Instruction Set Architecture

Wei Wang

Optional Readings from Textbooks

- “Computer Organization and Design,” Chapter 2 “Instructions: Language of the Computer.”
- “Computer Architecture: A Quantitative Approach,” Appendix A “Instruction Set Principles.”

Road Map

- Basics of Instruction Set Architecture
- ISA Design Choices and Classification
- CISC vs RISC
- ISA Implementation Overview
- SIMD Instructions
- Compiler Interactions

Basics of Instruction Set Architecture

Instruction Set Architecture

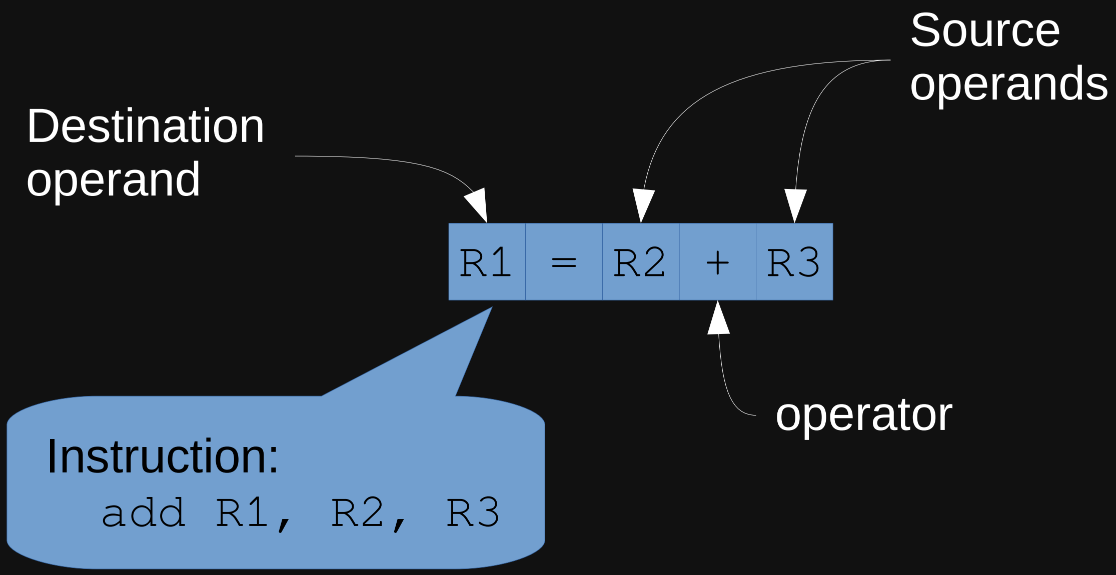
- “Instruction Set Architecture is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct (timing independent) program for that machine.” – IBM, Introducing the IBM 360 (1964)
- The ISA defines:
  - Operations that the processor can execute
  - Data Transfer mechanisms + how to access data
  - Control Mechanisms (branch, jump, etc)
  - “Contract” between programmer/compiler + HW

The Fundamental Requirement of ISA

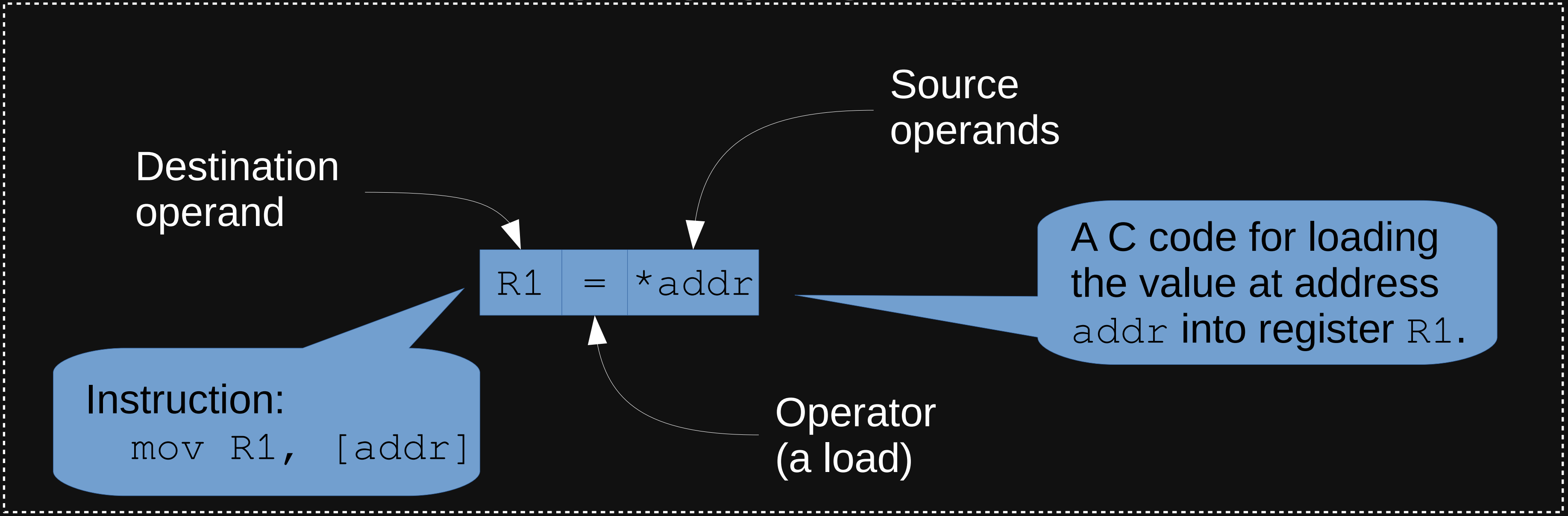
- ISA must be Turing complete.
  - So that it can run any computer programs.
- An architecture is Turing complete means:
  - The architecture can simulate any Turing machine
  - Intuitively, a Turing complete architecture can run any computer programs.

The Standard Structure of An Instruction

- An instruction typically has an operator (op or opcode), one or two source operands (src), and one destination operand (dest).

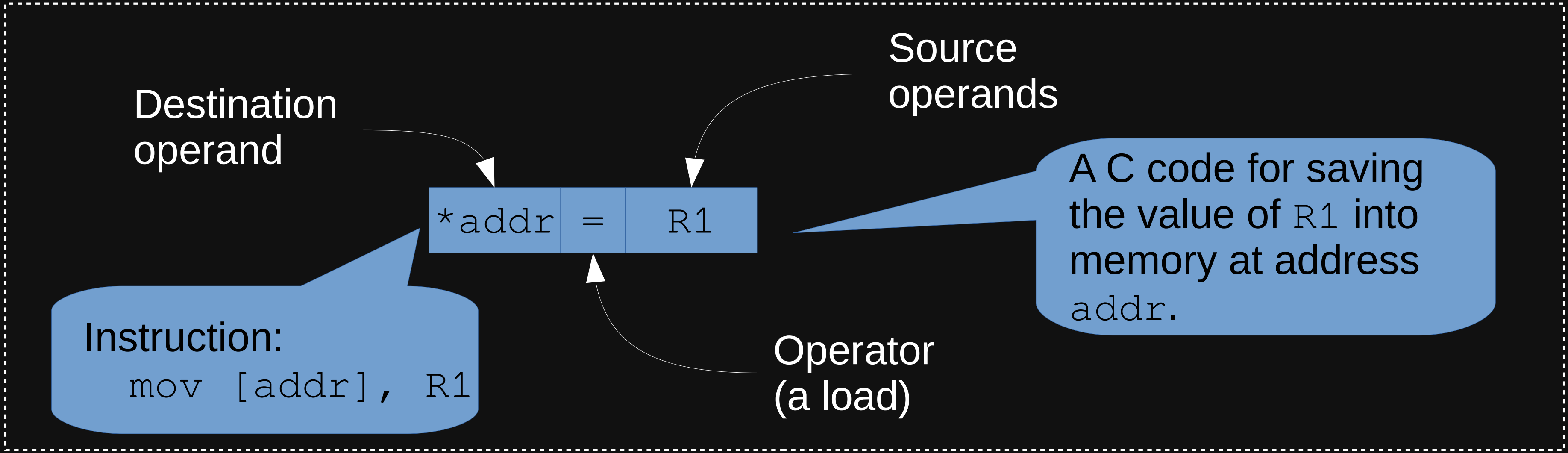


More Examples with Memory Instructions



A C code for loading the value at address `addr` into register `R1`.

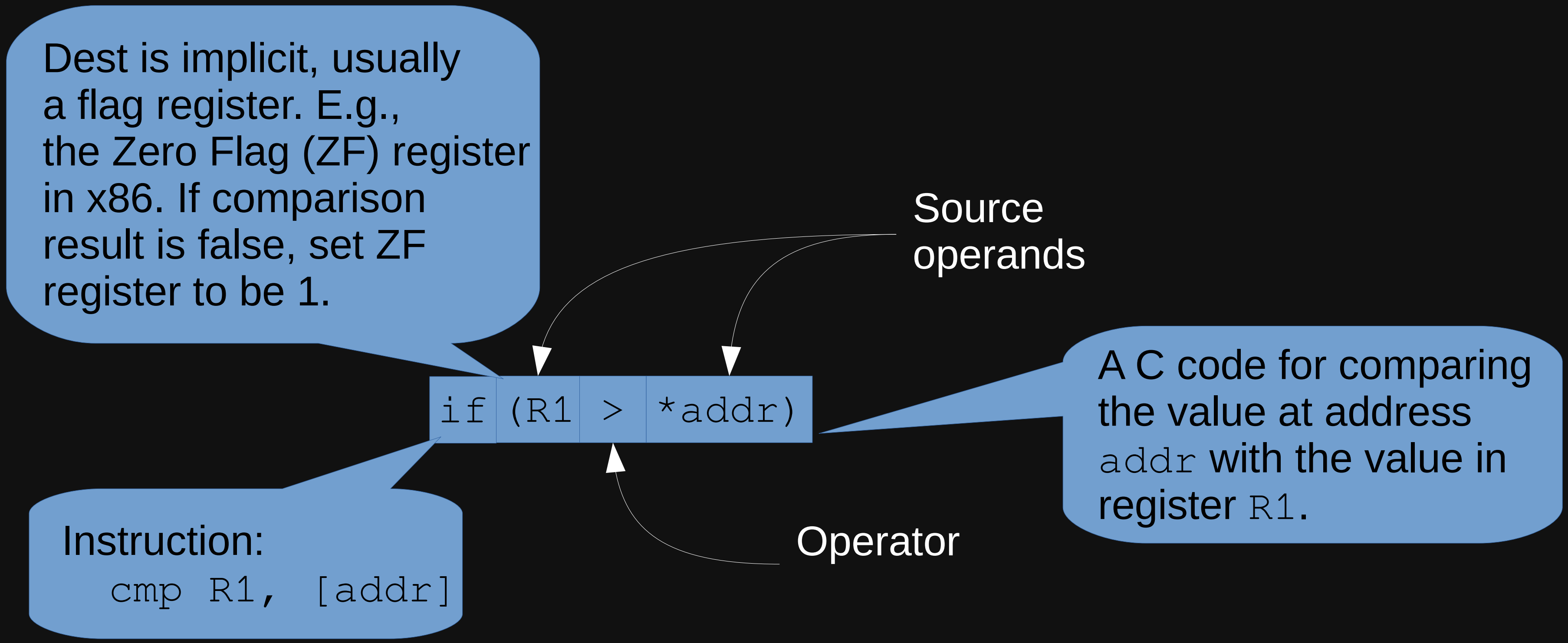
Instruction:
```
mov R1, [addr]
```



A C code for saving the value of `R1` into memory at address `addr`.

Instruction:
```
mov [addr], R1
```

More Examples with Logic Instruction



Dest is implicit, usually a flag register. E.g., the Zero Flag (ZF) register in x86. If comparison result is false, set ZF register to be 1.

