# COMP2300-COMP6300-ENGN2219
# Computer Organization & Program Execution

Convener: Shoaib Akram

shoaib.akram@anu.edu.au

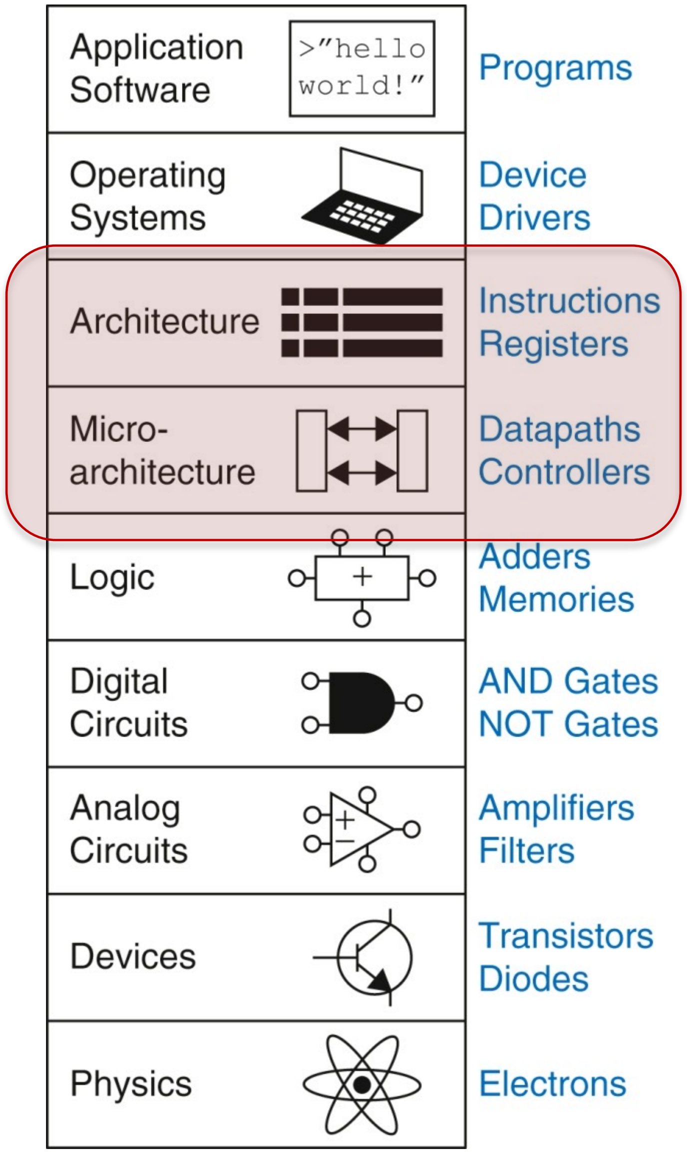
Australian National University

## Our Status

- We are done with digital logic fundamentals that we need to understand and build a CPU

- We are now (+ next week) at
  - Architecture layer

- Then
  - Microarchitecture layer



ISA then microarchitecture

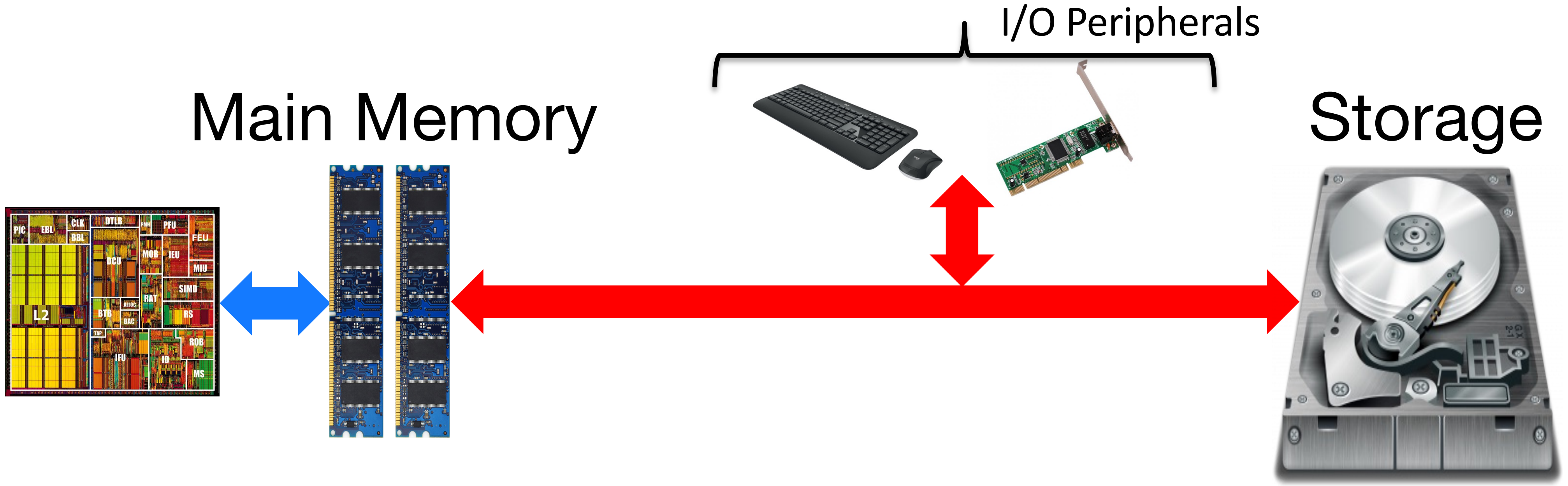
## Admin

- Quiz #1 has been marked
  - We will take the best two of four quizzes
- Marking of the checkpoint is underway
- Assignment 1 will be released this week
- Some % of assignment 1 grade comes from work you are doing in **Labs 4 – 6**

## Von Neumann Model

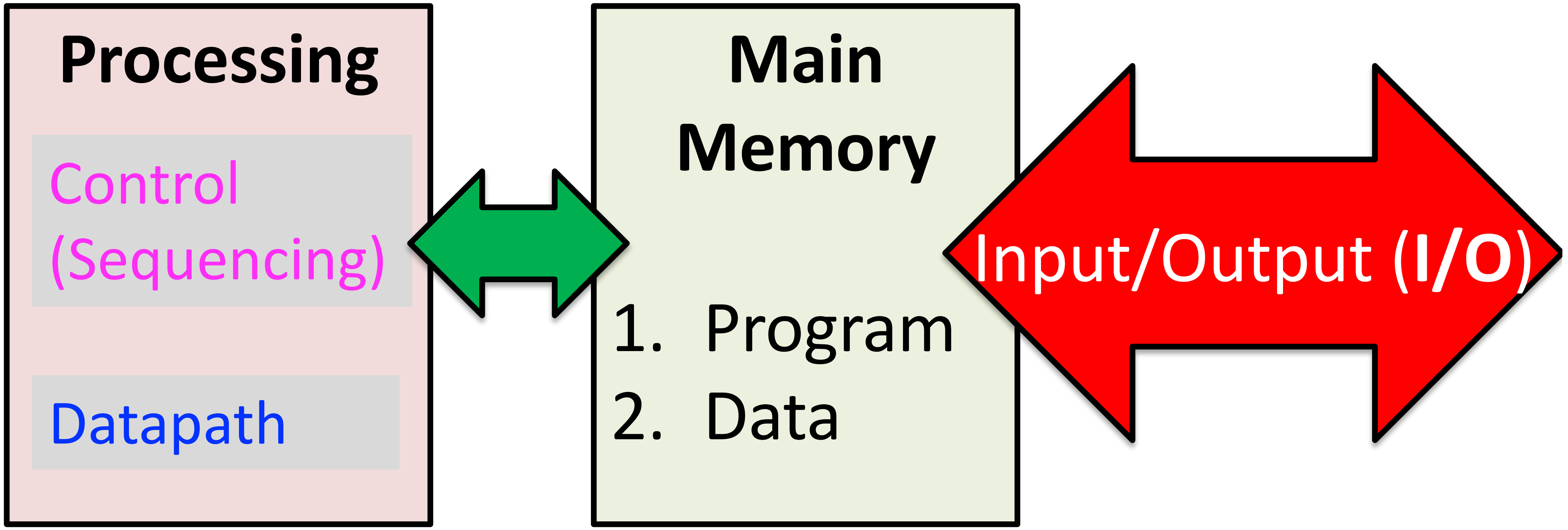
# Recall: A Computer System

- Key resources: **CPU**, **memory**, and Input/Output (**I/O**) devices
  - CPU (microprocessor) does the actual processing (computation)
  - Memory stores temporary data and forms a hierarchy (registers, SRAM, DRAM, ...)
  - Some fast **(small capacity)** memory called register file is close to the CPU and **rest is far**
  - Storage disk is an I/O device (much **slower** than memory, stores persistent data)
  - Memory is volatile, while disk is non-volatile (data is retained after a shutdown)
  - Other peripherals such as keyboard and network card are accessories to processing



## Another View: What is a Computer?

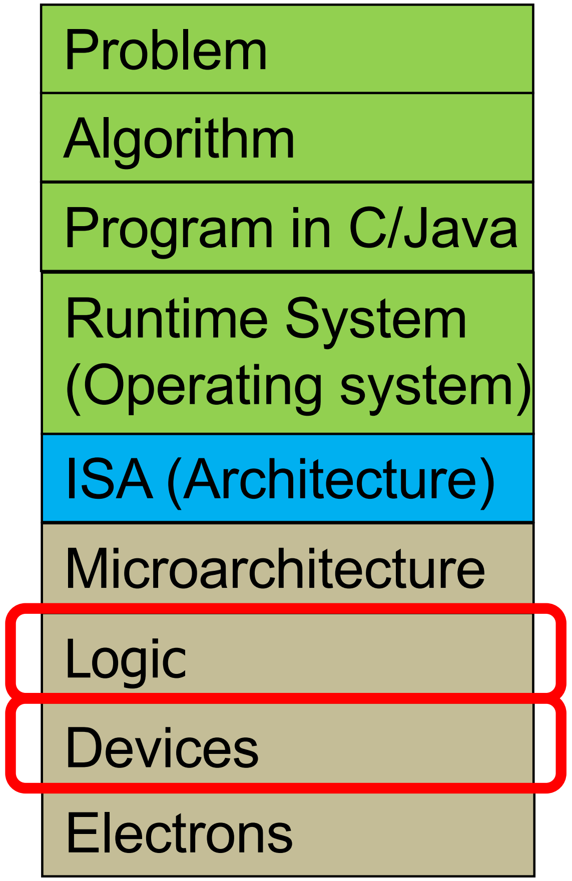
- Basic computer model proposed in the 1940s



- We will cover all three components

# Building up a Basic Computer Model

- In past lectures, we learned how to design
  - Combinational logic structures
  - Sequential logic structures
- With logic structures, we can build
  - Execution units
  - Decision units
  - Memory/storage units
  - Communication units
- All are basic elements of a computer
  - We will raise our abstraction level today
  - Use logic structures to construct a basic computer model

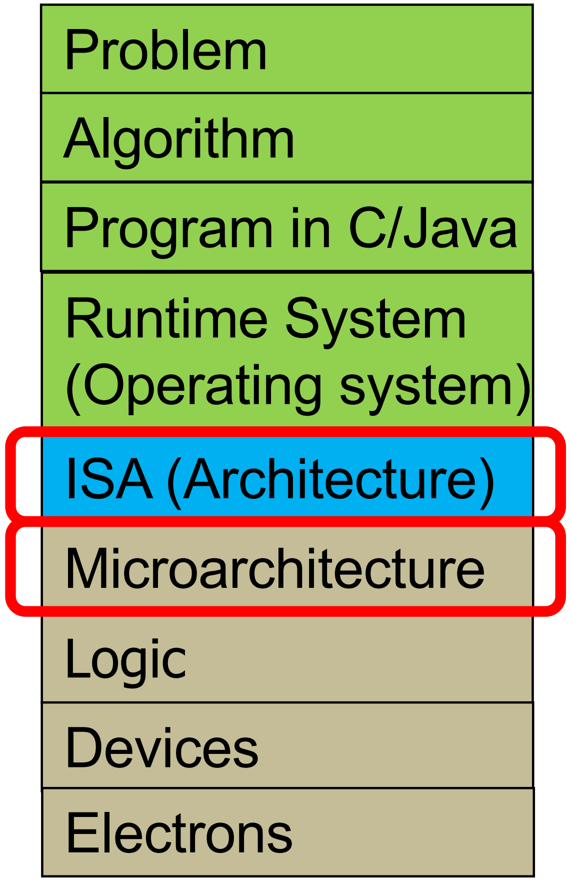


# Building up a Basic Computer Model

- **ISA:** Specification of the instructions computer can perform
  - An interface between the programs and hardware
    - **Programmer** needs to know ISA to be able to convey his wishes (instructions) to the hardware
    - Hardware builder (**computer architect**) needs to know the ISA to be able to build and organize circuits to carry out the instructions

- **Microarchitecture:** Circuit implementation of the specification

- **An important aspect to ponder:** Not every implementation detail is relevant to the programmer!
  - Just enough to be able to program the computer (as we will see)



# ISA vs. Microarchitecture

- What is part of ISA vs. Uarch?
  - Gas pedal: interface for “**acceleration**”
  - Internals of the engine: implement “**acceleration**”
- **Aspects of ISA**
  - The different instructions and their binary codes
  - Semantics (meaning) of each instruction
  - Word size, number of registers, memory addressability
- **Aspects of implementation**
  - Ripple-carry vs. carry-lookahead adder
  - Mux or tristate buffers
  - Canonical SOP or minimal Boolean expression for implementation
  - NAND gates only vs. AND/OR/NOT combination

## ISA vs. Microarchitecture

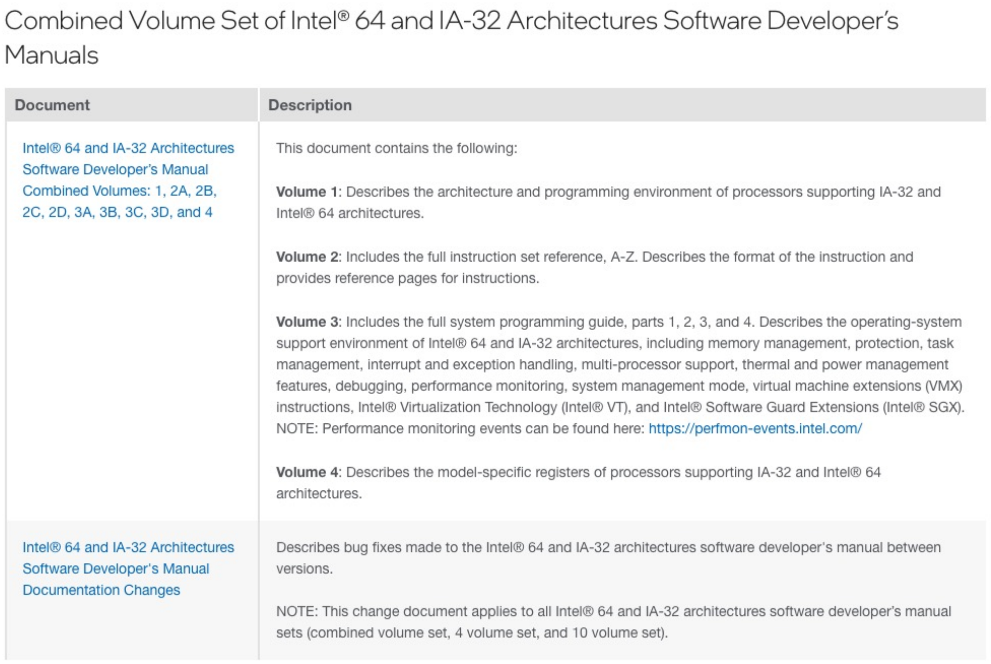
- **One ISA can have many microarchitectures**
  - One microarchitecture per student, but the QuAC ISA is the same on the course webpage

- **ISA is usually a one-time effort with incremental changes to enable new applications**
  - Only a few ISAs in the world but many microarchitectures
  - Microarchitecture changes faster than ISA
  - **Key insight:** ISA can enable simple vs. complex logic gate circuitry at the microarchitecture level (more in coming weeks ....

## ISA: Another View

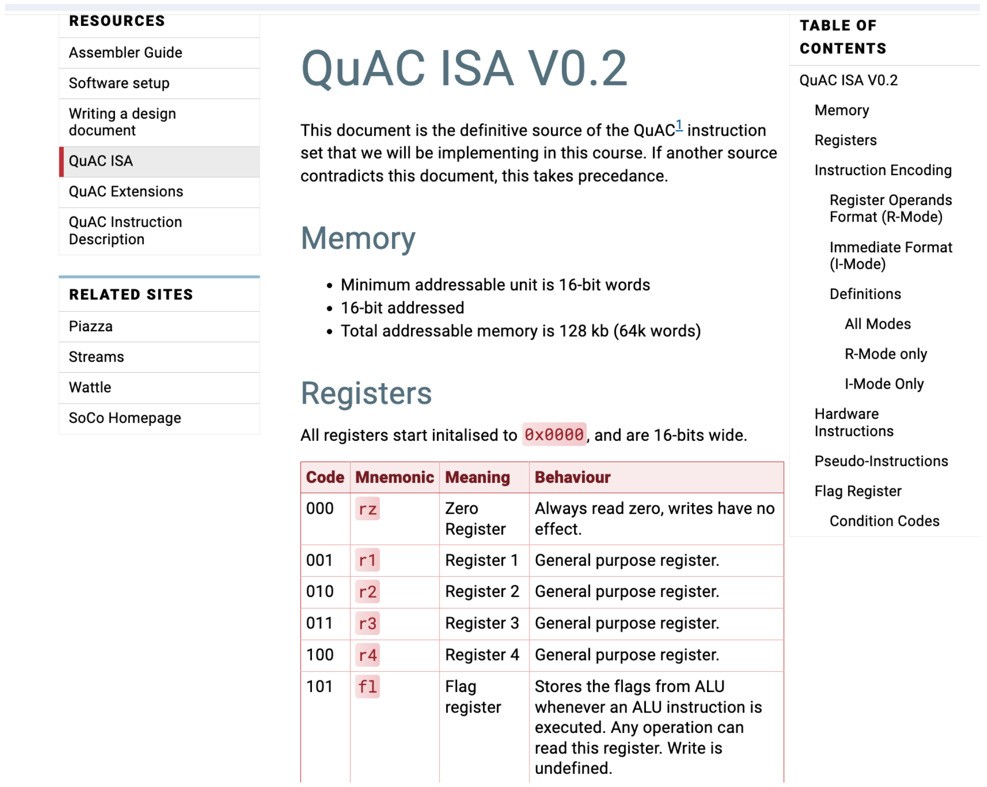
- **Most people don't write programs in the computer's own machine language (lowest level)**
- They prefer **high-level languages** such as C++, Java, or Python
- A compiler *translates* C++ or Java code into the computer's machine language
- ISA specifies everything in the computer that a *compiler writer* who wishes to translate programs from C++/Java to machine language need to know

# ISAs are a Good Bedtime Reading!



Combined Volume Set of Intel® 64 and IA-32 Architectures Software Developer's Manuals

| Document | Description |
|---|---|
| Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4 | This document contains the following:<br><br>**Volume 1**: Describes the architecture and programming environment of processors supporting IA-32 and Intel® 64 architectures.<br><br>**Volume 2**: Includes the full instruction set reference, A-Z. Describes the format of the instruction and provides reference pages for instructions.<br><br>**Volume 3**: Includes the full system programming guide, parts 1, 2, 3, and 4. Describes the operating-system support environment of Intel® 64 and IA-32 architectures, including memory management, protection, task management, interrupt and exception handling, multi-processor support, thermal and power management features, debugging, performance monitoring, system management mode, virtual machine extensions (VMX) instructions, Intel® Virtualization Technology (Intel® VT), and Intel® Software Guard Extensions (Intel® SGX).<br>NOTE: Performance monitoring events can be found here: https://perfmon-events.intel.com/<br><br>**Volume 4**: Describes the model-specific registers of processors supporting IA-32 and Intel® 64 architectures. |
| Intel® 64 and IA-32 Architectures Software Developer's Manual Documentation Changes | Describes bug fixes made to the Intel® 64 and IA-32 architectures software developer's manual between versions.<br><br>NOTE: This change document applies to all Intel® 64 and IA-32 architectures software developer's manual sets (combined volume set, 4 volume set, and 10 volume set). |



## QuAC ISA V0.2

This document is the definitive source of the QuAC[1] instruction set that we will be implementing in this course. If another source contradicts this document, this takes precedance.

### Memory

- Minimum addressable unit is 16-bit words
- 16-bit addressed
- Total addressable memory is 128 kb (64k words)

### Registers

All registers start initalised to `0x0000`, and are 16-bits wide.

| Code | Mnemonic | Meaning | Behaviour |
|---|---|---|---|
| 000 | `rz` | Zero Register | Always read zero, writes have no effect. |
| 001 | `r1` | Register 1 | General purpose register. |
| 010 | `r2` | Register 2 | General purpose register. |
| 011 | `r3` | Register 3 | General purpose register. |
| 100 | `r4` | Register 4 | General purpose register. |
| 101 | `fl` | Flag register | Stores the flags from ALU whenever an ALU instruction is executed. Any operation can read this register. Write is undefined. |

TABLE OF CONTENTS

# ISAs You Will Encounter @ ANU

- **QuAC**
  - An ISA for educational purposes developed at ANU
  - **Mainly covered in tutorials and required for assignment 1**
- **MIPS**
  - Pioneering **RISC** ISA developed by John Hennessy at **MIPS computer systems**
    - **M**icroprocessors without **I**nterlocked **P**ipelined **S**tages
  - Briefly covered in today's lecture for breadth

Ex-President of Stanford University
Chairman of Alphabet
Founder of MIPS Technologies
Turing Award Winner

- **ARM**
  - A popular **RISC** ISA developed by Arm Ltd.
  - **A**dvanced **R**ISC **M**achines
  - De facto choice for portable hand-held devices
  - **Covered extensively in lectures and required for assignment 2**
- **LC-3**
  - Little Computer 3 is an educational ISA developed by Yale N. Patt at UT-Austin
  - Briefly covered in today's lecture for breadth
- **x86-64**
  - A **CISC** ISA developed by Intel Corporation
  - **Most influential ISA** in the world and de facto choice for high-performance computing
  - Covered extensively in COMP2310

# What is a Computer?

- To get a task done by a (general-purpose) computer, we need
  - A computer program
    - That specifies what the computer must do
  - The computer itself
    - To carry out the specified task

- Program: A set of instructions
  - Each instruction specifies a well-defined piece of work for the computer to carry out
  - Instruction: the smallest piece of specified work in a program

- Instruction set: All possible instructions that a computer is designed to be able to carry out

## The Von Neumann Model

- In order to build a computer, we need an execution model for processing computer programs

