Instruction Set Architecture

A contract between software and hardware

Source material

- D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition*. Cambridge, MA: Morgan Kaufmann, 2018.
  - Chapter 2: Instructions: Language of the Computer
- J. L. Hennessy, D. A. Patterson, and K. Asanović, *Computer Architecture: A Quantitative Approach*, 5th ed. Waltham, MA: Morgan Kaufmann/Elsevier, 2012.
  - Appendix A: Instruction Set Principles
  - Appendix K: Survey of Instruction Set Architectures

- Arm ISA and extensions
  - Arm Architecture Reference Manual for A-profile architecture
  - SME Programmer's Guide
  - The Thumb instruction set

What is an “Instruction Set Architecture” (ISA)

The portion of the computer visible to the programmer or compiler writer.

Appendix A of Computer Architecture: A Quantitative Approach

- Dictionary that the hardware understands
  - Each “word” on the dictionary is called an “instruction”
- Exposes the hardware capabilities to the programmer
  - Types of operations (add, compare, data movement, etc.)
  - Types of operands (bytes, words, floating point numbers, etc.)
  - Number of operands
  - Interface with memory resources (registers, memory)
- Some examples: x86, armv9, RISC-V

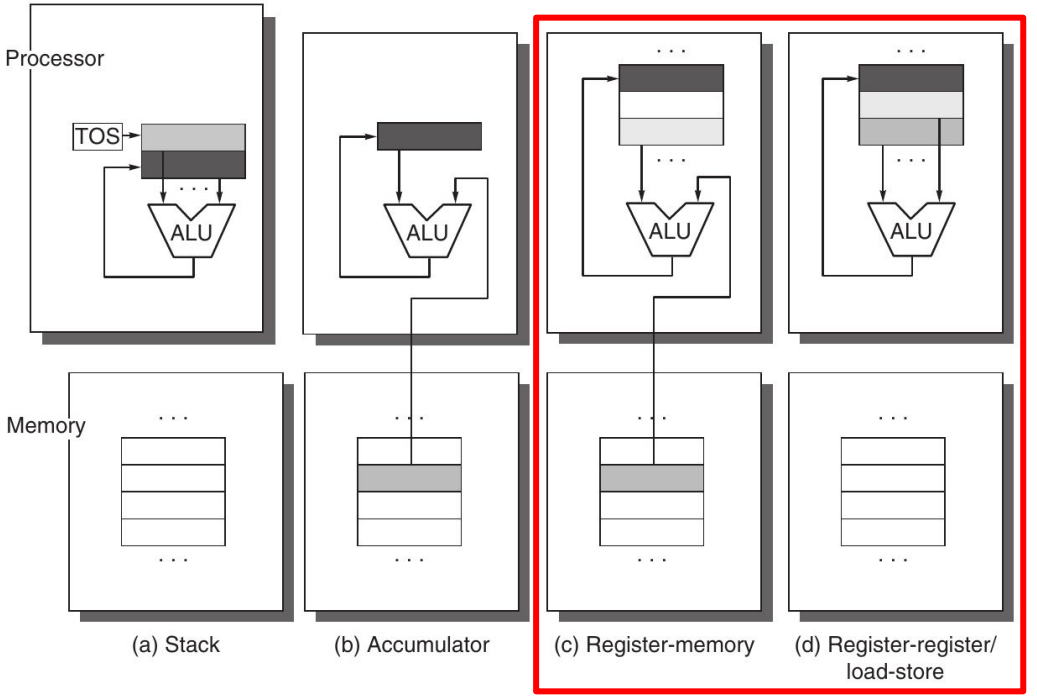
An ISA…

- Tells you **WHAT** an instruction does
  - Add 2 the values of two registers and store the result on a third register
  - Load the value at a certain memory address into a register
  - Store the value of a register into a certain memory address
- **WHAT** → *ISA* or *Architecture*

- ~~Does~~ Should not tell you **HOW** an instruction is executed (sometimes it does…)
  - Read the values of each source operand on a single cycle
  - Prefetch the next data to be loaded based on the data you just got
  - Buffer the data to be stored until the path to memory is available
- **HOW** →*Micro-architecture* or *Implementation*

Types of ISAs depending on… Operand location

- In practice, ISAs adopt more than one depending on the instruction
- ISAs in General Purpose Processors (GPP) typically expose
  - (c) Register-memory
  - (d) Register-register
- Specialized hardware might benefit from other types
  - (b) Accumulator → Arm's SME



Types of ISAs depending on… Memory addressing

- Defines how programmers access data
- Has direct impact on program size

- Simple addressing mode
  - Easy to implement in hardware
  - Requires more instructions
- Complex addressing mode
  - Hard to implement in hardware
  - Does more with less instructions

Mode	Example	Meaning
Register	`Add R4, R3`	`R4 ← R4 + R3`
Immediate	`Add R4, #3`	`R4 ← R4 + 3`
Reg. Indirect	`Add R4, (R1)`	`R4 ← R4 + Mem[R1]`
Displacement	`Add R4, 100(R1)`	`R4 ← R4 + Mem[100+R1]`
…		

- According to Hennessy&Patterson
  - Four addressing modes account for 97% of memory accesses
  - ISAs should support these: Displacement, Immediate, Register Indirect

