# Lec-02: Performance and Processor Architecture

**ECE-720t4: Innovations in Processor Design**

**Mark Aagaard**



## Schedule

| | | |
|---|---|---|
| 01 | Intro and Overview | In-order execution |
| 02 | Performance; Architecture | |
| 03 | Pipelining review | |
| 04 | Superscalar execution | O-o-O execution |
| 05 | Out-of-order execution: ctrl and data | |
| 06 | Out-of-order execution: reordering | |
| 07 | PowerPC 620 vs Intel P6 | Current practice |
| 08 | Processor Survey | |
| 09 | Processor Trends | |
| 10 | Data reuse and speculation | Recent advances |
| 11 | Multithreading concepts | |
| 12 | Multithreading techniques | |
| 13 | Review | |



## Outline of Lecture

- Historical definitions of performance
- Amdahl's law
- "Iron law" of computer performance
- Instruction set design
- Pipelining
- Instruction-level parallelism (not in lecture slides)



## Outline of Chapter 1

**Processor Design**

- 1.1 The Evolution of Microprocessors
- 1.2 Instruction-Set Processor Design
- 1.3 Principles of Processor Performance
- 1.4 Instruction-Level Parallel Processing
- 1.5 Summary

## Origins of Measuring Performance

- In early decades of computers, each new generation required a new technique to evaluate performance.

**Mid 1960s**

- Mainstream computers:
  - had reasonably similar instruction sets
  - each instruction took same length of time to perform
- Performance measure: time to perform a single instruction (e.g. add)

5

## Gibson Mix (1970)

- Computers evolved so that some instructions took longer to execute than others.
- Gibson proposed: time to execute an average instruction, based on weighted average of different instructions.
- Weights based on analyzing collection of typical programs
- Instruction frequency: static vs dynamic

6

## Whetstone (1973)

- Pipelining, caches, etc caused time to perform a single instruction dependent upon instructions before and after.
- Late 1960s: UK National Physical Laboratory benchmarked programs using the Whetstone interpreter for Algol 60.
- 1971: portable benchmark with real programs was becoming too time consuming
- Whetstone: synthetic benchmark program
  - Relatively short (quick and easy to run)
  - Reflected distribution and order of typical scientific program
- Measure was KWIPS, then KMIPS, then MIPS

7

## MIPS

- MIPS =
- Different types of MIPS
  - Fastest instruction (e.g. NOP)
  - Typical instruction (e.g. ADD)
  - Weighted average (Gibson, Whetstone, Dhrystone)
- Advantages of MIPS:
  - simpler to explain to management
  - bigger is better.

8

## Relative MIPS (1977)

- IBM marketing claimed IBM 370/158 was (first?) 1MIPS computer
- DEC VAX 11/780 developers ran programs on IBM 370/158 and on VAX 11/780
- Programs took same time on both computers, therefore performance of VAX 11/780 was 1 MIPS
- No one actually measured MIPS for VAX 11/780
- VAX 11/780 become de facto standard for 1 MIPS

- 1981: Joel Emer from DEC measured VAX 11/780 as 0.5 MIPS



## Dhrystone (1984)

- Synthetic benchmark program
- Focus on integer performance
- Created by Reinhold Weicker (Seimens AG)



## What Really Matters

- The real definition of performance is how long it takes to run your programs.



## SPEC Benchmarks

- Collection of real programs run for realistic lengths of time
- Updated roughly every 5 years
- Benchmarks for integer, floating point, graphics, multiprocessors, Java client/server, mail servers, network file systems, web servers.
- spec.org: great resource before you buy your next computer!