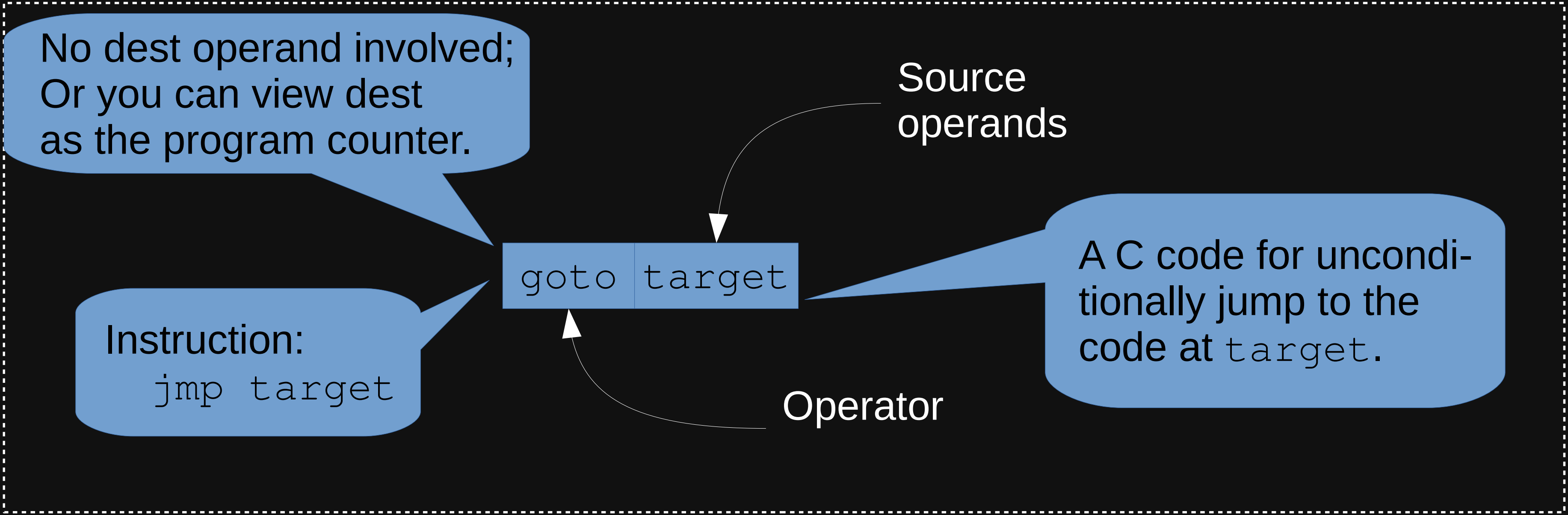
```
if (R1 > *addr)
```

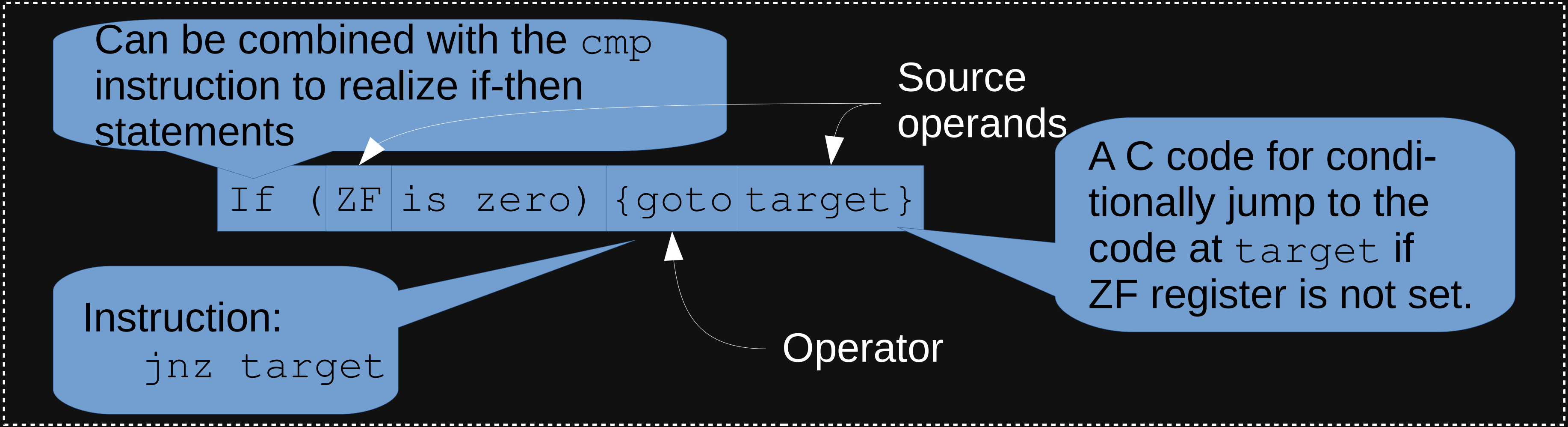
A C code for comparing the value at address `addr` with the value in register `R1`.

Instruction:

```
  cmp R1, [addr]
```

More Examples with Jump Instructions




Can be combined with the cmp instruction to realize if-then statements
Source operands
If (ZF is zero) {goto target}
A C code for conditionally jump to the code at target if ZF register is not set.
Instruction: jnz target
Operator

Confusing Instruction Syntaxes

- Intel syntax for x86 instructions:
  `op  dest, src1, src2`
- ATT syntax for x86 instructions:
  `op src1, src2, dest`
- MIPS syntax for MIPS instructions:
  - mostly `op dest, src1, src2`
  - But for store, it is "`sw src, dest`"
- ARM syntax for ARM instructions:
  `op register, src or dest`
  - Whether the second operand is src or dest depends on op.
- I will use a syntax similar to the Intel syntax for x86 instructions; for MIPS and ARM, I will use their standard format.

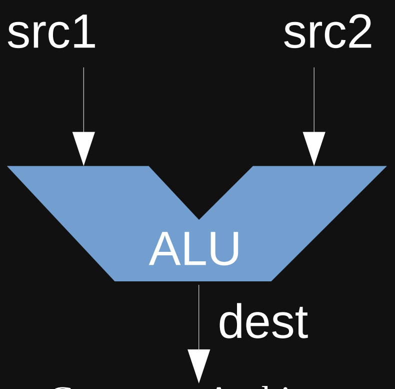
Key ISA Decisions

- Operations:
  - How many?
  - Which ones?
- Operands:
  - How many?
  - Location?
  - Types?
  - How to specify?
- Instruction format
  - How many bytes per instruction?
  - How many formats?

ISA Design Choices and Classification

Arithmetic Logic Unit

- An arithmetic logic unit (ALU) is a digital electronic circuit that performs arithmetic operations on source operands and produces result.
  - An ALU can be any arithmetic or bitwise logic operations
  - The operands must have integer values.



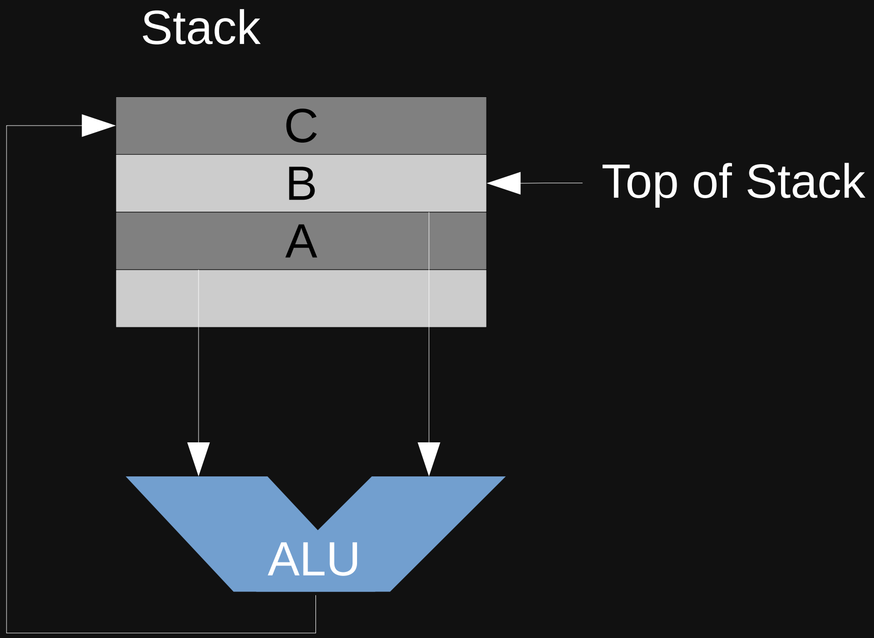
Standard illustration of ALU

ISA Classification based on Operands

- Operands may be from
  - Stack
  - Accumulator
  - Register
  - Register and Memory

Classifying ISA: Stack ISA

- The operands are read from a stack (inside CPU, not memory)..

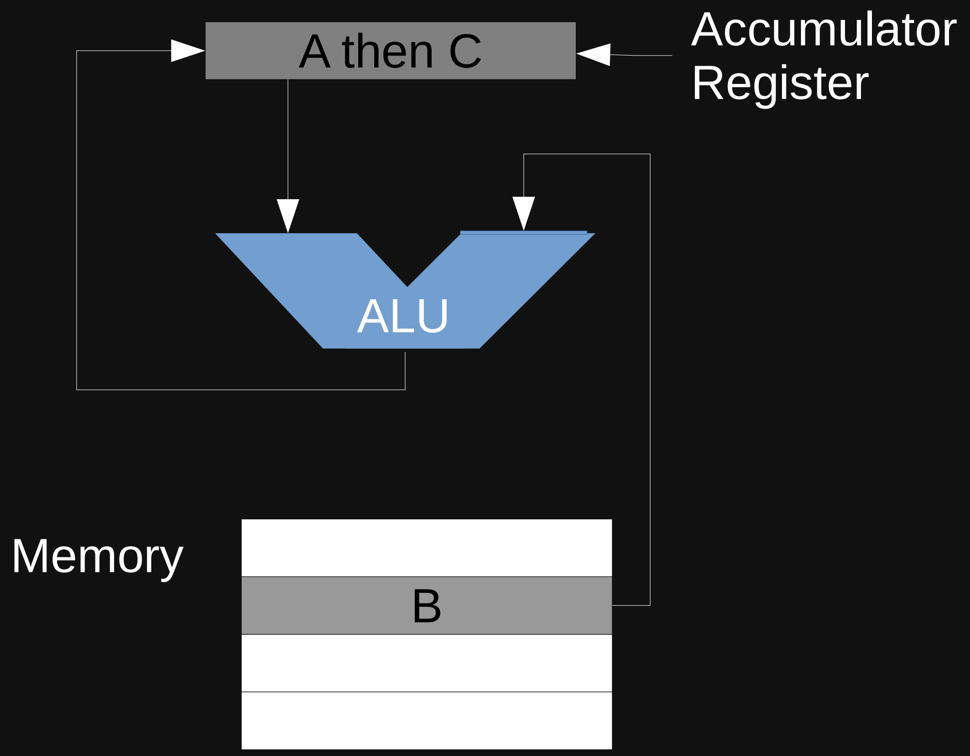


Stack

- Architectures with implicit “stack”
  - Acts as source(s) and/or destination, Top of Stack (TOS) is implicit
  - Push and Pop operations have 1 explicit operand
- Example: C = A + B
  - Push A   // S[++TOS] = Mem[A]
  - Push B   // S[++TOS] = Mem[B]
  - Add   // Tem1 = S[TOS--], Tem2 = S[TOS--] ,
    S[++TOS] = Tem1 + Tem2 (this is C)
  - Pop C   // Mem[C] = S[TOS--]
- x86 FP uses stack (complicates pipelining)

Classifying ISA: Accumulator

- The operands are read from memory and an accumulator

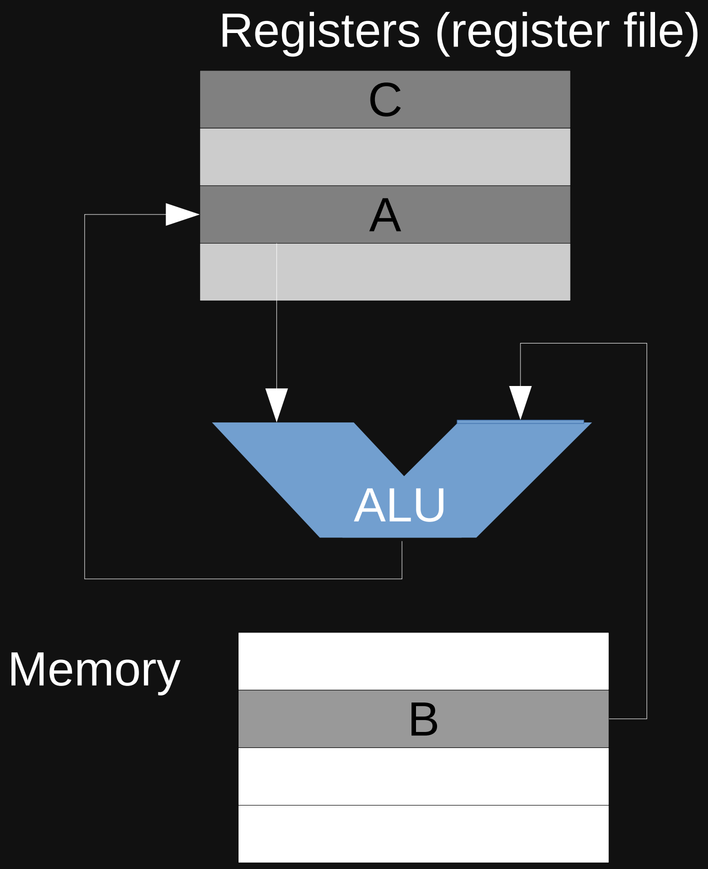


Accumulator

- Architectures with one implicit register
  - Acts as source and/or destination
  - One other source explicit
- Example: C = A + B
  - Load A   // (Acc)umulator <= A
  - Add [B]   // Acc <= Acc + B
  - Store C   // C <= Acc
- Accumulator is implicitly used, and can become a performance bottleneck
  - Although in real CPUs, accumulator instructions are just conceptual. The actually implementation is register-register.
- x86 uses accumulator *conceptually* for most integer operations.