Types of ISAs depending on… Operand type and size

- Operand types
  - Boolean, Integer, Floating Point, String
- Operand sizes
  - Bit, Byte, Word (32-bit), Double-word (64-bit), Single-precision, Double-precision, Pointer (64-bit*)
- Number of operands
  - 1 source + 1 destination, 2 source + 1 destination, 1 source + 2 destination

- Instructions tend to operate only on a specific data type/size
  - Add two 32-bit integers
  - Multiply two 64-bit floating point numbers
- There are special instructions to convert between types/sizes
  - Convert 32-bit integer to single-precision floating point (R1 → F1)
  - Widen 32-bit integer to 64-bit integer

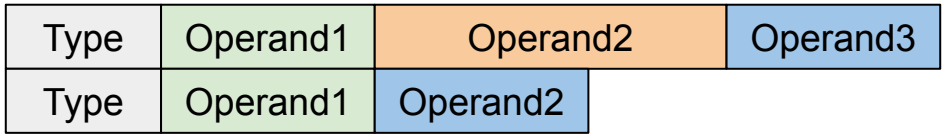
Types of ISAs depending on… Operations

- Arithmetic and logical (both integer and floating point)
  - Add, subtract, and, or, multiply, divide
- Data transfer
  - Move between registers, move from/to main memory
- Control
  - Branch, jump, call
- System
  - System call, virtual memory management

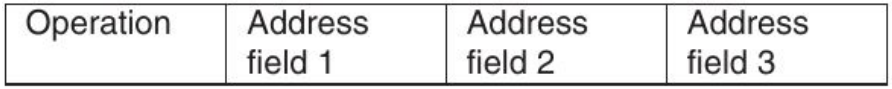
- “Graphics” → SIMD and/or Vector 👈 More on these tomorrow

Encoding instructions

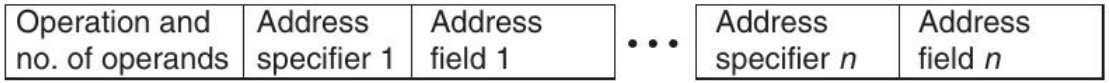
- Instruction type and operands is defined by a sequence of fields encoded as bits



- Instruction set must define
  - Which fields each instruction has
  - Size of each fields
  - Length of the instruction (fixed vs variable)



(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

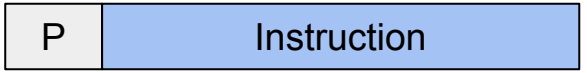


(a) Variable (e.g., Intel 80x86, VAX)

Real use case: ARM

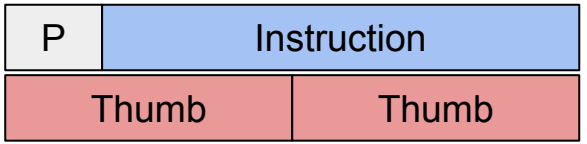
Look for me during break time and we can discuss ARM vs Arm

- Base instruction set that incrementally grows (armv7)
  - Basic functionality
  - Fixed length encoding (32-bit)
  - Predicated execution!!



Predicates (P) determines if the instruction should act as a NOP or execute as normal

- Compressed instructions (Thumb)
  - Reduced functionality
  - Fixed length encoding (16-bit)
  - Change between “normal” and “Thumb” by setting a bit
  - Code size efficiency



- SIMD extension called ASE (a.k.a NEON instructions)
  - Graphics processing and high-performance applications

I want to design my own ISA!

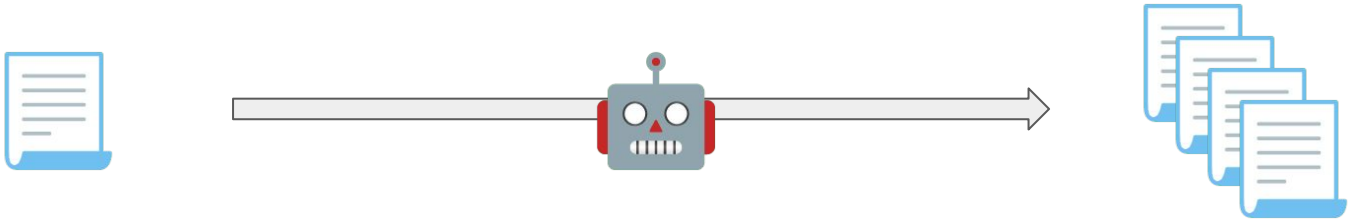
1. Consider ISA adoption
   - Programmers value binary compatibility and ecosystem maturity above all else
   - Example: Intel's x86 is not the "most elegant" ISA, but has remained strong because of wide adoption
   - Example: Arm was considered "emerging" until compilers and libraries were ported

2. Design for the common case
   - Favor small ISA that covers 95% of use cases
   - Avoid defining instructions that fit 1% of situations

3. The compiler and calling conventions
   - Compilers and system software agree on some coding conventions (e.g. ABI)
   - If a non-critical functionality can be implemented by software, let the software handle it

The biggest lie about ISAs

- The ISA is the common ground between hardware and software
- In reality…
  - Software developers program in high-level languages (C, C++, FORTRAN, Java, etc.)
  - Hardware implementations use an internal representation to control execution of instructions