## Defining Equations for Performance

$$\text{Performance} = \frac{\text{Work}}{\text{Time}}$$

- To double performance
  - do the same amount of work in half the time
  - OR: do twice the work in the same amount of time
- Time is easy to measure
- Challenge is to define or measure work
- Benchmarketing is all about defining work to make your product appear faster than your competitors'



## Tradeoffs

$$\downarrow \frac{\text{Instrs}}{\text{Program}} \; :$$

$$\downarrow \frac{\text{Cycles}}{\text{Instr}} \; :$$

$$\downarrow \frac{\text{Time}}{\text{Cycle}} \; :$$



## "Iron Law" of Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instrs}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instr}} \times \frac{\text{Time}}{\text{Cycle}}$$



## Other Performance Equations

$$\text{Speedup} = \frac{\text{TSlow}}{\text{TFast}}$$

$$\text{n\%faster} = \text{Speedup} - 1$$

$$= \frac{\text{TSlow}}{\text{TFast}} - 1$$

$$= \frac{\text{TSlow} - \text{TFast}}{\text{TFast}}$$

$$\text{WeightedAvg} = \sum_{i=1}^{n} \%i \times T_i$$

## Example: Integer and Floating Point

- Average time to execute an integer and floating-point instruction on two computers:

| | Int | Float |
|---|---|---|
| CPU1 | 2.0ns | 3.5ns |
| CPU2 | 2.5ns | 3.0ns |

- ***Q:*** *Which CPU has greater performance for integer programs, and how much faster is it?*

- ***Q:*** *If the average program is 90% integer instructions and 10% floating-point instructions, which CPU has greater performance, and how much greater is the performance?*



## Example: Integer and Floating Point

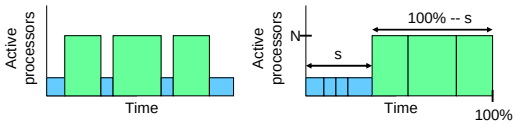
- ***Q:*** *If we want to optimize the slower CPU to match the performance of the faster CPU, should we optimize integer or floating-point instructions?*

- ***Q:*** *If you have to fire all of your computer architects because your stock price plummets, how can you get the slower CPU to match the performance of the faster CPU?*



## Amdahl's Law

- Supercomputers in 70s and 80s:
  - multiple processors
  - scalar instructions: run on main processor
  - vector instructions: run on all processors (useful for matrix and array operations)

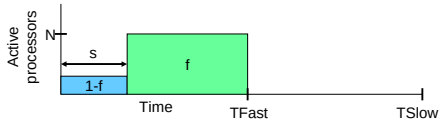



- Amdahl's definition of efficiency

$$\frac{1\times s + N\times(100\% -- s)}{N}$$



## Amdahl's Speedup




f = percentage of work done in vector mode

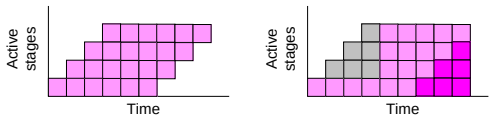
100% -- f = percentage of work done in scalar mode

TFast = Time if have N processors

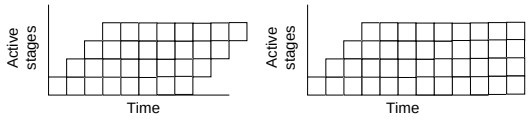
TSlow = Time if have just 1 processor

Speedup =

## Amdahl and Pipelining




Affect of stalls:






## Performance and Time

- In the computer field, performance increases at an exponential rate.
- On average, doubles every 18 – 24 months.
- Moore's law says that number of available transistors doubles every 18 – 24 months.
- It's up to the compiler writers, computer architects, computer engineers, and electrical engineers to use the ever increasing number of transistors to improve performance.
- Equation for performance increasing by factor of n every k units of time:

$$P(t) = n^{(t/k)}$$



## Example: Perf. and Time to Market

- ***Q:*** *If performance doubles every 2 years, how much does it increase each week?*
- ***Q:*** *You are considering a performance optimization that will delay your schedule by 4 weeks, but increase your performance by 5%, should you do the optimization?*



## Overview of Instruction Sets

## Levels of Abstraction

- ESL (electronic system level): several processors, software, custom hardware. Software model of entire system to predict behaviour and performance.
- Transaction level: example transaction = (transfer packet from main-cpu to cryptography engine; encrypt; transfer packet back to main-cpu)
- ISA (instruction-set architecture)
- architecture: 5—15 blocks per processor
- microarchitecture: building blocks are pipeline stages and memory arrays
- HLM (high-level model): behavioural description of hardware
- RTL (register-transfer level): 100k – 1M lines of code per processor



