# Lecture-3 (Performance Metrics and ISA)
# CS422-Spring 2020

Biswa@CSE-IITK

# What about Multi-core Systems?

Application *i* running on an N-core system

Throughput = $\sum$ IPC (i)

Individual Slowdown (i) = CPI-together (i) / CPI-alone (i)

Weighted Speedup = $\sum$ (IPC-together(i) / IPC-alone (i))

Harmonic Mean of Speedups = N/$\sum$ (IPC-alone(i)/IPC-together (i))

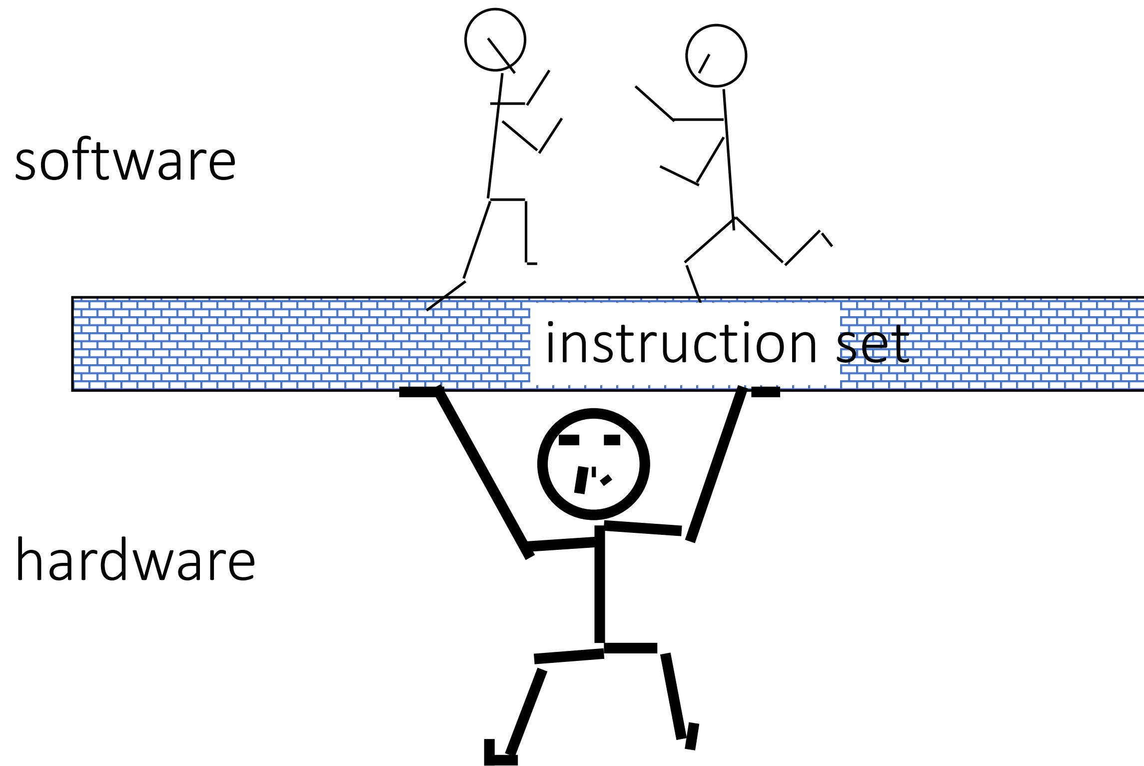
Unfairness =
Max-Slowdown/Min-Slowdown =
max(Individual slowdowns)/min(individual slowdowns)

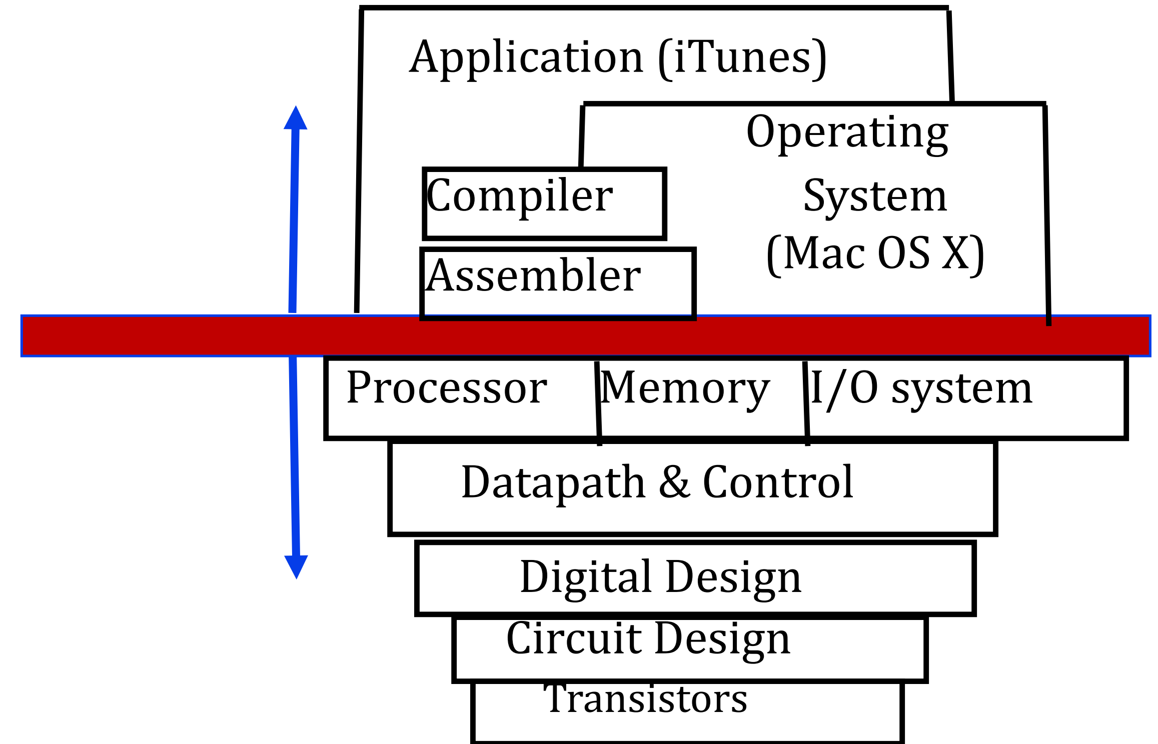
# Example

| | Single Program | IMTEL | AND |
|---|---|---|---|
| App. A | IPC=1 | IPC=0.5 | 1 |
| App. B | IPC=2 | IPC=1 | 0.5 |

| | IMTEL | AND |
|---|---|---|
| Weighted Speedup | 1 | 1.25 |
| **Hmean of Speedups** | **0.5** | **0.4** |
| Amean of IPCs | 0.75 | 0.75 |
| Hmean of IPCs | 0.66 | 0.66 |