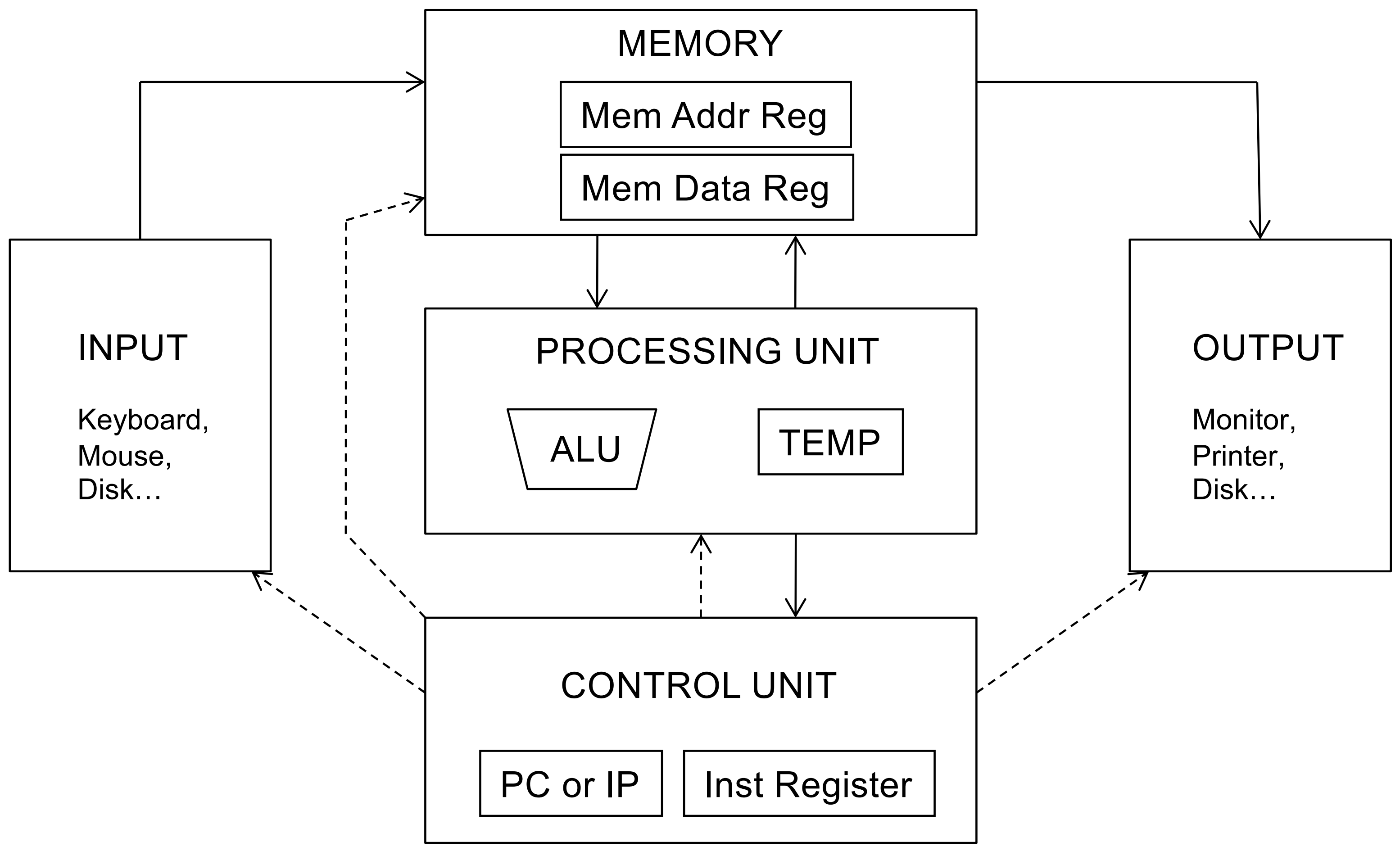
- John von Neumann proposed a fundamental model in 1946

- The von Neumann Model consists of 5 components
  - Memory (stores the program and data)
  - Processing unit
  - Input
  - Output
  - Control unit (controls the order in which instructions are carried out)

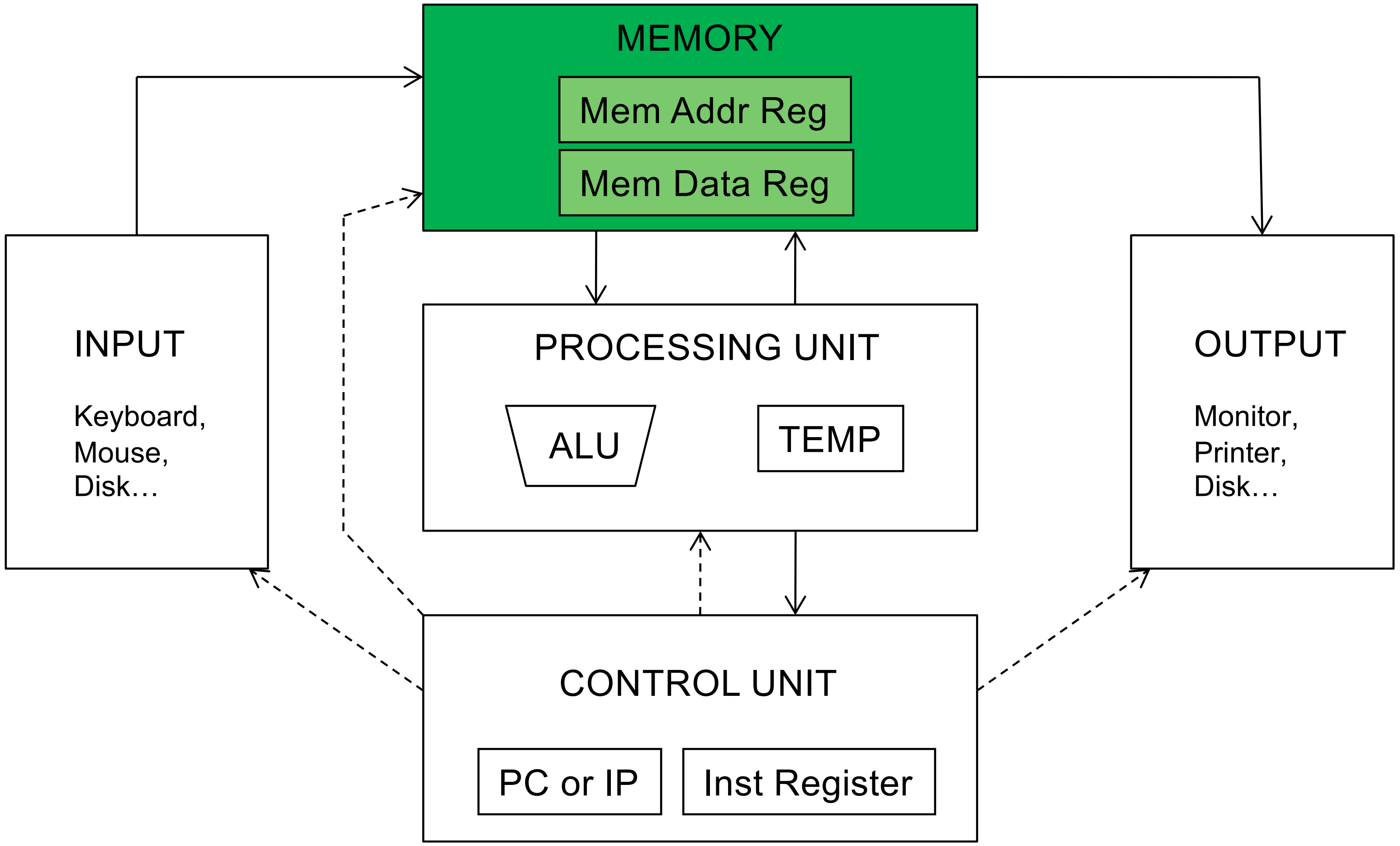
Burks, Goldstein, von Neumann, "Preliminary discussion of the logical design of an electronic computing instrument," 1946.

**All general-purpose computers today use the von Neumann model**

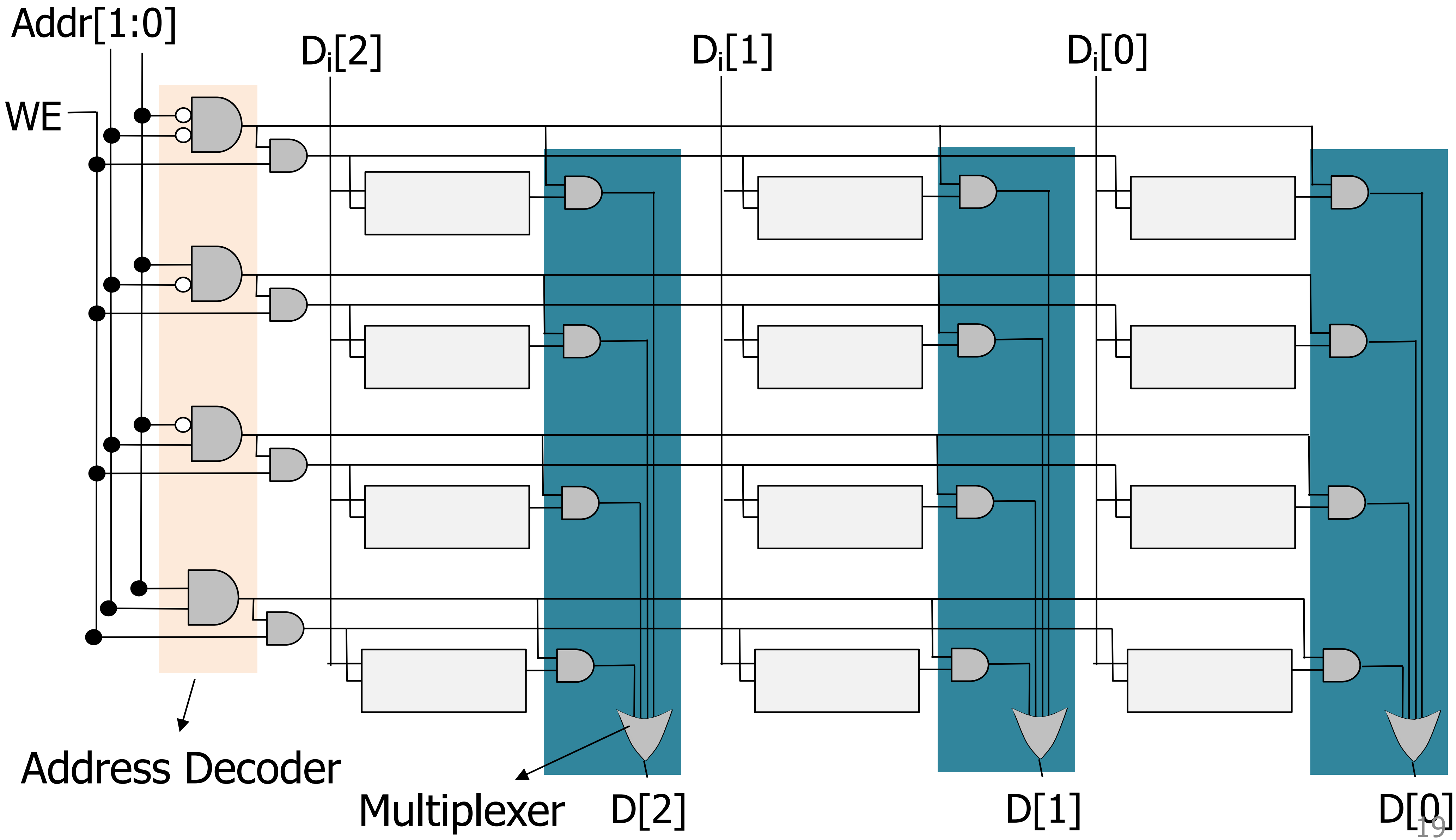
# The Von Neumann Model



## The Von Neumann Model

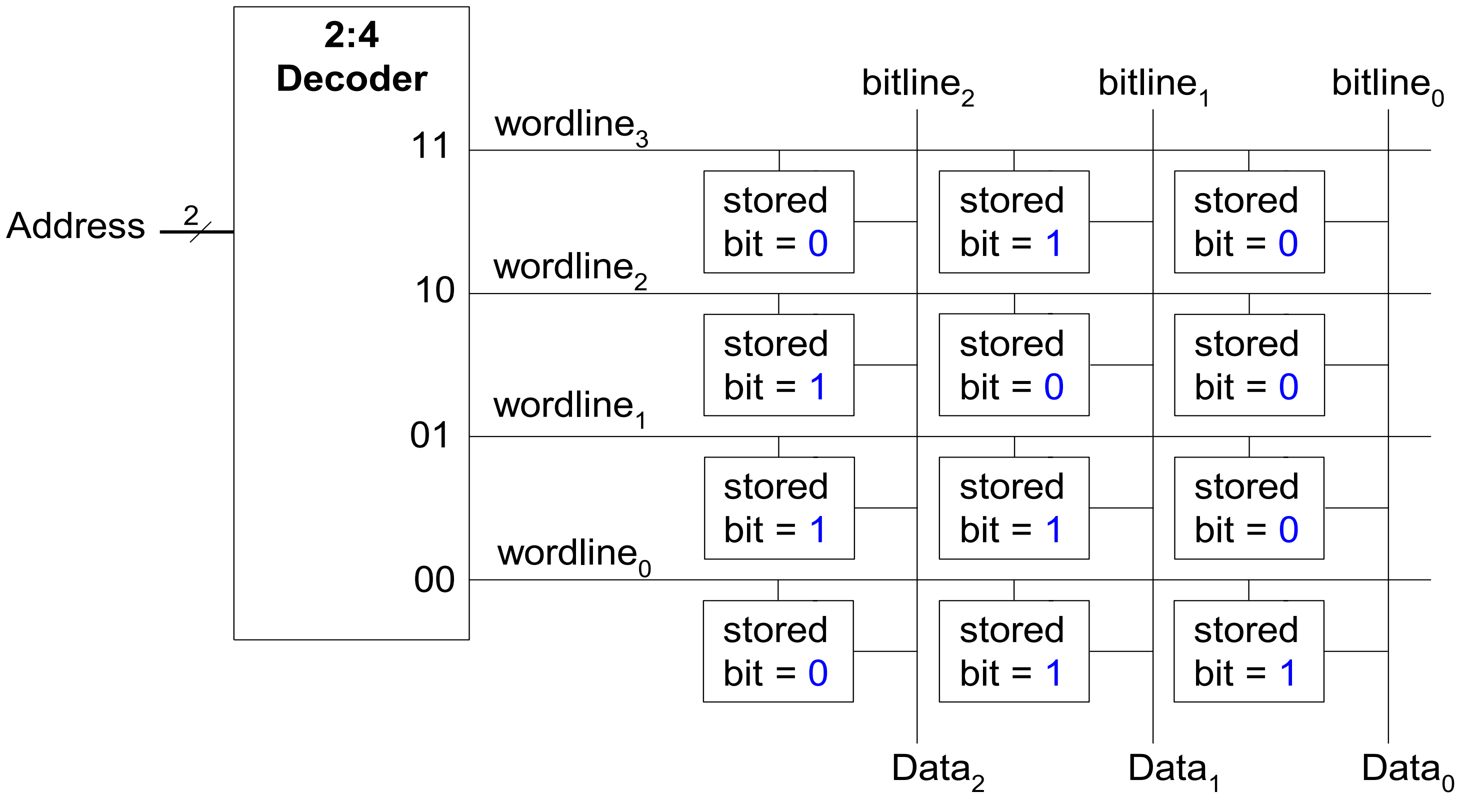


## Recall: A Memory Array (4 locations X 3 bits)


Addr[1:0]
$D_i[2]$
$D_i[1]$
$D_i[0]$
WE
Address Decoder
Multiplexer
D[2]
D[1]
D[0]

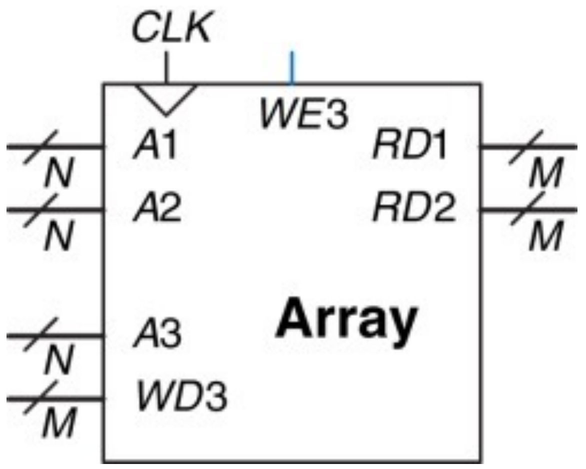
# Recall: Memory Array Organization

- Decoder drives the wordline HIGH based on the address
- Data on the selected row appears on the bitlines



# Recall: Memory Ports

- Each memory port gives **read** or **write** access to one memory address
- **Multiported memories** can access **multiple** addresses **simultaneously**
- Example of three-ported memory
  - Port 1 reads the data from address A1 onto the read data output RD1
  - Port 2 reads the data from address A2 onto the read data output RD2
  - Port 3 writes the data from the write data input WD3 into address A3 on the rising clock edge if WE3 is **TRUE**



## Memory

- Memory stores
  - Programs
  - Data

- Memory contains bits
  - Bits are logically grouped into bytes (8 bits) and words (e.g., 8, 16, 32 bits)

- Address space: Total number of uniquely identifiable locations
  - In MIPS, the address space is $2^{32}$
    - 32-bit addresses
  - In ARM, the address space is $2^{32}$
    - 32-bit addresses
  - In x86-64, the address space is (up to) $2^{48}$
    - 48-bit addresses

- Addressability: How many bits are stored in each location (address)
  - E.g., 8-bit addressable (or byte-addressable)
  - E.g., word-addressable
  - A given instruction can operate on a byte or a word

## A Simple Example

- A representation of memory with 8 locations
- Each location contains 8 bits (one byte)
  - Byte addressable memory with an address space of 8
  - Value 6 is stored in address 4 & value 4 is stored in address 6

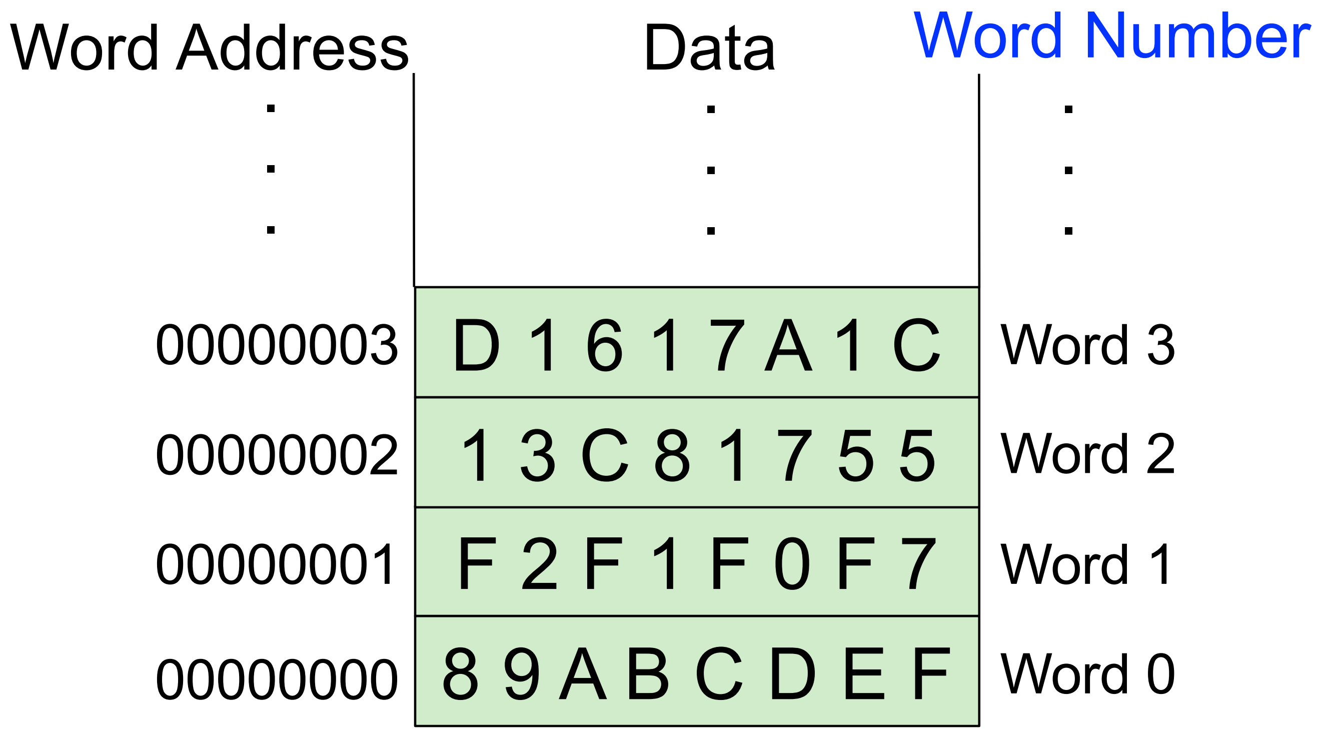
| Address | Data Value |
|---|---|
| 000 | |
| 001 | |
| 010 | |
| 011 | |
| 100 | 00000110 |
| 101 | |
| 110 | 00000100 |
| 111 | |

**Question:**
**How can we make same-size memory bit addressable?**

**Answer:**
**64 locations**
**Each location stores 1 bit**

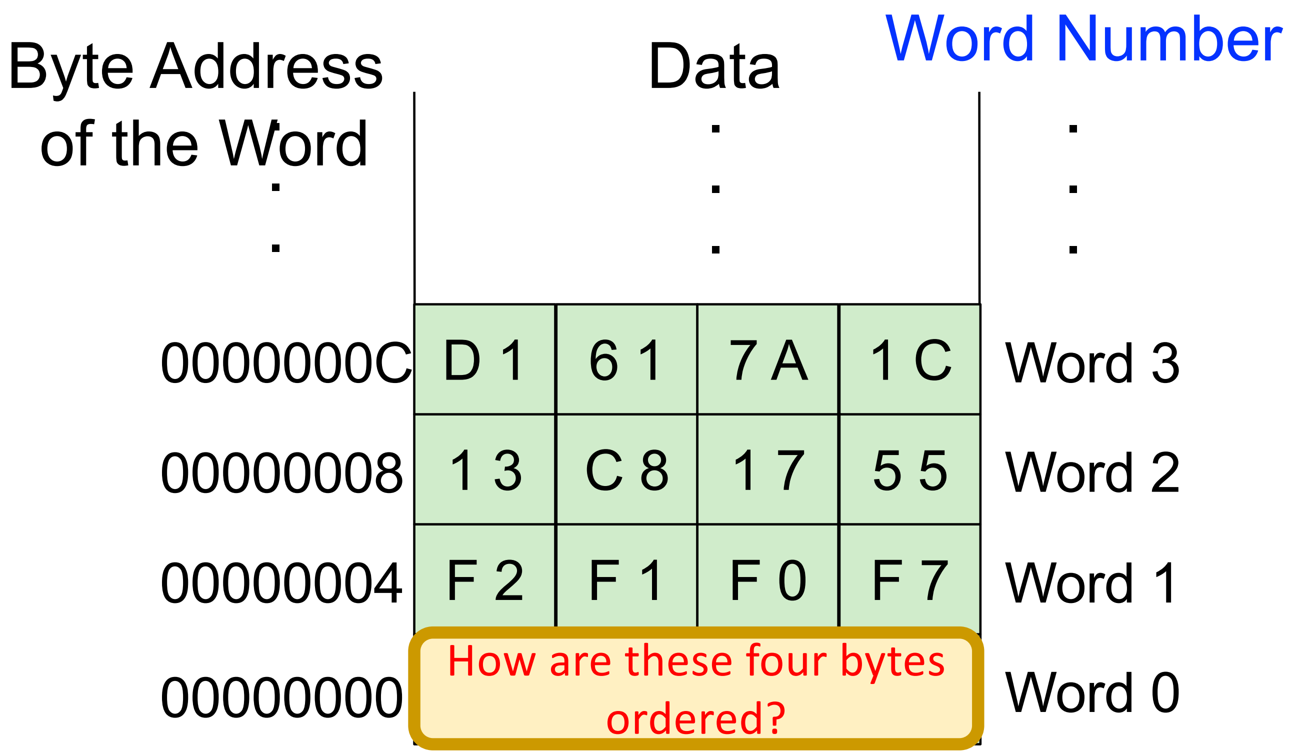
# Word-Addressable Memory

- Each data word has a unique address
  - In MIPS, a unique address for each 32-bit data word (not word-addressable)
  - In **QuAC**, a unique address for each 16-bit data word (word addressable)



# Byte-Addressable Memory

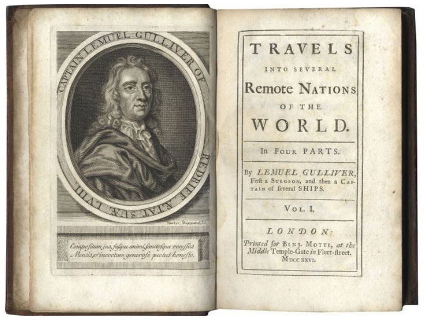
- Each byte has a unique address
  - MIPS is actually byte-addressable
  - ARM is also byte-addressable

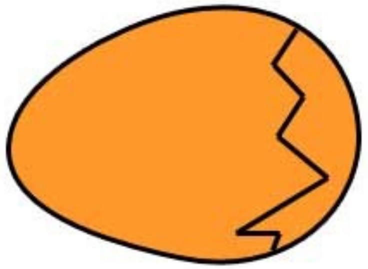


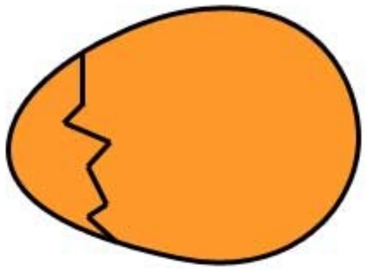
Which of the four bytes is most vs. least significant?