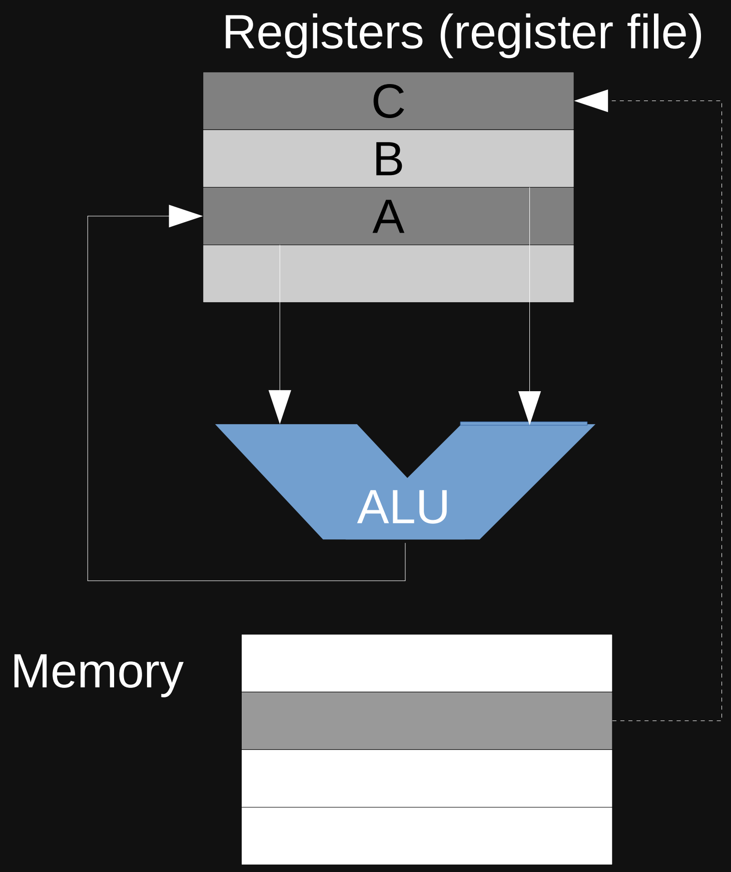
Classifying ISA: Register-Memory

- The operands are read from random-accessible memory and register file.



Classifying ISA: Register-Register/Load-Store

- The operands are fed into the ALU from registers. But the values of the operands are explicitly read from memory.



Register-Memory and Register-Register with Load/Store

- Most common approach
  - Fast, temporary storage (small)
  - Explicit operands (register IDs)
- Example: C = A + B

Register-memory Instructions

```
Load R1, [A]
Add  R3, R1, [B]

Store [C],   R3
```

Reg-reg with load/store instructions

```
Load R1, [A]
Load R2, [B]
Add   R3, R1, R2
Store [C], R3
```

- x86 ISA is mostly register-memory.
- All RISC ISAs are register-register with load/store.
  - E.g., ARM ISA is RISC and register-register.

Clarification on x86 ISA Operands

- x86 uses a stack for floating point (FP) operations
- x86 integer instructions operates in an accumulator fashion
  - Most instructions have a syntax of “`op dest, src`”, where `dest` used as both source and destination, much like an accumulator
  - But `dest` is still a register in the register file, not a real accumulator
- x86 integer instructions are really just register-memory. There is no real accumulator in x86 processors.

ISA Classification based on Operand Addressing Mode

- Addressing mode specifies how operands are located from memory and/or registers.
- Common addressing modes
  - Register
  - Immediate
  - Register indirect
  - Displacement
  - Indexed
  - Direct
  - Memory indirect
  - Auto-increment and auto-decrement
  - Scaled

Common Addressing Modes

- Register:
  - When an operand is in a register
  - Example,
    - For operation: `R1 = R2 + R3`
    - The instruction: `add R1, R2, R3`
- Immediate:
  - When an operand is a constant value
  - Example
    - For operation: `R1 = R3 + 3`
    - The instruction: `add R1, R3, 3`

Common Addressing Modes cont.

- Register Indirect (aka register deferred)
  - Accessing an operand whose memory address stored in register
  - Often used for C pointers
  - Example
    - Assuming the address of 2nd operand is in `R3`.
    - For operations:
      `R1 = R2 + *R3;`
    - The instruction: `add R1, R2, [R3]`

Common Addressing Modes cont.

- Displacement
  - Accessing an operand whose address is computed based on a base address and an offset (displacement)
  - Often used for local variables on the memory stack, where the base address is the stack frame pointer
  - Example
    - Assuming base address is in `R3`, and offset is `100`
    - For operations:
      ```
      addr = R3 + 100;
      R1 = R2 + *addr;
      ```
    - The instruction: `add R1, R2, [R3+100]`

Common Addressing Modes cont.

- Indexed:
  - Very useful for accessing an element of an array in memory.
  - The beginning address and the offset of the element are stored in registers.
  - Example:
    - Consider a summation with x[5] as an operand:
      ```
      R1 = R2 + x[5]
      ```
    - Assuming R3 has the base address of x, and R4 has the offset:
      ```
      int x[10];
      R3 = x; // set the beginning address of x to R3
      R4 = 5 * 4; // the offset of x[5], which is 5 * 4 bytes = 20
                     bytes
      ```
    - For operation:
      ```
      addr = R3 + R4
      R1 = R2 + *addr
      ```
    - The instruction: `add R1, R2, [R3+R4]`

Common Addressing Modes cont.

- Direct:
  - When the memory address of operand is known at compile time. That is, the value of the address is known at compile time.
  - Typically used for loading static and global variables.
  - Example:
    - Assuming one operand is at memory address 1000
    - The operation:
      ```
      addr = 1000
      R1 = R2 + *addr
      ```
    - The instruction: `add R1, R2, [1000]`

Common Addressing Modes cont.

- Memory indirect (aka, memory deferred):
  - When accessing nested pointers.
  - Example:
    - Assuming the first address is in `R3`
    - The operation:
      ```
      // basically, R1 = R2 + mem[mem[R3]]
      addr = *R3
      R1 = R2 + *addr;
      ```
    - The instruction: `add R1, R2, @(R3)`

Common Addressing Modes cont.

- Auto-increment:
  - Useful when stepping through an array.
  - The address of current element is stored in an register. And this register is automatically incremented as the code proceeding to iterate over every element in the array.
  - Example:
    - Assume the beginning address of array `x` is in `R2`. That is,
      ```
      int x[…];
      R2 = x;
      ```
    - The operation:
      ```
      while(true){
           R1 += *R2;
           R2 += 4; // advance 4 bytes to next element
      }
      ```
    - The instructions:
      ```
      Loop: add R1, R1, (R2)+
            goto loop
      ```
- Auto-decrement is similar, expect the the address is automatically decremented.

Common Addressing Modes cont.

- Scaled:
  - Used to access an array element with the array’s based address, the element index and the size of an element (scale)
  - Can handle arrays with any type of elements, such as chars, integers, floats and doubles
  - Example:
    - Assuming the beginning address of array `x` is in `R3:`
      ```
      double x[..];
      R3 = x;
      ```
    - The operation:
      ```
      // basically R1 = R2 + x[10]
      R4 = 10;
      R1 = R2 + R3[R4];
      ```
    - The instruction: `add R1, R2, [R3 + R4 * 8] //a double takes 8 bytes`

A More Complex Addressing Example

- Consider a case where an element of a two dimensional array is used as operand:
  `double x[10][10]`
  `R1 = R2 + x[3][5]`
- The location of `x[3][5]` is `x + 240 + 5 * 8`
  - `x[3][5]` starts from the 4th row, the first 3 rows occupies `3 rows * 10 elements/row * 8 bytes/element = 240 bytes`
  - `x[3][5]` is the 6th element on the 4th row. Therefore, it is starts after the first 5 elements, i.e., `5 elements *  8 bytes`.
  - In other words, the offset/displacement for the 4th row is `240` bytes, the index is `5` within the 4th row, and the element size/scale is `8`
- Again, assuming the beginning address of `x` is in `R3`. And the index `5` is in `R4`.
- The instruction for `R1 = R2 + x[3][5]` is then:
  `add R1, R2, [R3 + R4 * 8 + 240]`
- This addressing mode, `[base + index * scale + disp]`, is the memory addressing mode used by x86. This addressing mode can replace or include most of the addressing modes we have discussed so far.

ISA Classification by Instruction Length

- Instructions are eventually encoded with `0`s and `1`s.
  - Each instruction is encoded into several bytes of binary numbers.
- There are two types of ISA in terms of instruction length:
  - Variable-length ISA
  - Fixed-length ISA
- The choice in encoding affects,
  - How hard it is to decode an streams of bytes back into an instruction.
  - How much memory space a program can take.

Variable-length instructions

- Variable-length instructions (x86, VAX) require multi-step fetch and decode, but allow for a much more flexible and compact instruction set.
  - Low on memory usage, since many simple instructions only use one or two bytes.
  - Hard to decode. The decoder needs to first determine the length of next instruction in the memory before decoding it.
- For example, in x86-64:
  - The encoding for "`add r12, r11`" is "`0xdc014d`", which has three bytes
  - The encoding for "`mov r14, [r13]`" is "`0x00758b4d`", which has four bytes

Fixed-length instructions

- Fixed-length instructions allow easy fetch and decode, and simplify pipelining and parallelism
  - Easy to decode. There is no need to determine instruction length as each instruction always has the same length.
  - Take more memory space. As the length of the instructions is determined by the most complex instruction.
- For example
  - MIPS instructions are always 32 bits, with 6 bits for opcode and the rest for operands
  - Standard ARM instructions are also 32 bits, with variable length for opcodes.

How Many Operands?

- Most instructions have three operands (e.g., `z = x + y`).
- Most ISAs specify 0-3 (explicit) operands per Instruction.
- Operands can be specified implicitly or explicitly.
  - An accumulator-styled instruction (e.g., multiplication in x86) uses dest implicitly as a src operand.
- Generally, only one operand can be from memory.

The Ultimate Classification

- As we have seen, there are many ways to classify ISAs.
- There is one classification that considers most of the features we have discussed so far, which includes two classes of ISAs: CISC and RISC
  - CISC: Complex Instruction Set Computers
  - RISC: Reduced Instruction Set Computers

CISC and RISC

What leads to a good/bad ISA?