## Levels of Abstraction (2)

- Structural: HDL description of gates (e.g. C <= A + B)
- Gates / Cells: graphical description of design, often with sizing information
- Transistors
- Masks
- Silicon



## Application Domains for Processors

- Servers
- Desktop
- Embedded
  - Low-power
  - Signal processing
  - Graphics
  - Network
  - (Re)configurable



## ISA Features

- To justify a new instruction set, it must offer twice the performance of existing instruction sets
- Hardware architecture optimization must provide 3% improvement
- Performance increases 1% per week on average
- Takes up to ten years to design an ISA before production, and then at least five years from first product release to full deployment with complete software distribution

## ISA History

| | |
|---|---|
| • IBM 360: | 195x |
| • Intel x86: | 1971 |
| • Motorola 6400,68000, coldfire | 1974 |
| • ARM | 1985 (??) |
| • IBM Power: | 1990 |
| • PA-Risc: | 1990 |
| • MIPS | 1990 (?) |
| • Sun SPARC | 1990 (?) |
| • DEC Alpha | 1990 |
| • Itanium | 1998 |

30

## Dynamic (HW) vs Static (SW)

- ISA defines dynamic/static (e.g. hw/sw) interface (Yale Patt??)
- Static: program is static, doesn't change
- Dynamic: execution trace of program in hardware is dynamic: can get different behaviour in different runs of the program
- Static optimizations done by compiler
- Dynamic optimizations done on-the-fly by hardware at runtime

31

| | Static (SW) | (Dynamic (HW) |
|---|---|---|
| Data structures | | |
| Algorithms | | |
| Effective addresses | | |
| Window of instrs | | |

32

## Ex: Instruction Sets and Performance

**Evaluate performance impact of adding a combined multiply/add instruction**

- Half of the multiply instructions are followed by an add that can be combined into a single multiply/add instruction
- ADD: CPI=0.8, 15%
- MUL: CPI=1.2, 5%
- Other: CPI=1.0, 80%
- Option1: No change
- Option2: add MAC instr, increase clock period by 20%, MAC has same CPI as MUL
- Option3: add MAC instr, keep same clock period, CPI of MAC is 50% greater than MUL
- ***Q:*** *Which option is faster, and by how much?*

33

## Performance Simulation

- Analytical models of performance are accurate only for very simple processors
- Most analysis done by performance simulation (aka “instruction set simulators”)
- Goals of performance simulation are to measure
  - number of clock cycles to execute a program
  - monitor resource usage
    - % of stalled cycles
    - % of time that a unit (e.g. adder) is idle
  - Often don’t care about computing a real result
- Want simulator to run as fast as possible
  - more experiments = explore more design options
  - more benchmarks = more accurate estimate of performance



## Performamance Simulation

- Two types of simulators
  - Execution driven: simulator actually executes the code and computes results
  - Trace driven:
    - Record execution trace as run program on a simulator on on instrumented hardware
    - Run trace through performance simulator
    - Don’t compute results, just compute how long each instruction takes
- Most simulators are now execution-driven
  - Traces a require large amounts of memory
  - Trace is applicable only to architectures similar to that used to generate the trace (e.g. can’t change branch prediction algo)



## Instruction Set Design Decisions

- Instructions
  - Operations
  - Data types
  - Addressing modes
- State (Registers and Memory)
- Encoding
- Legality rules
  - Sequencing
  - Alignment



## Operations

- Normal operations
- Special operations

## Computation and Memory Instructions

38

## Operand and Result Location

- Memory
- General purpose registers
- Stack
- Accumulator
- Special purpose registers
  - PC
  - CCR
  - Link register
  - Top of stack register
  - Return address register
  - Loop count register

40

## Operation and Memory Decisions

- Operand and result locations
- Operand and result addressing
- Memory addressing modes
- Data types

39

## Operand and Result Addressing

- Implicit
  - Stack
  - Accumulator
  - Special purpose registers
- Explicit
  - Memory
  - General purpose registers
- Types of registers
  - Data / Addr
  - Int / FP
  - Predicate / Numeric