# ISA



# ISA

- ***ISA***: Instruction Set Architecture
  - A well-defined hardware/software interface
- The "contract" between software and hardware
  - Functional definition of operations supported by hardware
  - Precise description of how to invoke all features
- No guarantees regarding
  - How operations are implemented
  - Which operations are fast and which are slow (and when)
  - Which operations take more energy (and which take less)

# ISA

*… the attributes of a [computing] system as seen by the programmer, i.e. the conceptual structure and functional behavior*, as distinct *from the organization of the data flows and controls the logic design, and the physical implementation.*

*– Amdahl, Blaaw, and Brooks, 1964*

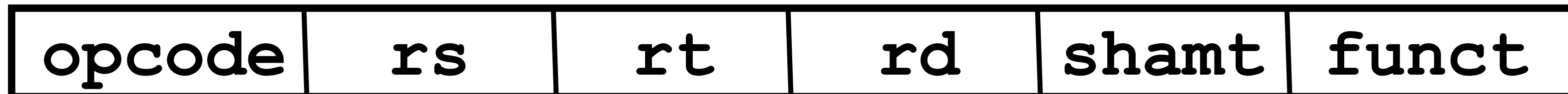
# ISA

- Programmer-visible states
  - Program counter, general purpose registers, memory, control registers
- Programmer-visible behaviors
  - What to do, when to do it

*ISAs last forever, don't add stuff you don't need*

# An Example of Instruction Encoding

Syntax: ADD \$8 \$9 \$10 Semantics: \$8 = \$9 + \$10




Binary: 000000 01001 01010 01000 00000 100000

# Features of the Ideal ISA

- Unambiguous
- Expressive
  - Easily describes all the algorithms that will run on this platform
- Instructions are used
  - Very complex instructions might not be used often
- (Relatively) easy to compile, easy to implement well
- Implementation provides good performance, cost, etc.
- ISAs often highly reliant on microarchitecture and vice-versa
  - Some ISAs easy to implement on some microarchitectures
  - Some microarchitectures make some instructions easy to implement

# Do not get confused with micro-architecture

- ISA
  - Agreed upon interface between software and hardware
    - SW/compiler assumes, HW promises
  - What the software writer needs to know to write and debug system/user programs
- Microarchitecture
  - Specific implementation of an ISA, Not visible to the software
- Microprocessor
  - **ISA, uarch**, circuits
  - "Architecture" = ISA + microarchitecture

# ISA

- Instructions
  - Opcodes, Addressing Modes, Data Types
  - Instruction Types and Formats
  - Registers, Condition Codes
- Memory
  - Address space, Addressability, Alignment, Virtual memory management
- Call, Interrupt/Exception Handling, Access Control, Priority/Privilege
- I/O: memory-mapped vs. instr, Task/thread Management, Power and Thermal Management
- Multi-threading support, Multiprocessor support

# Micro-architecture

- Implementation of the ISA under specific design constraints and goals
- Anything done in hardware without exposure to software
  - Pipelining
  - In-order versus out-of-order instruction execution
  - Memory access scheduling policy
  - Speculative execution
  - Superscalar processing (multiple instruction issue?)
  - Clock gating
  - Caching? Levels, size, associativity, replacement policy, Prefetching?
  - Voltage/frequency scaling? Error correction?

# What is What?

- ADD instruction’s opcode
- Number of general purpose registers
- Number of ports to the register file
- Number of cycles to execute the MUL instruction
- Whether or not the machine employs pipelined instruction execution

- Remember
  - Microarchitecture: Implementation of the ISA under specific design constraints and goals

# Design Point

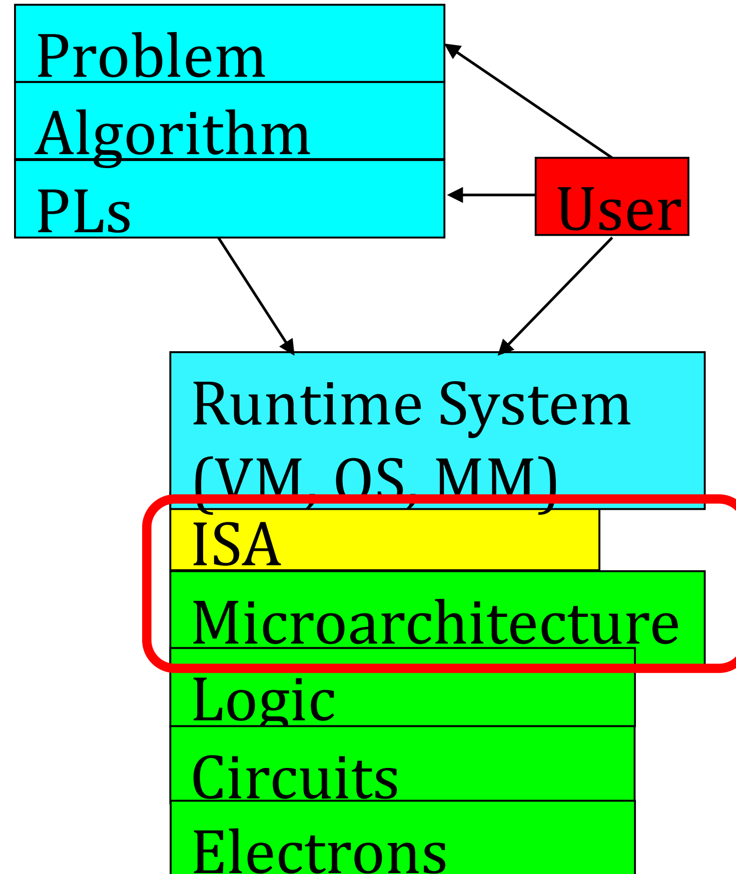
- A set of design considerations and their importance
  - leads to tradeoffs in both ISA and uarch
- Considerations
  - Cost
  - Performance
  - Maximum power consumption and Energy consumption (battery life)
  - Availability
  - Reliability and Correctness
  - Time to Market
- Design point determined by the "Problem" space (application space), the intended users/*market*