- Ease of Implementation (in processors)
  - Does the ISA lead itself to efficient implementations?
- Ease of Programming (for programmers)
  - Can the compiler use the ISA effectively?
- Future Compatibility
  - ISAs may last 30+yrs
  - Special Features, Address range, etc. need to be thought out

Implementation Concerns

- Simple Decoding (fixed length)
  - Instructions are encoded with 0/1 bits
  - Every instruction has the same length
    - E.g., the first 8 bits always represents the opcode; the next 8 bits for dest, then 8 bits for src1 and 8 bits for src2.
  - Most RISC ISAs, such as MIPS and ARM, have fixed length instruction
  - Simple decoding means simpler processor implementation
- Compactness (variable length)
  - Instructions have variable length
    - E.g, some instructions have 1 bit as opcode, some have 4 bits as opcode, some have 8 bits.
  - x86 ISA have variable length instructions.
  - Saves memory, but huge headache to implement
  - Most processors today internally used fixed-length micro-coded instructions (including x86) for simplicity.

Implementation Concerns cont.

- Simple Instructions (no register-memory instructions)
  - Things that get microcoded these days
  - Deterministic Latencies are key reason. That is, every instruction finished within the same time, making instruction scheduling much easier.
  - Instructions with multiple exceptions are difficult.
    - For register-memory instructions, an instruction may trigger both arithmetic exceptions and memory exceptions.
- More or less registers?
  - More registers are not always good. Once register file is too large, accessing it will be slow.
- Condition codes/Flags
  - Many instructions have side-effects. E.g., add has carry-over bits.
  - The selection of condition/flags registers affects instruction scheduling.

Programmability

- Well, at this moment you may think x86 is horrible as it has variable lengths and have complex register-memory instructions.
  - But programmability is also very important.
- Before mid 80s, programmability is nearly the most important design issue
  - 1960s, early 70s
    - Code was mostly hand-coded
  - Late 70s, Early 80s
    - Most code was compiled, but hand-coded was better
  - CISC ISAs provides register-memory styled instructions and many special instructions for special use cases. Therefore, CISC ISAs were early winners.
- After mid-80s, programmability becomes less important for ISA, due to better compilers
  - Mid-80s to Present
    - Most code is compiled and almost as good as assembly
  - Why?
    - Optimizing large amount of code is too difficult for human.
- RISC ISAs made a successful come-back in 21st century partially as programmability becomes less important.
- Note that programmability is less important today only for ISAs. Programmability is still very important in the fight among CPUs/GPUs/FPGA/ASICs, and is still very important for high-level languages.

ISA Compatibility

- Backward-compatibility
  - Never abandon existing code base
  - Extremely difficult to introduce a new ISA
    - Alpha failed, IA-64 is done, best solutions may not win (Alpha and IA-64 are not the best BTW).
  - x86 most popular, is the least liked!
- Extensible for the future
  - Hard to think ahead, but...
    - ISA tweak may buy 5-10% today
    - 10 years later it may buy nothing, but must be implemented

CISC and RISC

- Debate raged from early 80s through 90s
  - Now it is fairly irrelevant
- Despite this debate, Intel (x86 => Itanium) and DEC/Compaq (VAX => Alpha) have tried to switch
- Research in the late 70s/early 80s led to RISC
  - IBM 801 -- John Cocke – mid 70s
  - Berkeley RISC-1 (Patterson)
  - Stanford MIPS (Hennessy)
  - Acron ARM2 (1985)

VAX ISA (CISC)

- 32-bit ISA, instructions could be huge (up to 321 bytes), 16 GPRs
- Operated on data types from 8 to 128-bits, decimals, strings
- Orthogonal, memory-to-memory, all addressing modes supported
  - Orthogonal ISAs supports all addressing modes, i.e, the instruction types and addressing modes are orthogonal (independent).
- Hundreds of special instructions
- Simple compiler, hand-coding was common
- An instruction takes more than 10 cycles to execute!

X86 (CISC)

- Variable length ISA (1-16 bytes)
- Floating point operations uses stack instead of registers
- 2 operand instructions (somehow similar to accumulator)
  - Register-register and register-memory support
- Has multiple instructions for one task
  - E.g., `inc r11` and `add r11, 1` do almost same thing.
  - Usually one generic instruction that can do the task with slower speed (e.g., `add r11, 1)`, and one specialized instruction for this particular task with faster speed (e.g., `inc r11`).
- Has special instructions optimized for special tasks.
  - E.g., `leave` for function epilogue.
- Scaled addressing modes besides common ones.
- Has been extended many times (e.g., MMX, AMD64, SSE, AVX…)
  - Intel, instead went to IA64, which was not very successful.

MIPS ISA (RISC)

- Microprocessor without Interlocked Pipeline Stages
  - Although interlocking is back around 2002
- 32-bits long instructions
- Register-register operands
  - Must explicitly load operands from memory into registers before using them
- Use register-indirect, direct, immediate, displacement and indexed addressing mode
  - Addresses are stored in registers as instructions are not long enough to hold 32-bits or 64-bits addresses.
- Also revised many times to evolve to 64-bits architectures and include modern features (e.g., SIMD)
- A very popular academia teaching ISA.

ARM (RISC)

- Advanced RISC Machine, originally Acorn RISC Machine
  - Co-invested by Acorn and Apple
- Standard ARM instructions are 32-bits long. ARM-Thumb instructions are 16-bits or 32-bits.
  - Thumb ISA is a subset of standard ARM for memory compactness.
- Register-register operands
  - Must explicitly load operands from memory into registers before using them
- Use register-indirect, immediate, direct, auto-increment/decrement, displacement and indexed addressing mode.
- Almost every cellphone uses an ARM processor.

CISC vs RISC

- CISC
  1) Supports register-register, register-memory, accumulator and stack operands
  2) Complex addressing modes
  3) Variable instruction length
  4) May have an instruction for any operation. Sometime even duplicated instructions.
  5) Instructions require variable numbers of cycles to execute
  6) Must spend a lot transistors on control logics
  7) Leaves the complexity of the programs to hardware

- RISC
  1) Usually Register-Register operands
  2) Mostly simple addressing modes
  3) Fixed instruction length
  4) Limited number of instructions.
  5) Instructions mostly takes uniformed cycles to execute
  6) Can spend a lot of transistors on registers
  7) Leaves the complexity of the programs to software

CISC vs RISC

- Programmability, CISC is clearly a winner.
  - Recall this example: C = A + B

    CISC (3 insns)   RISC (4 insns)

    ```
    Load R1, [A]          Load R1, [A]
    Add  R3, R1, [B]      Load R2, [B]
                          Add   R3, R1, R2
    Store [C],   R3       Store [C], R3
    ```
  - CISC requires fewer instructions and thus is easier to write by hand. Fewer codes also mean fewer bugs.
  - Nowadays, compilers do the heavy-lifting, nearly no need to write assembly by hand, except for some cases of manual optimizations.
- Code sizes, CISC is clearly a winner, at least in the old days.
  - In nowadays, code sizes mostly depend on compilers.
  - Code size was important in 80s and 90s as memory was small.
- Because of these two advantages, CISC was a winner back in the 80s and 90s.
  - Although programmability and code sizes are less important today for ISAs.

CISC vs RISC cont'd

- For extensibility and backward-compatibility, CISC ISAs and RISC ISAs are both flexible enough.
- CISC ISAs appear to be more extensible. However, so far, RISC ISAs are also keeping up with technology changes.

CISC vs RISC cont'd

- Implementation complexity, RISC is clearly a winner
  - CISC requires too many control units to properly fetch and decode instruction, and to schedule instructions with variable execution times
  - CISC CPUs are usually much fatter

	ARM Cortex A9	Intel Atom N270	Intel i7 960	Intel Pen4
# of transistors	2600k	47000k	731000k	55000k

CISC vs RISC cont'd

- Implementation complexity
  - Due to the complexity, there are always more RISC implementations/processors than CISC
    - AMD has been simulating x86 with RISC from the beginning
    - Even Intel switched to a RISC implementation by translating x86 instructions into their uOPs
      - However this extra translation step has some negative impacts on power consumption
      - Pentium 4 is the last hard-wired Intel CPU. Some Atom processors reused Pen4's design to reduce power consumption

CISC vs RISC cont’d

- Other implications due to implementation complexity
  - More transistors mean high power consumption. That’s why we don’t see any CISC processors in embedded applications.
  - More control units mean less usable registers
    - x86 has 8 general purpose (GP) registers, MIPS has 32 GP registers, and ARM has 37 registers
    - x86 has more internal registers, but they are used to handle register-memory instructions and are not directly available to programmers.

CISC vs RISC cont’d

- I’d say RISC is the winner now, as there are barely true CISC implementations any more. We are mostly keeping CISC for the sake of backward compatibility.
- But note that RISC wins with the help of compilers, it doesn’t win by itself.

ISA Implementations Overview

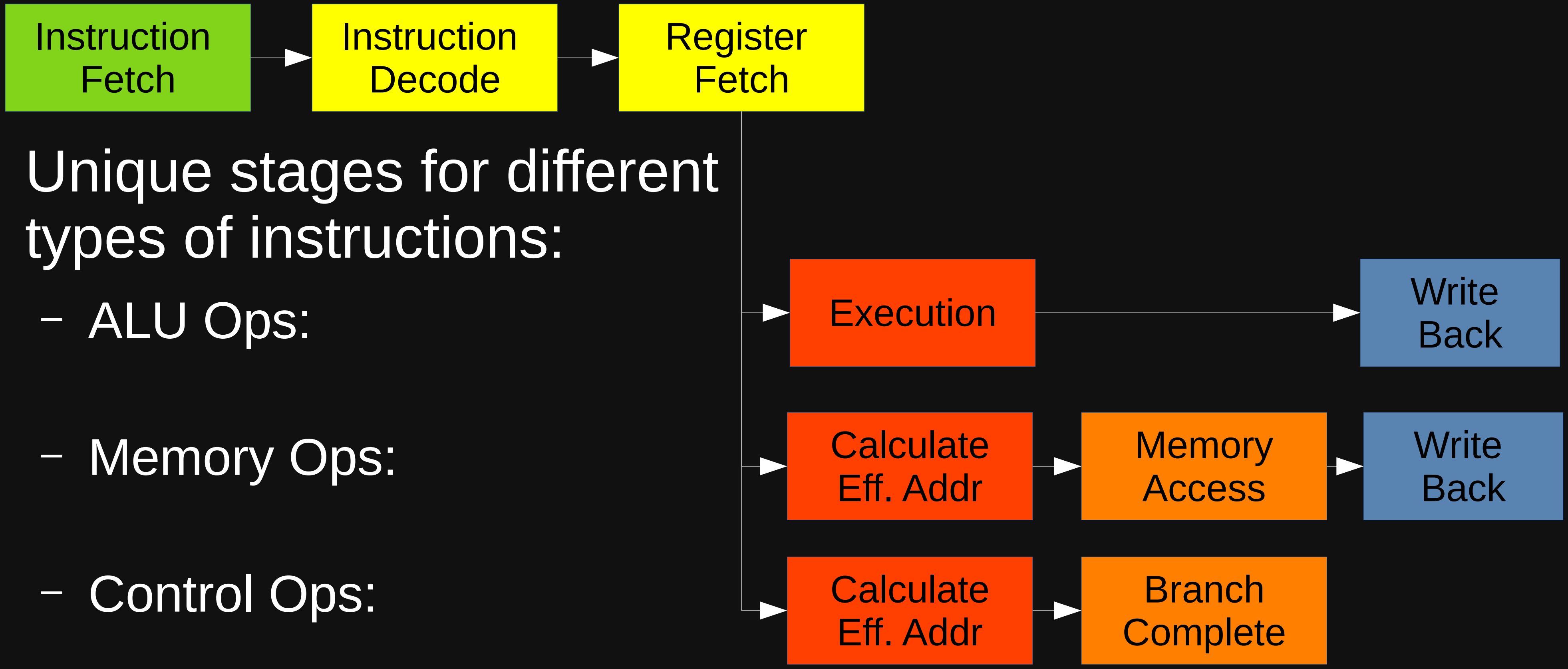
Instruction Execution Stages

- When implementing ISA, we typically partition the execution of an instruction into stages and implement each stage with transistors separately.
- Why
  - Many instructions share common steps in executions. Therefore, they can shared the common functional units.
  - Break down into stages with well-defined execution times (in CPU cycles) makes instruction scheduling and management easier.
  - Another divide-and-conquer or abstraction, simplifies the designing process.

Stages of Instruction Execution

- Common stages for all instructions:

- Unique stages for different types of instructions:
  - ALU Ops:
  - Memory Ops:
  - Control Ops:



Execution Stages with RISC

- For the following slides, we will see implementation examples for a RISC ISA.
  - That is, the ISA is register-register and has fixed instruction lengths.
- We will discuss the implementation of CISC after these examples.