41

## Memory Addressing

- Alignment
  - Byte, Half-word, Word, Double-word
- Addressing modes
  - Register
  - Immediate
  - Displacement
  - Register indirect
  - Indexed
  - Direct (or absolute)
  - Memory indirect
  - Pre/Post inc/dec rement
  - Scaled

42

## Data Width

- Most common data widths:
  - 8b byte
  - 16b short
  - 32b word
  - ***64b double***
- Special cases
  - Internal registers of greater width to improve precision of arithmetic
  - Vectors
    - MMX vs general vector instructions
  - CISC strings

43

## Control Flow Instructions

44

## Programming Language Control Flow

**Programming language constructs that lead to control flow instructions**

- Case/switch
- If-then-else
- Subroutine, procedure, function call
- Subroutine, procedure, function return
- Loop

45

## Control Instruction Design Decisions

- All types of control instructions:
  - Addressing modes for target
- Conditional branches:
  - Locations that can be tested
  - Tests that can be performed
- Procedure call and return
  - State save / restore
  - Argument / result passing



## Types of Control-Flow Instructions

| | Int | FP |
|---|---|---|
| • Conditional branches | 82% | 75% |
| • Jumps | 10% | 6% |
| • Procedure / function / subroutine calls<br>• Procedure / function / subroutine returns | 8% | 19% |

- Special case of conditional branches:
  - Loop closing
- Recent innovation
  - Predicated instructions (e.g. conditional move)



## Addressing Modes

- Performance advantage to calculate branch target quickly
  - Why?
- Most common addressing mode for control instructions: displacement



## Locations to Test

- CCR
- General Purpose Register
- Predicate Register

## Tests

- Flag set / unset
- Comparison
- <0, ≤0, =0, ≥0, >0

50

## State Save/Restore

- What's the problem?

- In CISC processors, often could save or restore state with a single instruction.
  - VAX Calls instruction: extremely general, extremely slow, not used.
- RISC approach:
  - let compiler do the work
  - establish compiler conventions
    - caller-save or callee-save
    - registers for stack pointer and link pointer
- Register windows

51

## Argument/Result Passing

- Similar to state save/restore
  - CISC: special purpose instructions
  - RISC: let the compiler do the work
- RISC alternative: register windows
  - Berkeley RISC I, SUN Sparc, Intel Itanium
  - Architectural carbunkle or elegant alternative to register renaming?

52

## Example: Braniac vs Speed Demon

- AMD Athlon 1.2 GHz, 409 SPECint
- Fujitsu SPARC64: 675 MHz, 443 SPECint
- Assume that it requires 20% more instructions to write a program in Sparc64 than in IA-32
- ***Q:*** *Which processor has higher performance?*

- ***Q:*** *What is the ratio between the CPIs?*

- ***Q:*** *What is the CPI of the AMD Athlon?*

53

## Ex: Instruction Encoding

- Design a simple 16-bit instruction set
  - Number of instructions:
  - Number of addresses per instruction:
  - Number of registers:

54

## A Simple RISC ISA

55

## Defining an Instruction Set

- Instructions
  -
  -
  -
-
-

56

## Operations: Arith and Control

- Arithmetic: *func* **R***dst* **R***src1* **R***src2*
  - Regs[Rdst] ← Regs[Rsrc1] func Regs[Rsrc2]
- Branch: **B***cond* (**R***tst*)*imm*
  - Conds: <0, <=0, =0, !=0, >=0, >0
  - if cond(Regs[Rtst]) then { PC ← PC + 4 +imm } else { PC ← PC+4 }
- Jump:
  - **J** *#imm* : PC ← PC + 4 + imm
  - **JR R***tgt* : PC ← Regs[Rtgt]
  - **JAL** *#imm* : R31 ← PC+4; PC ← PC+4+imm
  - **JALR R***tgt* : R31 ← PC+4; PC ← Regs[Rtgt]