# Big Endian vs. Little Endian

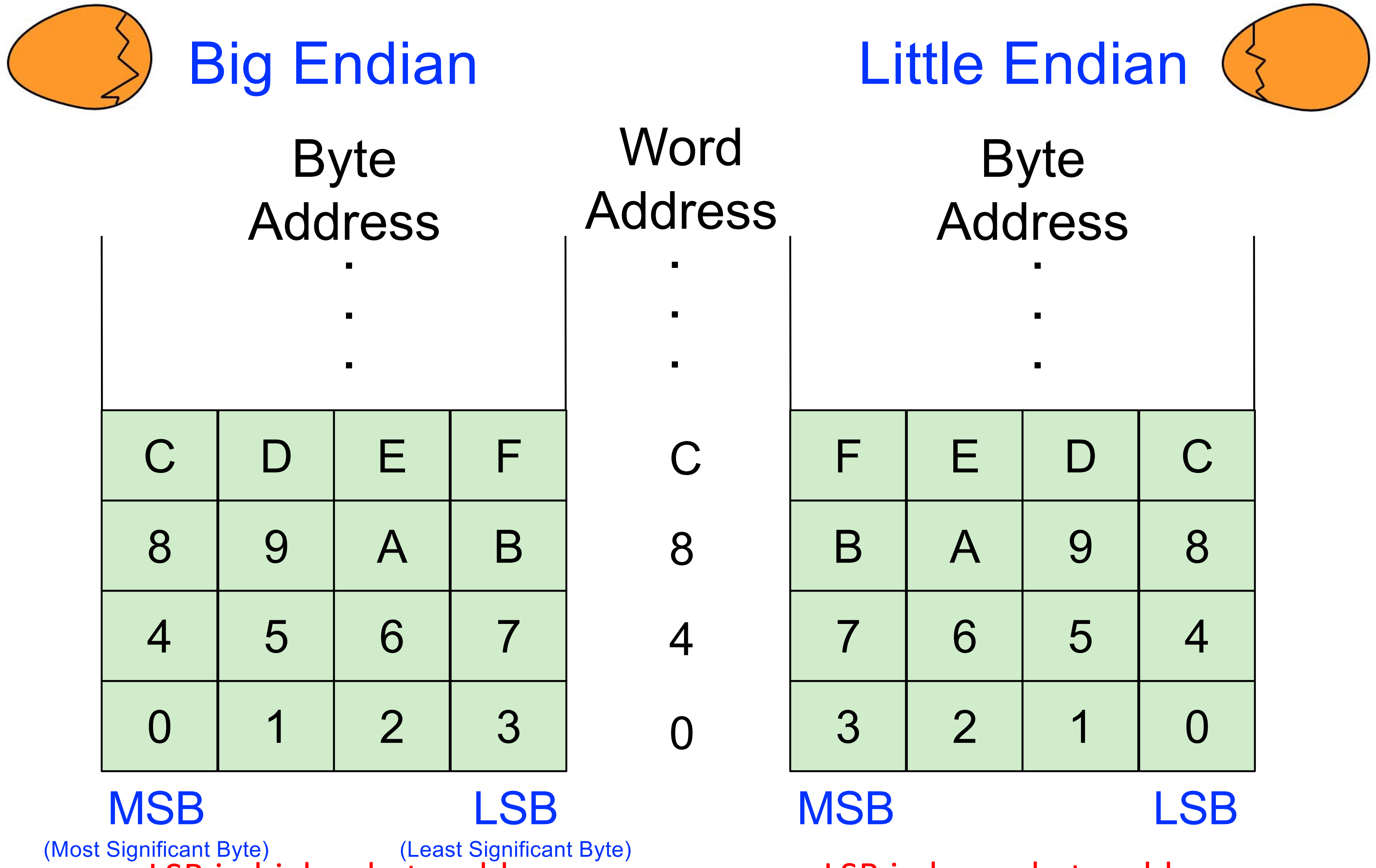
- Jonathan Swift's Gulliver's Travels
  - Big Endians broke their eggs on the big end of the egg
  - Little Endians broke their eggs on the little end of the egg


TRAVELS
INTO SEVERAL
Remote NATIONS
OF THE
WORLD.
In FOUR PARTS.
By LEMUEL GULLIVER,
VOL. I.
LONDON:


BIG ENDIAN - The way people always broke their eggs in the Lilliput land


LITTLE ENDIAN - The way the king then ordered the people to break their eggs

# Big Endian vs. Little Endian

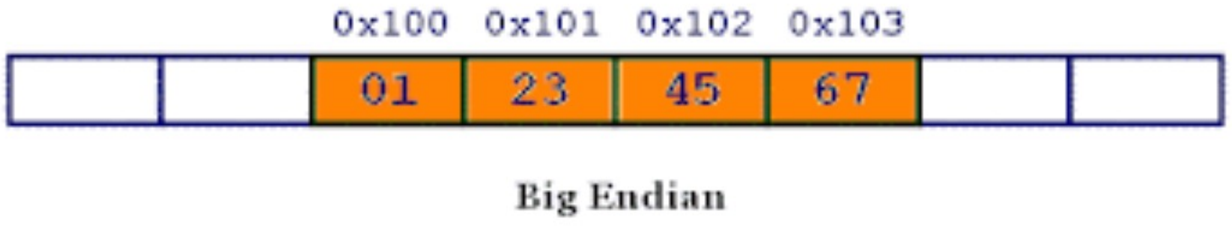


LSB in higher byte address

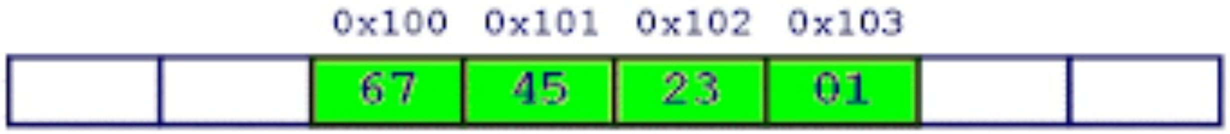
LSB in lower byte address

## Big Endian vs. Little Endian

- 0x01234567
- Memory addresses start at 0x100

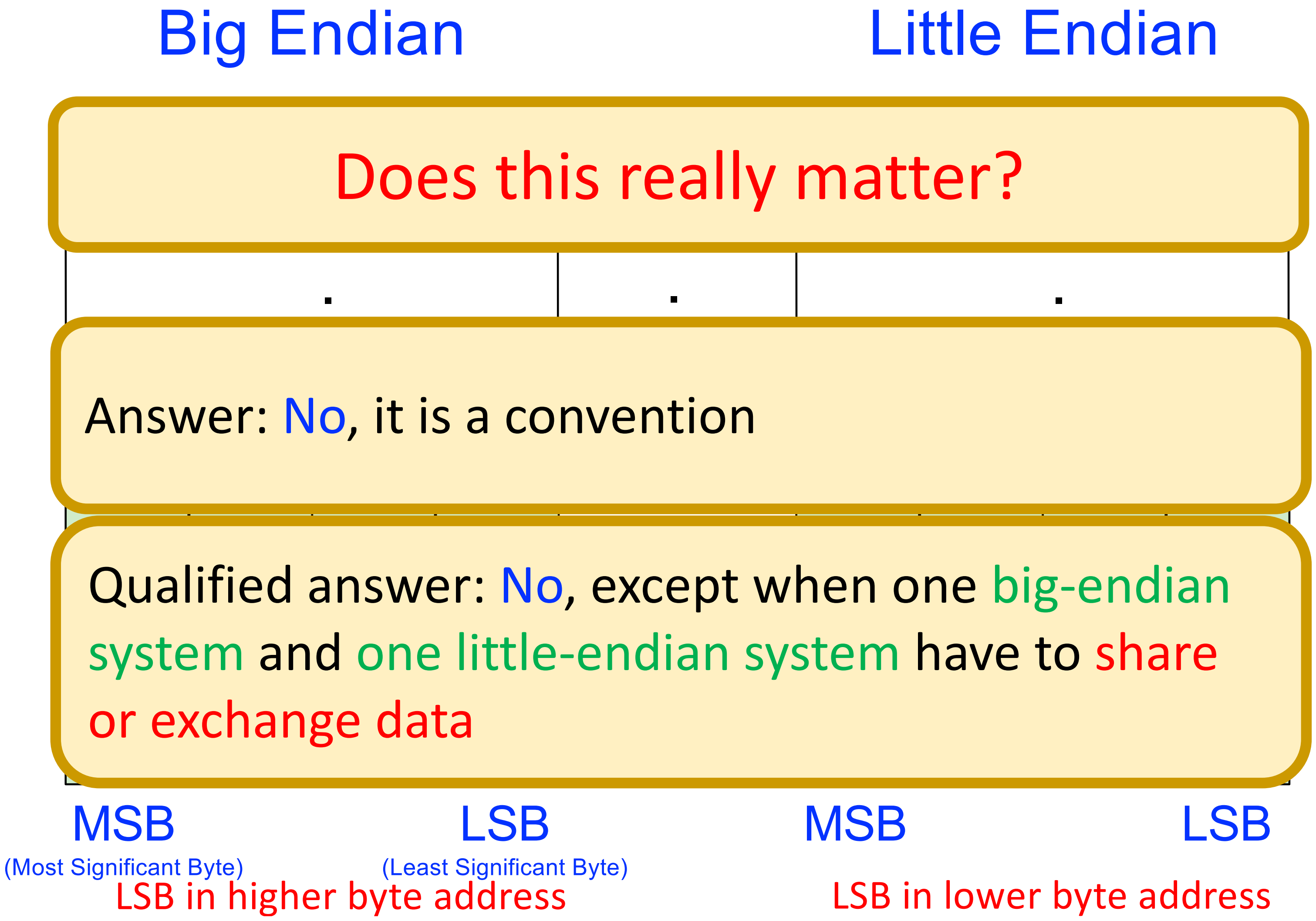


Big Endian



Little Endian

## Big Endian vs. Little Endian



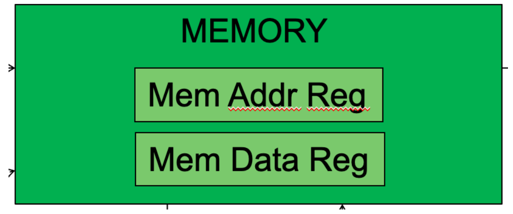
Does this really matter?

Answer: No, it is a convention

Qualified answer: No, except when one big-endian system and one little-endian system have to share or exchange data

## Accessing Memory: MAR and MDR

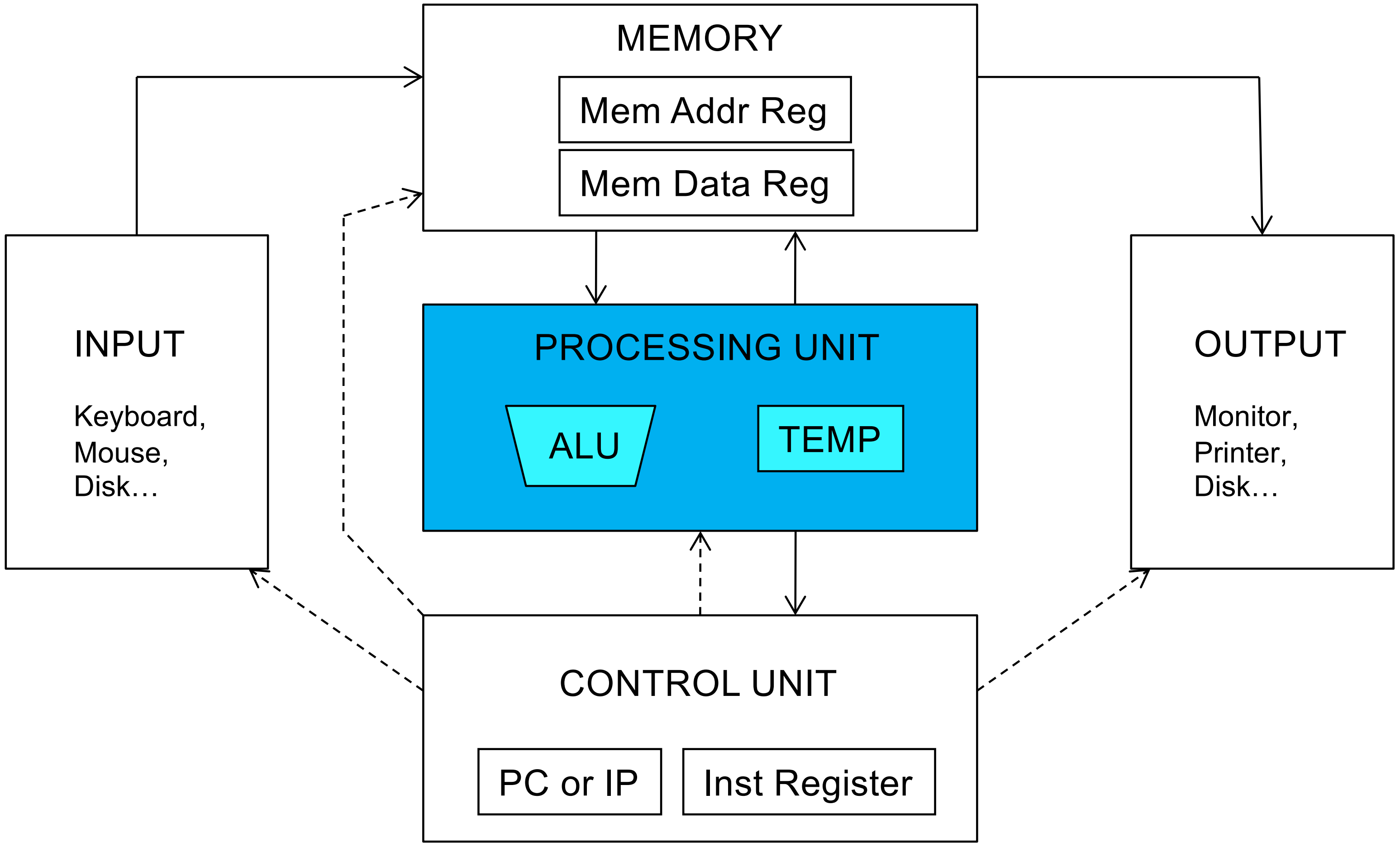
- There are two ways of accessing memory
  - Reading or loading data from a memory location
  - Writing or storing data to a memory location
- Two registers are usually used to access memory
  - Memory Address Register (MAR)
  - Memory Data Register (MDR)



- To read
  - Step 1: Load the MAR with the address we wish to read from
  - Step 2: Data in the corresponding location gets placed in MDR
- To write
  - Step 1: Load the MAR with the address and the MDR with the data we wish to write
  - Step 2: Activate Write Enable signal → value in MDR is written to address specified by MAR

## Learn to Distinguish Address from Data

# The Von Neumann Model



# Processing Unit

- Performs the actual computation(s)

- The processing unit can consist of many functional units

- We start with a simple Arithmetic and Logic Unit (ALU), which executes computation and logic operations
  - ARM: ADD, AND, NOT, SUB
  - MIPS: add, sub, mult, and, nor, sll, slr, slt…

- The ALU processes quantities that are referred to as words
  - Word length in ARM**v4** is 32 bits (**v8** is 64 bits)
  - Word length in MIPS is 32 bits
  - Word length in QuAC is 16 bits

## Recall: Arithmetic & Logic Unit (ALU)

- Combines a variety of arithmetic and logical operations into a single unit (that performs only one function at a time)
- Usually denoted with this symbol:

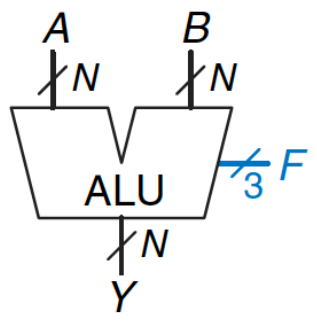


**Figure 5.14 ALU symbol**

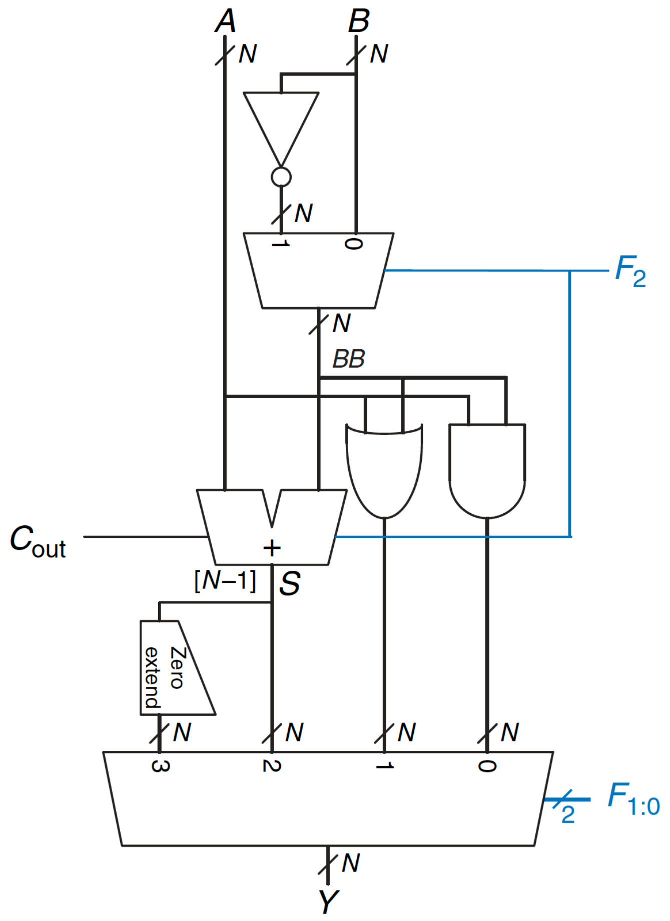
**Table 5.1 ALU operations**

| $F_{2:0}$ | Function |
|---|---|
| 000 | A AND B |
| 001 | A OR B |
| 010 | A + B |
| 011 | not used |
| 100 | A AND $\overline{B}$ |
| 101 | A OR $\overline{B}$ |
| 110 | A – B |
| 111 | SLT |

# Recall: Arithmetic & Logic Unit (ALU)

**Table 5.1 ALU operations**

| $F_{2:0}$ | Function |
|---|---|
| 000 | A AND B |
| 001 | A OR B |
| 010 | A + B |
| 011 | not used |
| 100 | A AND $\overline{\text{B}}$ |
| 101 | A OR $\overline{\text{B}}$ |
| 110 | A – B |
| 111 | SLT |



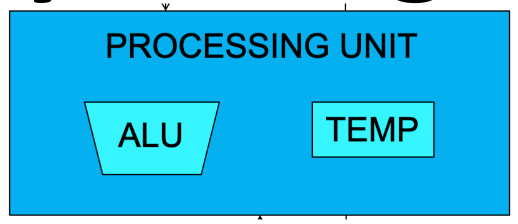
## Processing Unit: Fast Temporary Storage

- It is almost always the case that a computer provides a small amount of storage very close to ALU
  - Purpose: to store temporary values and quickly access them later

- E.g., to calculate ((A+B)*C)/D, the intermediate result of A+B can be stored in temporary storage
  - Why? It is too slow to store each ALU result in memory & then retrieve it again for future use
    - A memory access is much slower than an addition, multiplication or division
  - Ditto for the intermediate result of ((A+B)*C)

- This temporary storage is usually a set of registers
  - Called Register File

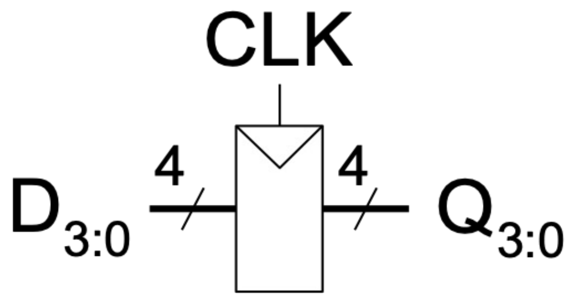
# Registers: Fast Temporary Storage



- Memory is large but slow
- Registers in the Processing Unit
  - Ensure fast access to values to be processed in the ALU
  - Typically one register contains one word (same as word length)
- Register Set or Register File
  - Set of registers that can be manipulated by instructions
  - ARM has 16 general purpose registers (GPRs)
    - R0 to R15: 4-bit register number
    - Register size = Word length = 32 bits
  - MIPS has 32 general purpose registers
    - More elaborate naming scheme: 5-bit register number (or Register ID)
    - Register size = Word length = 32 bits
  - QuAC has 8 general purpose registers (one undefined)

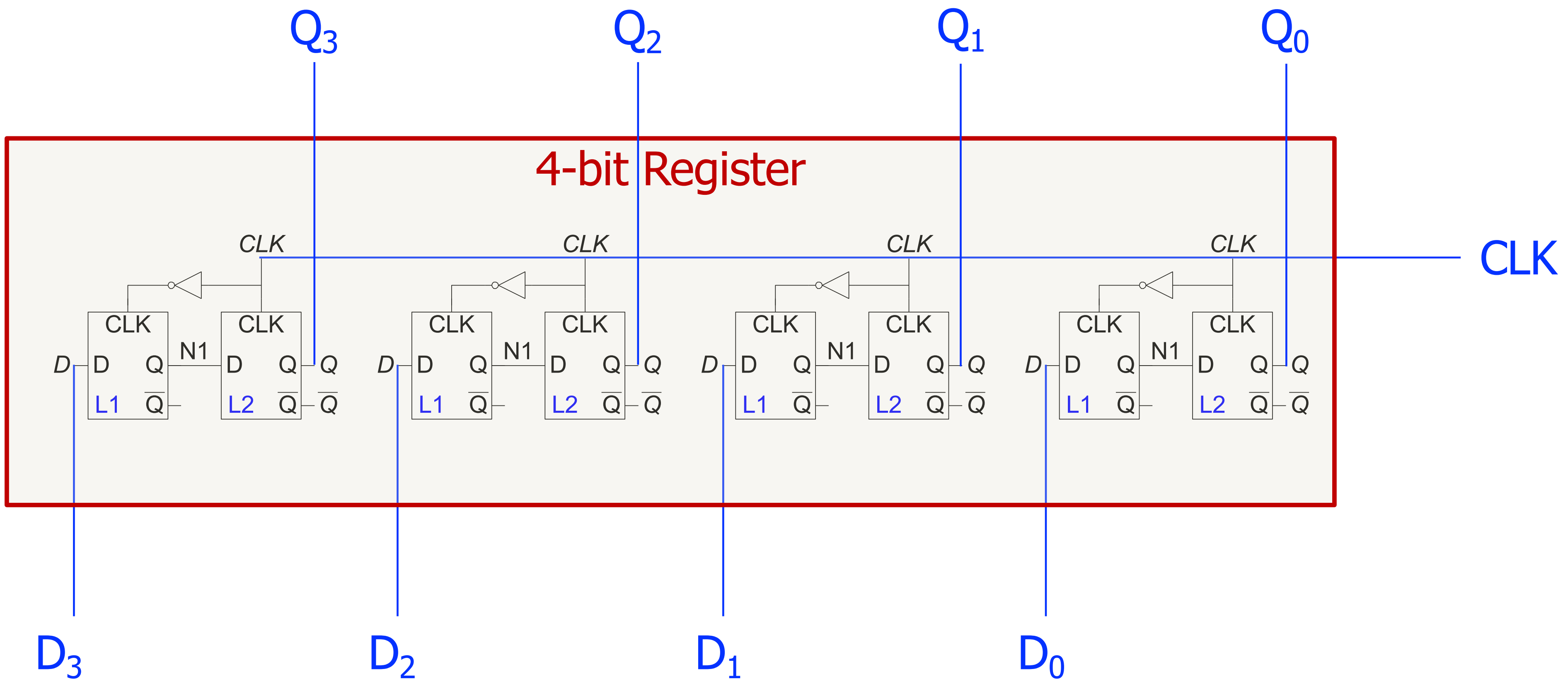
## Recall: Register

- **How can we use flipflops to store more than one bit?**
  - **Principle of modularity:** Use more flipflops!
  - A single CLK to simultaneously write to all flipflops