Instruction Fetch

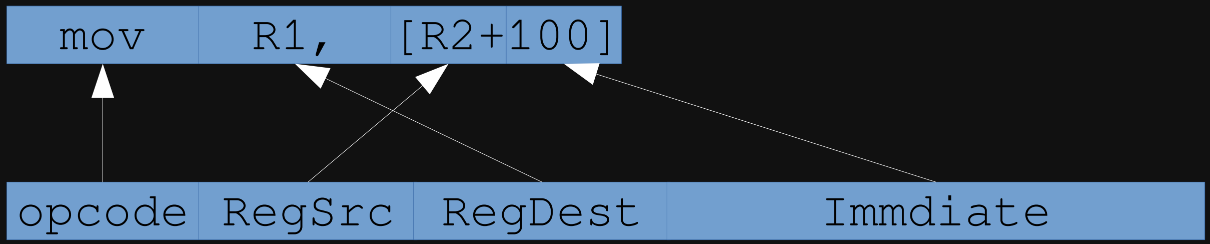
- Send the Program Counter (PC) to memory
- Fetch the current instruction from memory into the instruction register (IR)
  - `IR <= Mem[PC]`
- Update the Next PC (NPC) to be the next sequential instruction
  - `NPC <= PC + 4` (for simplicity, assuming 4-bytes per instruction)
- Optimizations
  - Instruction Caches, Instruction Prefetch
- Performance Affected by
  - Code density, Instruction size variability (CISC/RISC)

Instruction Decode/Reg Fetch

- Decide what type of instruction we have
  - ALU, Branch, Memory
  - Decode Opcode
- Get operand(s) from register file
  - Part of the instruction determines where the operands are from. E.g.,
    - `A <= Regs[IR`$_{RegSrc1}$`]; B <= Regs[IR`$_{RegSrc2}$`];`
    - `Imm <= SignExtend(IR`$_{immd}$`)`
    - `RegSrc1`, `RegSrc2`, `RegDest`, and `immd` represents certain bits of the instruction. For example, `RegSrc1` may be bits 20 to 15.
- Performance Affected by
  - Regularity in instruction format, instruction length

Calculate Effective Address: Memory Ops

- Calculate memory address for the data using ALU
- Addressing modes here direct how to compute the effective memory address
  - E.g., in the following instruction, the effective address is computed by summing the value in `R2` and `100`.



Calculate Effective Address: Branch/Jump Ops

- Calculate target for branch/jump operation using ALU
  - That is, the effective memory address for the jump target in memory
- The actually address depends on the addressing modes.
  - The actual address calculation is similar to memory access instructions.

Execution: ALU Ops

- Perform the computation
- Register-Register
  - $\texttt{ALU}_{\texttt{output}}$ `<= A op B`
- Register-Immediate
  - $\texttt{ALU}_{\texttt{output}}$ `<= A op Imm`

Memory Access

- Take effective address, perform Load or Store
- Load
  - `Load_Buffer <= Mem[ALU`$_{output}$`]`
- Store
  - `Mem[ALU`$_{output}$`] <= Register`
- Note that `ALU`$_{output}$ is the calculated effective address.

Branch Completion

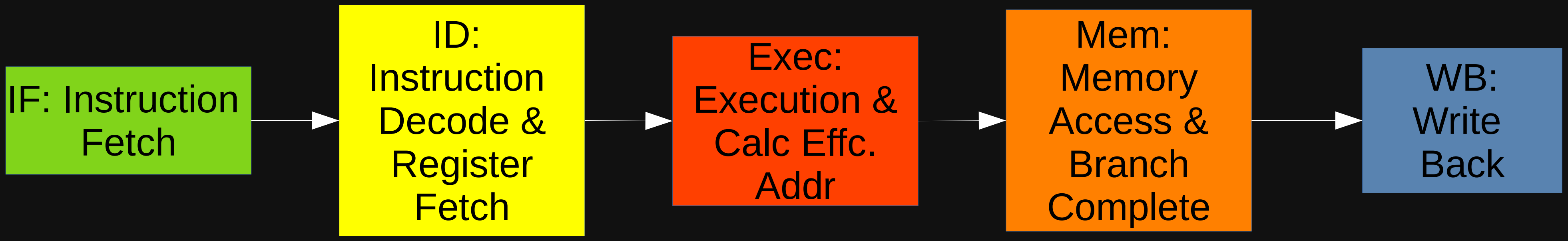
- If unconditional jump, set the PC to the calculated effective address
- If conditional jump, set the PC to calculated effective address is condition is met
  -
```
If (cond)
        PC <= ALU_output
else
        PC <= next instruction’s address
```

Write-Back

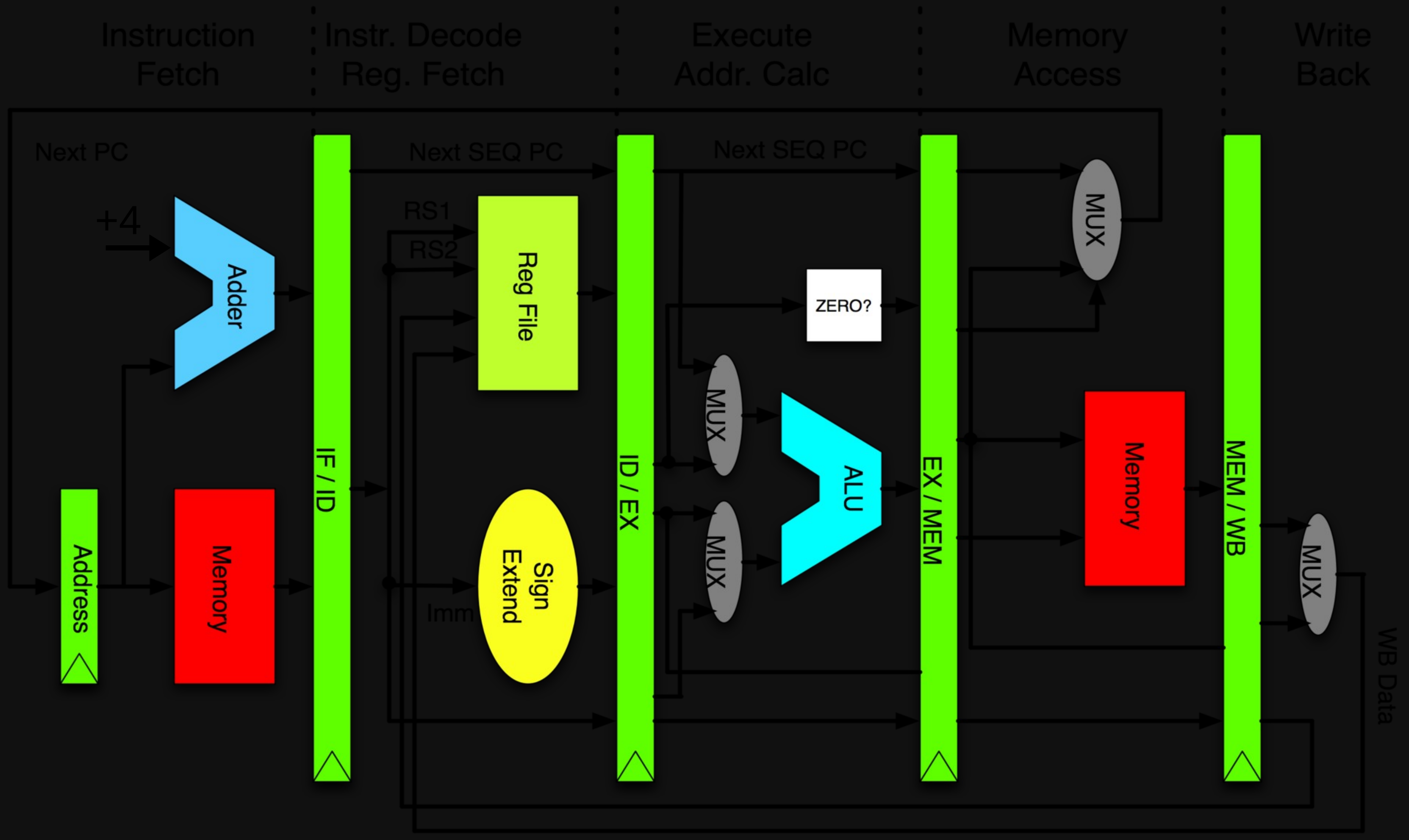
- Send results back to register file
- Normal ALU instruction
  - $\texttt{Regs[IR}_{\texttt{RegDest}}\texttt{] <= ALU}_{\texttt{output}}$
- Load Instruction
  - $\texttt{Regs[IR}_{\texttt{RegDest}}\texttt{] <= Load\_Buffer}$
- This is a stage for register writing, not memory writing.

Putting All Stages Together

- In a typical yet simple RISC CPU implementation, the processor is partitioned into 5 connected stages.
- Every instruction goes through all stages, although this instruction does not necessarily trigger the functional units of every stage.

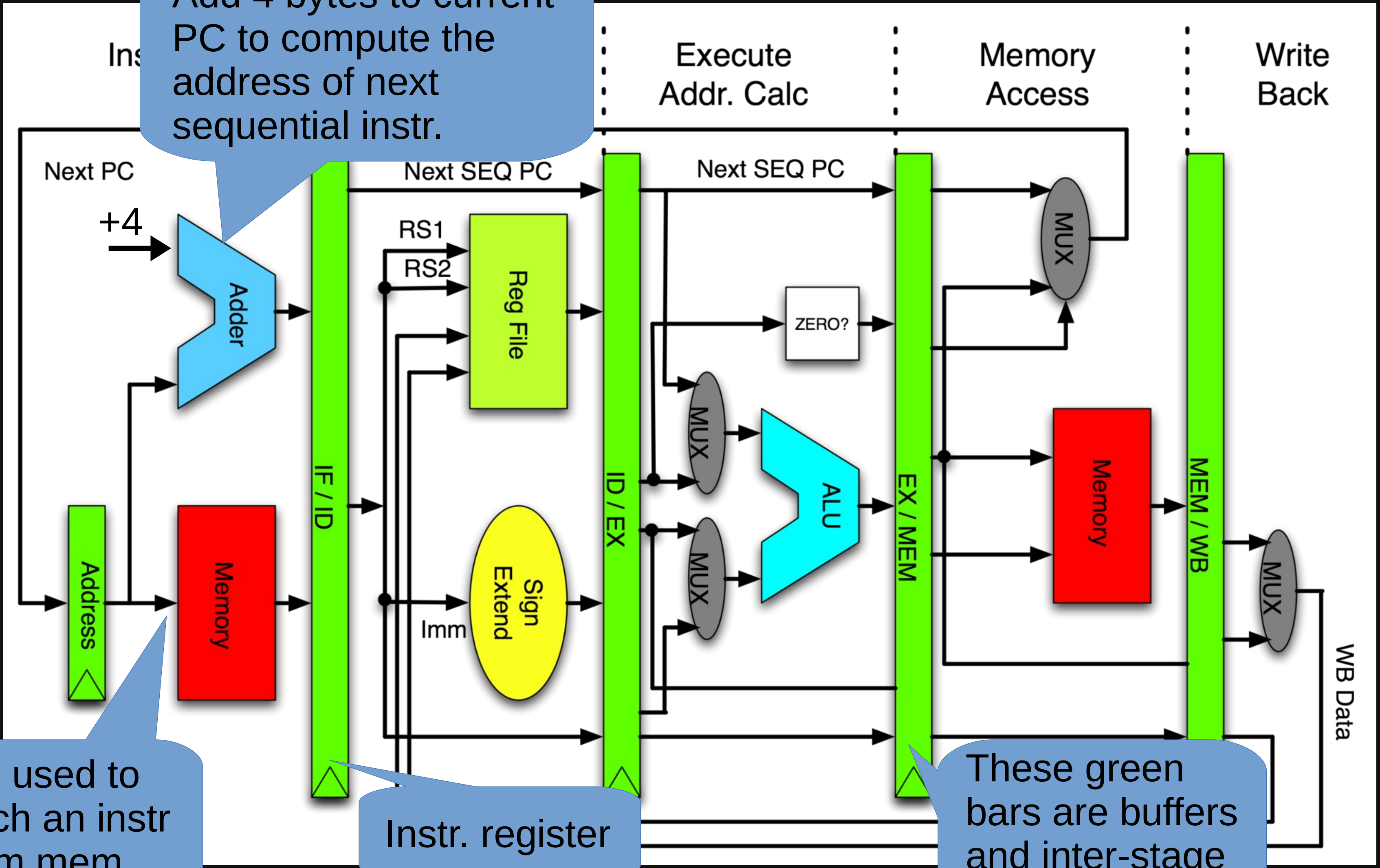

IF: Instruction Fetch
ID: Instruction Decode & Register Fetch
Exec: Execution & Calc Effc. Addr
Mem: Memory Access & Branch Complete
WB: Write Back