57

## Operations: Load and Store

- Load: **L***width/type* **R***dst* **R***addr imm*
  - Widths: **B**yte, **H**alf-word, **W**ord, **D**ouble
  - Type: signed or **U**nsigned
  - Regs[Rdst] ← Mem[ Regs[Raddr]+imm ]
- Store: **S***width* **R***addr* **R***src imm*
  - Widths: **B**yte, **H**alf-word, **W**ord, **D**ouble
  - Mem[ Regs[Raddr]+imm ] ← Regs[Rdst]

- ***Other operations (e.g. traps, exceptions, special registers, floating point) are not included in this lecture***



## Architectural and Physical State

- Architectural State
  - PC program counter
  - R register file
  - Mem memory
- Physical (hidden) state
  - IR instruction register
  - Imm immediate data
  - ALUoutput
  - LMD load memory data



## Basic Steps of Execution

in-flight

- Fetch: get instruction from memory
- Issue: begin decoding, enter pipeline
- Dispatch: wait until operands are ready, send to execution units
- Execute: perform computation
- Writeback: put result in speculative state
- Retire: put result in non-speculative state

- Notes:
  - "dispatch" = "disperse" = "schedule"
  - "retire" = "commit" = "finalize"
  - "state" = registers and/or memory



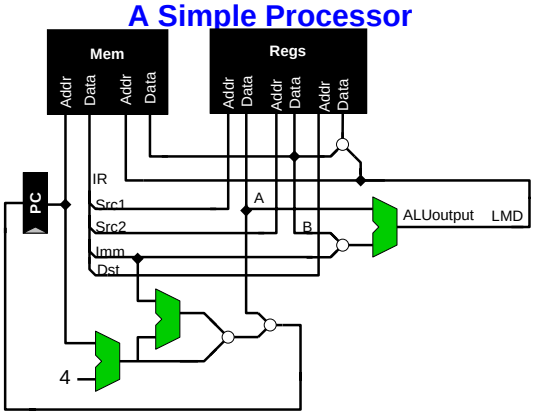
## Basic Steps of Execution

- IR ← Mem[PC]
- A ← Regs[ IR.src1 ]
- B ← Regs[ IR.src2 ]
- Imm ← IR.imm
- ALUoutput ← A func B
- **OR:** ALUoutput ← A func Imm

} func = arith func, address calc, or condition test

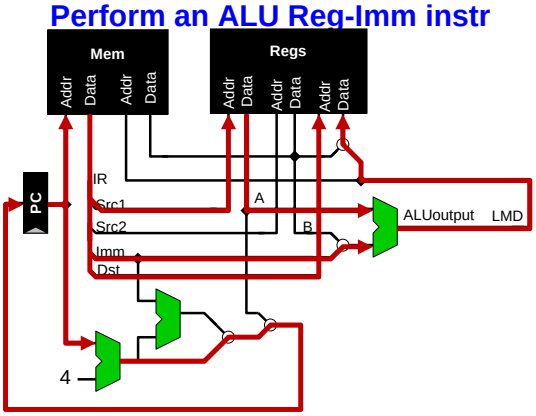
- PC ← PC+4
- **OR:** PC ← PC+4+imm
- **OR:** PC ← A
- LMD ← Mem[ B ]
- Mem[ ALUoutput] ← B
- Regs[ IR.dst ] ← ALUoutput
- **OR:** Regs[ IR.dst ] ← LMD