Front-End
Instructions from ISA

Instruction Decoding
and translation

Back-End
Internal micro-ops

- An instruction from the front-end is translated into one or more internal “operations”
- In some implementations, the back-end representation is called “micro-operation”

Pipelining

How to schedule house chores

Source material

- D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition*. Cambridge, MA: Morgan Kaufmann, 2018.
  - Chapter 4.5: An Overview of Pipelining
  - Chapter 4.6: Pipelined Datapath and Control
- J. L. Hennessy, D. A. Patterson, and K. Asanović, *Computer Architecture: A Quantitative Approach*, 5th ed. Waltham, MA: Morgan Kaufmann/Elsevier, 2012.
  - Appendix C: Pipelining: Basic and Intermediate Concepts
  - Chapter 3: Instruction-Level Parallelism and Its Exploitation

From instruction definition to implementation

`add R1, R2, R3`

- Resources
  - Instruction Memory
  - Decoding logic
  - Register file
  - Arithmetic-Logic Unit (ALU)
  -
- Steps
  - Instruction Fetch (IF)
  - Instruction Decode (ID)
  - ReaD operans (RD)
  - EXecute operation (EX)
  -
  - WriteBack result (WB)

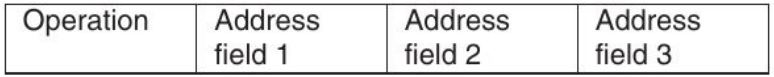
`load R2, 100(R1)`

- Resources
  - Instruction Memory
  - Decoding logic
  - Register file
  - Address Generation Unit (AGU)
  - Data Memory
- Steps
  - Instruction Fetch (IF)
  - Instruction Decode (ID)
  - ReaD operans (RD)
  - Address Generation → EXecute (EX)
  - MEMory access (MEM)
  - WriteBack result (WB)

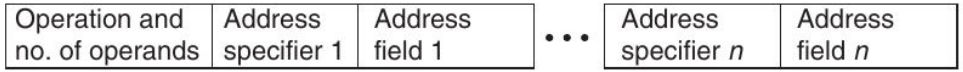
Resources - Decoding logic

- Translate a sequence of bits (instruction) into control signals
  - Type of instruction
  - Operands

- ISA design impact
  - Fixed size instructions → Decoding logic always has to read the same amount of bits to decode an instruction
  - Variable size instructions → Decoding logic needs to decode the instruction by chunks as they become available and the control signals indicate that more chunks are needed



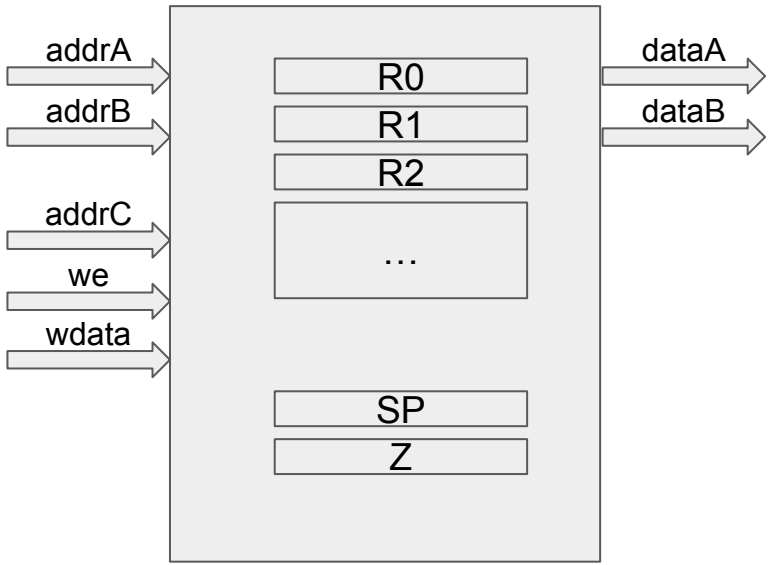
(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)



(a) Variable (e.g., Intel 80x86, VAX)

Resources - Register file

- Contains a set of registers
  - Must support ISA-defined registers
  - Might include more (physical) registers
- Number of Read/Write ports
  - How many reads per cycle?
  - How many writes per cycle?
- What happens if read & write happen during the same cycle?
  - Write available on the same cycle?
  - Write available on the next cycle?