# Trade-offs

- ISA-level tradeoffs
- Microarchitecture-level tradeoffs
- System and Task-level tradeoffs
  - How to divide the labor between hardware and software
- *Computer architecture is the science and art of making the appropriate trade-offs to meet a design point*
  - *Why art?*

# Mantra

New demands
from the top
(Look Up)




New demands and
personalities of users
(Look Up)

New issues and
capabilities
at the bottom
(Look Down)

*We do not (fully) know the future (applications, users, market) and it changes*

# ISA in Details

- Which architecture to use: Stack, Accumulator, R-R, R-M
- #Registers
- Trade-offs in terms of #operands, #registers, ….
- Big vs Little Endian
- Importance of Alignment
- Which instructions are dominant and which addressing modes are dominant ?
- Fixed vs Variable Encoding , RISC vs CISC
- Pseudo - instructions
- ISA vs micro-architecture

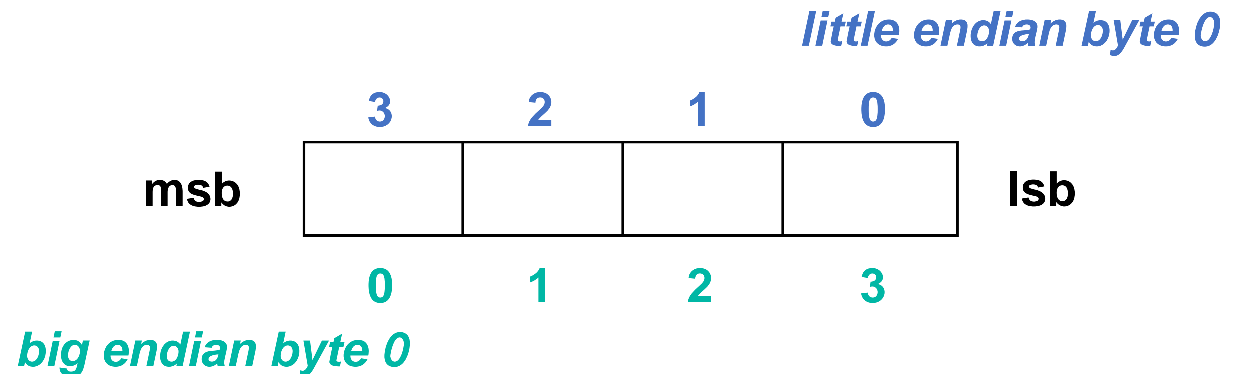
# Architecture, registers, operands, etc

- Instruction processing style
  - Specifies the number of “operands” an instruction “operates” on and how it does so
  - 0, 1, 2, 3 address machines: 0-address: stack machine (push A, pop A, op)
    - 1-address: accumulator machine (ld A, st A, op A)
    - 2-address: 2-operand machine (one is both source and dest)
    - 3-address: 3-operand machine (source and dest are separate)
  - Tradeoffs? Larger operate instructions vs. more executed operations
    - Code size vs. execution time vs. on-chip memory space

# Endianness (Byte Ordering within a Word)

- Big Endian: address of most significant byte = word address (xx00 = Big End of word)
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

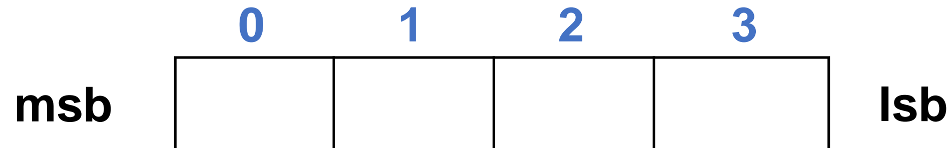
- Little Endian: address of least significant byte = word address (xx00 = Little End of word)
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



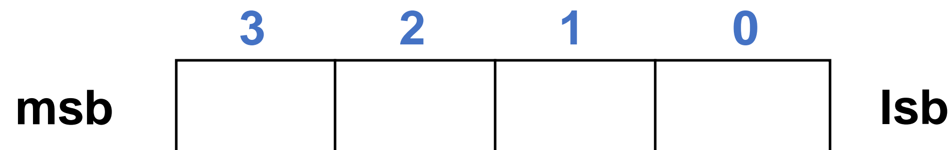
# Example

- int x = ABCD

- Big-endian:




Little-endian:




Big –endian: A B C D, Little-endian: D C B A

But why to have two options? Why not an universal one? What about bi-endianness? Can we check the endianness of our machine? Why not have endianness at the bit level?

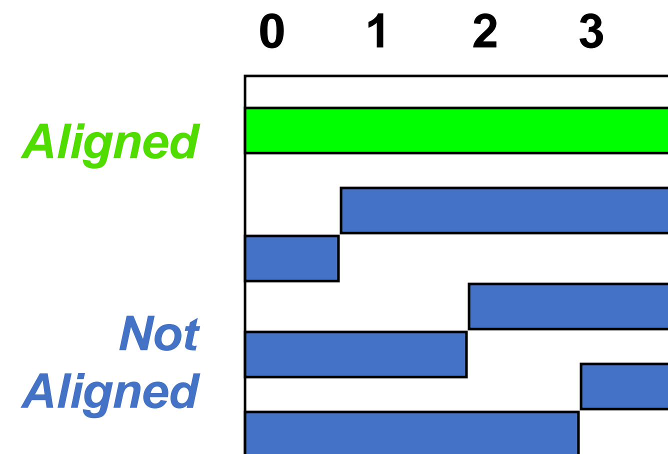
# Alignment (A myth may be ☺)

- Object of size s bytes at byte address A is aligned if
A mod s = 0

- Alignment for faster transfer of data ?

- Why fast ??

- Think about memory.