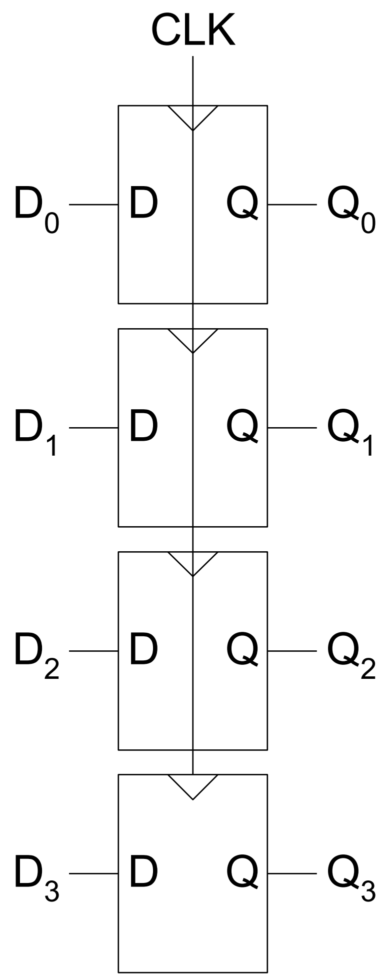
- **Register:** A structure that stores more than **one bit** of information and can be read from and written to
- This **register** holds 4 bits, and its data is referenced as Q[3:0]

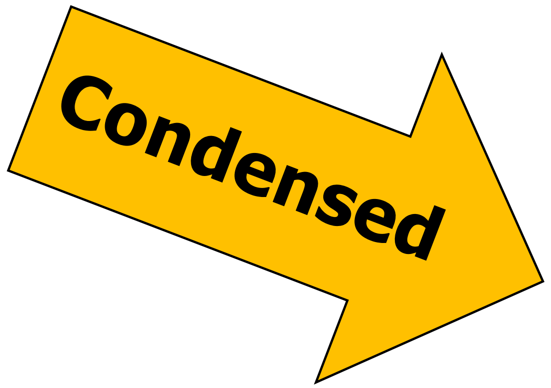
# Recall: 4-bit Register

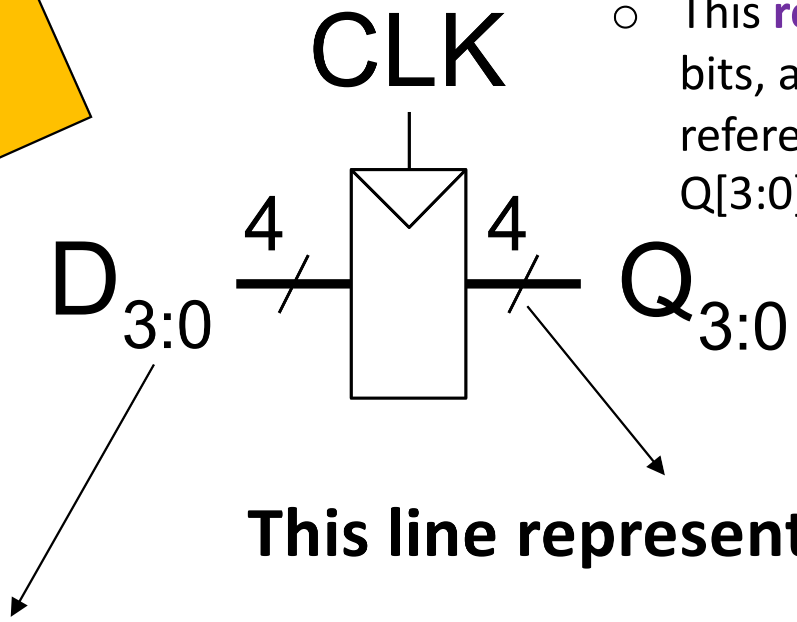


To build an **N-bit** register, use a bank of **N** flipflops with a shared **CLK**

## Recall: 4-bit Register







**This line represents 4 wires**

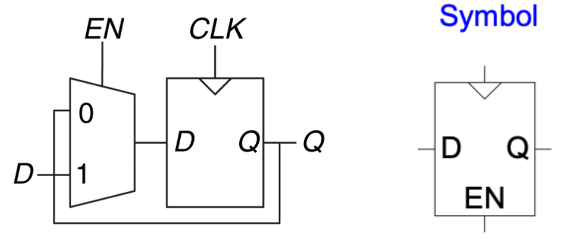
**This register stores 4 bits**

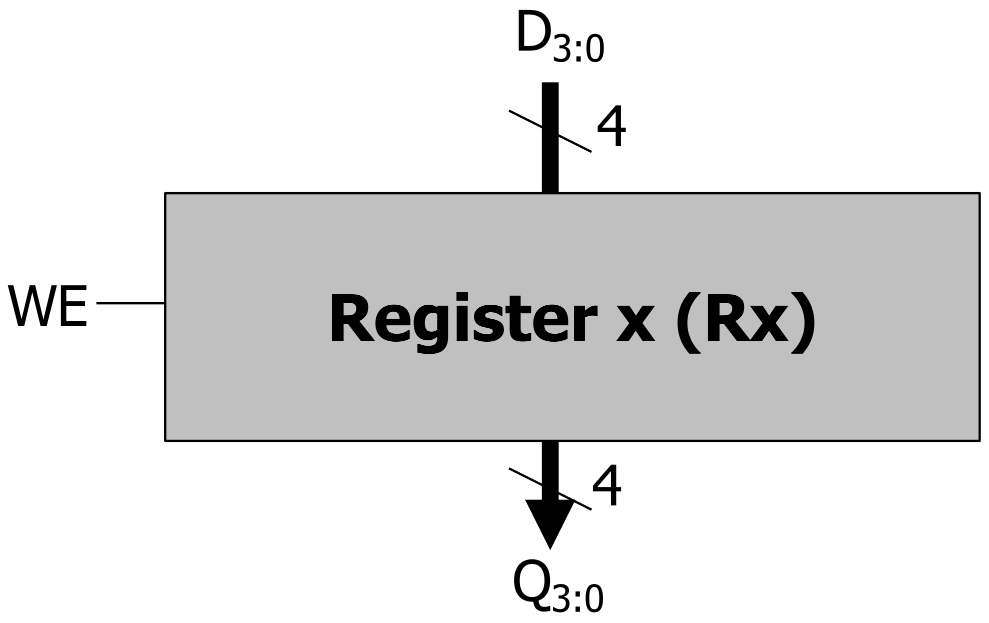
- Here we have a **register,** or a structure that stores more than one bit and can be read from and written to
- This **register** holds 4 bits, and its data is referenced as Q[3:0]

# More Realistic Register

- A single WE signal for all flip-flops for simultaneous writes

Enabled Flip-Flop





## How Registers are Addressed?

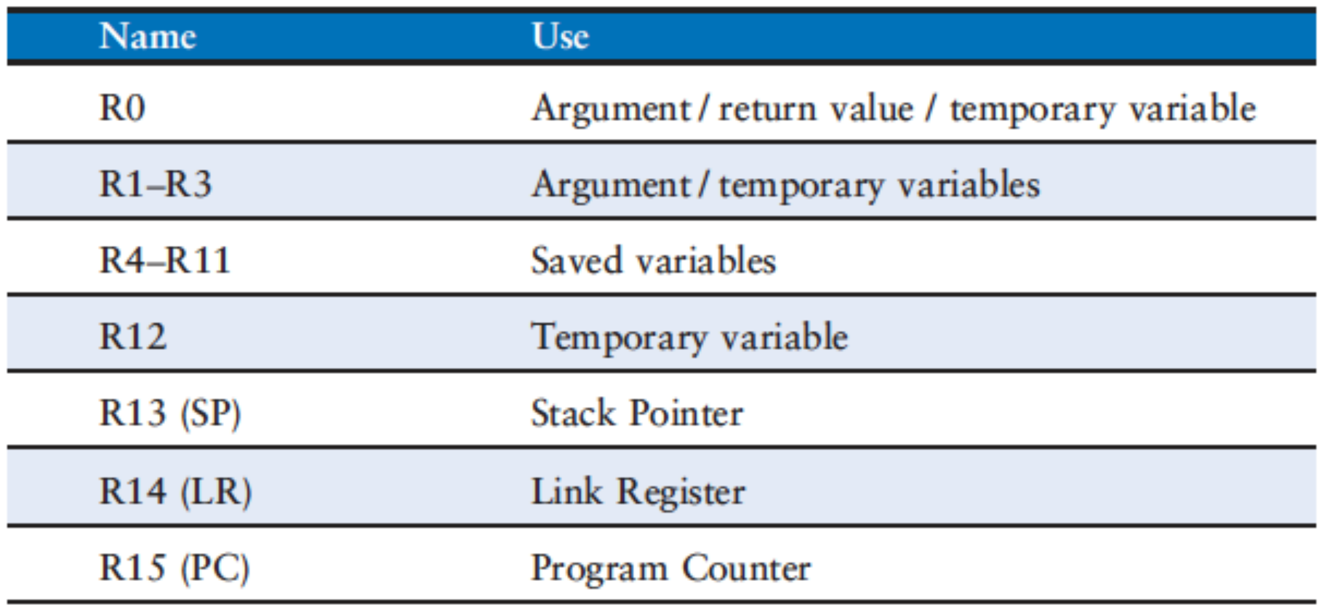
- **Each ISA gives a set of general-purpose registers with special names**
- So, an assembly programmer can use convenient names
- How they are translated into binary addresses is up to the implementation
- Let’s see

# MIPS Register File

| Name | Register Number | Usage |
|---|---|---|
| `$0` | 0 | the constant value 0 |
| `$at` | 1 | assembler temporary |
| `$v0-$v1` | 2-3 | function return value |
| `$a0-$a3` | 4-7 | function arguments |
| `$t0-$t7` | 8-15 | temporary variables |
| `$s0-$s7` | 16-23 | saved variables |
| `$t8-$t9` | 24-25 | temporary variables |
| `$k0-$k1` | 26-27 | OS temporaries |
| `$gp` | 28 | global pointer |
| `$sp` | 29 | stack pointer |
| `$fp` | 30 | frame pointer |
| `$ra` | 31 | function return address |

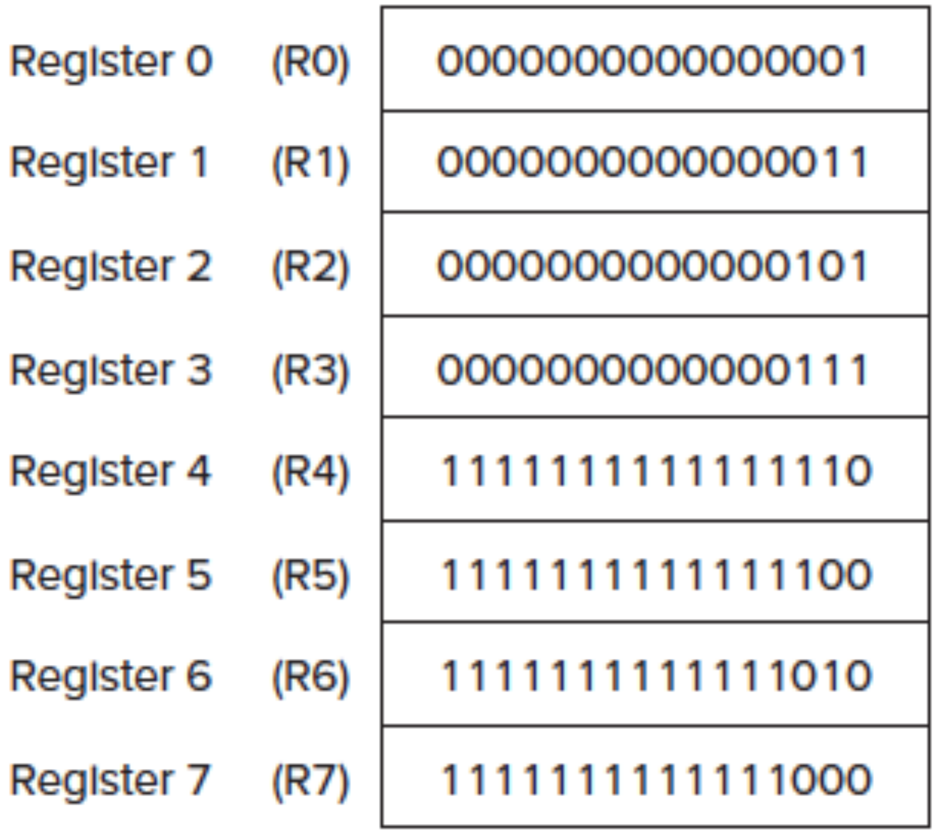
## ARM Register File

Table 6.1 **ARM register set**



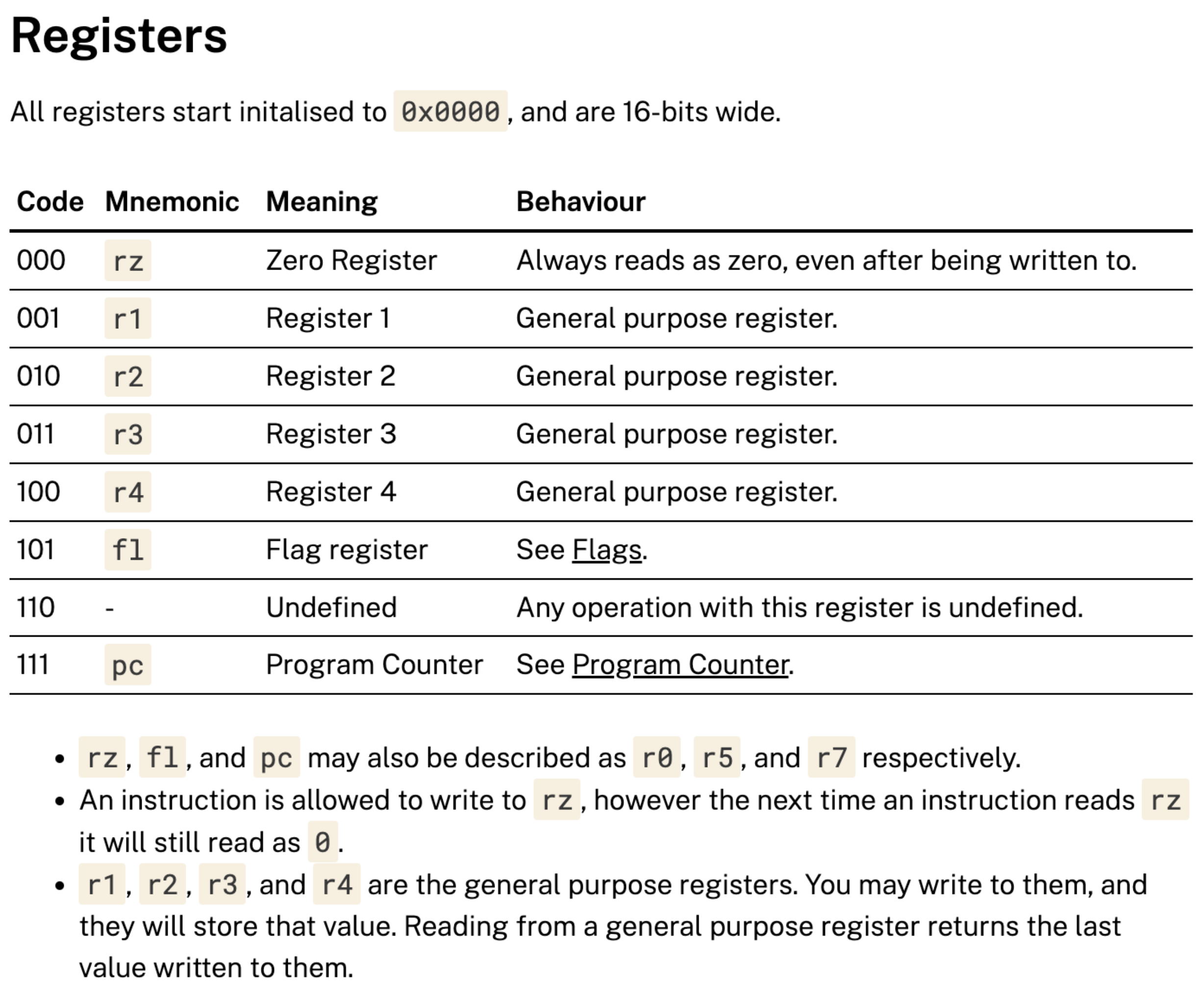
| Name | Use |
|---|---|
| R0 | Argument / return value / temporary variable |
| R1–R3 | Argument / temporary variables |
| R4–R11 | Saved variables |
| R12 | Temporary variable |
| R13 (SP) | Stack Pointer |
| R14 (LR) | Link Register |
| R15 (PC) | Program Counter |

# LC-3 Register File (with Contents)



| | | |
|---|---|---|
| Register 0 | (R0) | 0000000000000001 |
| Register 1 | (R1) | 0000000000000011 |
| Register 2 | (R2) | 0000000000000101 |
| Register 3 | (R3) | 0000000000000111 |
| Register 4 | (R4) | 1111111111111110 |
| Register 5 | (R5) | 1111111111111100 |
| Register 6 | (R6) | 1111111111111010 |
| Register 7 | (R7) | 1111111111111000 |

# QuAC Register File



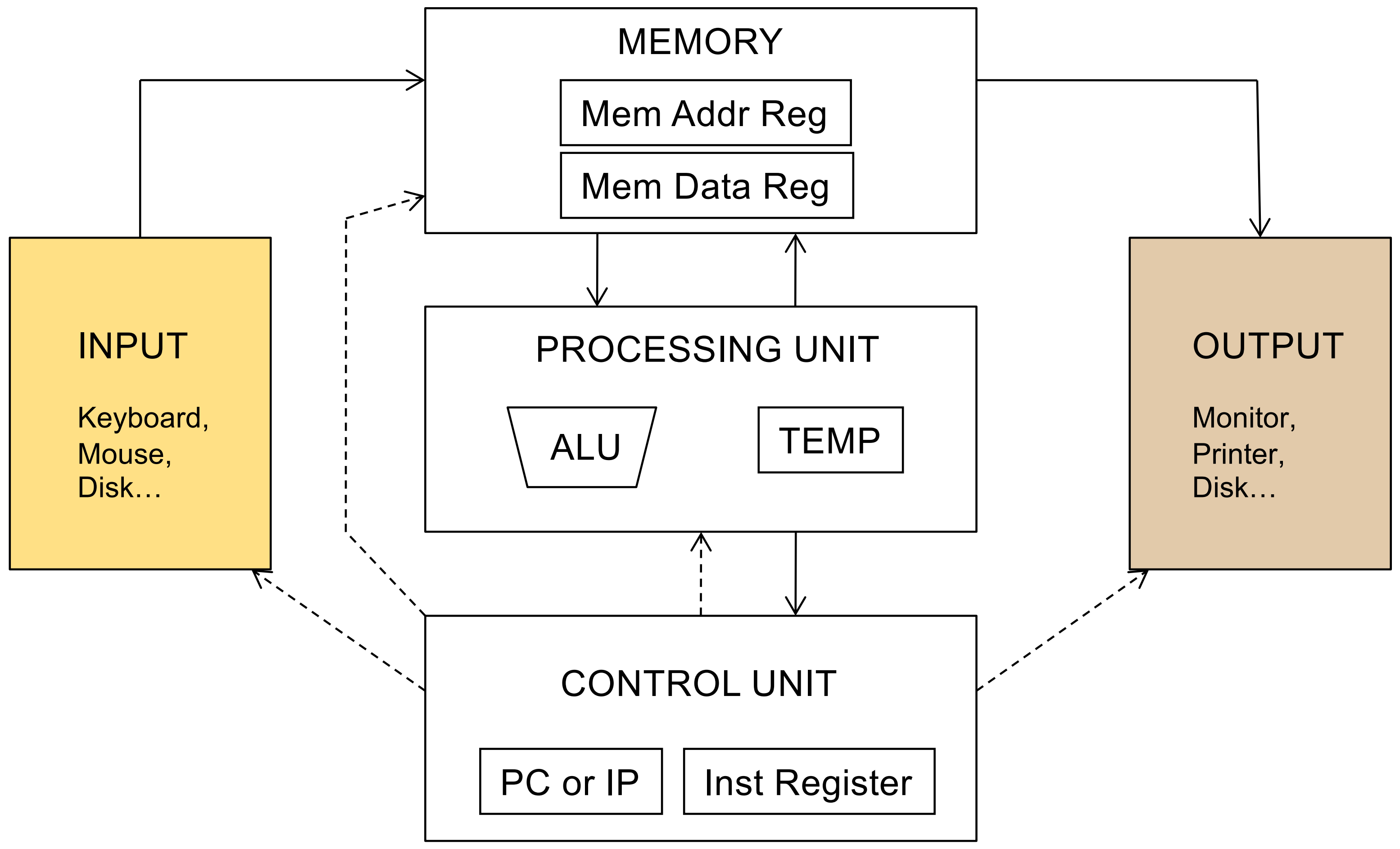
## Registers

All registers start initalised to `0x0000`, and are 16-bits wide.

| Code | Mnemonic | Meaning | Behaviour |
|---|---|---|---|
| 000 | `rz` | Zero Register | Always reads as zero, even after being written to. |
| 001 | `r1` | Register 1 | General purpose register. |
| 010 | `r2` | Register 2 | General purpose register. |
| 011 | `r3` | Register 3 | General purpose register. |
| 100 | `r4` | Register 4 | General purpose register. |
| 101 | `fl` | Flag register | See Flags. |
| 110 | - | Undefined | Any operation with this register is undefined. |
| 111 | `pc` | Program Counter | See Program Counter. |

- `rz`, `fl`, and `pc` may also be described as `r0`, `r5`, and `r7` respectively.
- An instruction is allowed to write to `rz`, however the next time an instruction reads `rz` it will still read as `0`.
- `r1`, `r2`, `r3`, and `r4` are the general purpose registers. You may write to them, and they will store that value. Reading from a general purpose register returns the last value written to them.

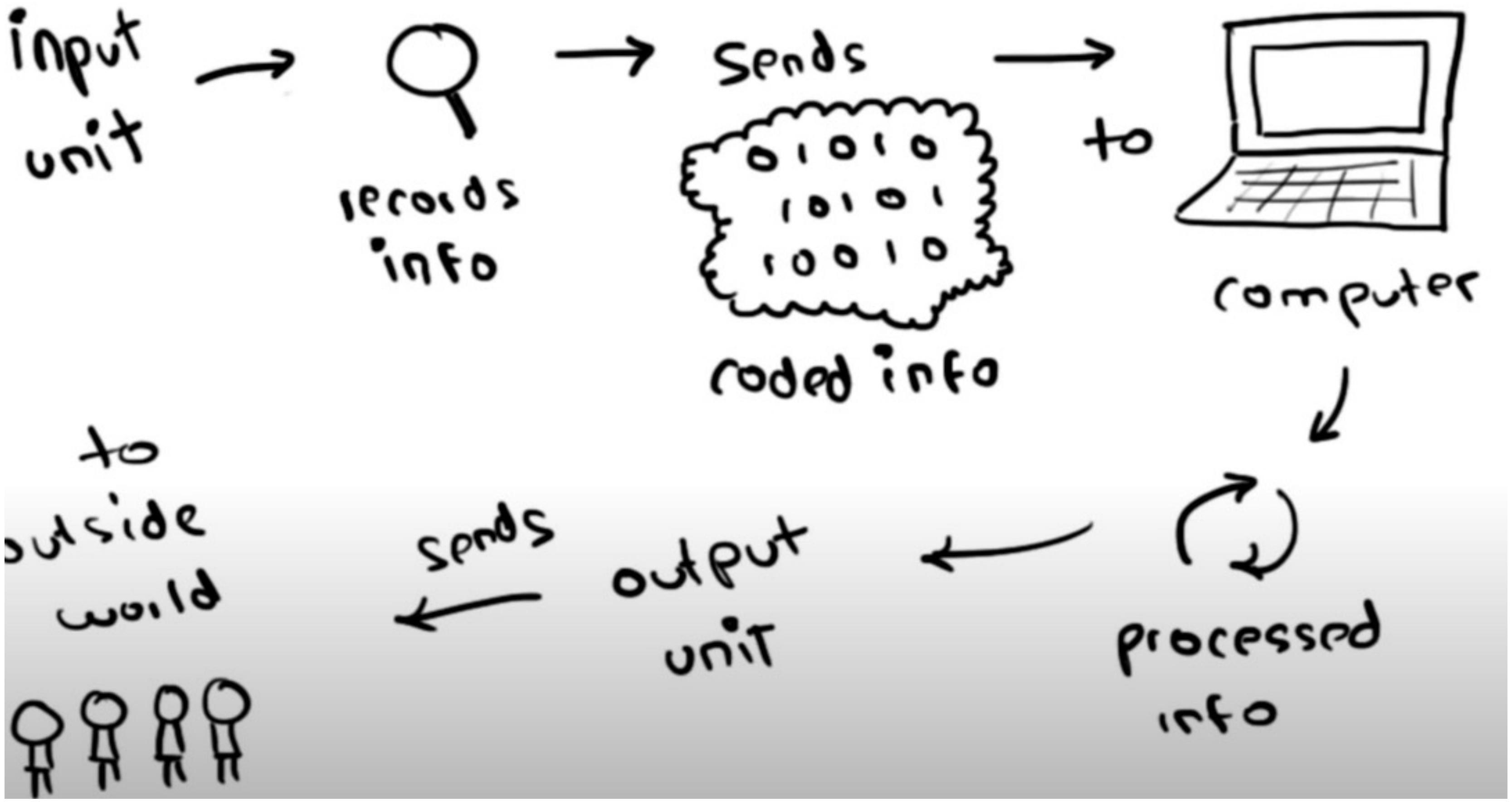
## The Von Neumann Model



## Input and Output

- Enable information to get into and out of a computer
- Many devices can be used for input and output
- They are called peripherals
  - Input
    - Keyboard
    - Mouse
    - Scanner
    - Disks
    - Etc.
  - Output
    - Monitor
    - Printer
    - Disks
    - Etc.