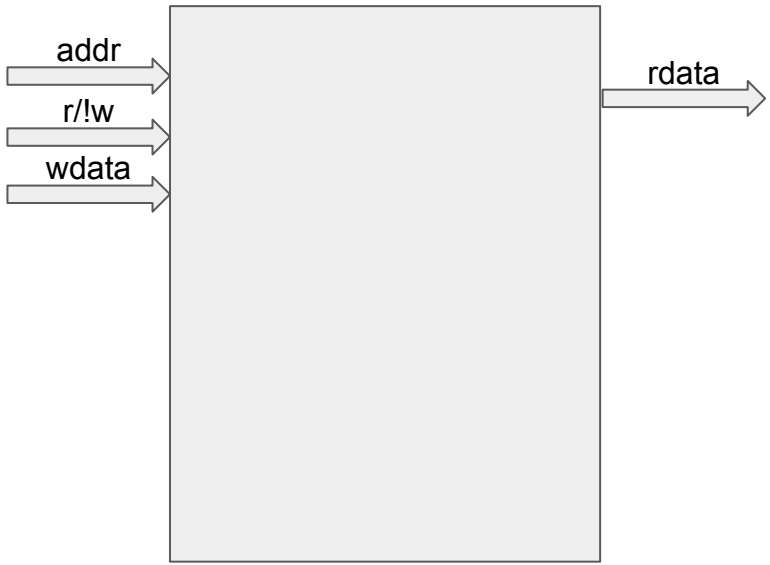


A made up register file

1 Write port 2 Read ports

Resources - Memory

- Addressable space
  - How many bits does an address have?
  - Addressable size? (Byte, Word, etc.)
- Same questions as with register file
  - Number of Read/Write ports
  - Read & Write during the same cycle?
- What kind of information does it store?
  - Data or Instructions?
  - What about if it stores both?
- Access latency
  - Much higher than register file
  - Will be covered in a later session
  - For now, assume accesses to memory are completed in one cycle

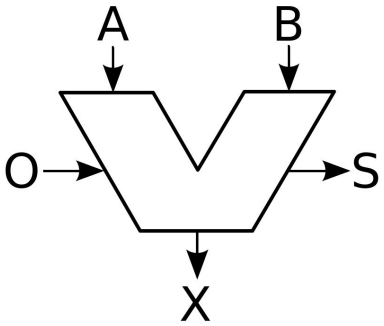


A made up memory bank

1 Write port 1 Read port

Resources – ALU

- Performs ISA-defined operations
  - Takes input operands and operator
  - Outputs operation result
- Number of operands
  - Depending on the ISA, might require to support more than 2 input operands
- Status information
  - Was the result zero?
  - Did an overflow occur?
- Operation latency
  - Some operations are slower than others
  - Example: addition vs multiplication
  - For now, assume all operations are completed in one cycle



Conventional way to draw an ALU

- Input operands: A, B
- Type of operation: O
- Output: X
- Status information: S

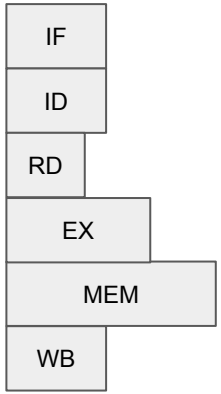
Resources - Address Generation Unit

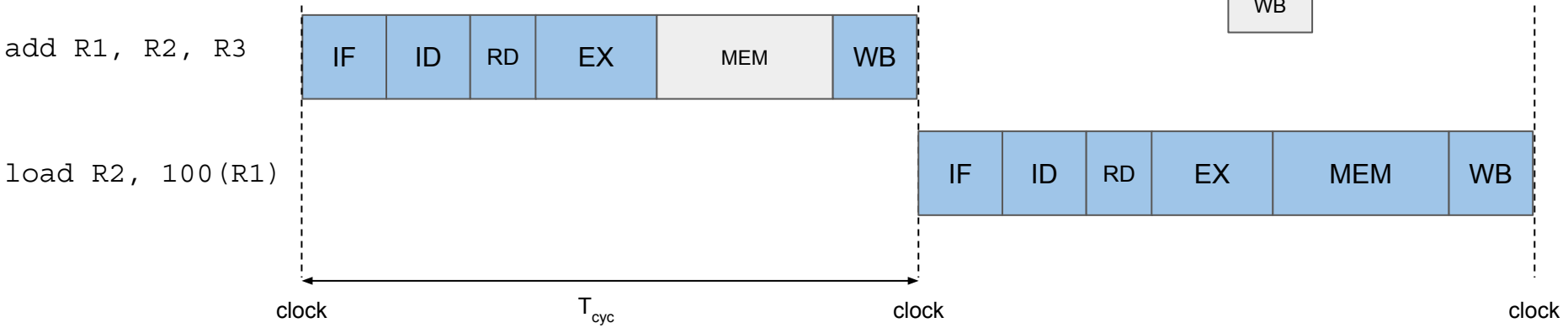
- Special module to compute addresses for memory operations
  - Effective address = base address + offset * datatype_size
  - Strictly integer operations

- Implementation approaches
  - Simple and cheap → Use the ALU
  - A bit more complicated → Add specific AGU module

Single-cycle execution

- Not all steps/stages take the same amount of time
- All instructions take one cycle of duration T_{cyc}