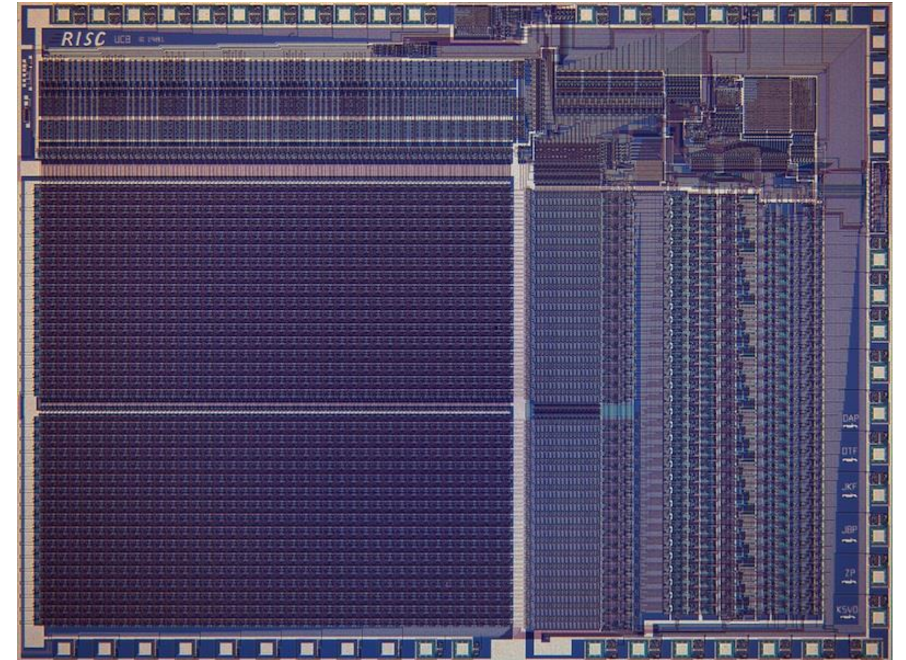
# Dominant Instructions

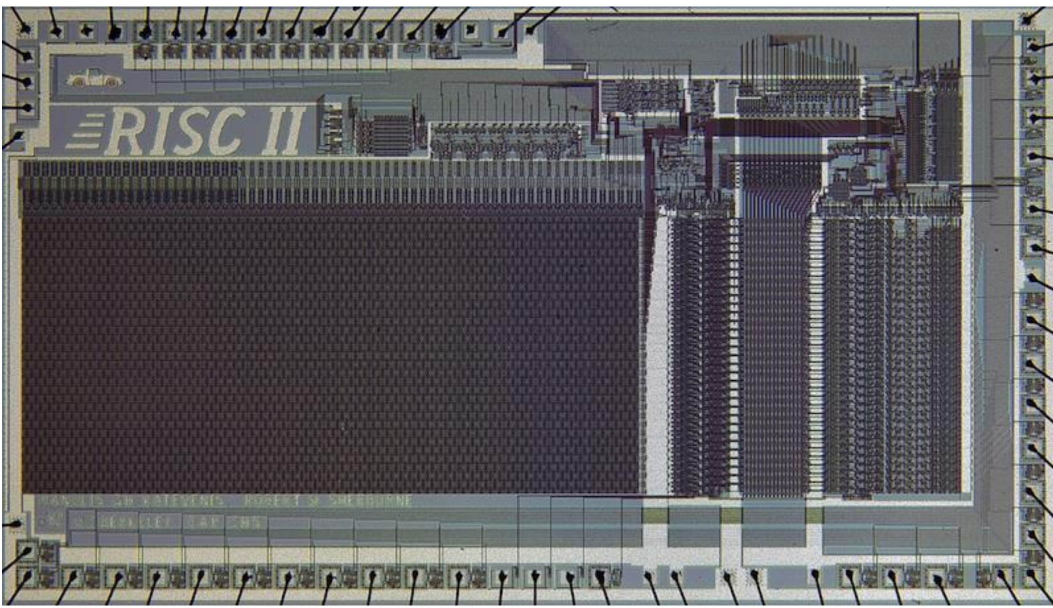
| Rank | instruction | Integer Average Percent total executed |
|---|---|---|
| 1 | load | 22% |
| 2 | conditional branch | 20% |
| 3 | compare | 16% |
| 4 | store | 12% |
| 5 | add | 8% |
| 6 | and | 6% |
| 7 | sub | 5% |
| 8 | move register-register | 4% |
| 9 | call | 1% |
| 10 | return | 1% |
| | Total | 96% |

- **Simple instructions dominate instruction frequency**

# RISC vs CISC

RISC-I (1982) Contains 44,420 transistors, fabbed in 5 μm NMOS, with a die area of 77 mm$^2$, ran at 1 MHz. This chip is probably the first VLSI RISC.




RISC-II (1983) contains 40,760 transistors, was fabbed in 3 μm NMOS, ran at 3 MHz, and the size is 60 mm$^2$.

# RISC vs CISC

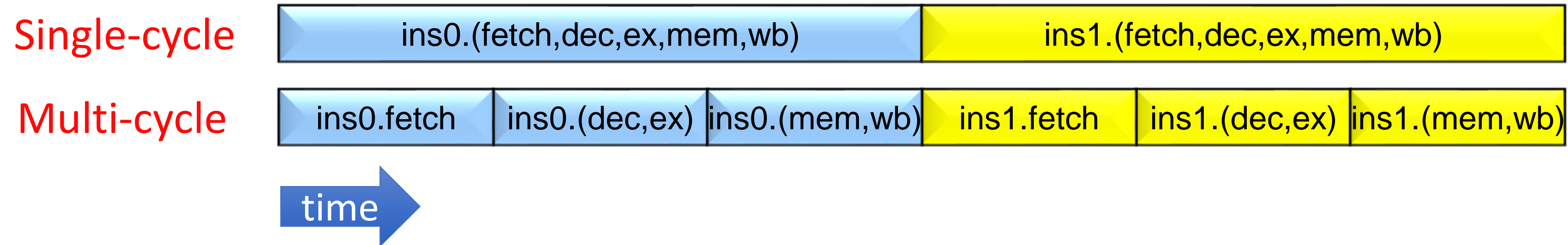
- CISC vs. RISC
  - Complex instruction set computer → complex instructions
    - Initially motivated by "not good enough" code generation
  - Reduced instruction set computer → simple instructions
    - John Cocke, mid 1970s, IBM 801, Goal: enable better compiler control
- RISC motivated by
  - Memory stalls (no work done in a complex instruction when there is a memory *stall*?)
  - Simplifying the hardware → lower cost, higher frequency
  - Enabling the compiler to optimize the code better
    - Find fine-grained parallelism to reduce *stalls*

# Warm-up on Scribing: Due January 21th

- ISA, RISC-CISC, RISC-V
- https://people.eecs.berkeley.edu/~kubitron/courses/cs252-F00/handouts/papers/patterson80.pdf
- http://www.cs.uccs.edu/~xzhou/teaching/CS4520/Assignment/papers/R1_Comment_Case_RISC_SigArchNews_patterson.pdf
- https://engineering.berkeley.edu/magazine/spring-2015/simplify-risc-story