## Input and Output



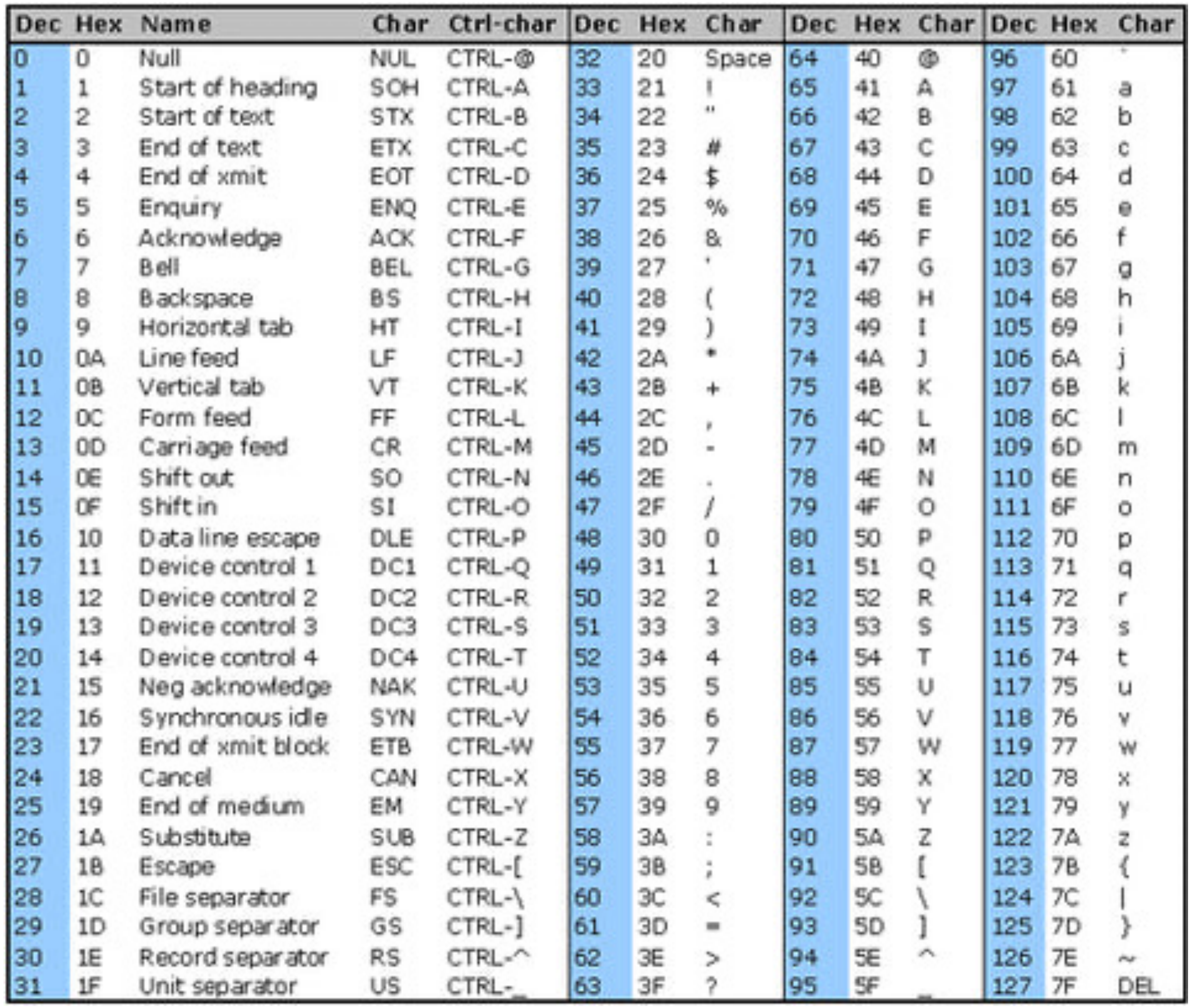
## Keyboard and Monitor

- The simplest keyboard has two registers
  - Keyboard data register **(KBDR)** for holding the ASCII code of keys struck
  - Keyboard status register **(KBSR)** for maintaining status information about the keys struck

- The simplest monitor has two registers
  - Display data register **(DDR)** for holding the ASCII code of something to be displayed on the screen
  - Display status register **(DSR)** for maintaining associated status information

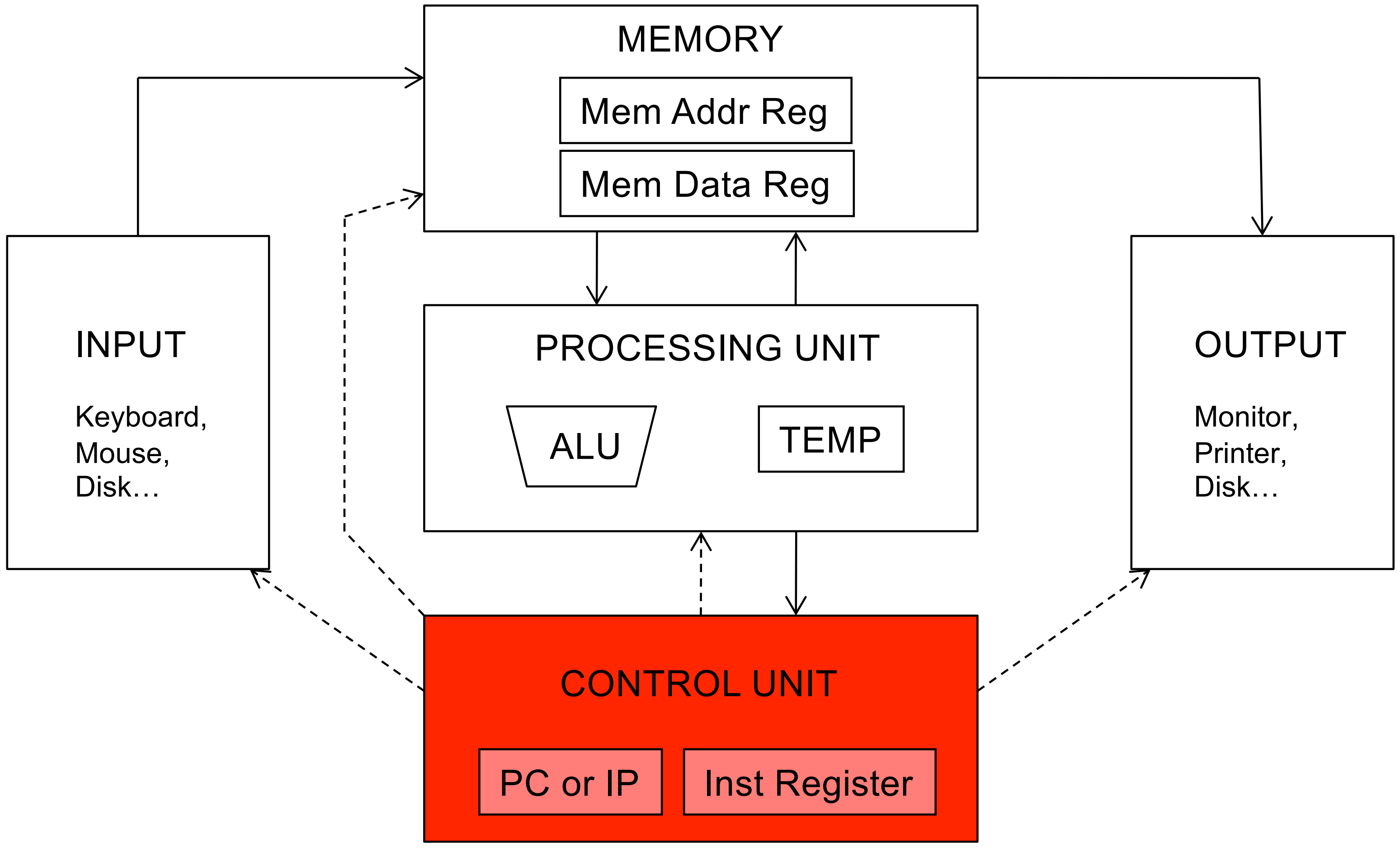
# ASCII Encoding

- ASCII stands for American Standard Code for Information Interchange
- It ranges from 0 to 255 in Decimal or 00 to FF in Hexadecimal
- All characters on an English keyboard can be represented using 8-bit codes



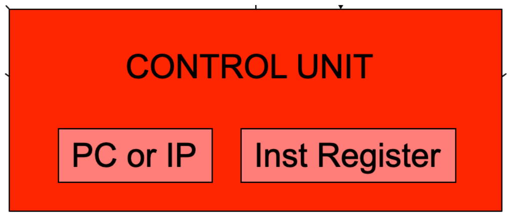
| Dec | Hex | Name | Char | Ctrl-char | Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | Null | NUL | CTRL-@ | 32 | 20 | Space | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 1 | Start of heading | SOH | CTRL-A | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 2 | Start of text | STX | CTRL-B | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 3 | End of text | ETX | CTRL-C | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 4 | End of xmit | EOT | CTRL-D | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 5 | Enquiry | ENQ | CTRL-E | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | Acknowledge | ACK | CTRL-F | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | Bell | BEL | CTRL-G | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 8 | Backspace | BS | CTRL-H | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 9 | Horizontal tab | HT | CTRL-I | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | 0A | Line feed | LF | CTRL-J | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | 0B | Vertical tab | VT | CTRL-K | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | 0C | Form feed | FF | CTRL-L | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | 0D | Carriage feed | CR | CTRL-M | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | 0E | Shift out | SO | CTRL-N | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | 0F | Shift in | SI | CTRL-O | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | Data line escape | DLE | CTRL-P | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | Device control 1 | DC1 | CTRL-Q | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | Device control 2 | DC2 | CTRL-R | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | Device control 3 | DC3 | CTRL-S | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | Device control 4 | DC4 | CTRL-T | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | Neg acknowledge | NAK | CTRL-U | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | Synchronous idle | SYN | CTRL-V | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | End of xmit block | ETB | CTRL-W | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | Cancel | CAN | CTRL-X | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | End of medium | EM | CTRL-Y | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | Substitute | SUB | CTRL-Z | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | Escape | ESC | CTRL-[ | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | File separator | FS | CTRL-\ | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 29 | 1D | Group separator | GS | CTRL-] | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | Record separator | RS | CTRL-^ | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | Unit separator | US | CTRL-_ | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | DEL |

# The Von Neumann Model

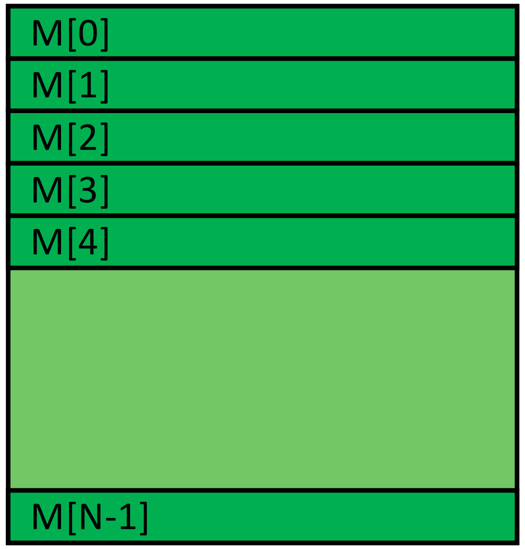


## Control Unit

- The control unit is like the conductor of an orchestra
- It conducts the step-by-step process of executing (every instruction in) a program
- It keeps track of which instruction being processed, via
  - Instruction Register (IR), which contains the instruction
- It also keeps track of which instruction to process next, via
  - Program Counter (PC) or Instruction Pointer (IP), another register that contains the address of the (next) instruction to process



# Programmer Visible (Architectural) State



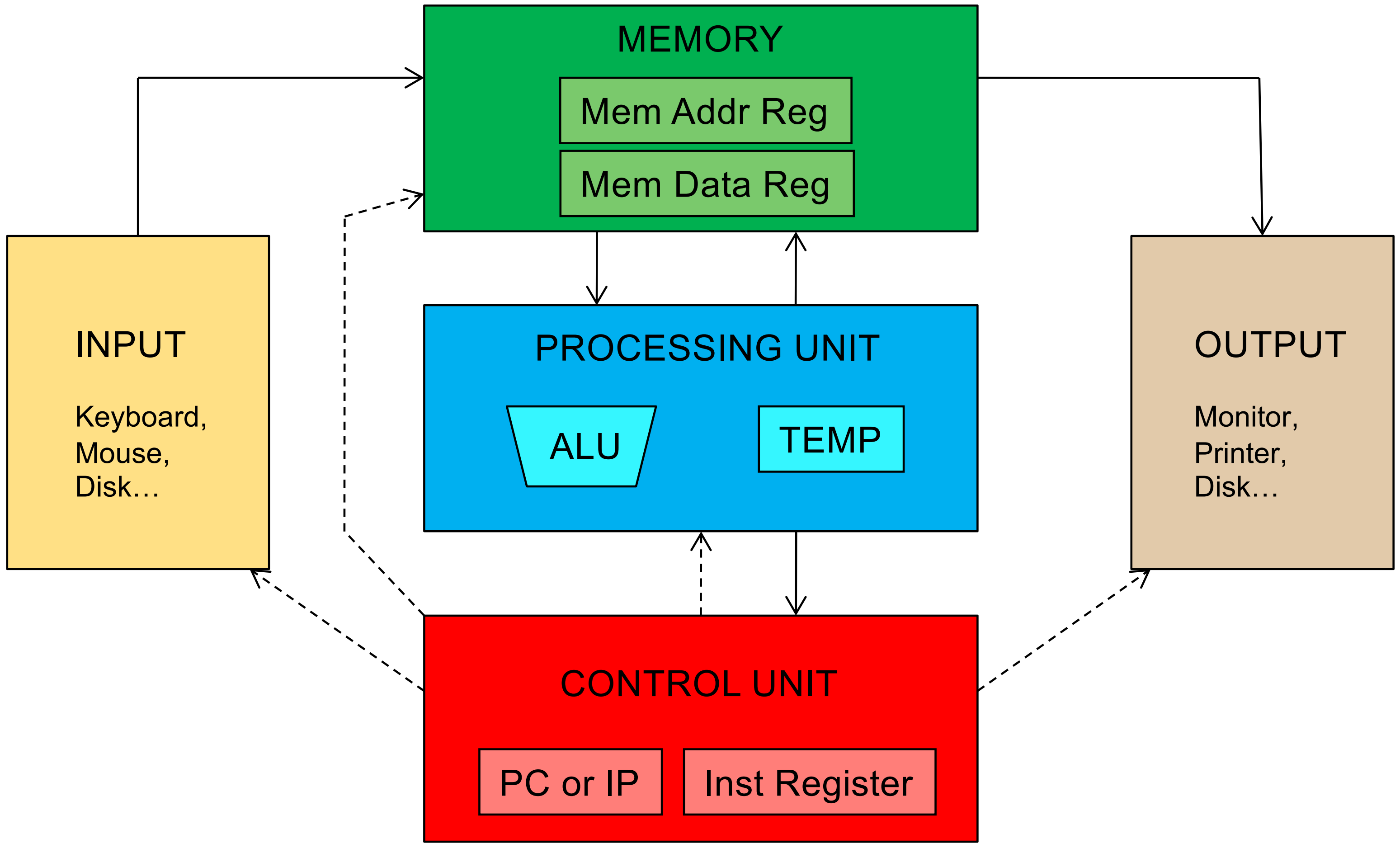
**Memory**
array of storage locations
indexed by an address

**Registers**
- given special names in the ISA
  (as opposed to addresses)
- general vs. special purpose

**Program Counter**
memory address
of the current (or next) instruction

Instructions (and programs) specify how to transform
the values of programmer visible state

# The Von Neumann Model



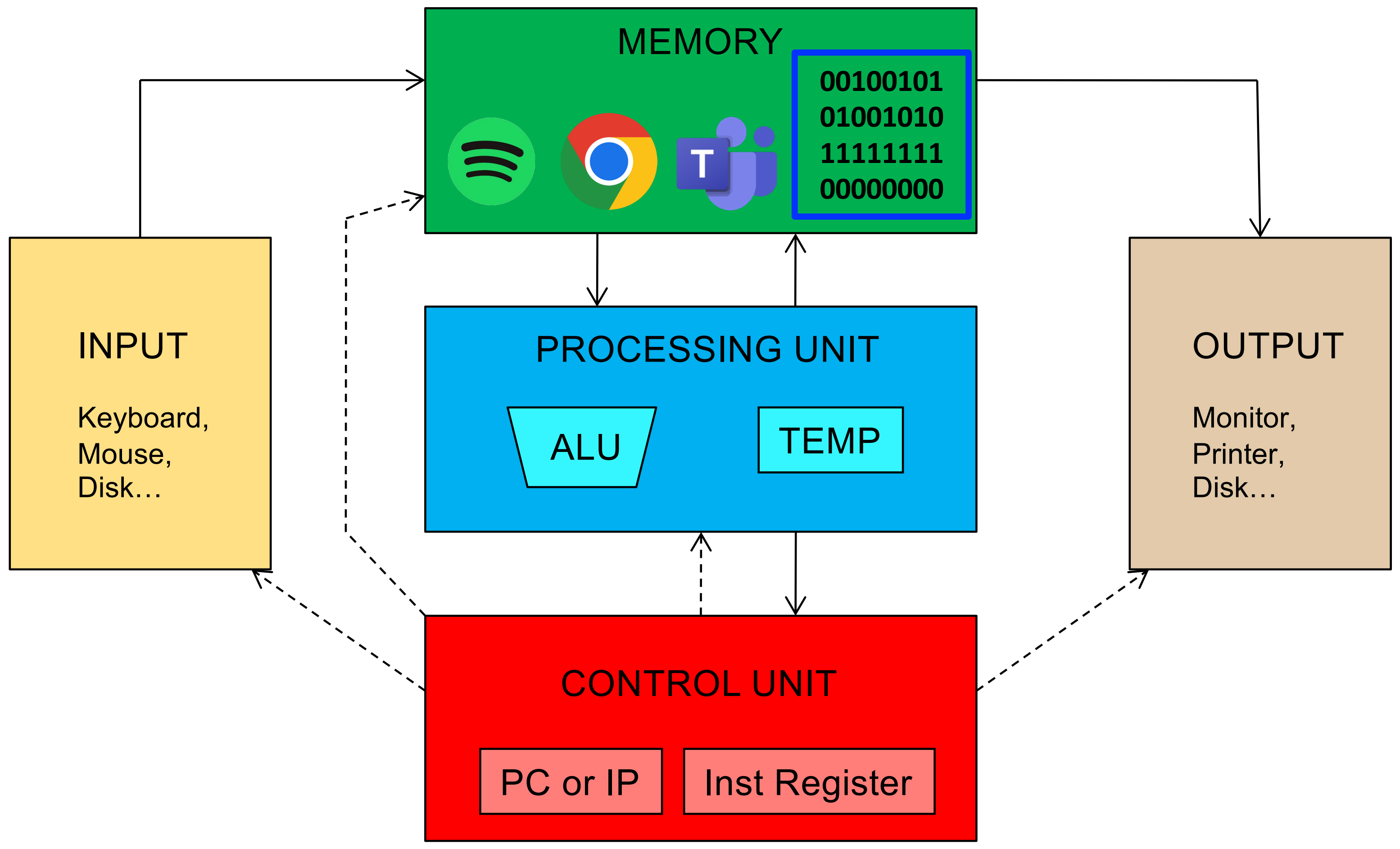
## Von Neumann Model: Two Key Properties

- Von Neumann model is also called ***stored program computer*** (***instructions in memory***). It has two key properties:

- Stored program
  - Instructions stored in a linear memory array
  - Memory is unified between instructions and data
    - The interpretation of a stored value depends on the control signals

- Sequential instruction processing
  - One instruction processed (fetched, executed, completed) at a time
  - Program counter (instruction pointer) identifies the current instruction
  - Program counter is advanced sequentially except for control transfer instructions