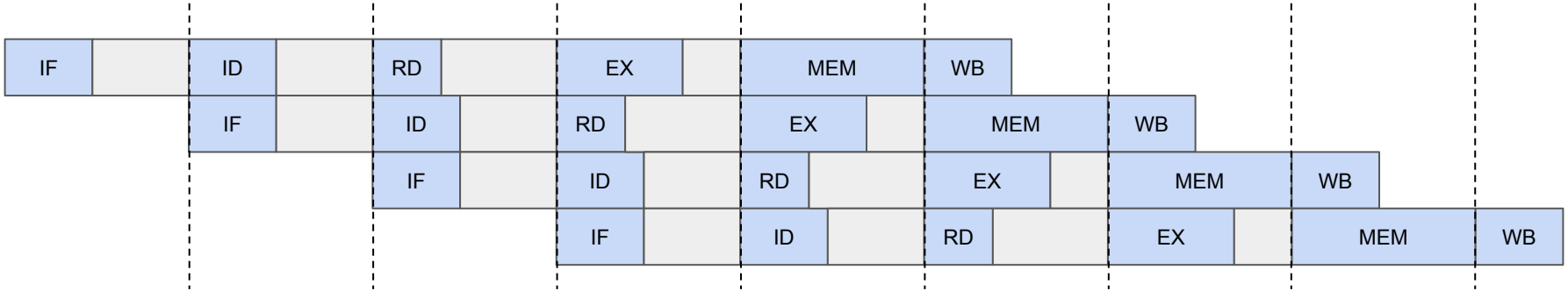




Pipelined execution

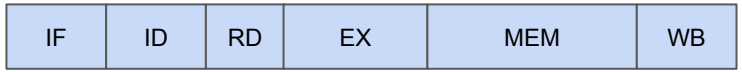
- All steps/stages take the same amount of time (the slowest)
- All instructions take six cycles of duration T_{cyc}

- Overlapping the execution of multiple instructions at different stages of execution
- This overlap is called: ***Instruction Level Parallelism***