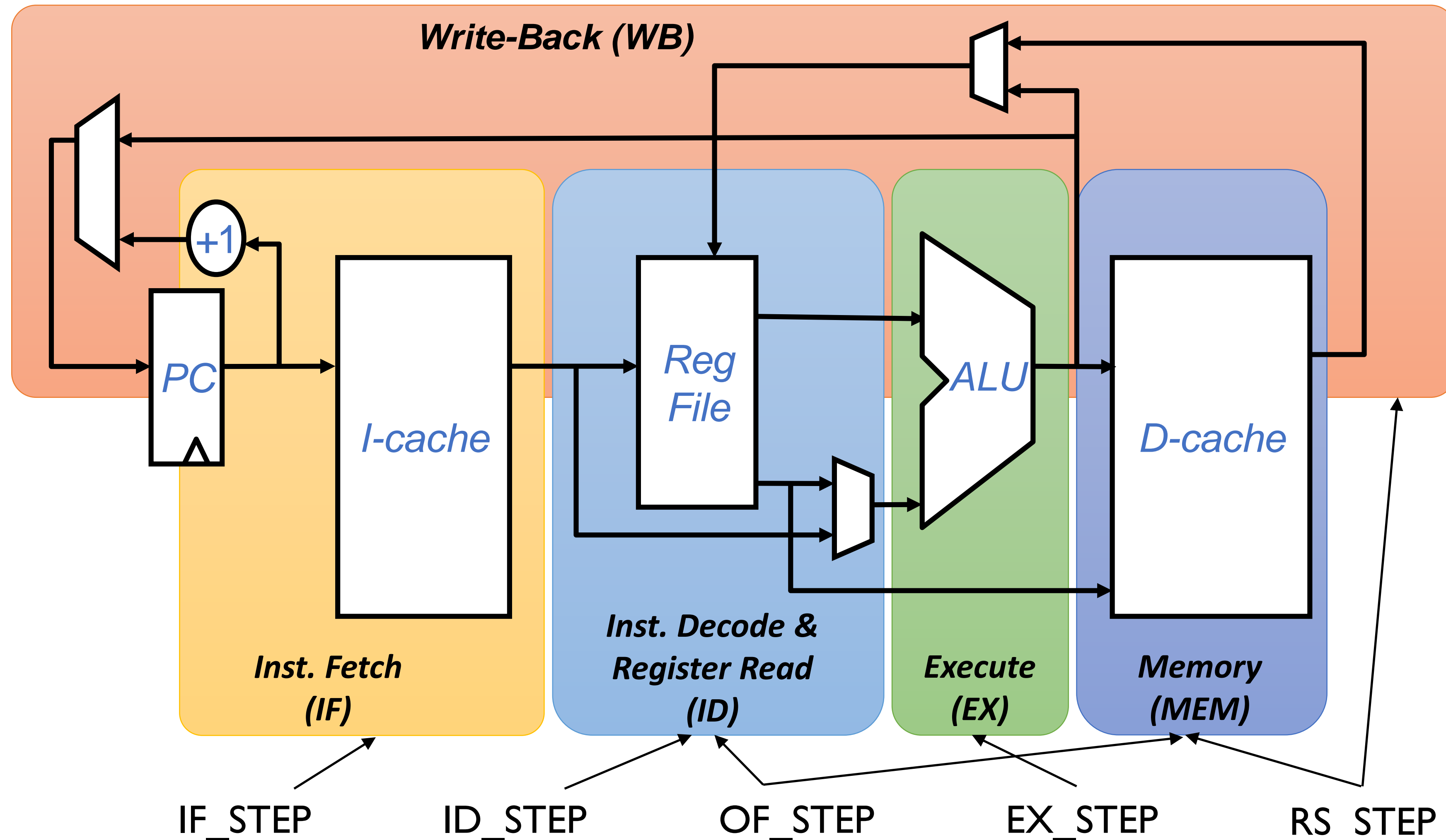
# Instruction Pipelining

# Single-Instruction Datapath




- Process one instruction at a time
- *Single-cycle* control: hardwired
  - Low CPI (1)
  - Long clock period (to accommodate slowest instruction)
- *Multi-cycle* control: typically micro-programmed
  - Short clock period
  - High CPI
- Can we have both low CPI and short clock period?
  - Not if datapath executes only one instruction at a time
  - No good way to make a single instruction go faster