## The Von Neumann Model



## Examples of von Neumann Machines

# LC-3: A von Neumann Machine

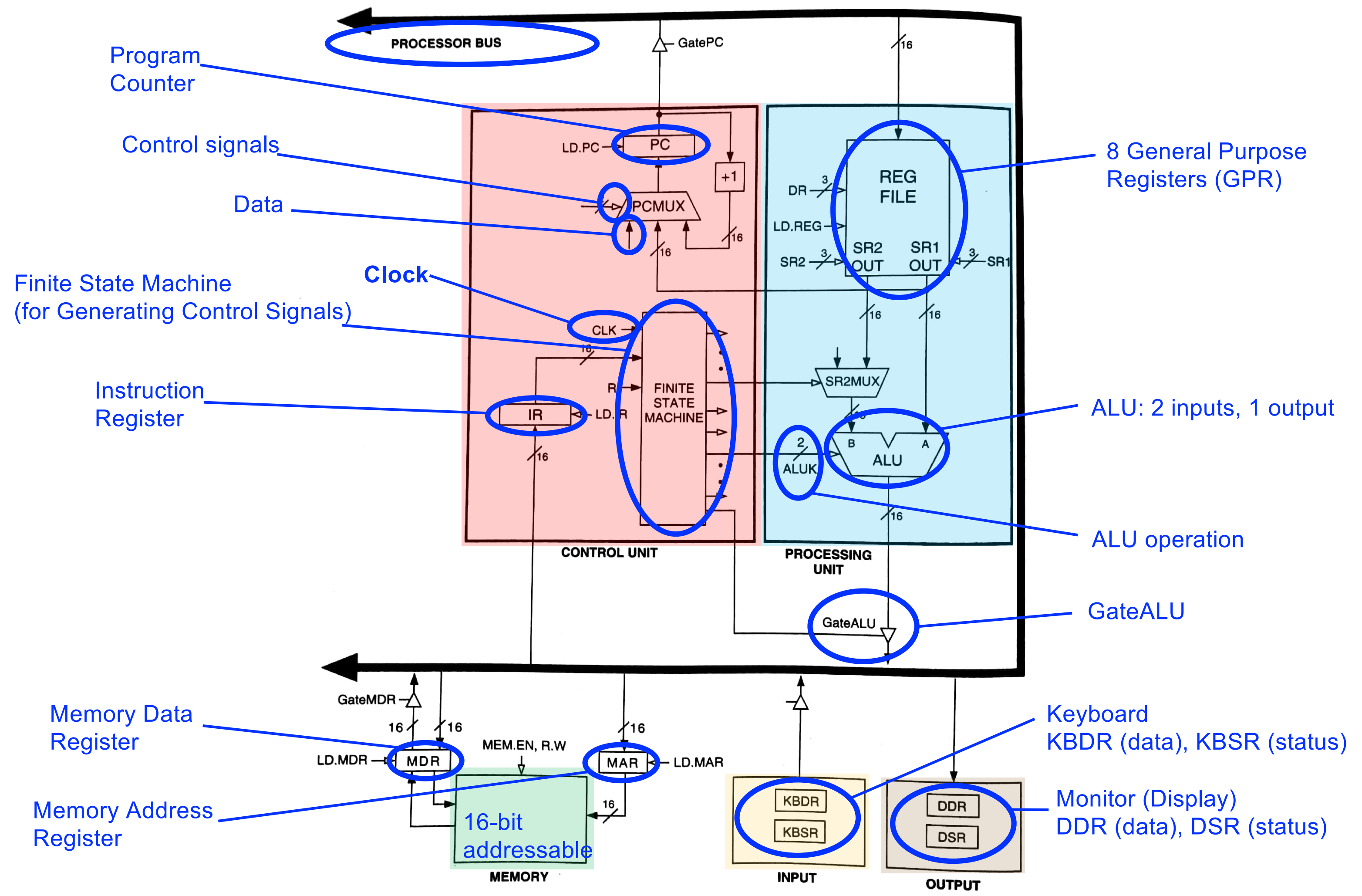
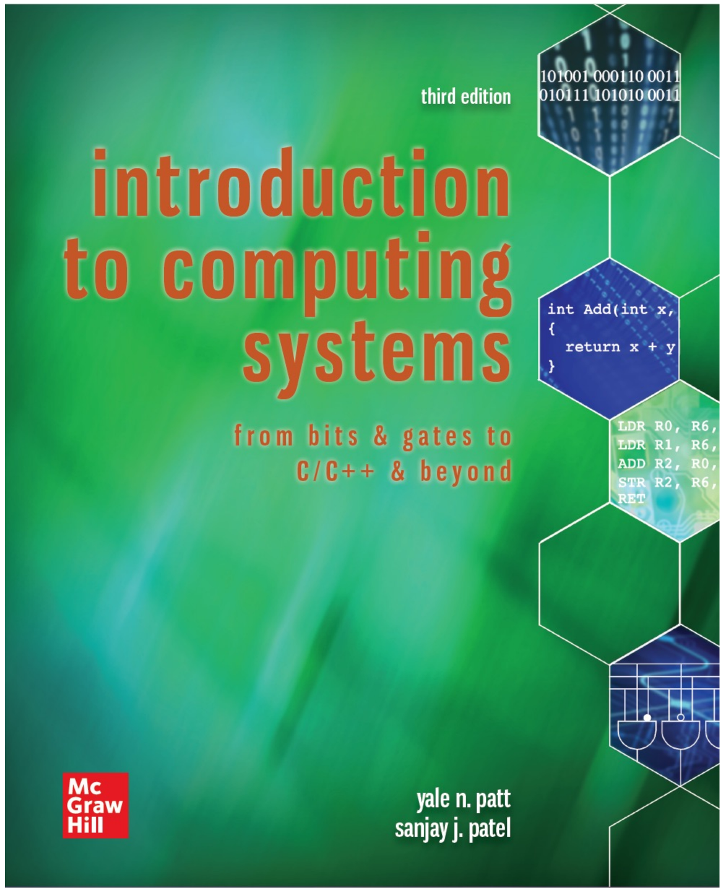
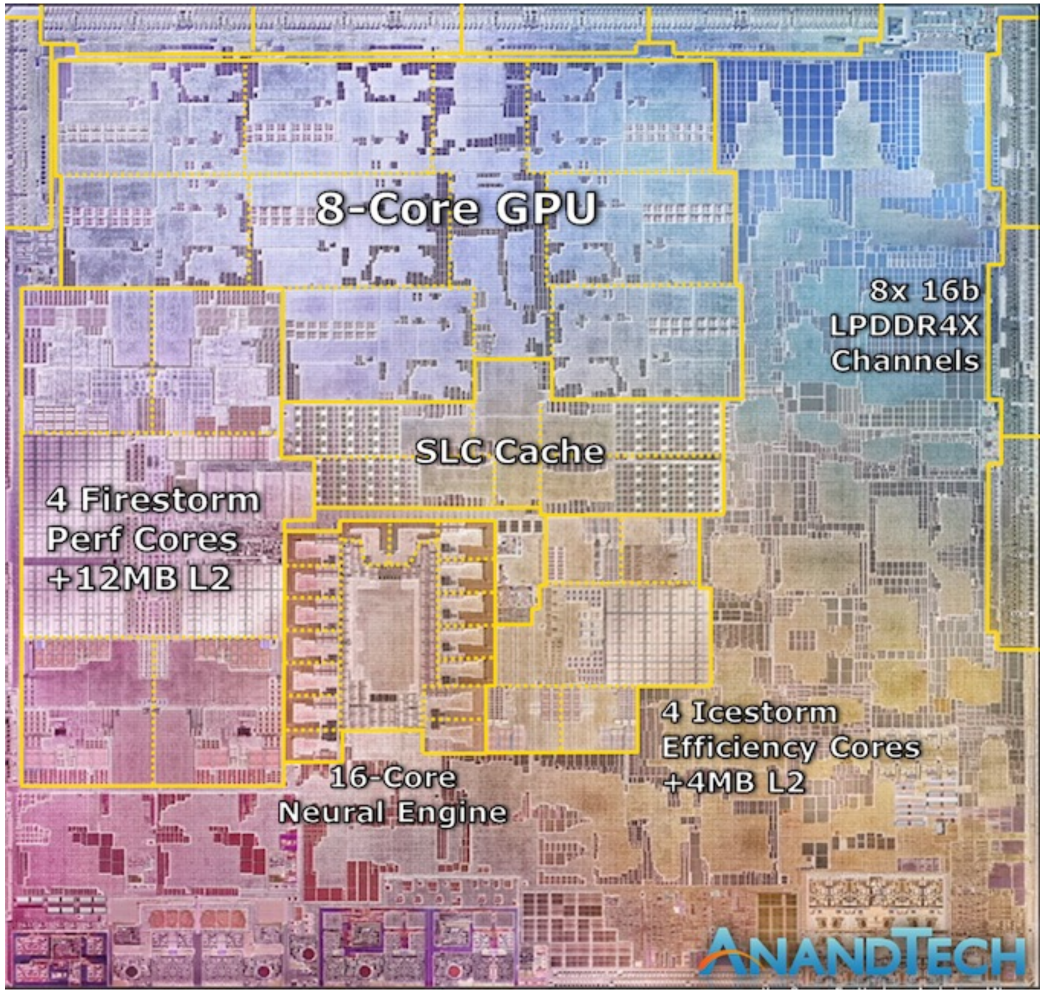


**Figure 4.3** The LC-3 as an example of the von Neumann model

## LC-3: A von Neumann Machine


third edition
introduction to computing systems
from bits & gates to C/C++ & beyond
yale n. patt
sanjay j. patel
Mc Graw Hill

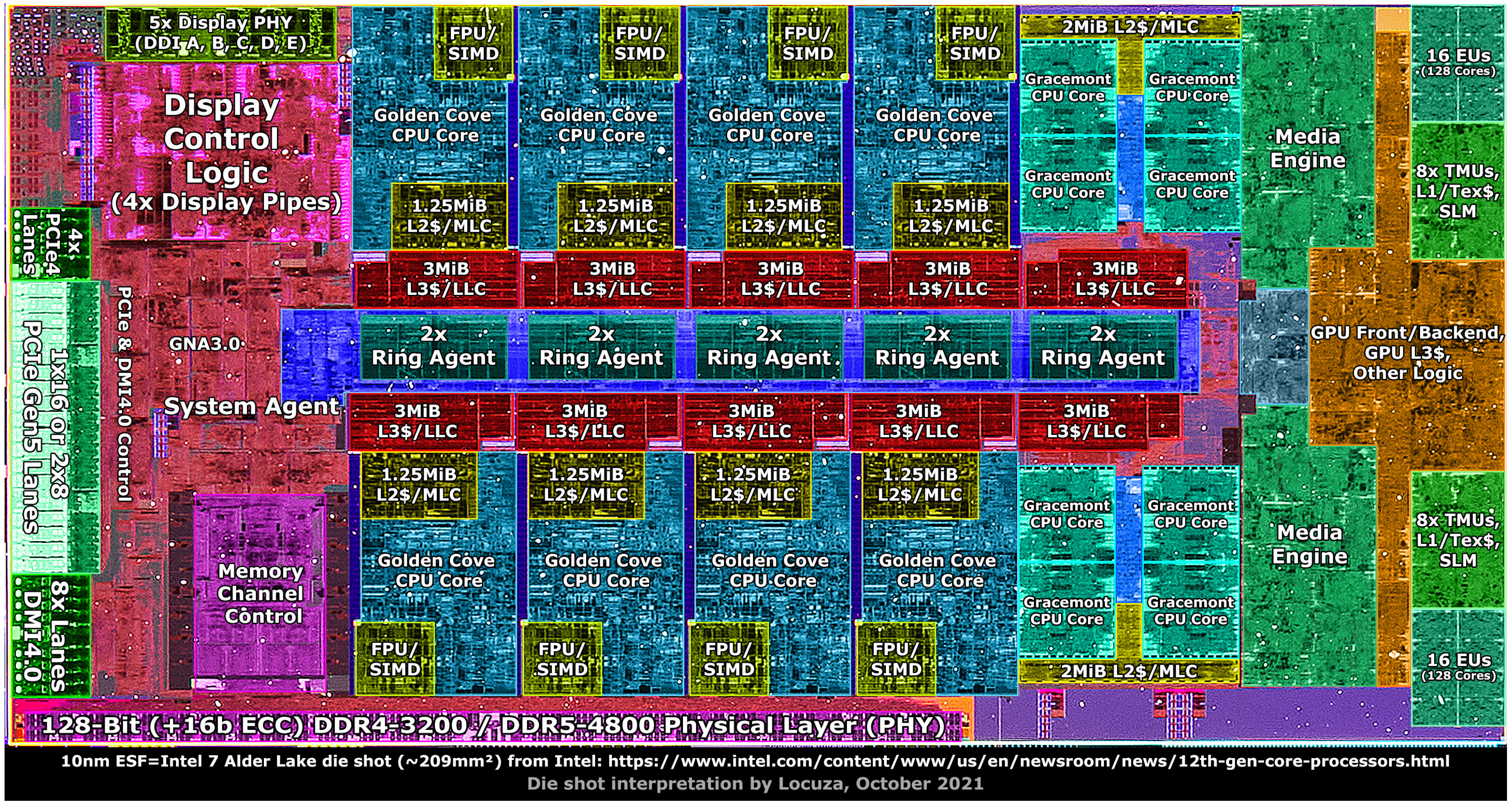
# Another Von Neumann Machine



Apple M1, 2021

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested

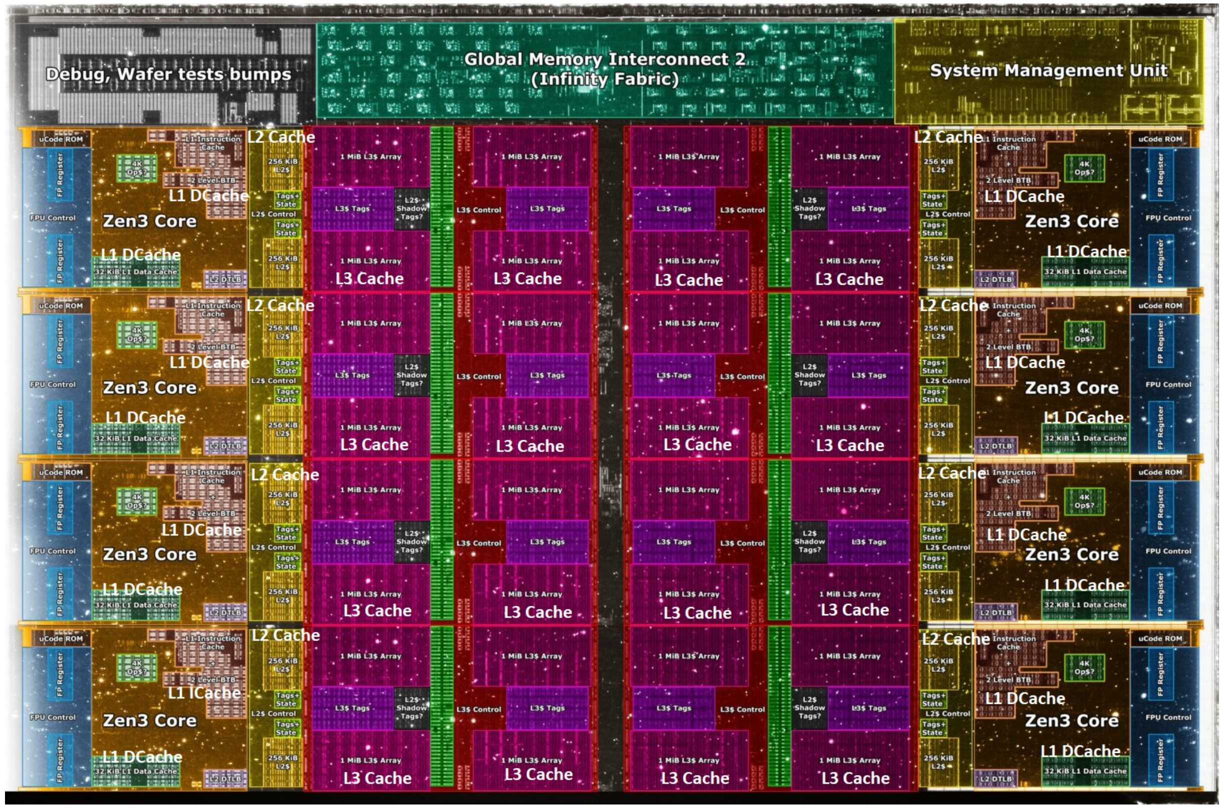
# Another Von Neumann Machine


5x Display PHY (DDI A, B, C, D, E)
Display Control Logic (4x Display Pipes)
4x PCIe4 Lanes
1x16 or 2x8 PCIe Gen5 Lanes
8x Lanes DMI4.0
PCIe & DMI4.0 Control
GNA3.0
System Agent
Memory Channel Control
FPU/ SIMD
Golden Cove CPU Core
1.25MiB L2$/MLC
3MiB L3$/LLC
2x Ring Agent
2MiB L2$/MLC
Gracemont CPU Core
Media Engine
16 EUs (128 Cores)
8x TMUs, L1/Tex$, SLM
GPU Front/Backend, GPU L3$, Other Logic
128-Bit (+16b ECC) DDR4-3200 / DDR5-4800 Physical Layer (PHY)
10nm ESF=Intel 7 Alder Lake die shot (~209mm²) from Intel: https://www.intel.com/content/www/us/en/newsroom/news/12th-gen-core-processors.html
Die shot interpretation by Locuza, October 2021

Intel Alder Lake, 2021

Source: https://twitter.com/Locuza_/status/1454152714930331652

## Another Von Neumann Machine



Core Count:
8 cores/16 threads

L1 Caches:
32 KB per core

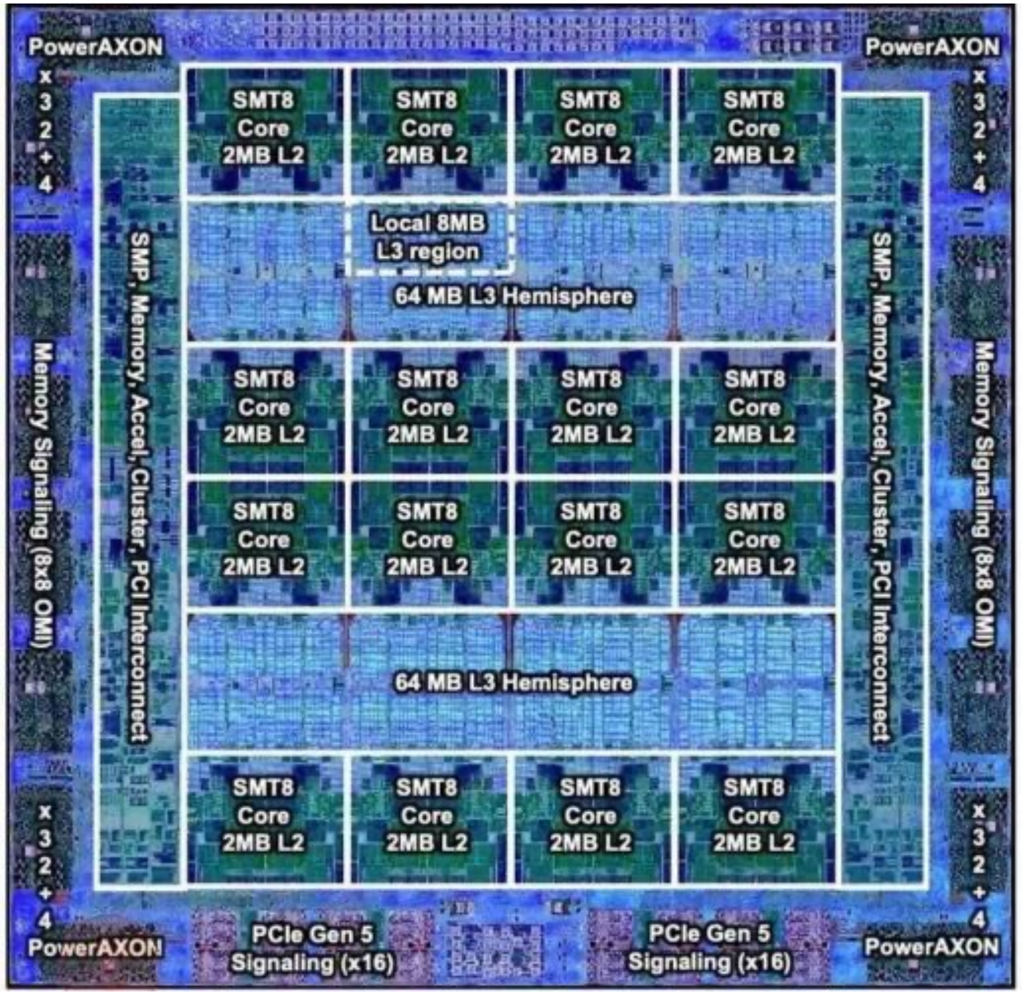
L2 Caches:
512 KB per core

L3 Cache:
32 MB shared

AMD Ryzen 5000, 2020

https://wccftech.com/amd-ryzen-5000-zen-3-vermeer-undressed-high-res-die-shots-close-ups-pictured-detailed/

# Another Von Neumann Machine


PowerAXON
PowerAXON
PowerAXON
PowerAXON
x 32 + 4
x 32 + 4
x 32 + 4
x 32 + 4
SMT8 Core 2MB L2
Local 8MB L3 region
64 MB L3 Hemisphere
64 MB L3 Hemisphere
SMP, Memory, Accel, Cluster, PCI Interconnect
SMP, Memory, Accel, Cluster, PCI Interconnect
Memory Signaling (8x8 OMI)
Memory Signaling (8x8 OMI)
PCIe Gen 5 Signaling (x16)
PCIe Gen 5 Signaling (x16)

IBM POWER10, 2020

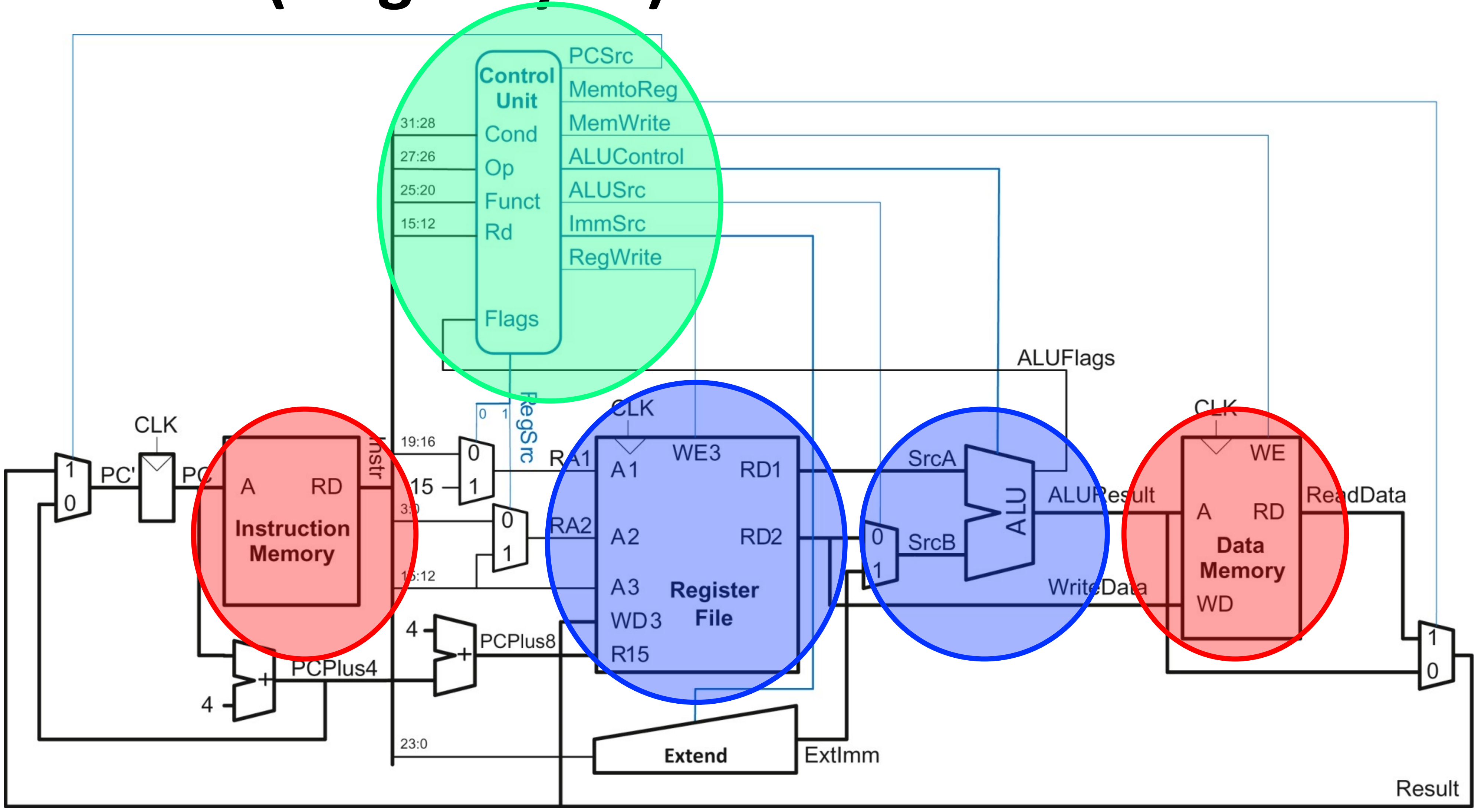
Cores:
15-16 cores,
8 threads/core

L2 Caches:
2 MB per core

L3 Cache:
120 MB shared

https://www.it-techblog.de/ibm-power10-prozessor-mehr-speicher-mehr-tempo-mehr-sicherheit/09/2020/