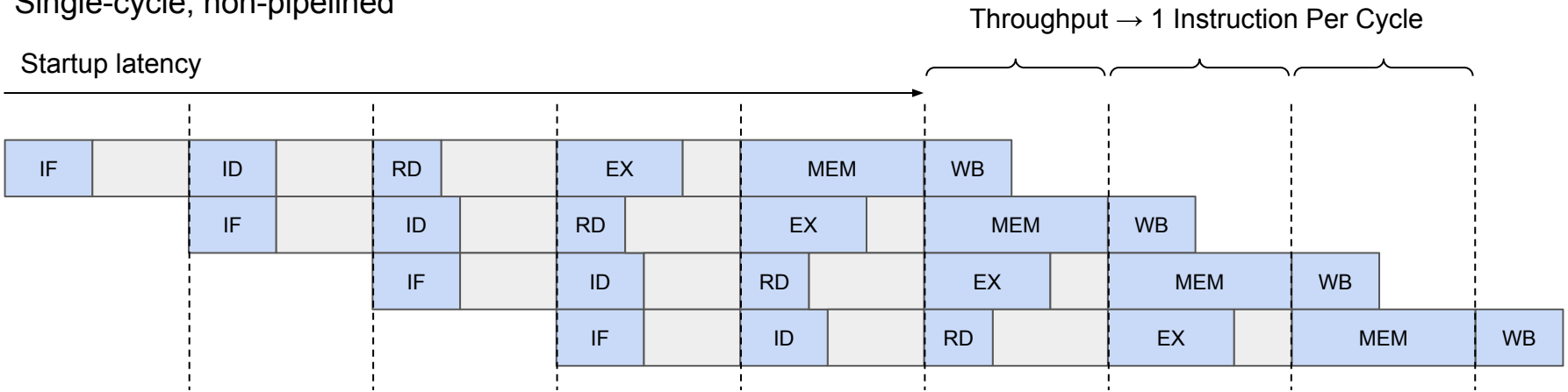


Design for Throughput, not for Latency

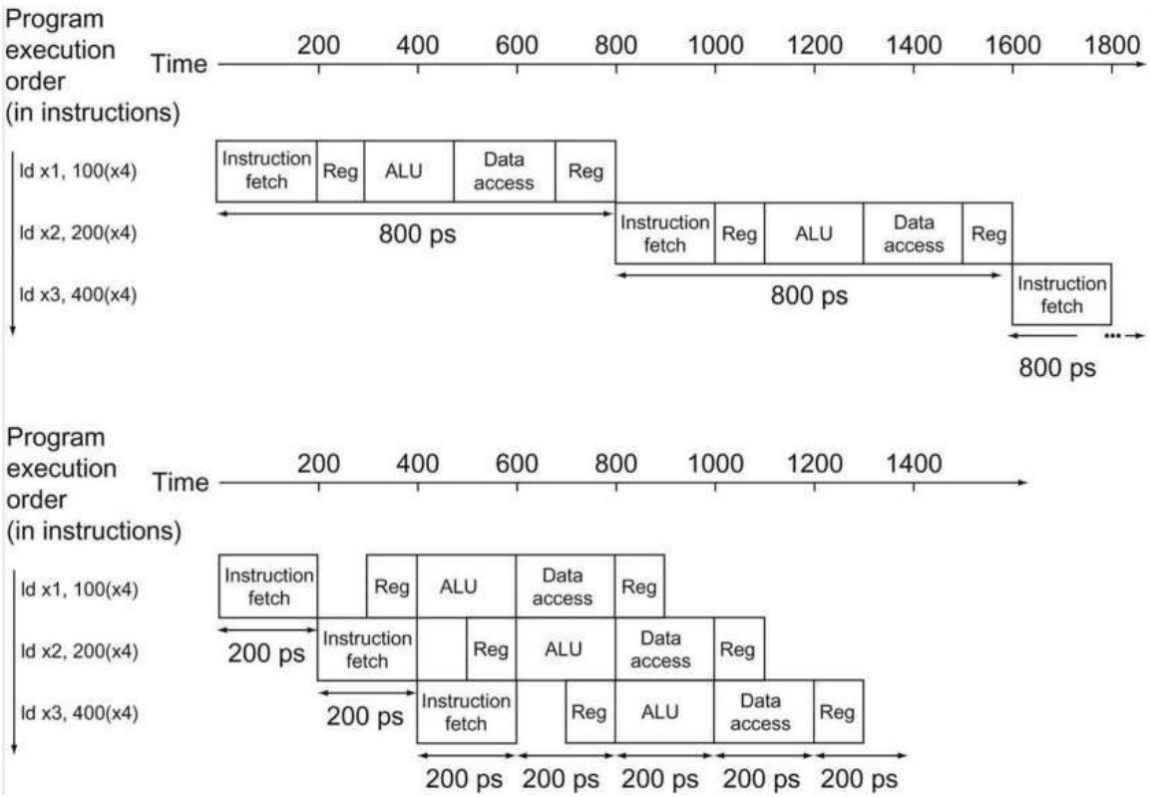
Single-cycle, non-pipelined



Single-cycle, non-pipelined



Example from literature



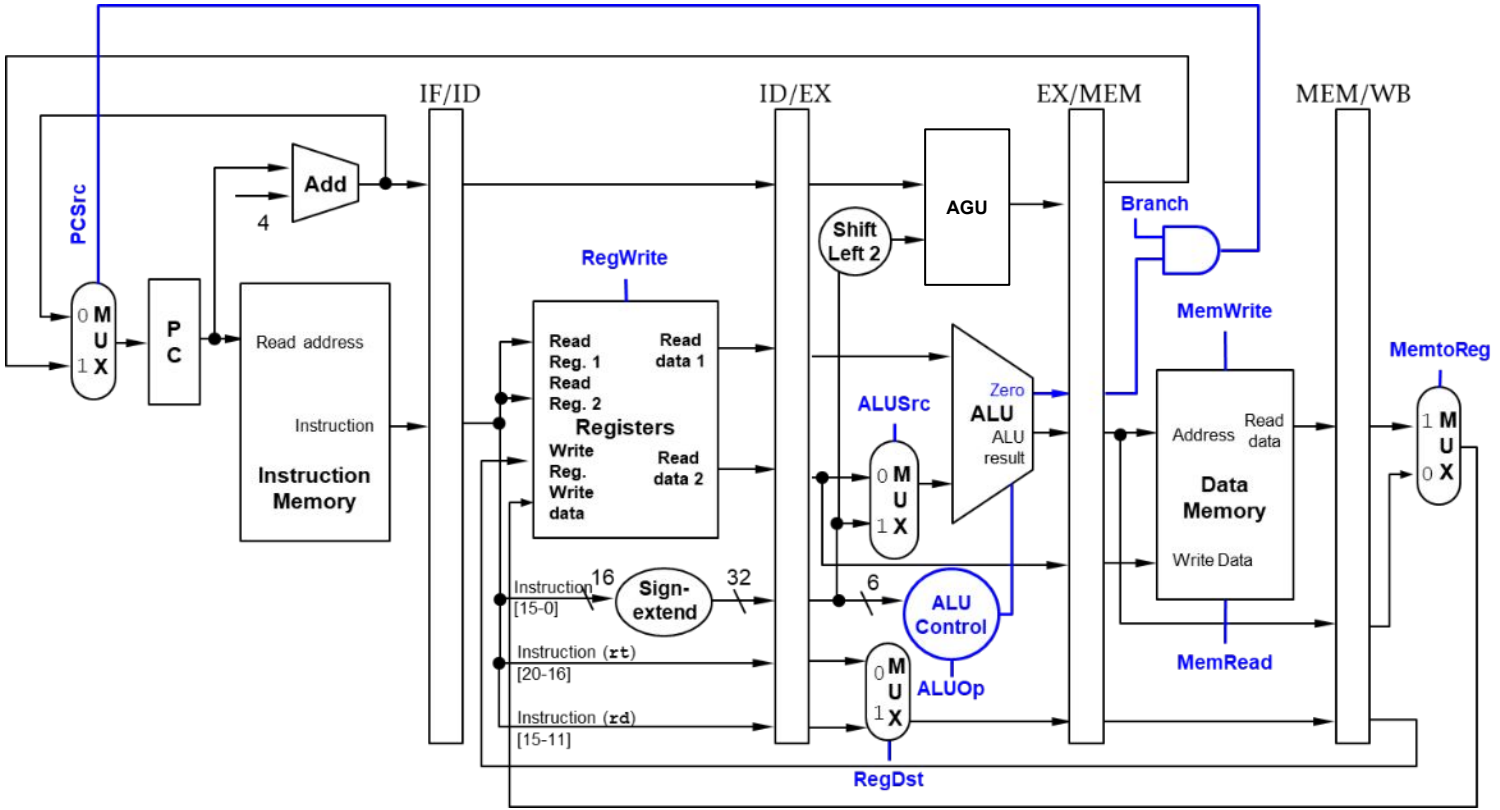
Design for Throughput, not for Latency

- Single-cycle instructions have lower latency
- Pipelined instructions have higher throughput