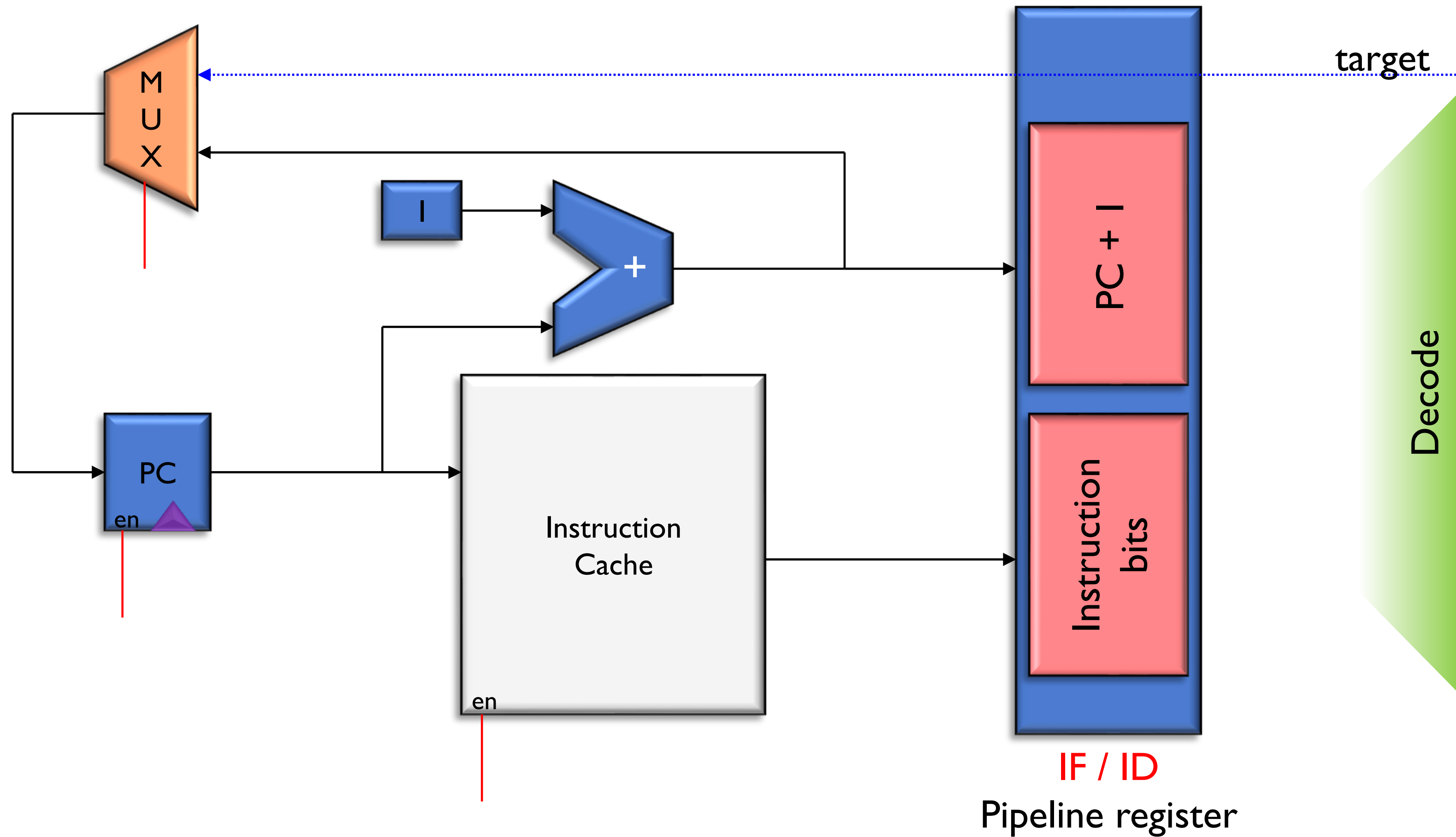
# 5-Stage MIPS Datapath



# Stage 1: Fetch

- Fetch instruction from instruction cache
  - Use PC to index instruction cache
  - Increment PC (assume no branches for now)
- Write state to the pipeline register (IF/ID)
  - The next stage will read this pipeline register

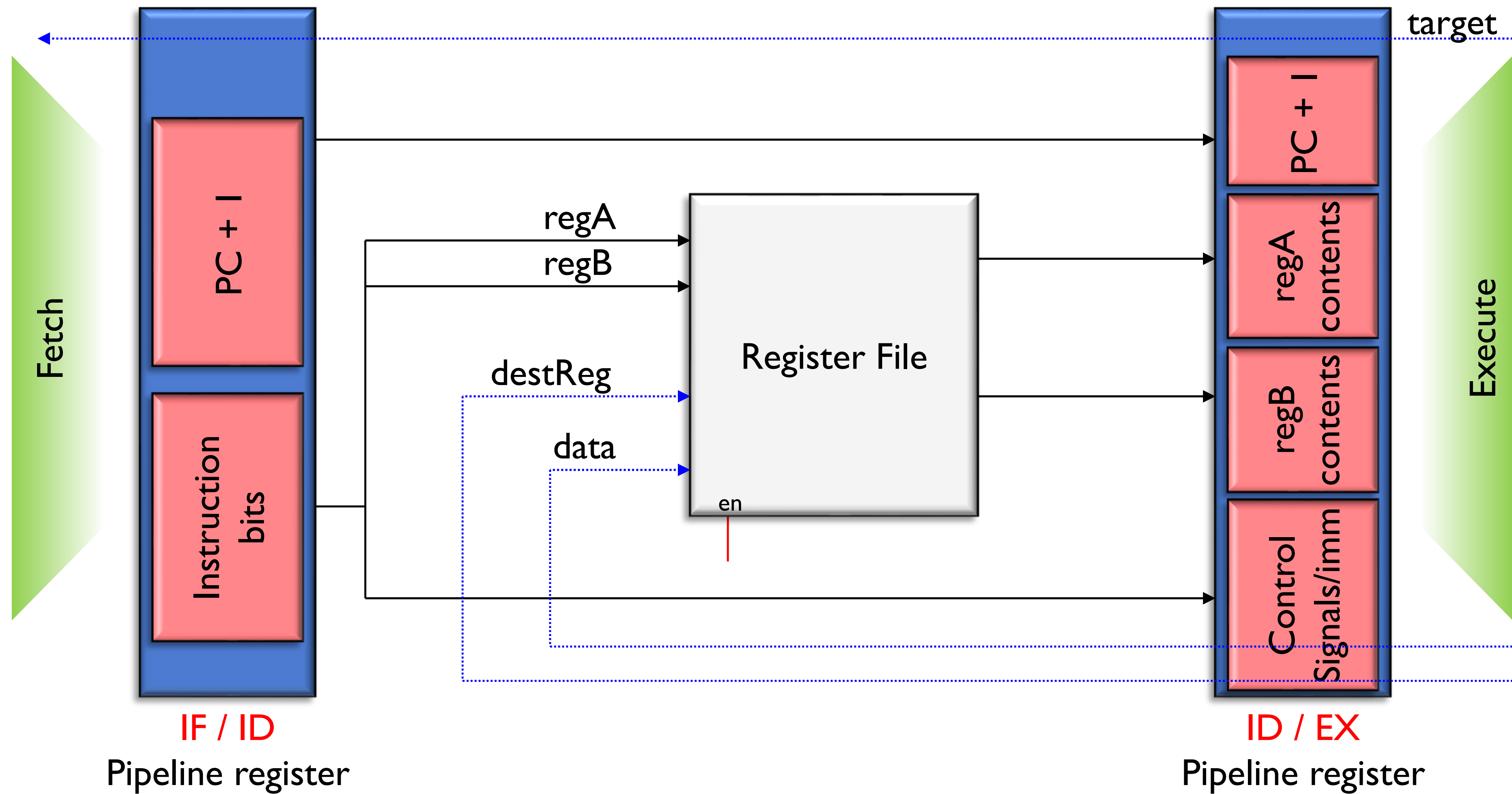
# Stage 1: Fetch Diagram



# Stage 2: Decode

- Decodes opcode bits
  - Set up Control signals for later stages
- Read input operands from register file
  - Specified by decoded instruction bits
- Write state to the pipeline register (ID/EX)
  - Opcode
  - Register contents, immediate operand
  - PC+1 (even though decode didn't use it)
  - Control signals (from insn) for opcode and destReg