A Simple Implementation of the 5-stage RISC CPU



* figure by Hellisp from Wikibooks.org

Instruction Fetch in Implementation



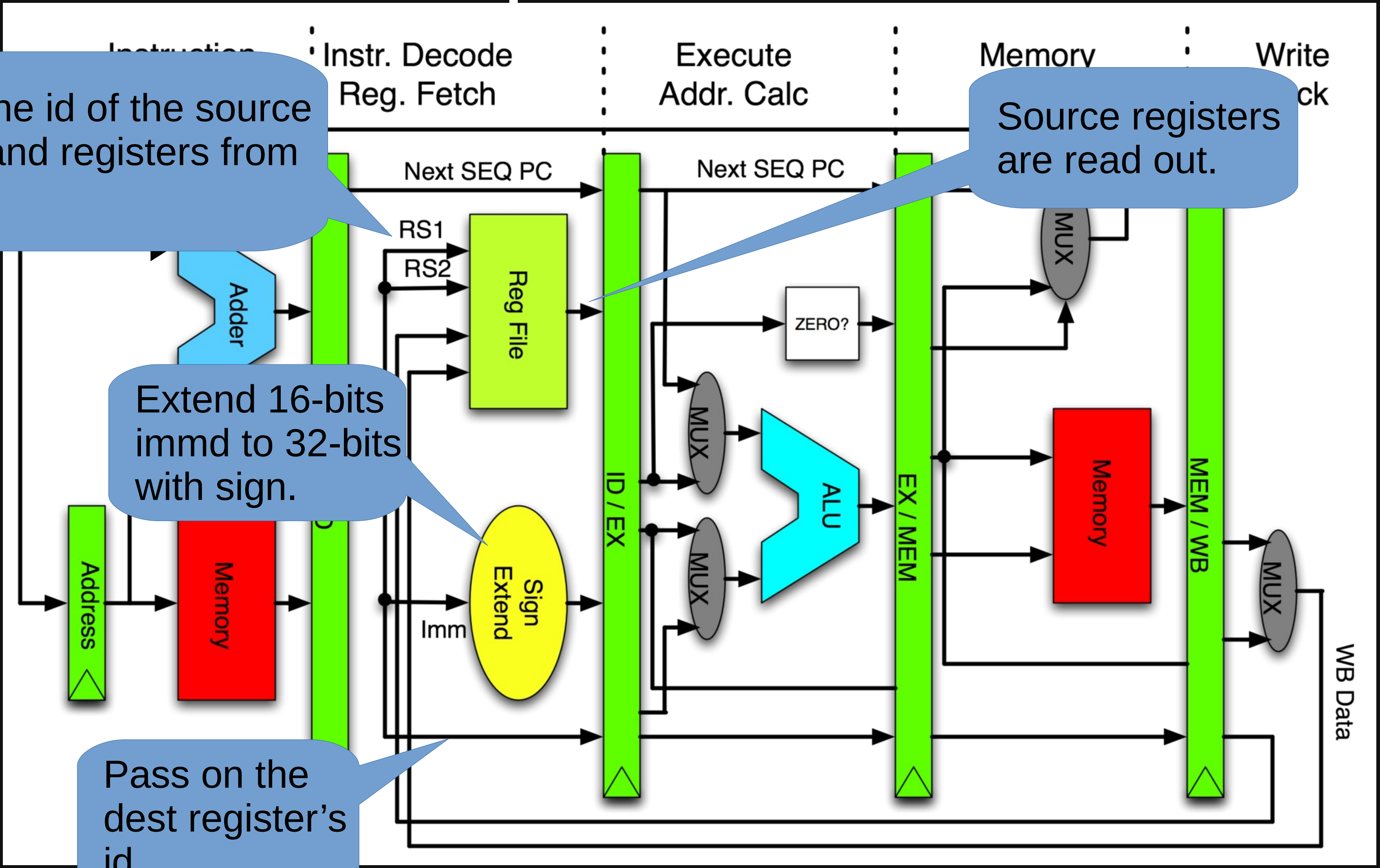
Add 4 bytes to current PC to compute the address of next sequential instr.

PC used to fetch an instr from mem.

Instr. register

These green bars are buffers and inter-stage storage registers.

Instruction Decode & Register Fetch in Implementation



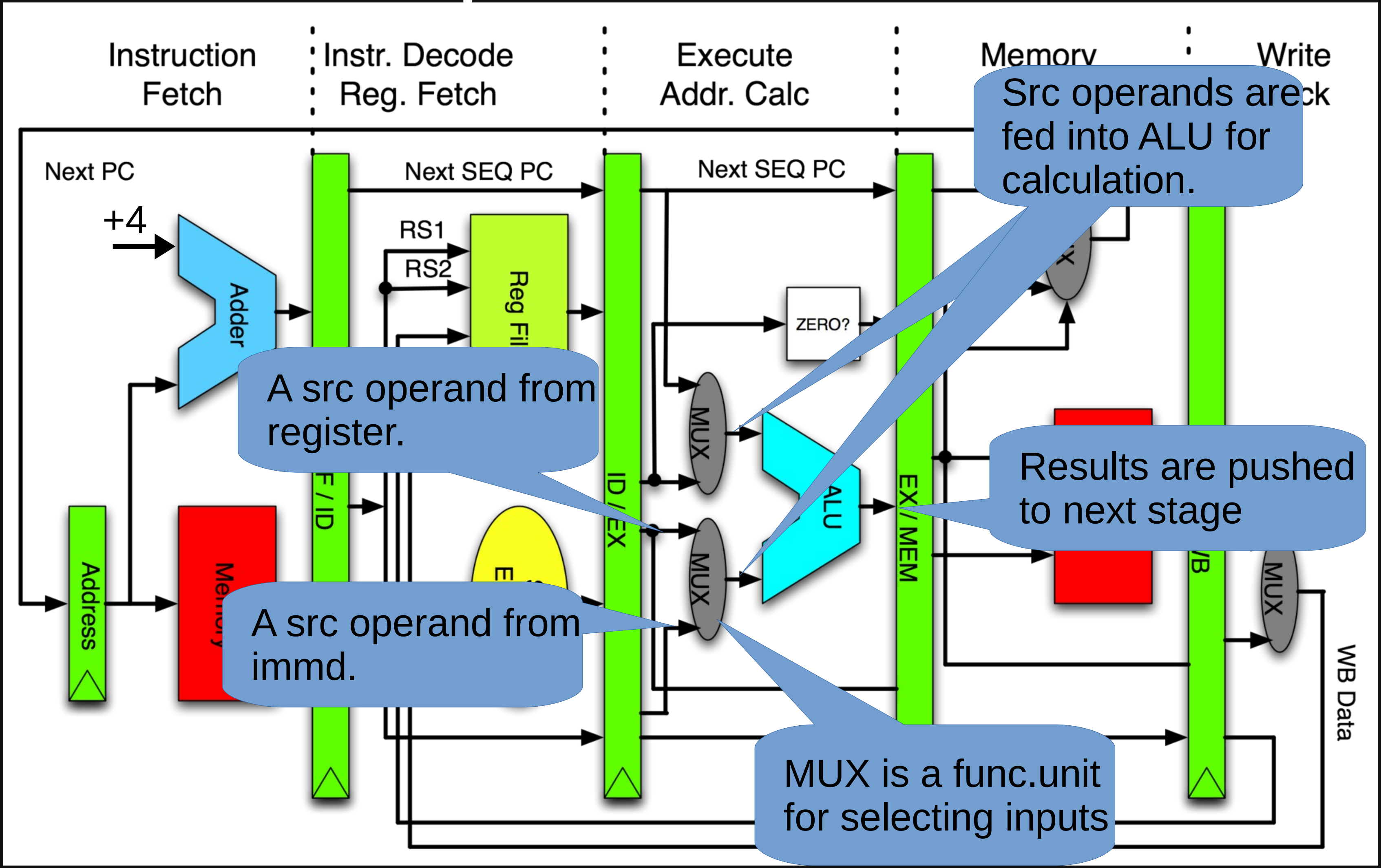
Get the id of the source operand registers from IR.

Source registers are read out.

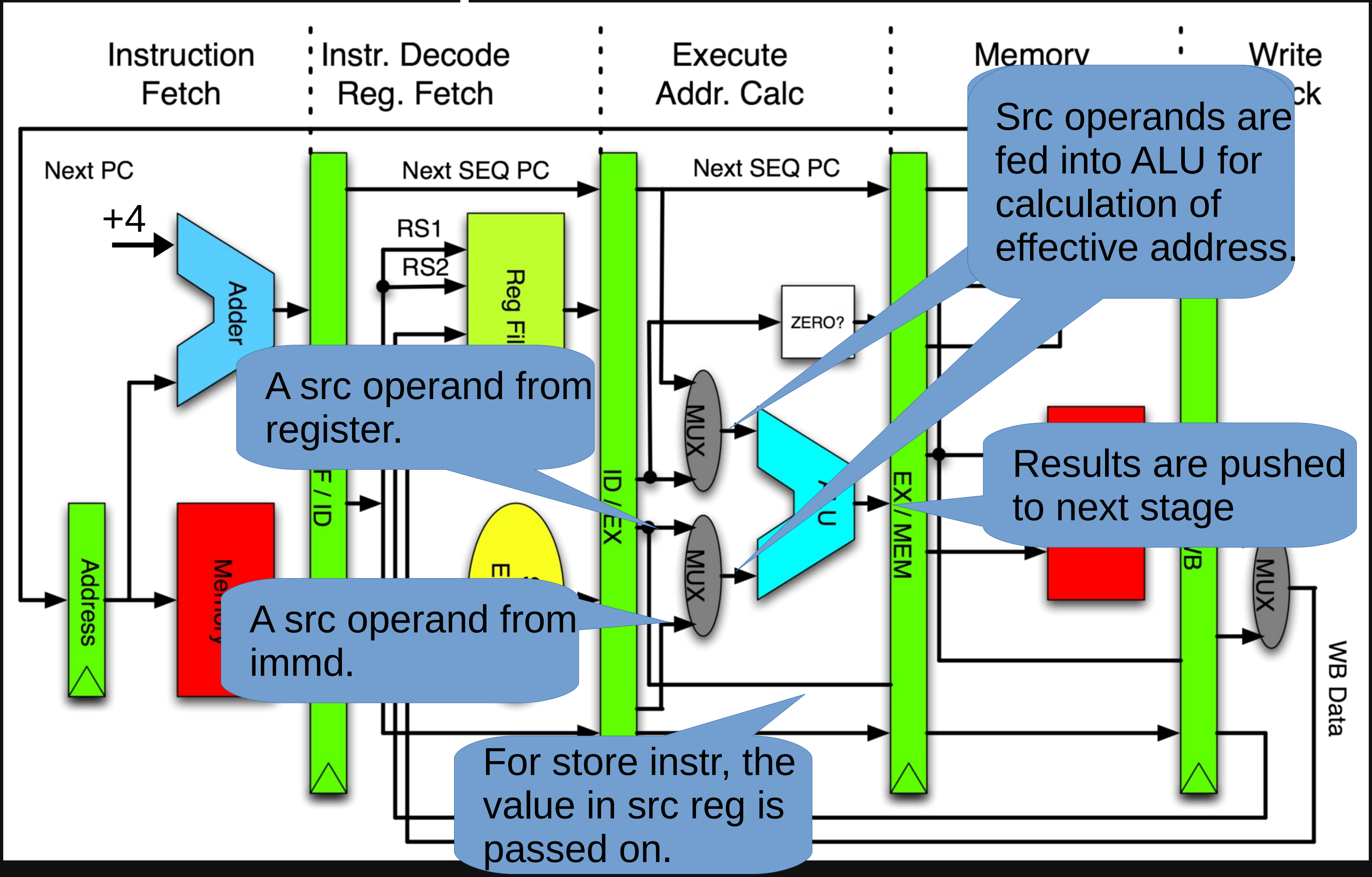
Extend 16-bits immd to 32-bits with sign.

