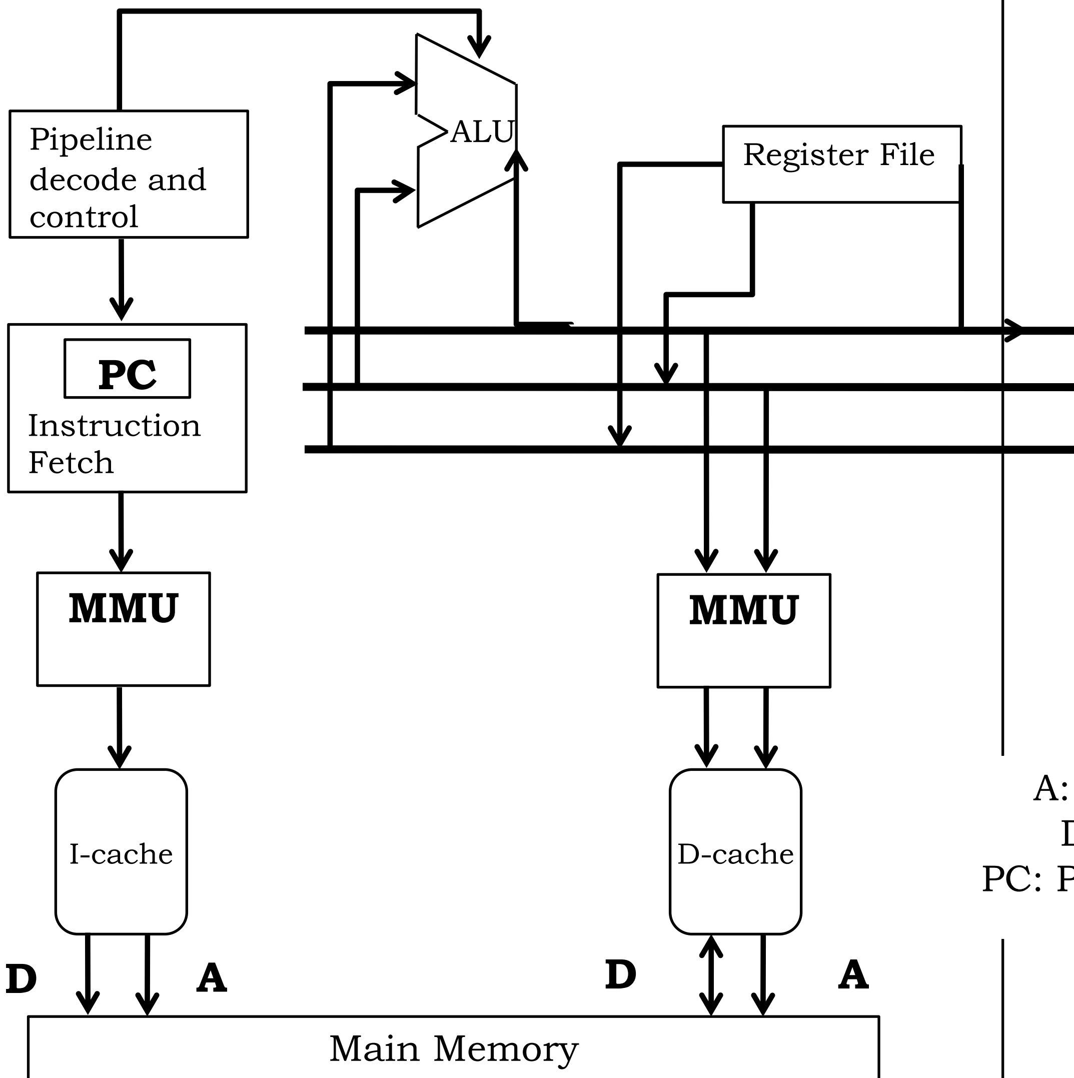
Write to cache

2. Write-back cycle (WB): Write the result into the register whether from the memory or the ALU

Pipeline

Block diagram

ALU only talks to the register buses



Data and instructions with separate paths and cache

A: Address Line
D: Data Line
PC: Program Counter

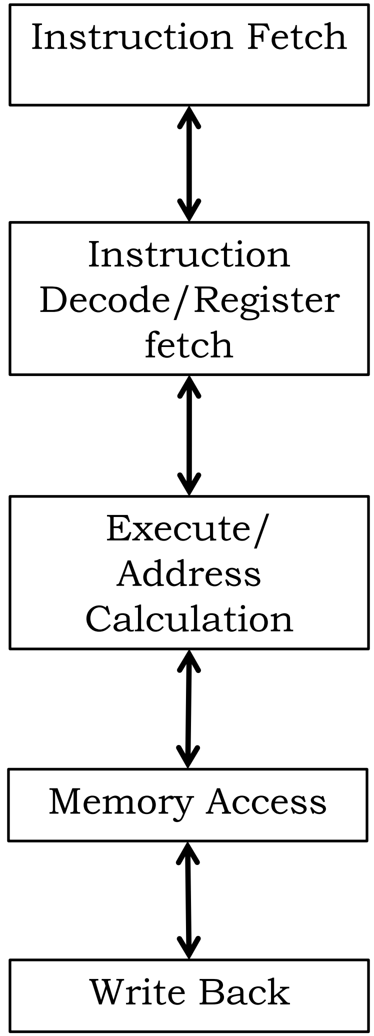
Pipeline

11 Pipeline

Overview

This pipeline has five stages.

n stage pipeline



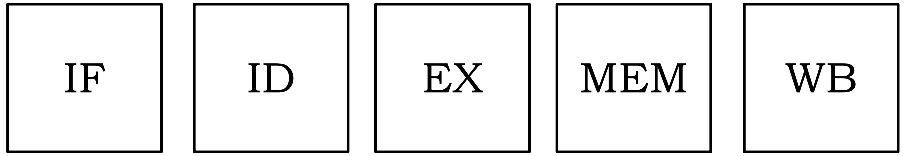
Each stage should take one cycle.
While the IF is fetching an instruction then Decode is decoding the previous instruction; and execute is doing the one before.

No pipeline – time for 1 instruction is k cycles.
For n instructions
non pipeline = $n*k$
Pipeline = k (for first instruction)
+(n-1) (for the other n-1) = $k+n-1$

Speed-up = $(n*k)/(k+n+1)$ =
$k/(k/n + (1+1/n)$ = k (as n goes to infinity)
Speed-up = number of stages
(ignoring stalls)