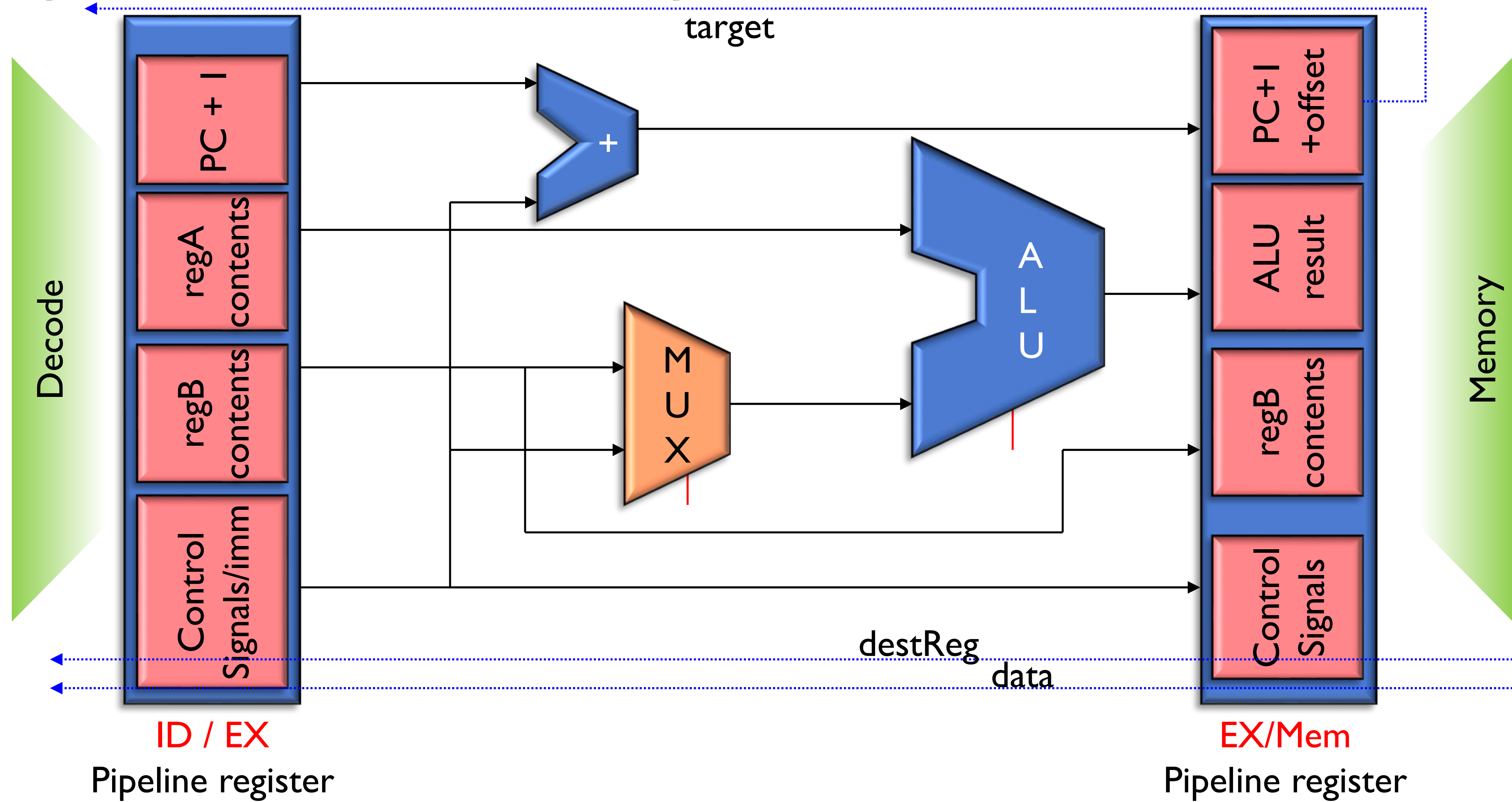
# Stage 2: Decode Diagram



# Stage 3: Execute

- Perform ALU operations
  - Calculate result of instruction
    - Control signals select operation
    - Contents of regA used as one input
    - Either regB or constant offset (imm from insn) used as second input
  - Calculate PC-relative branch target
    - PC+1+(constant offset)
- Write state to the pipeline register (EX/Mem)
  - ALU result, contents of regB, and PC+1+offset
  - Control signals (from insn) for opcode and destReg