# ARMv4 (Single-Cycle) 32-bit



# ARMv4 (Multi-Cycle) 32-bit

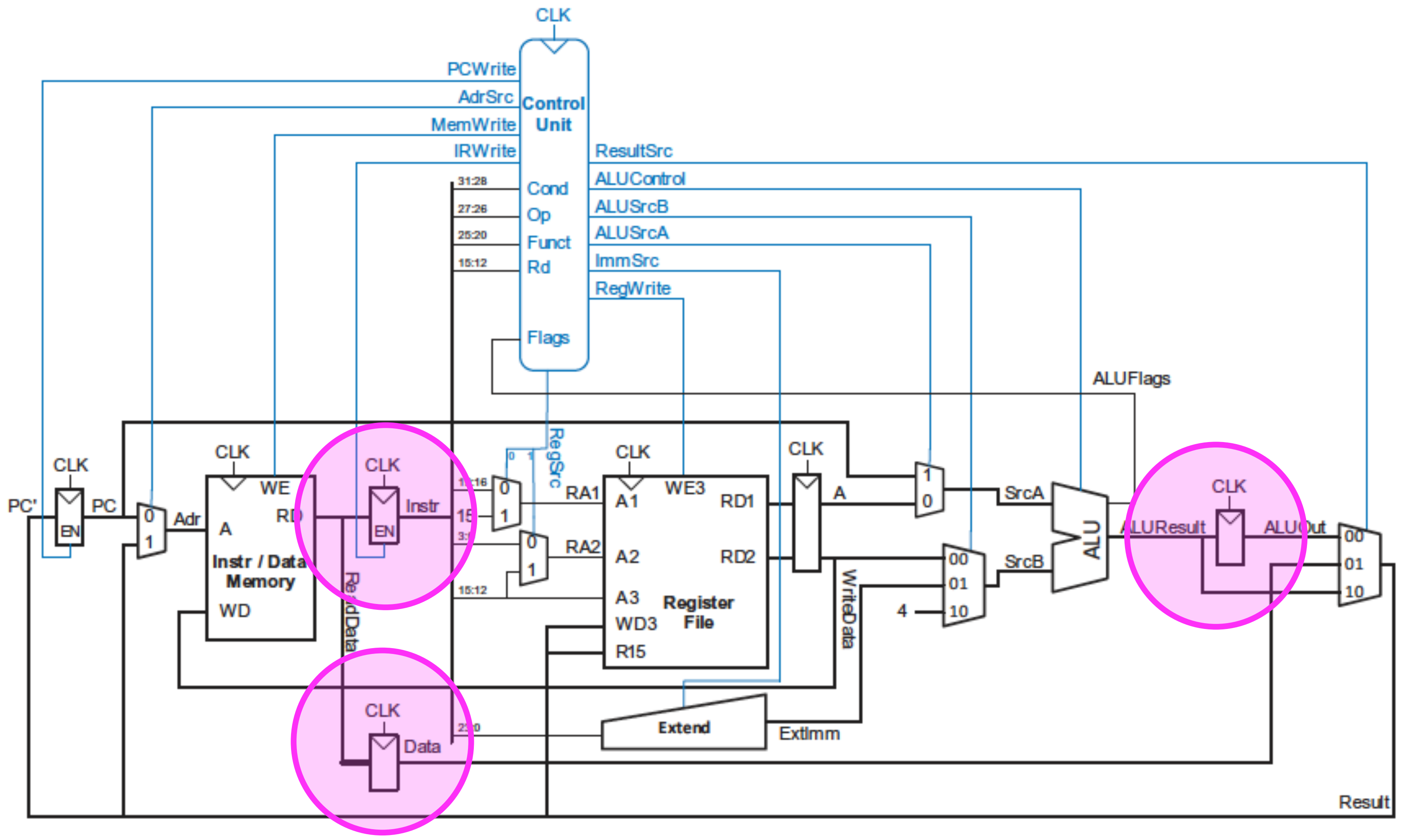
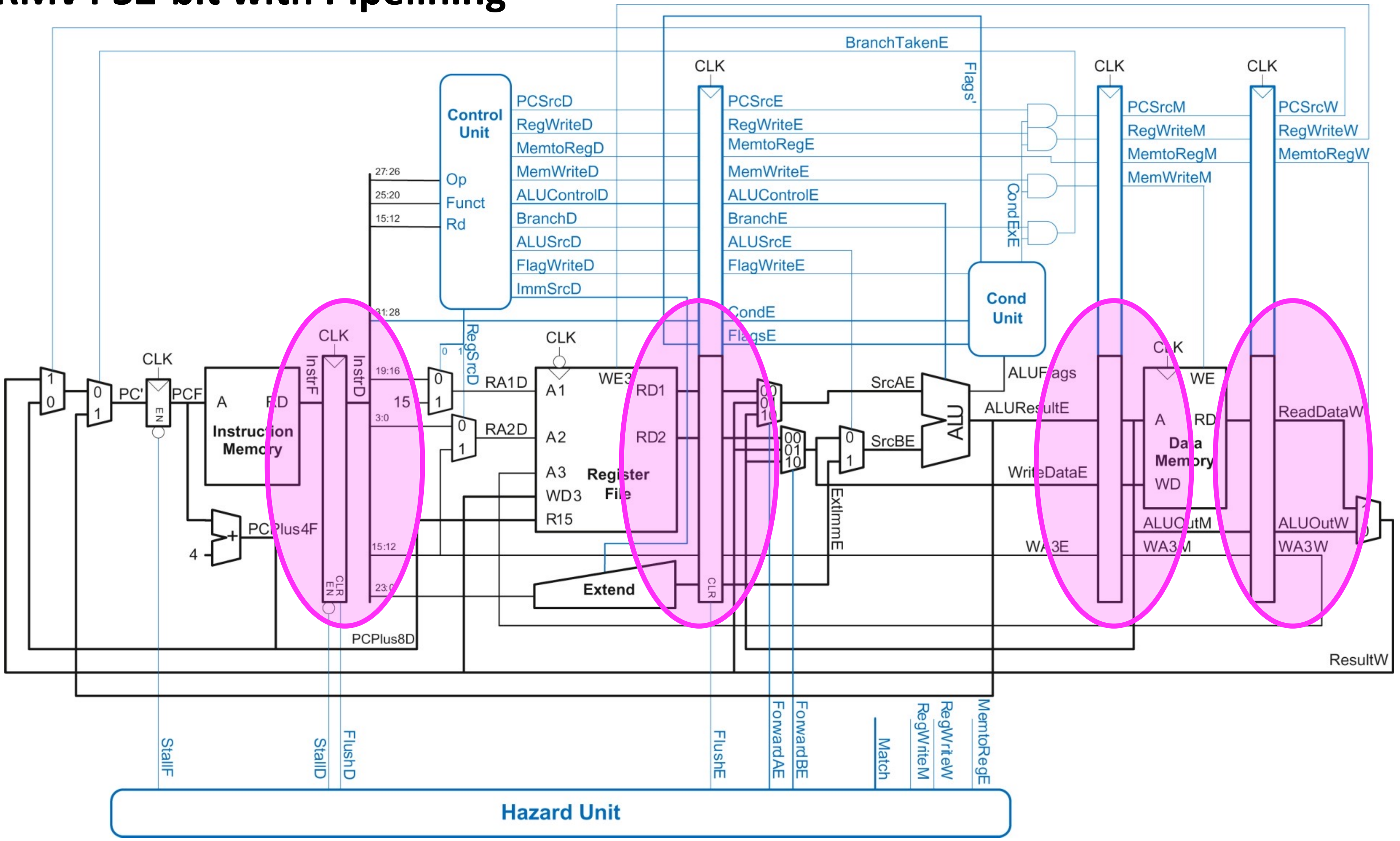
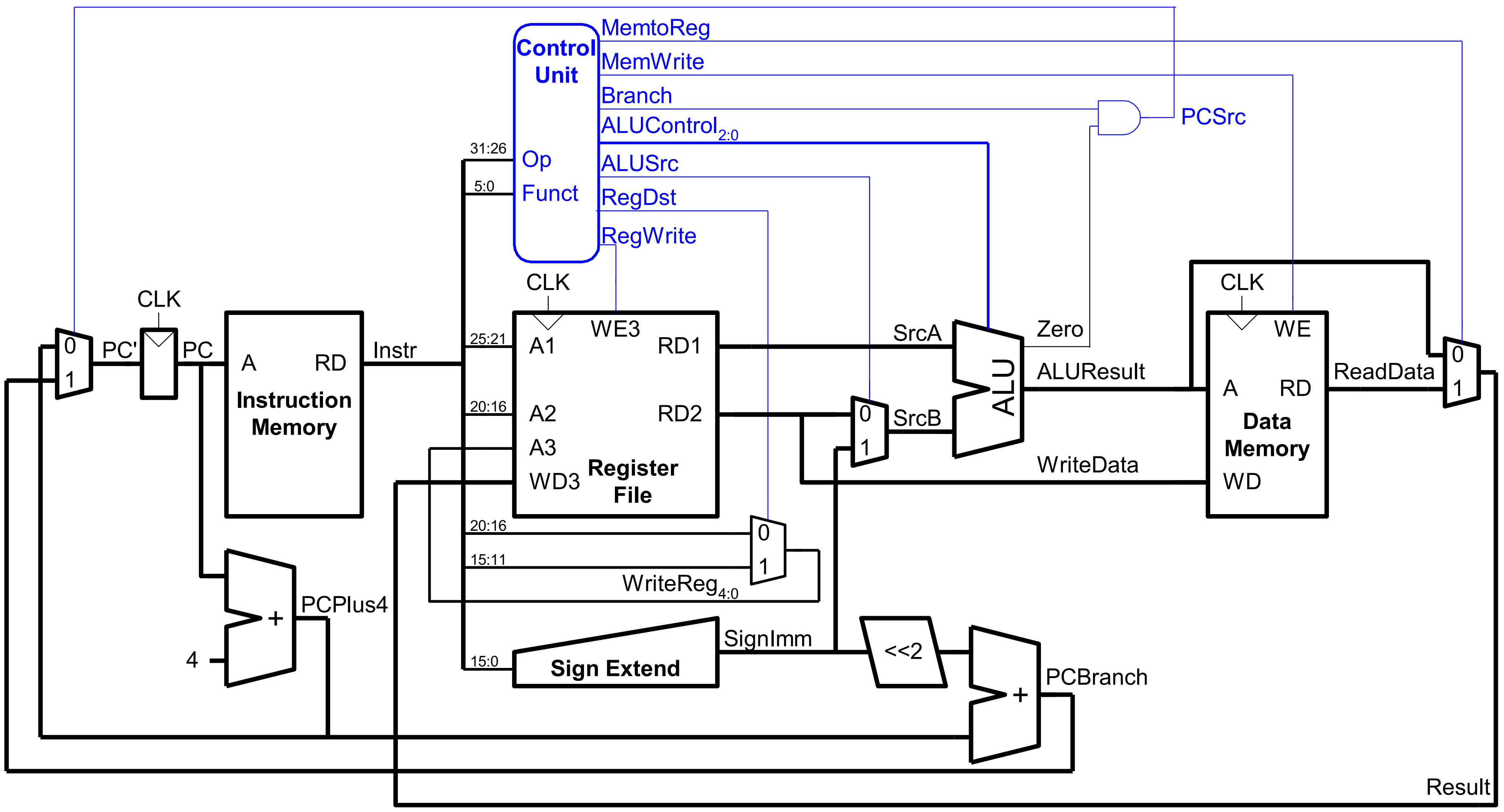


Figure 7.30 Complete multicycle processor

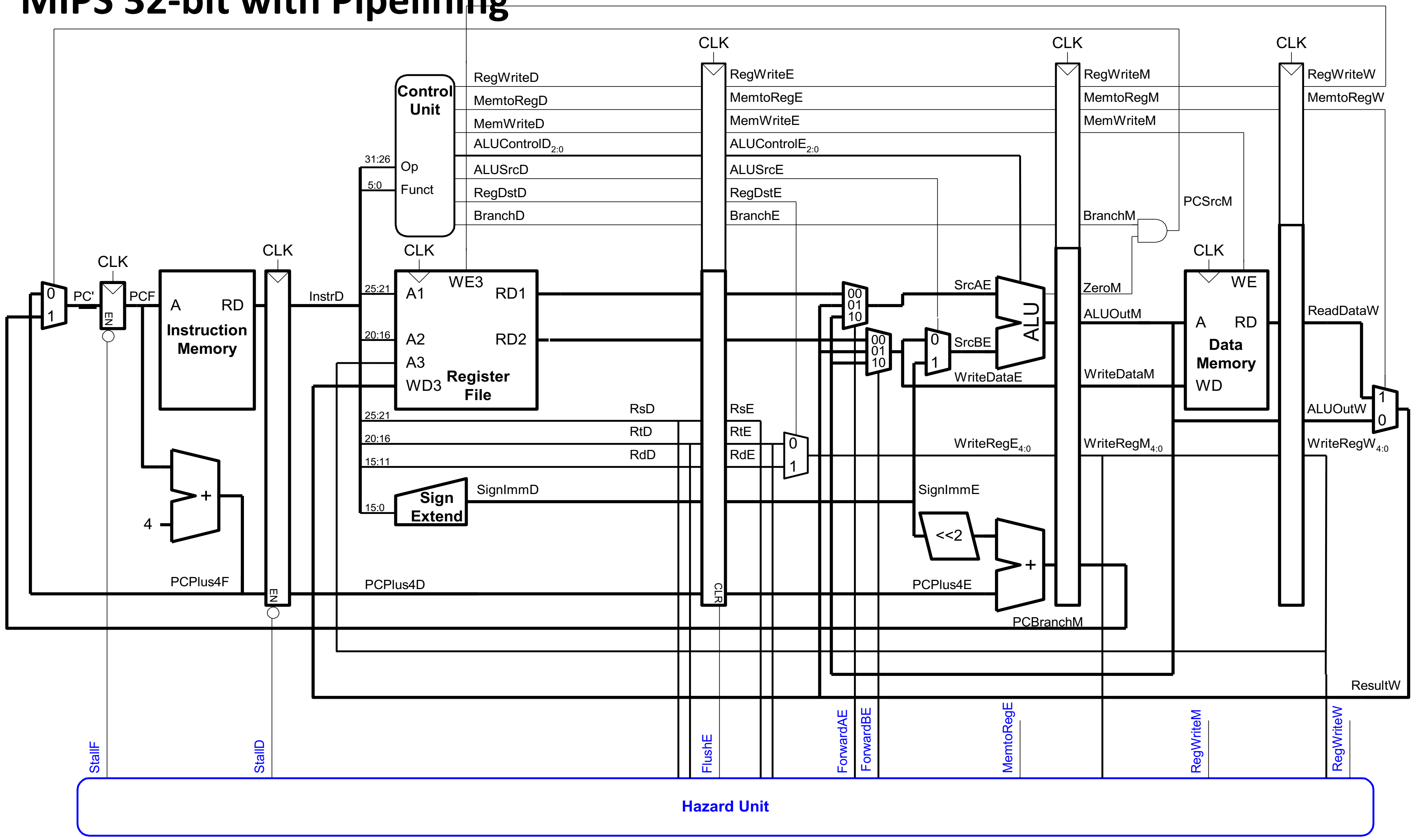
## ARMv4 32-bit with Pipelining



# MIPS (Single-Cycle) 32-bit



# MIPS 32-bit with Pipelining



## Key to Understanding Computers

- The key principles and fundamentals are the same
- Put your understanding of key principles to practice in labs
- **The exam/quiz is not structured to test your skills in memorizing slides!**

## The Concept of Sequential Execution

## Stored Program and Sequential Execution

- Instructions and data are stored in memory
  - Typically the instruction length is the word length
- The processor fetches instructions from memory sequentially
  - Fetches one instruction
  - Decodes and executes the instruction
  - Continues with the next instruction
- The address of the current instruction is stored in the program counter (PC)
  - If word-addressable memory, the processor increments the PC by 1 (in QuAC)
  - If byte-addressable memory, the processor increments the PC by the instruction length in bytes (4 in MIPS and ARM)
    - **Assume the OS sets the PC to 0x00400000 (start of a program)**

## A sample ARM program stored in memory

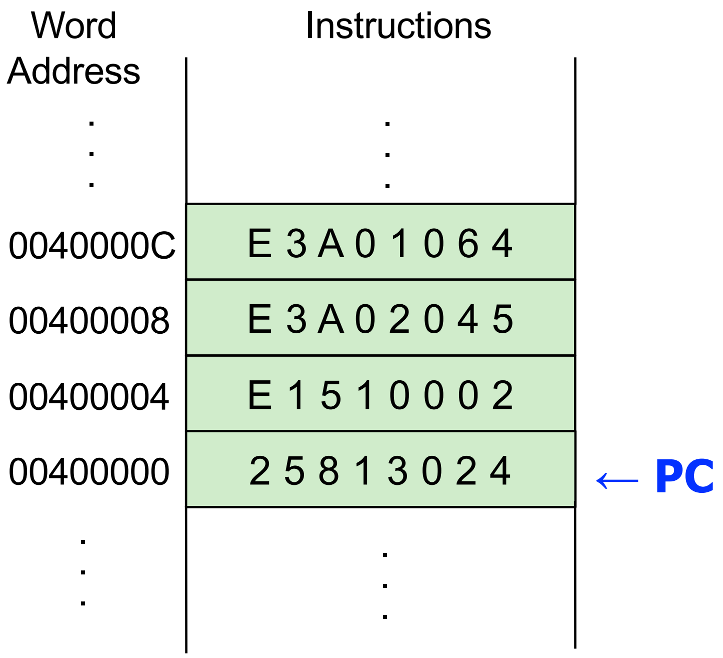
- A sample **ARM** program
  - 4 instructions stored in consecutive words in memory
    - No need to understand the program now. We will get back to it

ARM assembly code

```
MOV   R1,  #100
MOV   R2,  #69
CMP   R1,  R2
STRHS R3,  [R1, #0x24]
```

Machine code (encoded instructions)

```
0xE3A01064
0xE3A02045
0xE1510002
0x25813024
```



## A sample program: MIPS Example

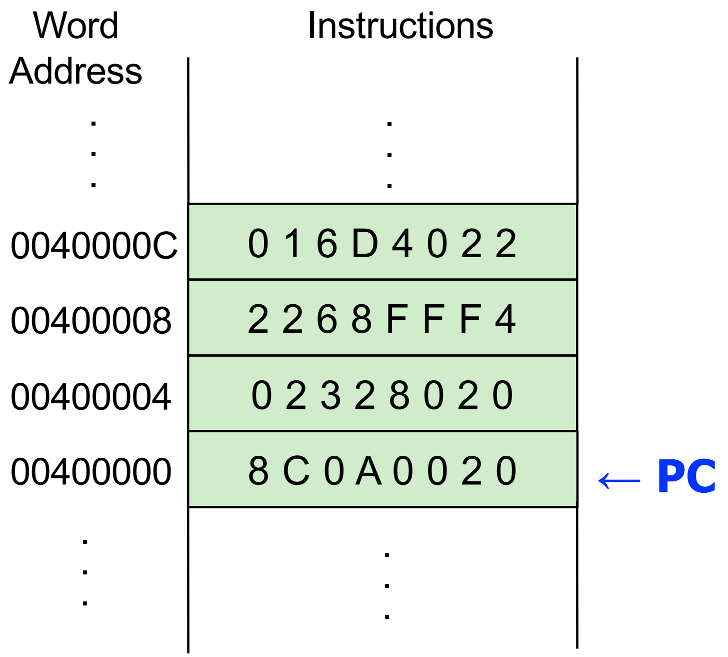
- A sample MIPS program
  - 4 instructions stored in consecutive words in memory
    - No need to understand the program now. We will get back to it

MIPS assembly

```
lw   $t2, 32($0)
add  $s0, $s1, $s2
addi $t0, $s3, -12
sub  $t0, $t3, $t5
```

Machine code (encoded instructions)

```
0x8C0A0020
0x02328020
0x2268FFF4
0x016D4022
```



## The Instruction

- An instruction is the most basic unit of computer processing
  - Instructions are words in the language of a computer
  - Instruction Set Architecture (ISA) is the vocabulary
- The language of the computer can be written as
  - Machine language: Computer-readable representation (that is, 0s and 1s)
  - Assembly language: Human-readable representation
- We will study ARM (in detail in lectures) and QuAC (in tutorials and assignment 1) and other ISAs for broader understanding
  - **Principles are similar** in all ISAs (x86, SPARC, RISC-V, …)

## The Instruction: Opcode & Operands

- An instruction is made up of two parts
  - Opcode and Operands

- Opcode specifies what the instruction does
- Operands specify who the instruction is to do it to

- Both are specified in instruction format (or instruction encoding)
  - A MIPS and ARM instructions consists of 32 bits (bits [31:0])
  - QuAC instructions consist of 16 bits (bits [15:0])

# The Instruction: Examples

- **MIPS** example: Bits [31:26] specify the opcode → up to 64 distinct opcodes
  - Bits [25:11] are used to figure out where the **operands** are

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

- **QuAC** example: Bits [15:12] specify the opcode → up to 16 distinct opcodes
  - Bits [10:0] are used to figure out where the **operands** are

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| op | | | | cond | rd | | | 0 | ra | | | 0 | rb | | |

- **ARM** example: Bits [27:26] specify the opcode → up to 4 distinct opcodes
  - Bits [19:0] are used to figure out where the **operands** are

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|---|---|
| cond | op | I | cmd | S | Rn | Rd | Src2 |

## Instruction Types

- There are three main types of instructions

- Operate (data processing) instructions
  - Execute operations in the ALU

- Data movement (memory) instructions
  - Read from or write to memory

- Control flow (branch/jump) instructions
  - Change the sequence of execution (decision making)

- Let us start with some example instructions

## An Example Operate Instruction

- Addition

High-level code

```
a = b + c;
```

QuAC Assembly

```
add a, b, c
```

- add: mnemonic to indicate the operation to perform
- b, c: source operands
- a: destination operand
- a ← b + c

# Registers

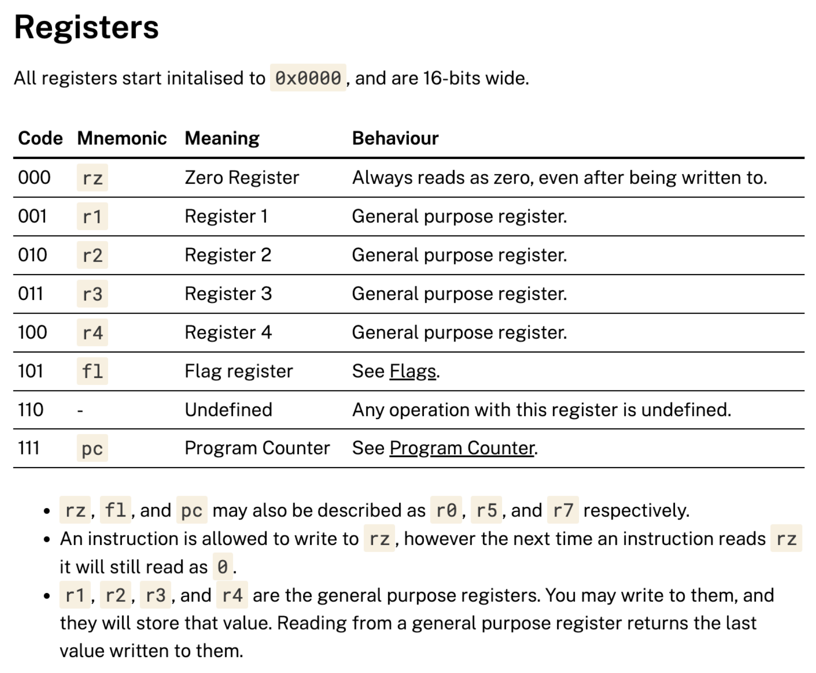
- We map variables to registers

Assembly

```
add a, b, c
```

ARM registers

```
b = R1
c = R2
a = R0
```



### Registers

All registers start initalised to `0x0000`, and are 16-bits wide.

| Code | Mnemonic | Meaning | Behaviour |
|---|---|---|---|
| 000 | `rz` | Zero Register | Always reads as zero, even after being written to. |
| 001 | `r1` | Register 1 | General purpose register. |
| 010 | `r2` | Register 2 | General purpose register. |
| 011 | `r3` | Register 3 | General purpose register. |
| 100 | `r4` | Register 4 | General purpose register. |
| 101 | `fl` | Flag register | See Flags. |
| 110 | - | Undefined | Any operation with this register is undefined. |
| 111 | `pc` | Program Counter | See Program Counter. |

- `rz`, `fl`, and `pc` may also be described as `r0`, `r5`, and `r7` respectively.
- An instruction is allowed to write to `rz`, however the next time an instruction reads `rz` it will still read as `0`.
- `r1`, `r2`, `r3`, and `r4` are the general purpose registers. You may write to them, and they will store that value. Reading from a general purpose register returns the last value written to them.