- We measure throughput in
  - Instructions Per Cycle (IPC) → Goal: Maximize
  - Cycles Per Instruction (CPI) → Goal: Minimize

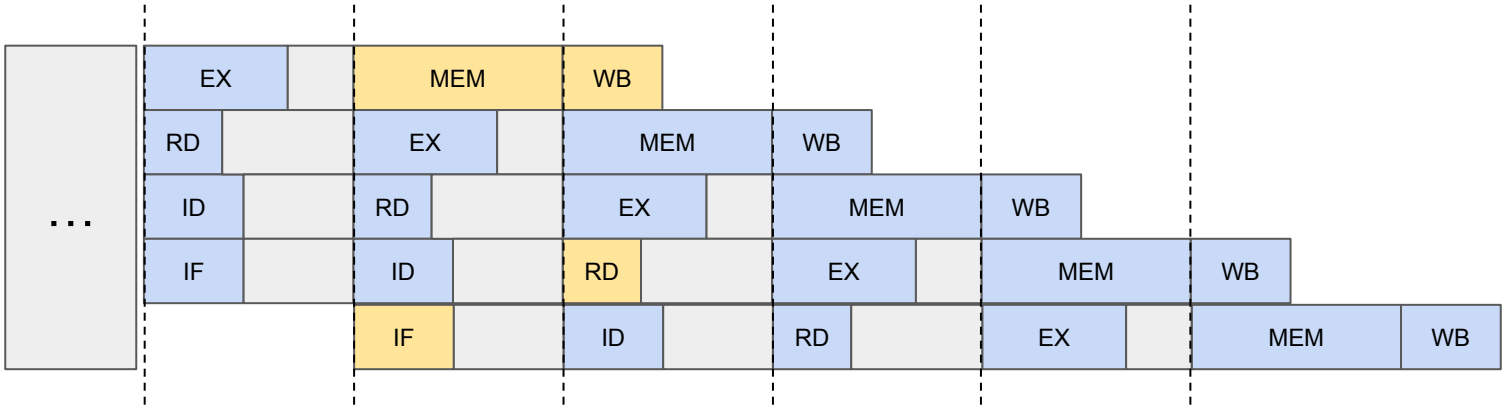
- Ideal scalar pipeline throughput: IPC = 1

Mapping stages to resources: The Datapath



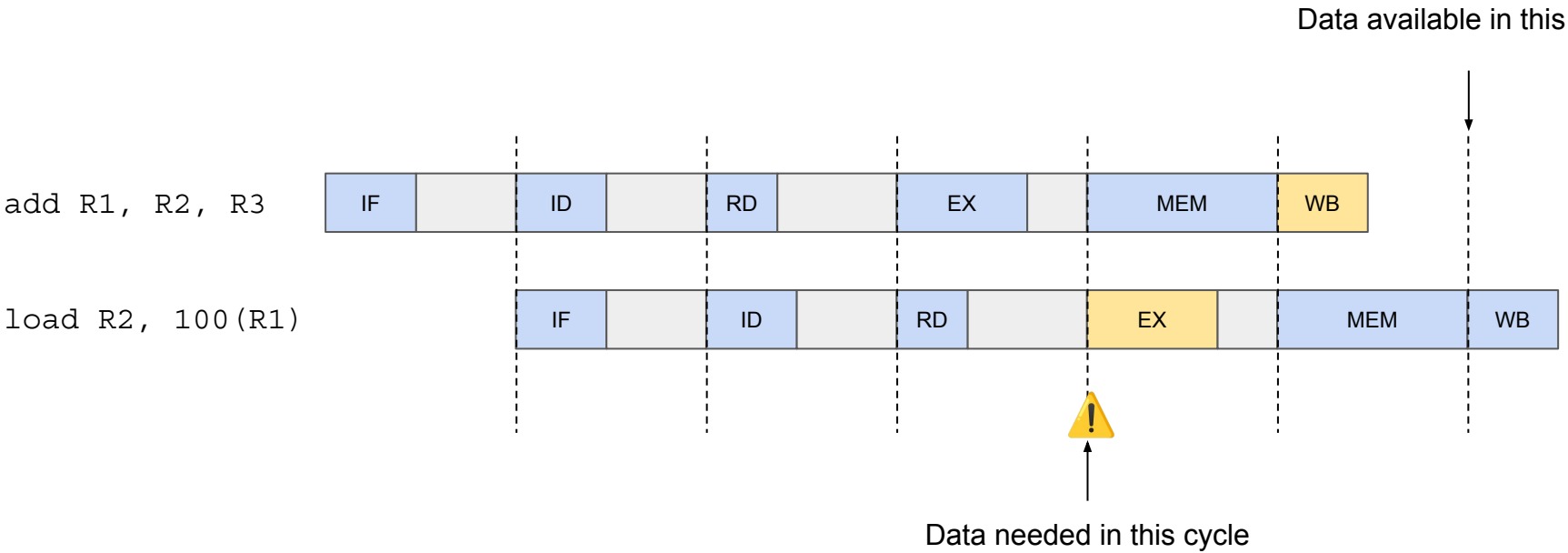
Dangers of pipelining: Structural Hazards

- Two instructions that need to access the same resource during the same cycle
- Example: MEM and IF both access memory
  Do we have separate memory for data and instructions?
- Example: WB writes to the register file and RD reads from it
  Can we support both operations on the same cycle?