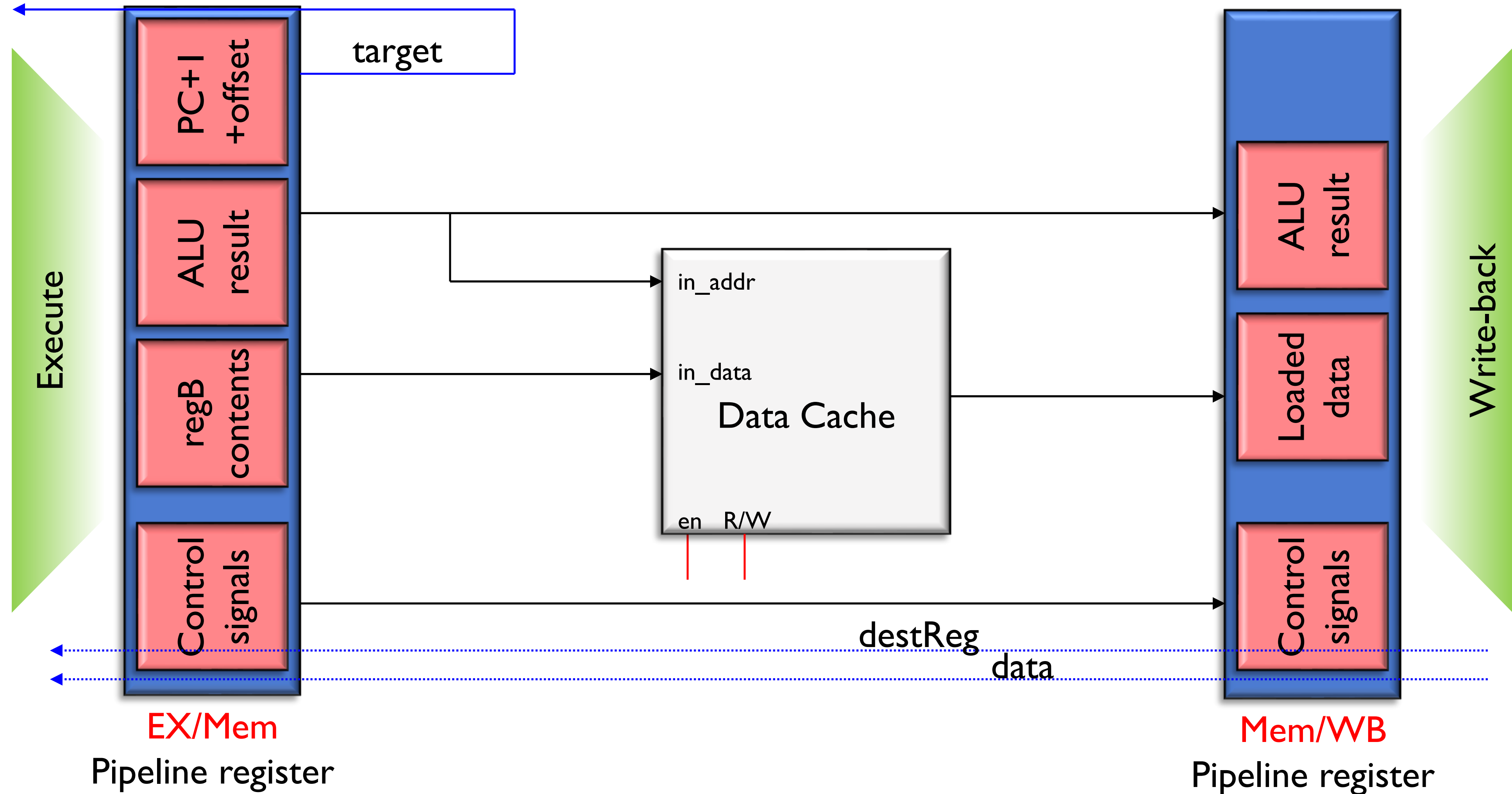
# Stage 3: Execute Diagram



# Stage 4: Memory

- Perform data cache access
  - ALU result contains address for LD or ST
  - Opcode bits control R/W and enable signals
- Write state to the pipeline register (Mem/WB)
  - ALU result and Loaded data
  - Control signals (from insn) for opcode and destReg

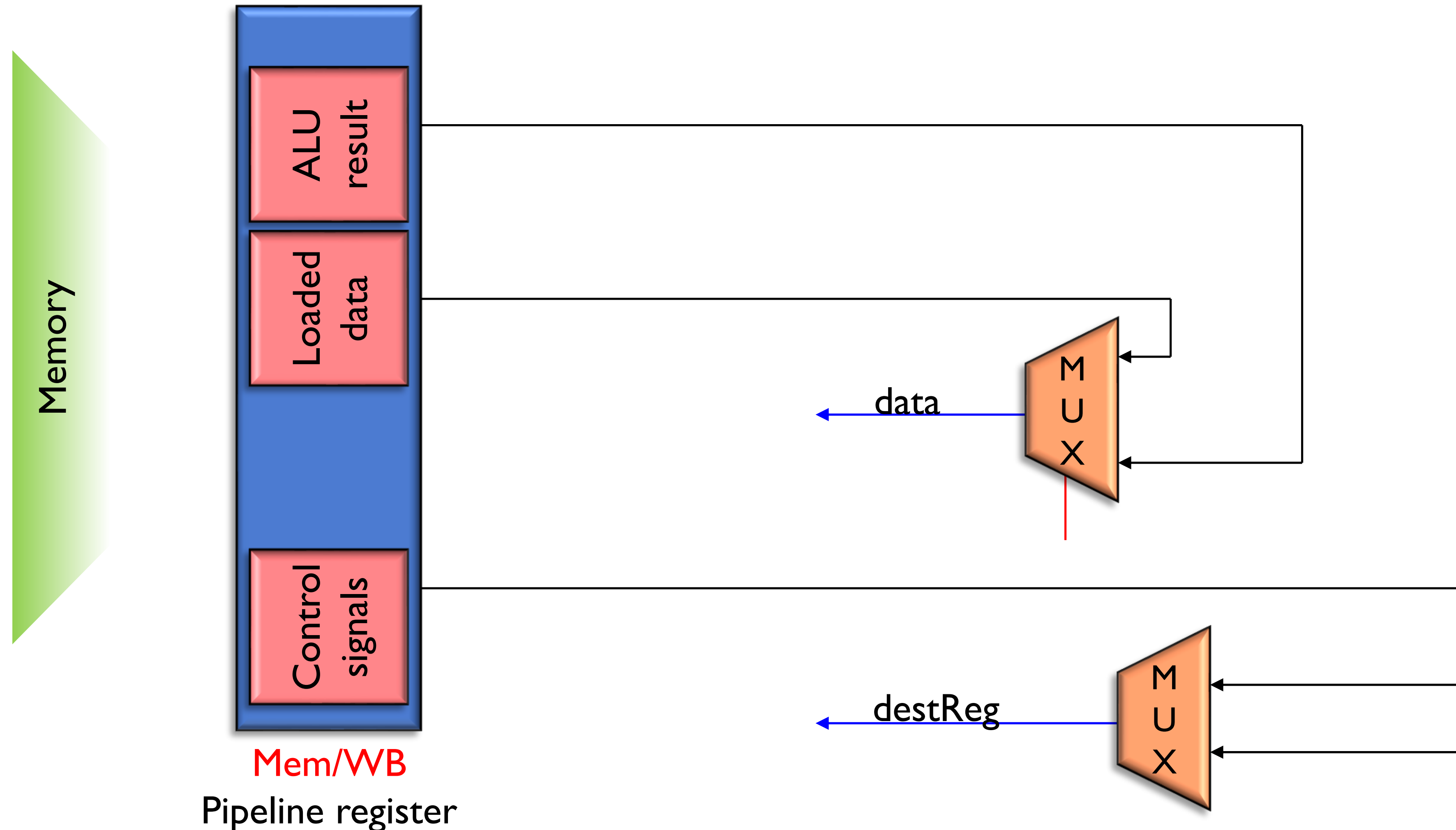
# Stage 4: Memory Diagram



# Stage 5: Write-back

- Writing result to register file (if required)
  - Write Loaded data to destReg for LD
  - Write ALU result to destReg for ALU insn
  - Opcode bits control register write enable signal

# Stage 5: Write-back Diagram



# Putting It All Together