MIPS.
See H&P, P&H and
various other text books

Instruction stages



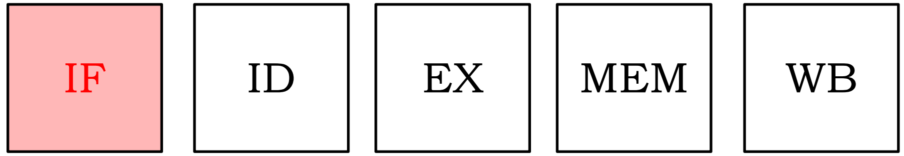
Look at an instruction architecture from the "pipeline"

It has 5 stages each one takes one clock cycle.

They are

IF	Instruction Fetch
ID	Instruction Decode
EX	Execute
MEM	Memory
WB	Write Back

Instruction Fetch



IF Instruction Fetch

Get the instruction from memory (or more usually cache)
Instructions often in I-cache. The cache for instructions (D-cache for data)

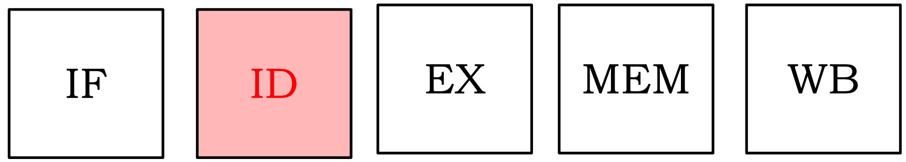
PC incremented by 4 – points to next instruction.

If there has been a branch or jump set the PC from that instruction.

Jump – non sequential alteration of the PC. Always taken.

Branch – non sequential alteration of the PC, conditionally taken on the basis of values in the registers.

Instruction decode & register fetch



ID — Instruction Decode

Different actions depending on the sort of instruction.