## A Simple Processor



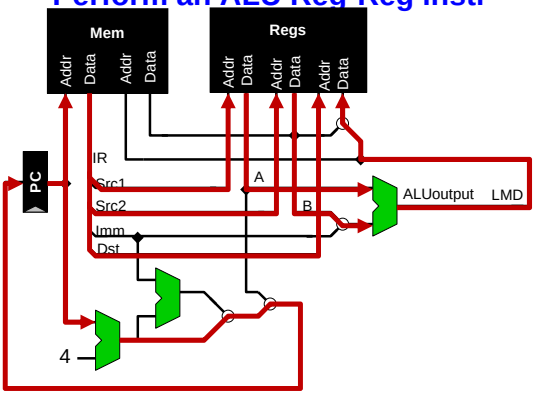

62

## Perform an ALU Reg-Imm instr



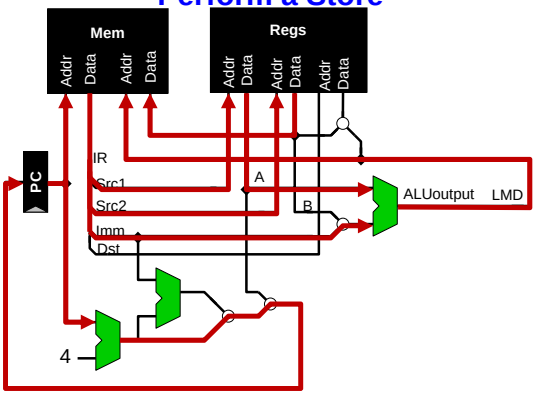

64

## Perform an ALU Reg-Reg instr



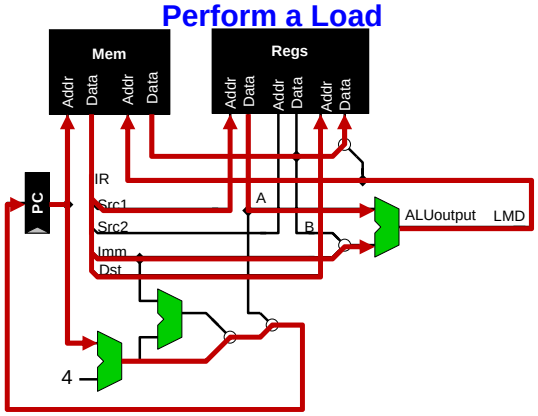

63

## Perform a Store



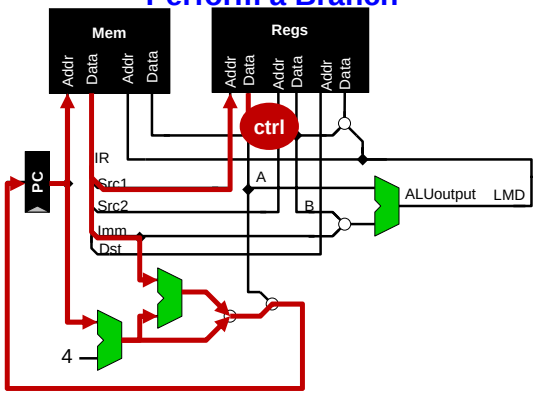

65

## Perform a Load



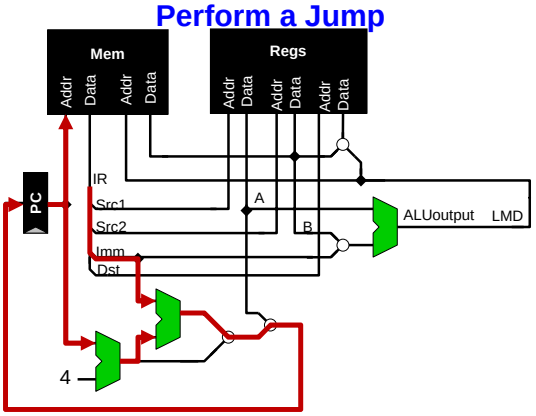

66

## Perform a Branch



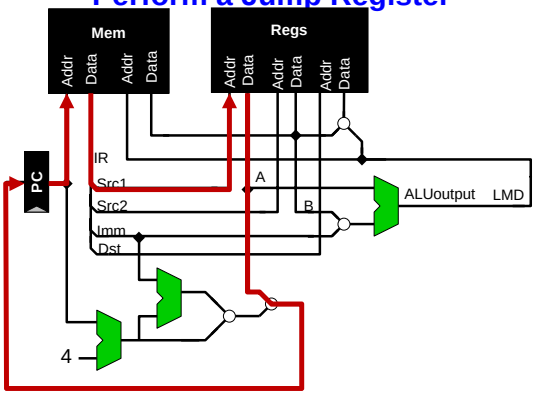

67

## Perform a Jump




68

## Perform a Jump Register




69

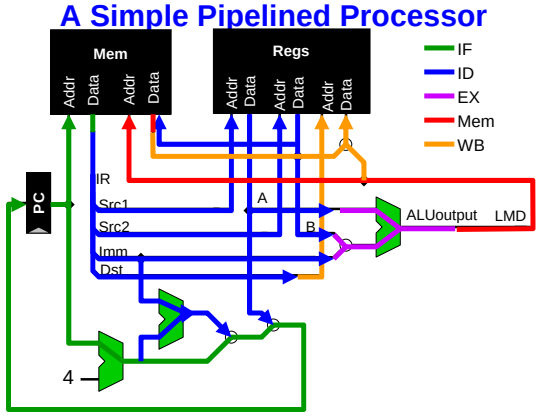
## Allocate Ops in Stages

- IR ← Mem[PC]
- A ← Regs[ IR.src1 ]
- B ← Regs[ IR.src2 ]
- Imm ← IR.imm
- ALUoutput ← A func B
  ALUoutput ← A func Imm
- PC ← PC+4
  PC ← PC+4+imm
  PC ← A
- LMD ← Mem[ B ]
  Mem[ ALUoutput] ← B