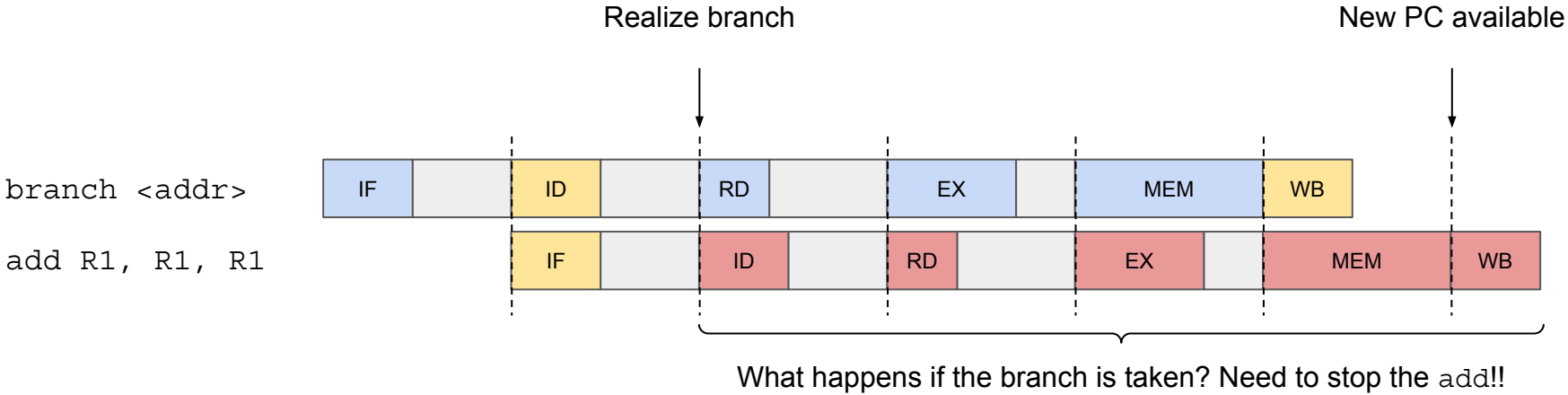
Dangers of pipelining: Data Hazards

- Consider a code with two back-to-back instructions that have a data dependency
- The result of instruction (1) is not ready when instruction (2) starts executing



Dangers of pipelining: Control/Sequence Hazards

- Branch/jump instruction executed → Program Counter (PC) written during the WB stage
- While the next instruction is fetching (IF)
  - We are still decoding the branch instruction
- While the next instruction is decoding (ID)
  - We do not know if the branch is taken or not
  - In case the branch is taken, we do not know which is the target address

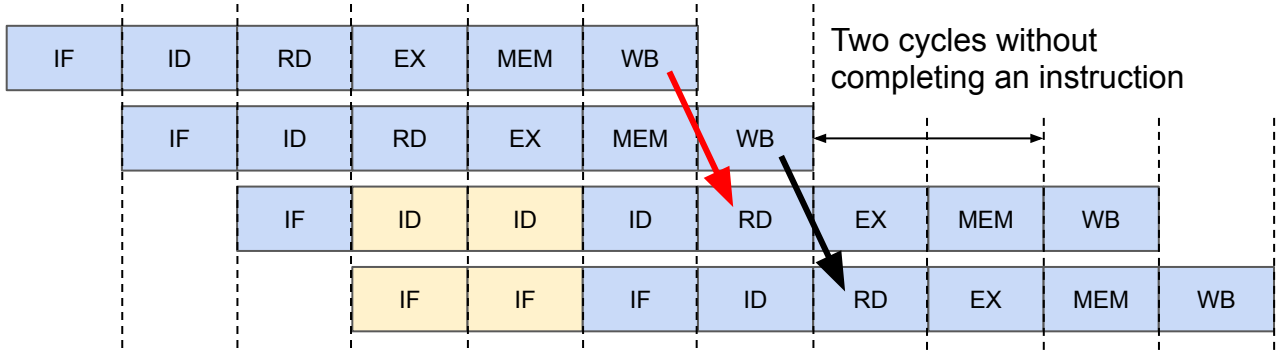


Stalling while hazards are resolved: Bubbles

- The simplest and most conservative approach to solve hazards is to ***stall*** the pipeline
- This is also known as introducing a ***bubble***
- Hazards are identified and pipeline is stalled during the decoding stage
- When an instruction leaves the decoding stage, we say it has been ***issued***

- One cycle were pipeline is stalled → One cycle without completing an instruction

```
xor R1, R1, R1
xor R2, R2, R2
add R1, R1, #5
add R2, R2, #3
```



Remarks about pipelining

1. Throughput over Latency
   - Latency of one instruction (both in cycles and in time) is higher than non-pipelined implementation
   - After startup latency, pipelined implementation completes one Instruction Per Cycle (IPC = 1)

2. Dangers of pipelining
   - Overlapping execution of multiple instructions creates hazards
   - We circumvent hazards by stalling the pipeline

Next session: Improving throughput by increasing Instruction Level Parallelism