Register-Register

Memory reference

Control transfer

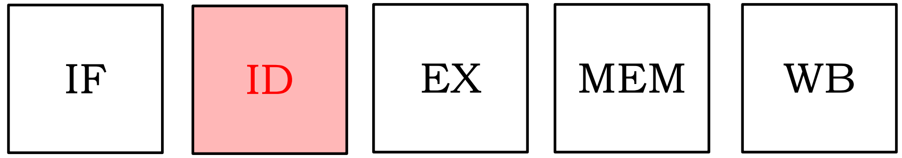
Register-Register: modify the values of a register depending on values in other registers.

and, or, add, sub

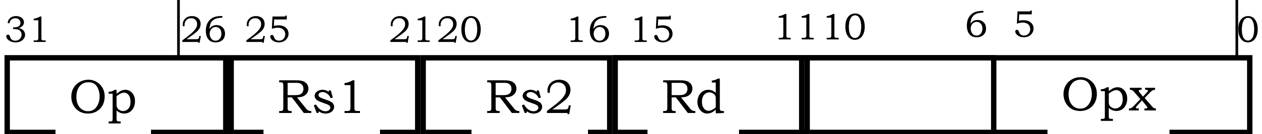
Memory reference: commonly in RISC machines and certainly here. *lw* load from memory to registers or *sw* store from register back to memory

Control transfer: jump or branch

Instruction decode & register fetch



Register-Register



Registers always in the same place.
Opcode always the same length.

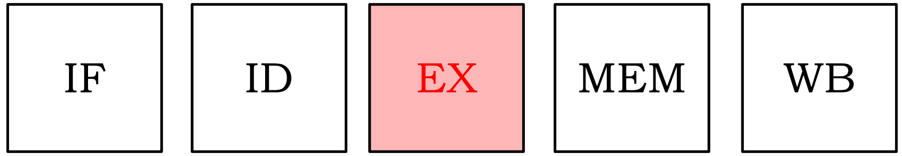
Can set up access to the registers, while decoding the instruction.

If registers not needed no penalty.
If needed already there.

Repeated theme, if you can do something without penalty, do it, even if not needed.

Note there is nearly always some penalty, even if it is only power

Execute



Here the ALU operates on the operands which have been prepared in the decode cycle

Memory reference:

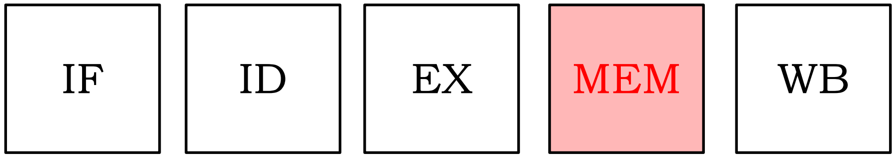
Calculates effective address by taking Base register and adding offset

Arithmetic

For register register op codes execute op code perform the arithmetic operation

Similarly for register

Memory Reference



Load or store accesses memory.
A-L writes result from ALUout register

Write Back

IF | ID | EX | MEM | WB

Write from place the memory reference placed the data, into the register

Step	**R-Type**	**Mem Ref**	**Branches**	**Jumps**
IF	Instruction Register⇐ Memory(PC) PC ⇐ PC + 4			
Instruction Decode/ Memory fetch	A ⇐ Reg[IR(25-21)] B ⇐ Reg[IR(20-16)] ALUOut ⇐ PC + signextend (IR(15:0))			
Execution	ALUOut ⇐ A op B	ALUOut ⇐ A + signextend IR(15:0)	if (A==B) PC ⇐ ALUout	PC ⇐ PC + IR(25:0) shift
Mem access R type comp	Reg(IR(15:11) ⇐ ALUout	Load MDR⇐ Mem[ALUout] Store memory[ALUout]) ⇐ B		
Mem read completion		Load Reg(IR(20:16) ⇐ MDR		Pipeline

Clock speed

Look at MIPS – used in Hennessey & Patterson (as well as Patterson & Hennessey) and other architecture books.

Clean architecture – not surprising designed by an academic

Easy to pipeline.

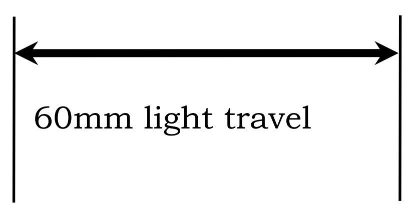
Start a new instruction on each clock cycle

Each cycle becomes a pipe *stage.*

Stage	IF	Register Read	ALU	Data	Register Write	Total
Time	200ps	100ps	200ps	200ps	100ps	800ps

Each stage must take the same time. So each stage must go as slow as the slowest.