- Regs[ IR.dst ] ← ALUoutput
  Regs[ IR.dst ] ← LMD

| Arith-reg | Arith-imm | Store | Load | Branch | Jump | Jump-reg |
|---|---|---|---|---|---|---|
| | | | | | | |

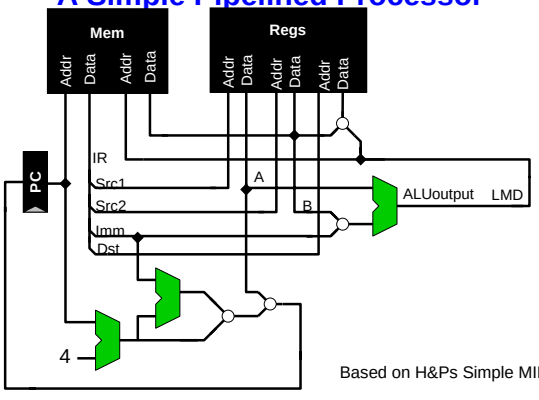
70

## A Simple Pipelined Processor




72

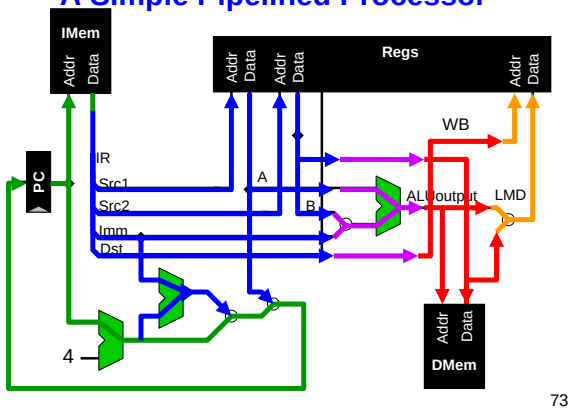
## A Simple Pipelined Processor




Based on H&Ps Simple MIPS

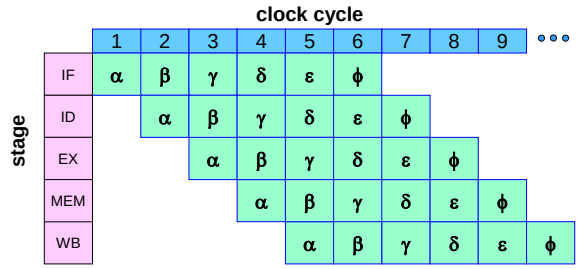
71

## A Simple Pipelined Processor



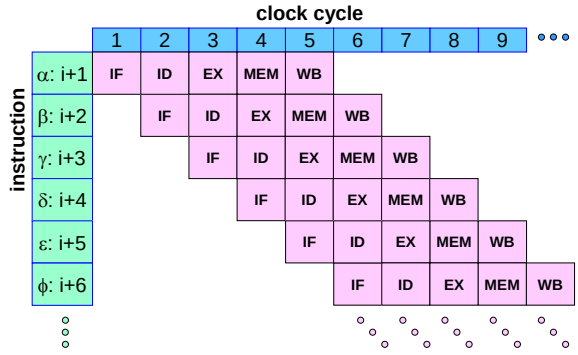

73

## Reservation-Table View




74

## Execution-Pattern View




75

## Pipelining: Ideal vs Reality

- Previous two slides showed ***ideal*** pipeline behaviour:
  - no hazards
  - speedup =_____
- Hazards
  - reduce speedup
  - complicate hardware (and maybe software) design