Pass on the dest register's id

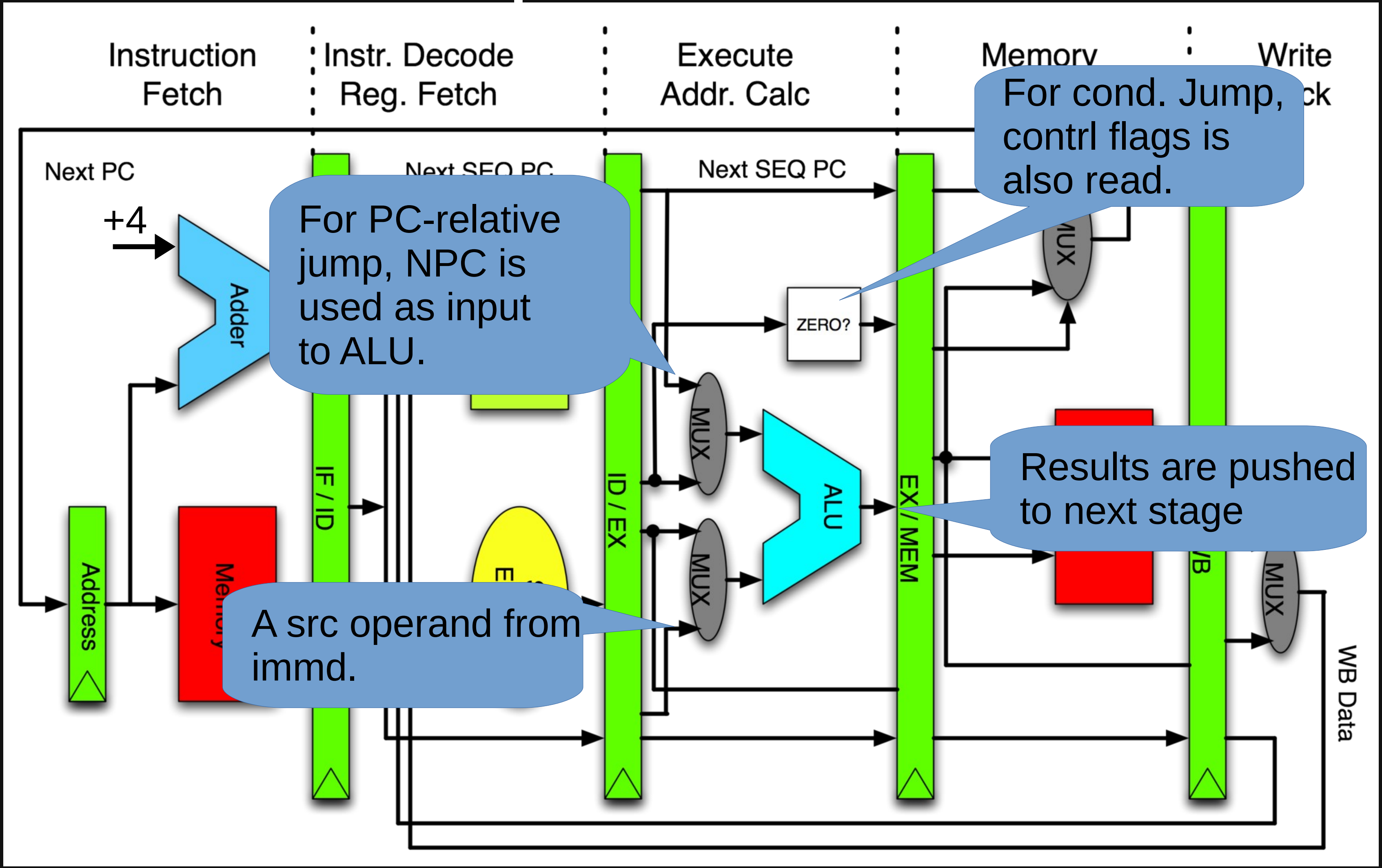
Exec Stage for ALU Instructions in Implementation



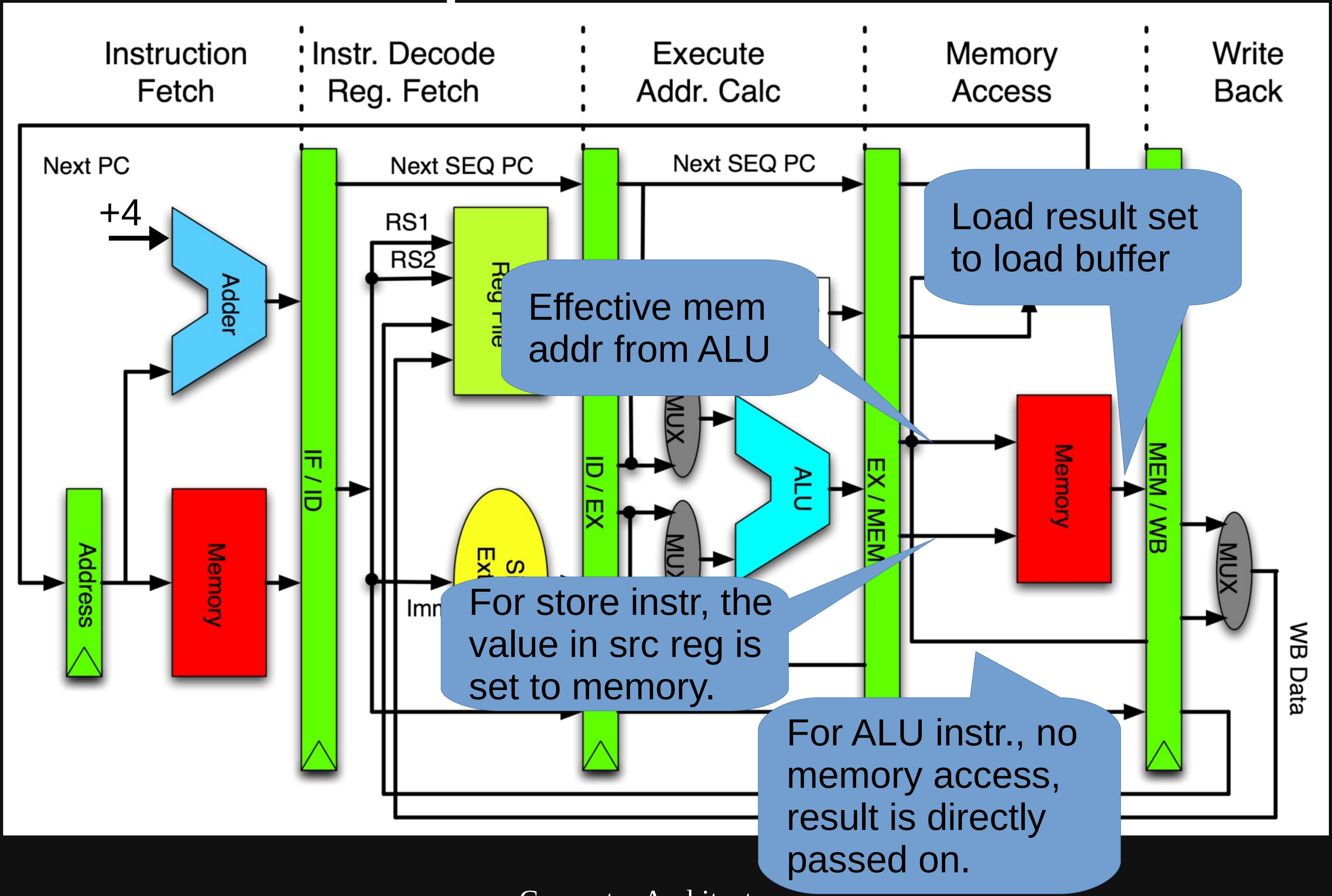
Exec Stage for Mem Instructions in Implementation