Clock must be 200ps. (5 GHz)



20 Five stage pipe

Instruction time

Many instructions – follow H&P in looking at five types

Instruction	IF	Register Read	ALU	Data	Register Write	Total
Load Word *lw*	200ps	100ps	200ps	200ps	100ps	800ps
Store Word *sw*	200ps	100ps	200ps	200ps		700ps
Arithmetic *add,sub,*	200ps	100ps	200ps		100ps	600ps
Branch *beq*	200ps	100ps	200ps			500ps

Each stage must take the same time. So each stage must go as slow as the slowest.

Clock must be 200ps. (5 GHz)

All instructions need to take the same time, (single cycle) so the instructions with missing stages do nothing at that point in the pipeline.

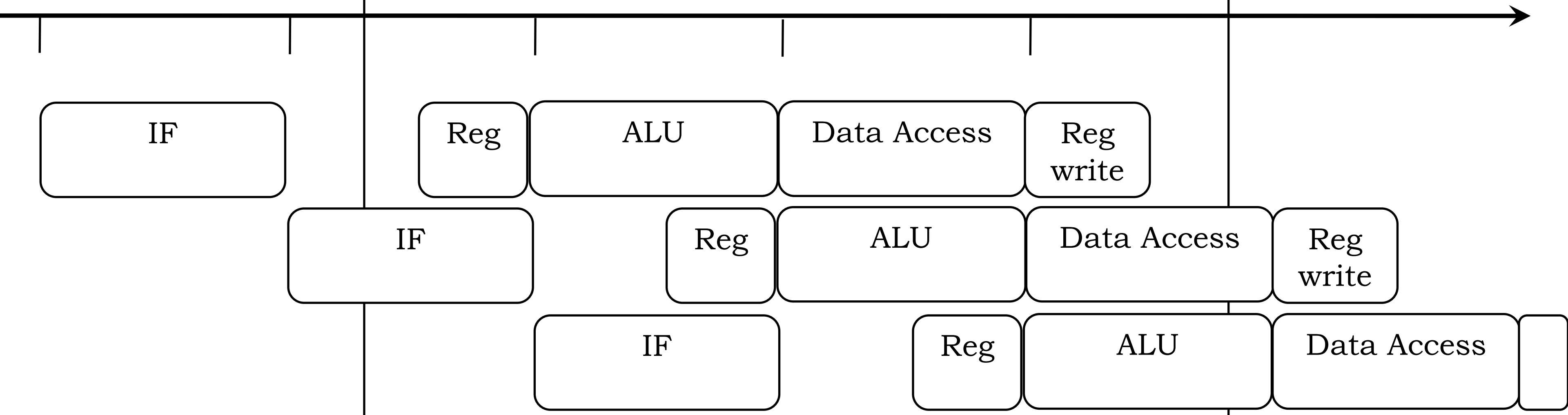
All instructions take 800ps.

Improves throughput **but** not latency

Non-pipelined instructions take 800ps each. Pipelined instructions finish every 200ps.

The speed up is approximately 1/stages. Assuming enough instructions to render the start up cost negligible (and no pipeline stalls).

Time flow



Ignoring stalls

In this case the first instruction takes 900ps, but the instruction rate is still one every 200ps.

n instructions take 800*n ps sequential.
900 + 200*n ps pipelined

$$\text{Speed up} = \frac{900 + 200}{800n} = \frac{1{,}125}{n} + 0.25 \rightarrow 0.25_{\ n\rightarrow\infty}$$

If the pipeline was perfectly balanced then the speed up for an k stage pipeline executing n instructions is k as n→∞

Designing for pipeline

MIPS all instructions the same length.

c.f. IA-32. Instructions vary in length. Would make pipelining very hard. **But** instructions translated to microcode. Microcode is MIPS like. Microcode is executed in a pipeline.
Complex to preserve backward compatibility

Source register in the same place in all instructions. Register file can be accessed as instruction is decoded.
Called “uniform decode”

If register position depends on instruction. Must decode first

Memory operands only appear in loads or stores. Can calculate the memory address here and access in following stage. Memory operands introduce an extra stage in the pipeline.

Operands must be aligned – transfers can always be completed in a single stage.