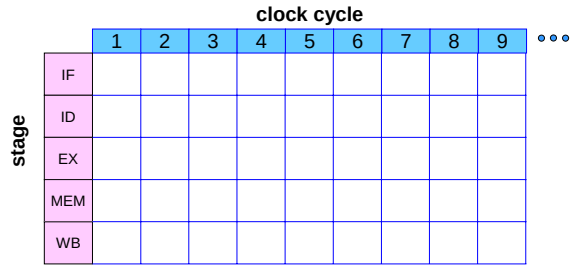
76

## Hazards in Our Simple Pipeline

- _____ Hazards
- _____ Hazards
- _____ Hazards

77

## Back-to-Back Arith Instrs

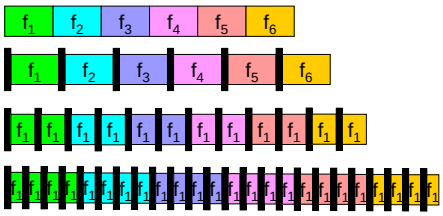



ADD: R8 ← R2 + R1
SUB: R10 ← R8 – R2
AND: R12 ← R8 AND R3
OR : R13 ← R8 OR R4
XOR: R14 ← R8 XOR R5

78

## Pipelining--The Basic Idea

- break task into smaller pieces
  - pipe stages or pipe segments




**latency**: time for an instruction to pass through the pipeline

**throughput:** number of instructions that exit the pipeline per unit time

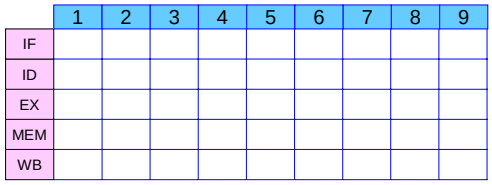
80

## Dealing Data Hazards

- Hardware techniques
  -
  -
  -
- Software techniques
  -

79

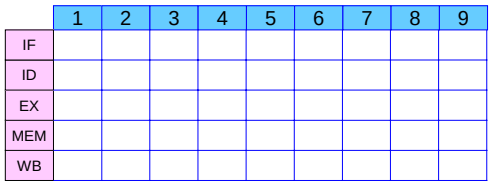
## Data-Dependencies on Loads




**LW R1,32(R6)**
**ADD R4,R1,R7**
**SUB R5,R1,R8**
**AND R6,R1,R7**

81

## Control Hazards--Branches/Jumps




82

## Branch Instruction Stats

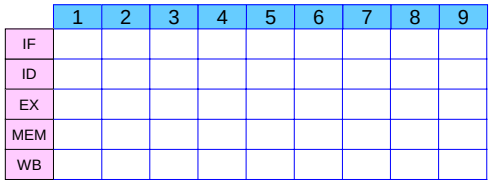
**branch/jump instruction frequencies**

| | **unconditional** | **conditional** |
|---|---|---|
| DLX | 2% | 11% |
| x86 | 7% | 14% |
| VAX | 8% | 17% |

**approx 53% of conditional branches executed are taken**

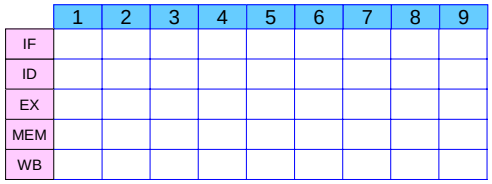
84

## Control Hazards--Branches/Jumps




83

## Predict-Not-Taken--Success




85

## Predict-Not-Taken--Fail

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| IF | | | | | | | | | |
| ID | | | | | | | | | |
| EX | | | | | | | | | |
| MEM | | | | | | | | | |
| WB | | | | | | | | | |