Exec Stage for Branch Instructions in Implementation



Memory Access Stage in Implementation



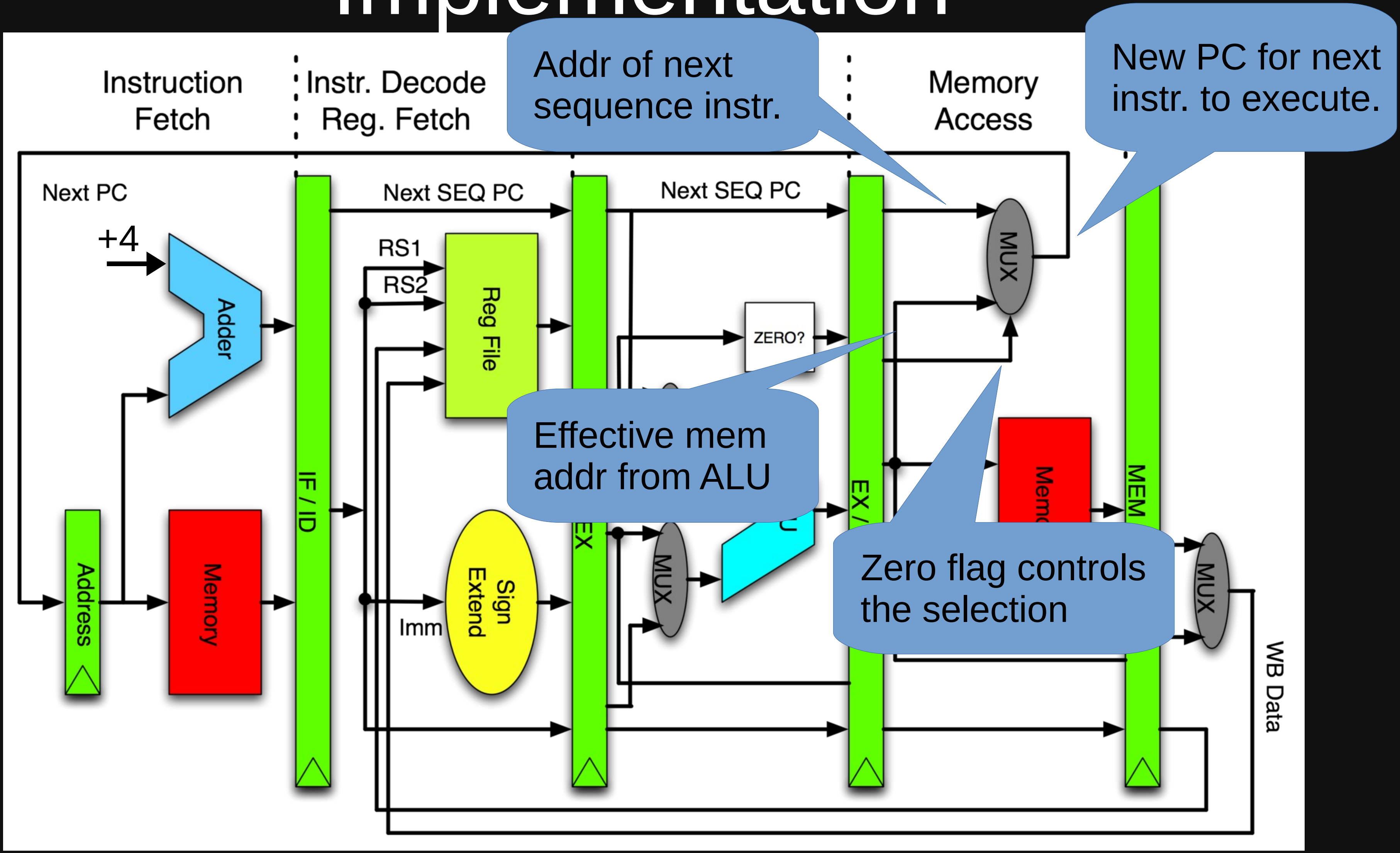
Load result set to load buffer

Effective mem addr from ALU

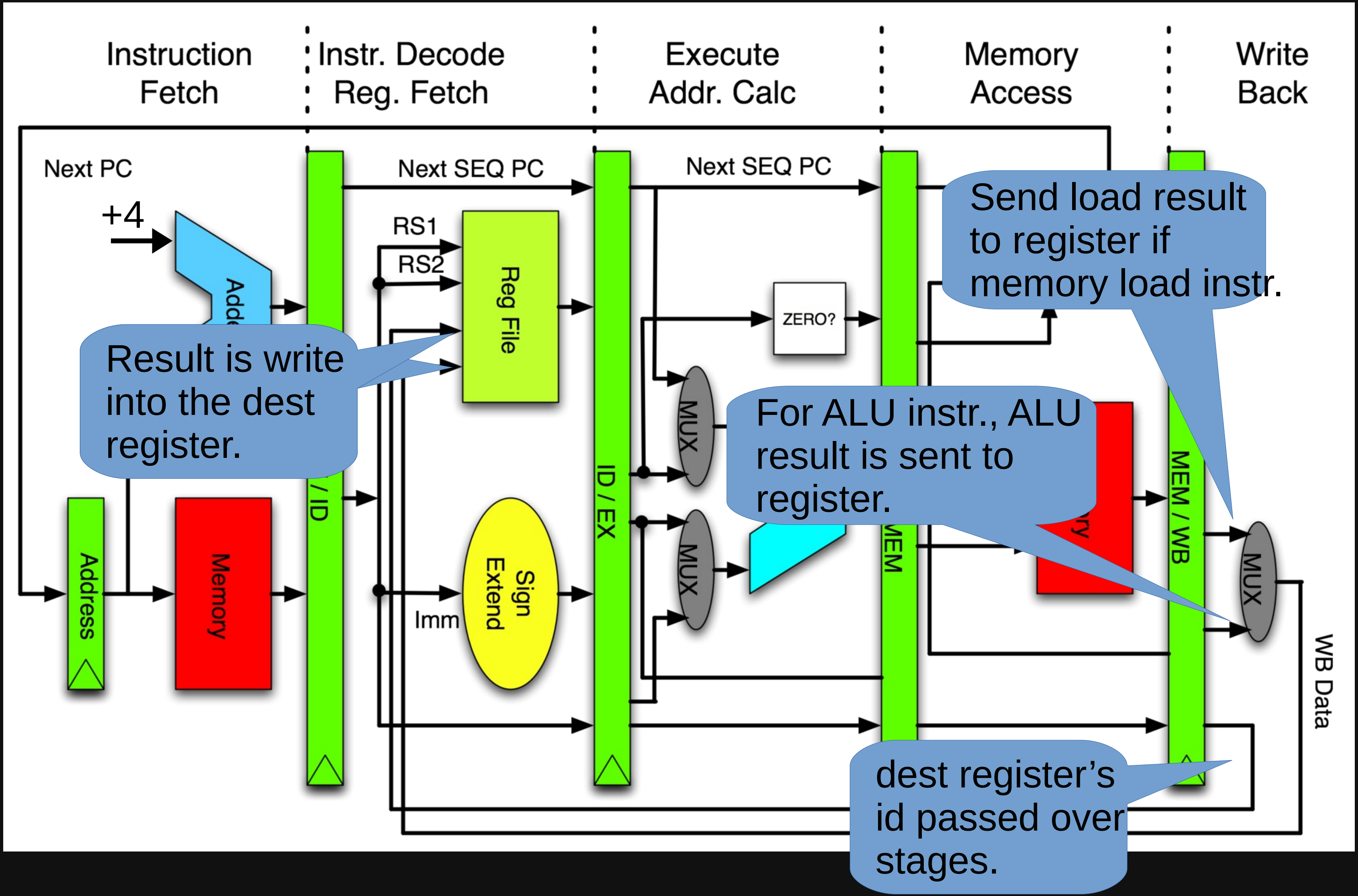
For store instr, the value in src reg is set to memory.

For ALU instr., no memory access, result is directly passed on.

Branch Complete Stage in Implementation



Write-back Stage in Implementation



SIMD and Parallel Instructions

The Requirement for Multi-media Processing

- Around 1997, desktop computers were becoming devices for entertainment.
- A key task for entertainment is to decode video and audio streams.
  - The decode process is essentially one linear algebra transformation on a stream of data.
- This requirement leaded to the invention of various multi-media instructions, such as Intel MMX and AMD 3DNow!.
- Luckily, with Moore’s law, there were spare transistors that manufactures could spend on functional units for multi-media instructions.
  - These functional units typically extends the exec stage.

From Multi-media To Science

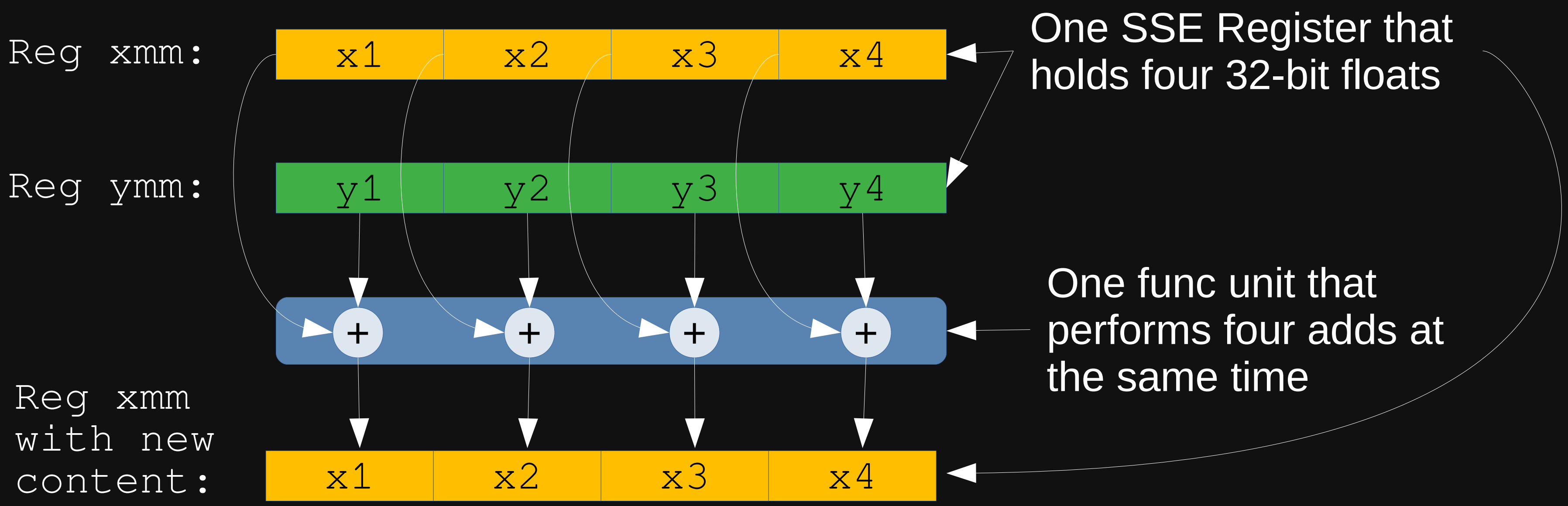
- Later, these multi-media instructions proven to be very helpful for scientific applications as well.
  - The same thing happened to GPU as well. Graphic workloads naturally resembles scientific (and machine learning) workloads, as they all rely heavily on linear algebra.
- These multi-media instructions evolved into more complex instructions such as SSE and AVX.

SIMD Instruction

- The common feature of MMX, 3DNow!, SSE and AVX instructions is that they can all perform the same math operation on multiple inputs using just one instruction.
- The common name for these instructions are SIMD – <u>S</u>ingle <u>I</u>nstruction <u>M</u>ultiple <u>D</u>ata.
  - A GPU does exactly the same. GPUs are practially SIMD processors.

A Simple Example of A SIMD Instr.

- A simple packed (parallel or vectorized) SSE add instruction `addps xmm, ymm` for $x_i = x_i + y_i$, can be implemented as,


Reg xmm:
x1
x2
x3
x4
One SSE Register that holds four 32-bit floats
Reg ymm:
y1
y2
y3
y4
+
+
+
+
One func unit that performs four adds at the same time
Reg xmm with new content:
x1
x2
x3
x4

SIMD Instructions In General

- Similarly, other SSE instructions can perform four substractions, multiplications and divisions on eight floats/integers simultaneously.
- There are also scalar (non-parallel) instructions in SSE for better programmability.
- SSE have been extended several times. We also have AVX instructions now that can handle four double operations at a time.

VLIW Instructions

- Except for multi-media, scientific and machine-learning applications, very few applications have the need to do multiple math operations at the same time.
- To improve the performance for these applications, system researchers tried to explore other parallel execution opportunities.
  - In general, this is called Instruction Level Parallelism (ILP).
- Note that, with Moore's Law, we are guaranteed to have more functional units. So the goal in CPU optimization has long been finding ways to use all these functional units in parallel to improve CPU utilization and efficiency.
  - Another extreme example is on-chip graphic processors.

VLIW Instructions cont’d

- For example, the following program can be re-arranged to execute simultaneously,

```
a[i] += a[i-1];
c[i] *= a[i] + c[i-1];

b[i] += b[i-1];
d[i] *= b[i] + d[i-1];
```

```
a[i] += a[i-1];
b[i] += b[i-1];
```

} Run in parallel

```
c[i] *= a[i] + c[i-1];
d[i] *= b[i] + d[i-1];
```

} Run in parallel

- Modern processors can actually identify this parallel opportunity.
  - The methodology to identify this parallel opportunity is called Out-of-Order (OoO) execution, which was invented in 1960s.

VLIW Instructions cont’d

- The problem with OoO execution is that it is very complex to design and consumes a lot of transistors that can be use as registers and normal computational units.
- A possible alternative to OoO is to shift the burden of identifying parallel opportunities to software.
  - Let compiler to find the operations that can be executed in parallel.
  - The compiler then generates ONE very-long instruction for several operations that can run in parallel.

VLIW Instructions cont'd

- A possible alternative to OoO is to shift the burden of identifying parallel opportunities to software.
  - Let compiler to find the operations that can be executed in parallel.
  - The compiler then generates ONE very-long instruction for several operations that can run in parallel.

```
a[i] += a[i-1];
b[i] += b[i-1];
```

$\Rightarrow$ Just one Instr.:

```
  some_special_op a, b, i
```

  - The CPU needs to provide such instructions in its ISA.
  - Essentially, the compiler tells the CPU the parallel parts of a program using these very-long instructions.
  - These very-long instructions are called Very Long Instruction Word (VLIW).

The Difficulty of VLIW

- However, it turns out VLIW CPUs/compilers are very hard to design.
  - It is hard (basically impossible) for compilers to find enough parallelism with an integer sequential program.
    - There are too many unknown memory addresses and data dependencies at the compilation time.
  - It is hard to provide a finite set of instructions to cover the infinite combinations of instructions.
- Intel went for VLIW for their 64-bit CPU design, and it was not successful.
  - The architecture is call IA-64, and the processors are called Itanium. The last Itanium processor shipped in 2017.
  - AMD went for extending 32-bit x86 ISA to 64 bits. Intel followed suit later. And that’s why 64-bit x86 are called AMD64 today.

Compiler Interactions

Compilers and ISAs

- Nowadays, the main users of instructions are compilers.
- Therefore, ISA design has a huge impact on compiler designs.
  - Complex ISA actually leads to complex compilers.
    - When there are many choices for addressing modes, an when there are many instructions that can do the same task, compiler designers will get confused.
  - Chip manufactures usually provides limited supports to open-source compiler writers. We rely on third-party experiments to deduce the internal operations of CPUs.
    - Check out Agner Fog.

Compilers and ISAs

- Ideally, architects can help compiler writers
  - Providing regularity (already discussed)
  - Primitives, not solutions (direct HLL-support has not succeeded)
  - Simplify trade-offs among alternatives
  - Provide instructions that bind compile-time constants

Compilers Support for SIMD Instructions

- Compilers have limited capacity when generating SIMD instructions.
- GCC generally do not generate SIMD instructions.
  - GCC provide intrinsic functions, but no automatic code generation.
- Compilers from manufactures can generate SIMD in some cases.
- Generally, manual implementation is required for code snippets that use SIMDs.

Acknowledgment

- These slides are partially based on the lecture notes from Dr. David Brooks and Dr. Gurpur Prabhu