**QuAC registers**

```
b = r1
c = r2
a = r0
```

MIPS registers

```
b = $s1
c = $s2
a = $s0
```

# From Assembly to QuAC Machine Code

- Addition

**QuAC assembly**

```
add  r0, r1, r2
```

- Instruction Fields

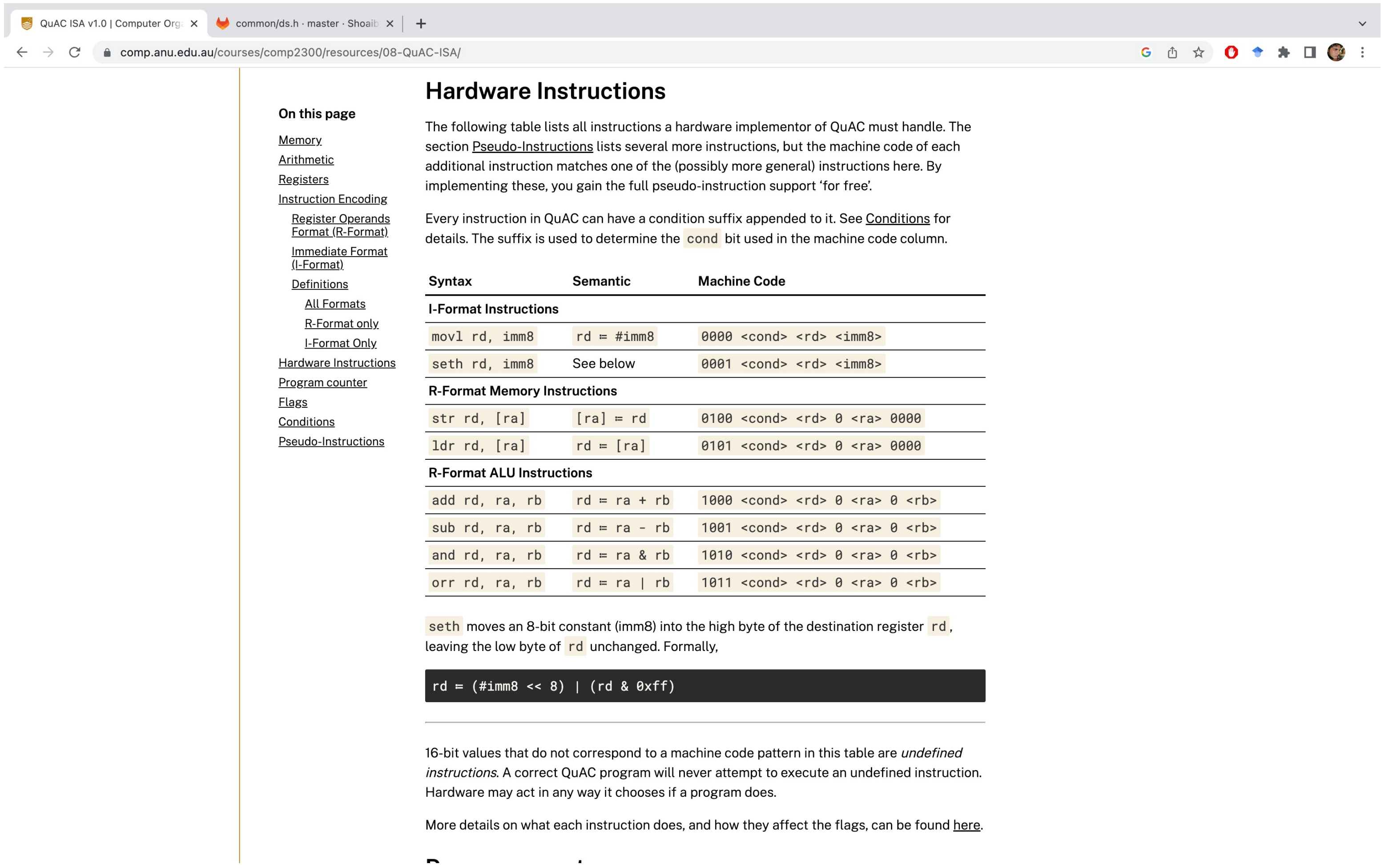
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| op 8 | | | | cond 0 | rd 0 | | | 0 | ra 1 | | | 0 | rb 2 | | |

- Machine code (Instruction Encoding)

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

- Machine code in short (hexadecimal)
  - 0x 8 0 1 2

# QuAC Opcodes


QuAC ISA v1.0 | Computer Org
common/ds.h · master · Shoaib
comp.anu.edu.au/courses/comp2300/resources/08-QuAC-ISA/

**On this page**

## Hardware Instructions

The following table lists all instructions a hardware implementor of QuAC must handle. The section Pseudo-Instructions lists several more instructions, but the machine code of each additional instruction matches one of the (possibly more general) instructions here. By implementing these, you gain the full pseudo-instruction support 'for free'.

Every instruction in QuAC can have a condition suffix appended to it. See Conditions for details. The suffix is used to determine the `cond` bit used in the machine code column.

| Syntax | Semantic | Machine Code |
|---|---|---|
| **I-Format Instructions** | | |
| `movl rd, imm8` | `rd = #imm8` | `0000 <cond> <rd> <imm8>` |
| `seth rd, imm8` | See below | `0001 <cond> <rd> <imm8>` |
| **R-Format Memory Instructions** | | |
| `str rd, [ra]` | `[ra] = rd` | `0100 <cond> <rd> 0 <ra> 0000` |
| `ldr rd, [ra]` | `rd = [ra]` | `0101 <cond> <rd> 0 <ra> 0000` |
| **R-Format ALU Instructions** | | |
| `add rd, ra, rb` | `rd = ra + rb` | `1000 <cond> <rd> 0 <ra> 0 <rb>` |
| `sub rd, ra, rb` | `rd = ra - rb` | `1001 <cond> <rd> 0 <ra> 0 <rb>` |
| `and rd, ra, rb` | `rd = ra & rb` | `1010 <cond> <rd> 0 <ra> 0 <rb>` |
| `orr rd, ra, rb` | `rd = ra \| rb` | `1011 <cond> <rd> 0 <ra> 0 <rb>` |

`seth` moves an 8-bit constant (imm8) into the high byte of the destination register `rd`, leaving the low byte of `rd` unchanged. Formally,

```
rd = (#imm8 << 8) | (rd & 0xff)
```

16-bit values that do not correspond to a machine code pattern in this table are *undefined instructions*. A correct QuAC program will never attempt to execute an undefined instruction. Hardware may act in any way it chooses if a program does.

More details on what each instruction does, and how they affect the flags, can be found here.

# From Assembly to ARM Machine Code

- Addition

**ARM assembly**

```
ADD  R0, R1, R2
```

- Instruction Fields

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|---|---|
| cond | op | I | cmd | S | Rn | Rd | Src2 |

- Machine Code (Instruction Encoding)

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|---|---|
| 1110 | 00 | 0 | 0100 | 1 | 0001 | 0000 | 000000000010 |

- Machine Code in short (hexadecimal)
  - 0x E 0 9 1 0 0 0 1

## Instruction Format

- A form of representation of an instruction composed of **fields** of binary numbers (we have seen already)
- **It is the layout of the instruction**
- The instruction is divided into segments, and each segment is called a **field**
- An ISA defines a few classes or types of formats, and each class or type has many different instructions for that type

# QuAC Instruction Formats

**info**

There's nothing enforcing future instructions fall into these two formats: R-Format and I-Format only *describe* the general pattern existing instructions follow. New instructions could follow an entirely different encoding format.

### Register Operands Format (R-Format)

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `op` | | | | `cond` | `rd` | | | 0 | `ra` | | | 0 | `rb` | | |

### Immediate Format (I-Format)

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `op` | | | | `cond` | `rd` | | | `imm8` | | | | | | | |

| Syntax | Semantic | Machine Code |
|---|---|---|
| **I-Format Instructions** | | |
| `movl rd, imm8` | `rd = #imm8` | `0000 <cond> <rd> <imm8>` |
| `seth rd, imm8` | See below | `0001 <cond> <rd> <imm8>` |
| **R-Format Memory Instructions** | | |
| `str rd, [ra]` | `[ra] = rd` | `0100 <cond> <rd> 0 <ra> 0000` |
| `ldr rd, [ra]` | `rd = [ra]` | `0101 <cond> <rd> 0 <ra> 0000` |
| **R-Format ALU Instructions** | | |
| `add rd, ra, rb` | `rd = ra + rb` | `1000 <cond> <rd> 0 <ra> 0 <rb>` |
| `sub rd, ra, rb` | `rd = ra - rb` | `1001 <cond> <rd> 0 <ra> 0 <rb>` |
| `and rd, ra, rb` | `rd = ra & rb` | `1010 <cond> <rd> 0 <ra> 0 <rb>` |
| `orr rd, ra, rb` | `rd = ra \| rb` | `1011 <cond> <rd> 0 <ra> 0 <rb>` |

# MIPS Instruction Formats

- Only three formats for simplicity of implementation
- One can see the **consistency** across formats

R (Register) Format:

| | | | | | |
|---|---|---|---|---|---|
| Opcode (6) | Rs (5) | Rt (5) | Rd (5) | Shamt (5) | Funct (6) |

Most arithmetic and logic instructions (except 'immediate')

I (Immediate) Format:

| | | | |
|---|---|---|---|
| Opcode (6) | Rs (5) | Rt (5) | 16-bit Immediate value (16) |

Data Transfer, Immediate, and Cond. Branch instructions

J (Jump) Format:

| | |
|---|---|
| Opcode (6) | 26-bit word address (26) |

Unconditional Jump instructions

- MIPS ISA is outside of scope and only shown for breadth

# Instruction Format: R Type in MIPS

- MIPS R-type Instruction Format (R = Register)
  - 3 register operands (register-based ALU operations)

| 0 | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

  - op = opcode = 0
  - rs, rt = source registers
  - rd = destination register
  - shamt = shift amount (only shift operations)
  - funct = operation in R-type instructions

| Name | Register Number | Usage |
|---|---|---|
| `$0` | 0 | the constant value 0 |
| `$at` | 1 | assembler temporary |
| `$v0-$v1` | 2-3 | function return value |
| `$a0-$a3` | 4-7 | function arguments |
| `$t0-$t7` | 8-15 | temporary variables |
| `$s0-$s7` | 16-23 | saved variables |
| `$t8-$t9` | 24-25 | temporary variables |
| `$k0-$k1` | 26-27 | OS temporaries |
| `$gp` | 28 | global pointer |
| `$sp` | 29 | stack pointer |
| `$fp` | 30 | frame pointer |
| `$ra` | 31 | function return address |

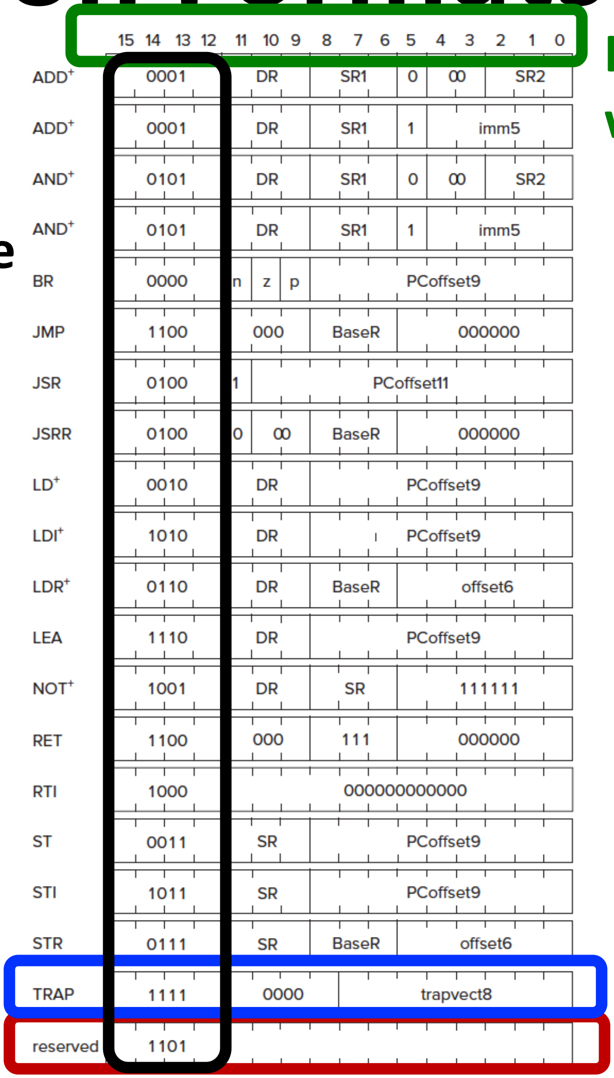
## Instruction Format: Data Processing (DP) in ARM

$$\begin{tikzcd}[row sep=large]
\text{ADD Rd,} \arrow[d] & \text{Rn} \arrow[d] & \text{Rm} \arrow[d] \\
\text{ADD R0,} & \text{R1,} & \text{R3}
\end{tikzcd}$$

- Rn and Rm are source registers and Rd is the destination register
- **Below is the instruction format (encoding)**
- op = opcode (what does the instruction do?)
  - 00 means operate instruction and **cmd = 0100** means **ADD**
  - Some bits are pre-set (**details later**)

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:4 | | | | | | | | 3:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1110 | op | 0 | cmd | S | Rn | Rd | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rm |

# LC-3 Instruction Formats



| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD$^+$ | 0001 | | | | DR | | | SR1 | | | 0 | 00 | | SR2 | | |
| ADD$^+$ | 0001 | | | | DR | | | SR1 | | | 1 | imm5 | | | | |
| AND$^+$ | 0101 | | | | DR | | | SR1 | | | 0 | 00 | | SR2 | | |
| AND$^+$ | 0101 | | | | DR | | | SR1 | | | 1 | imm5 | | | | |
| BR | 0000 | | | | n | z | p | PCoffset9 | | | | | | | | |
| JMP | 1100 | | | | 000 | | | BaseR | | | 000000 | | | | | |
| JSR | 0100 | | | | 1 | PCoffset11 | | | | | | | | | | |
| JSRR | 0100 | | | | 0 | 00 | | BaseR | | | 000000 | | | | | |
| LD$^+$ | 0010 | | | | DR | | | PCoffset9 | | | | | | | | |
| LDI$^+$ | 1010 | | | | DR | | | PCoffset9 | | | | | | | | |
| LDR$^+$ | 0110 | | | | DR | | | BaseR | | | offset6 | | | | | |
| LEA | 1110 | | | | DR | | | PCoffset9 | | | | | | | | |
| NOT$^+$ | 1001 | | | | DR | | | SR | | | 111111 | | | | | |
| RET | 1100 | | | | 000 | | | 111 | | | 000000 | | | | | |
| RTI | 1000 | | | | 000000000000 | | | | | | | | | | | |
| ST | 0011 | | | | SR | | | PCoffset9 | | | | | | | | |
| STI | 1011 | | | | SR | | | PCoffset9 | | | | | | | | |
| STR | 0111 | | | | SR | | | BaseR | | | offset6 | | | | | |
| TRAP | 1111 | | | | 0000 | | | | trapvect8 | | | | | | | |
| reserved | 1101 | | | | | | | | | | | | | | | |

Instructions are 16-bit words

opcode is in the same place for each instruction

Such "weird" instructions will make more sense in COMP2310 as they provide support for I/O and networking

Reserved for future use

## Read Operands from Memory

- With operate instructions, such as addition, we tell the computer to execute arithmetic (or logic) computations in the ALU
- We also need instructions to access the operands from memory
  - Load them from memory to registers
  - Store them from registers to memory
- Next, we see how to read (or load) from memory
- Writing (or storing) is performed in a similar way, but we will talk about that later

## Reading Byte-Addressable Memory

- ARM assembly (Load Register or LDR)

High-level code

```
a = A[2];
```

ARM assembly

```
LDR  R3, [R0, #8]
```

$R3 \leftarrow \text{Memory}[R0 + 8]$

- MIPS assembly (load word or lw)

High-level code

```
a = A[2];
```

MIPS assembly

```
lw   $s3, 8($s0)
```

$\$s3 \leftarrow \text{Memory}[\$s0 + 8]$

These instructions use a particular addressing mode
(i.e., the way the address is calculated), called base+offset

## Load Word in MIPS and ARM

- ARM assembly

```
LDR  R3, [R0, #8]
```

$R3 \leftarrow$ Memory[R0 + 8]

- MIPS assembly

```
lw   $s3, 8($s0)
```

\$s3 $\leftarrow$ Memory[\$s0 + 8]

- Byte address is calculated as: word_address * bytes/word
  - 4 bytes/word in MIPS and ARM
  - If QuAC were byte-addressable (i.e., QuAC v3), 2 bytes/word

# Load Word in Word-Addressable LC-3

- LC-3 assembly (Load Register or LDR)

High-level code

```
a = A[2];
```

LC-3 assembly

```
LDR  R3, [R0, #2]
```

R3 ← Memory[R0 + 2]

- Each word in LC-3 is 16 bits
- Therefore, We interrogate memory with word addresses (not byte addresses)
- If LC-3 were byte-addressable, the offset would be 4

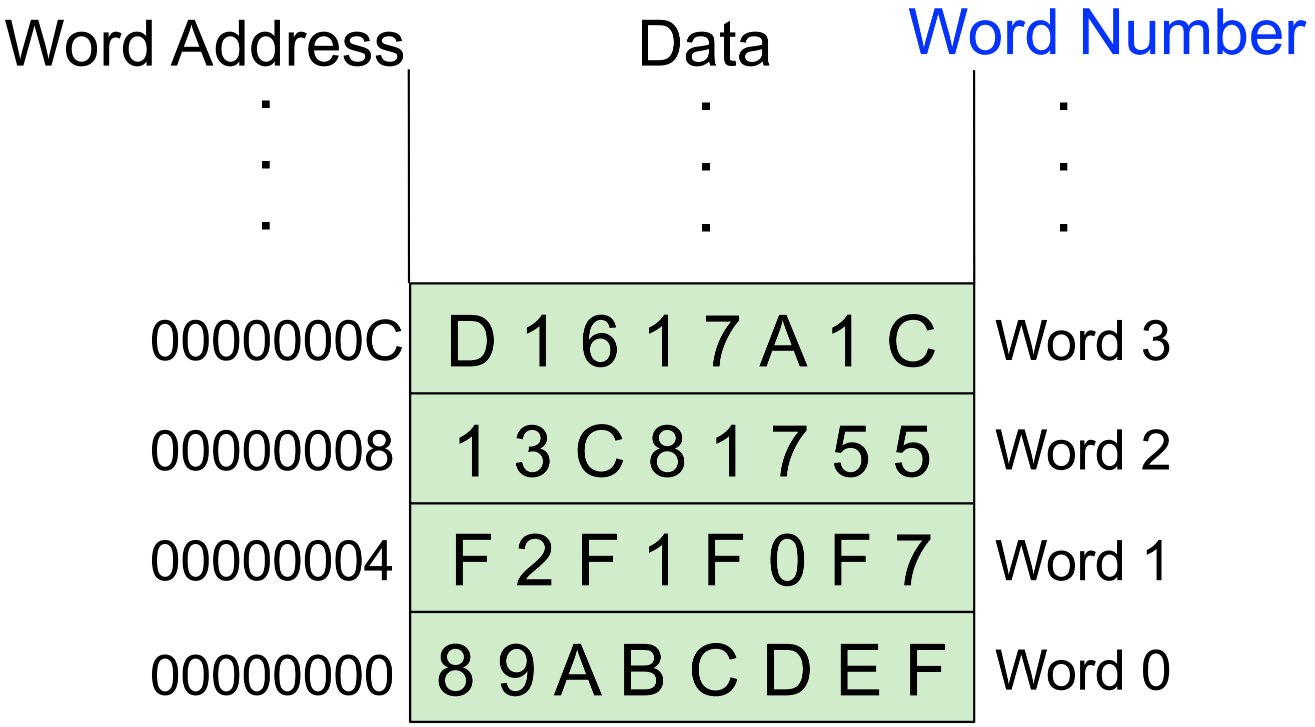
# Hypothetical 32-bit QuAC Memory

- If QuAC were 32-bit architecture, let's look at its memory view
  - Word-addressable QuAC
  - We use word numbers to address memory

| Word Address | Data | Word Number |
|---|---|---|
| ⋮ | ⋮ | ⋮ |
| 00000003 | D 1 6 1 7 A 1 C | Word 3 |
| 00000002 | 1 3 C 8 1 7 5 5 | Word 2 |
| 00000001 | F 2 F 1 F 0 F 7 | Word 1 |
| 00000000 | 8 9 A B C D E F | Word 0 |

# Hypothetical 32-bit QuAC Memory

- If QuAC were 32-bit architecture, let's look at its memory view
  - **Byte**-addressable QuAC
  - We use word numbers translated to byte addresses to read memory



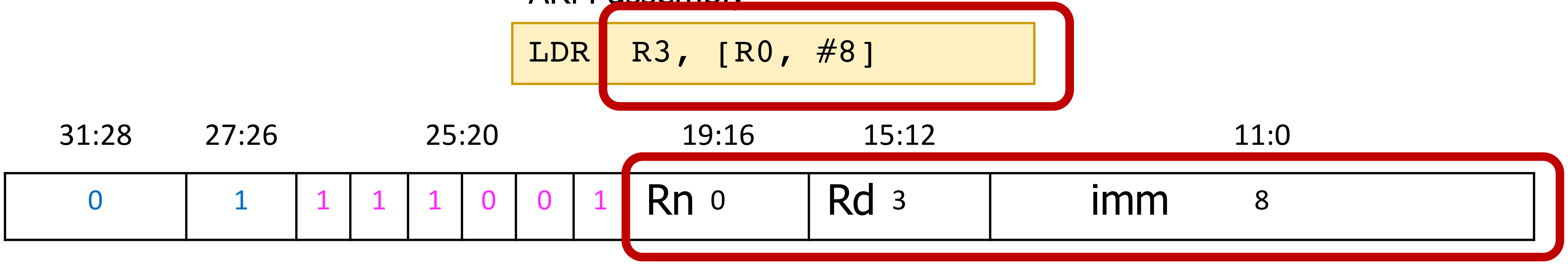
| Word Address | Data | Word Number |
|---|---|---|
| . . . | . . . | . . . |
| 0000000C | D 1 6 1 7 A 1 C | Word 3 |
| 00000008 | 1 3 C 8 1 7 5 5 | Word 2 |
| 00000004 | F 2 F 1 F 0 F 7 | Word 1 |
| 00000000 | 8 9 A B C D E F | Word 0 |

# Another Instruction Encoding

- ARM

ARM assembly

```
LDR  R3, [R0, #8]
```



- MIPS

MIPS assembly

```
lw  $s3, 8($s0)
```

Field Values

| op | rs | rt | imm |
|---|---|---|---|
| 35 | 16 | 19 | 8 |

This encoding has space for immediate values such as offsets.

## The Instruction Set

- It defines opcodes, operands, data types, and addressing modes

- Addressing mode = Formulas for figuring out operands
  - Register, Immediate, Base + Offset

- The datatype is the representation of the operands in 0s and 1s

- **ADD** and **LDR** in ARM assembly have been our first examples

## `ADD R0, R1, R2`

- What is the instruction mnemonic and opcode?
  - `ADD` (opcode = 0001 for LC-3)
- What is the addressing mode?
  - register mode
- What is the data type?
  - 2’s complement integer
- What does the instruction do?
  - The instruction **directs** the computer to perform a 2’s complement integer addition and specifies the locations (GPRs) where the computer can find **source operands** and the location of a GPR where the computer is to write the **result**

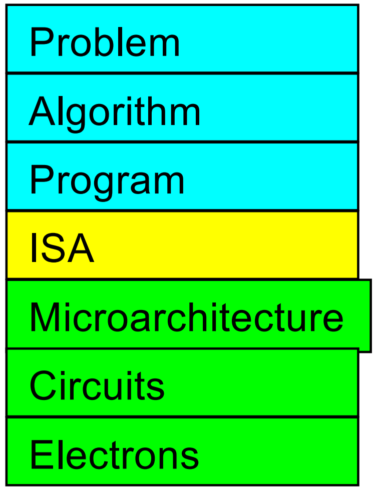
## `LDR R3, [R0, #8]`

- What is the opcode?
  - `LDR` (0110 for LC-3)
- What is the addressing mode?
  - base + offset (we will study in detail later)
- What is the data type?
  - bit vector
- What does the instruction do?
  - The instruction directs the computer to load a destination register with the contents of a memory location, where the location can be calculated using a formula: add the contents of a GPR (R8) to a constant number (#8)

# The Instruction Set Architecture

- The ISA is the interface between what the software commands and what the hardware carries out

- The ISA specifies
  - The memory organization
    - Address space (ARM: $2^{32}$, MIPS: $2^{32}$)
    - Addressability (ARM: 8 bits, MIPS: 8 bits, QuAC: 16 bits)
    - Word- or Byte-addressable
  - The register set
    - R0 to R15 in ARM
    - 32 registers in MIPS
  - The instruction set
    - Opcodes
    - Operands
    - Addressing modes
    - Length and format of instructions



## Two Questions

- What state of the computer is visible (or exposed to) the programmer?
  - What state can they manipulate by writing machine code?
  - **Answer:** The **Architectural** State
    - General-purpose registers, memory, program counter

- What does the ISA specify?
  - The memory organization

  - The register set

  - The instruction set

- **Meta-point:** Architectural state is part of the ISA specification

## Instruction (Processing) Cycle

## How are these instructions executed?

- By using instructions, we can speak the language of the computer

- Thus, we now know how to tell the computer to
  - Execute computations in the ALU by using, for instance, an addition
  - Access operands from memory by using the load word instruction

- But, how are these instructions executed on the computer?
  - The process of executing an instruction is called is the instruction cycle (or, instruction processing cycle)

## The Instruction Cycle

- The instruction cycle is a sequence of steps or phases, that an instruction goes through to be executed
  - FETCH
  - DECODE
  - EVALUATE ADDRESS
  - FETCH OPERANDS
  - EXECUTE
  - STORE RESULT

- Not all instructions require the six phases
  - LDR does **not** require EXECUTE
  - ADD does **not** require EVALUATE ADDRESS
  - Intel x86 instruction ADD [eax], edx is an example of instruction with six phases

## LC-3 Assembly

- We will use **LC-3** (**L**ittle **C**omputer v.3) architecture as  example

- ADD Operate instruction

```
ADD  R0, R1, R2
```

- Instruction for accessing memory

High-level code

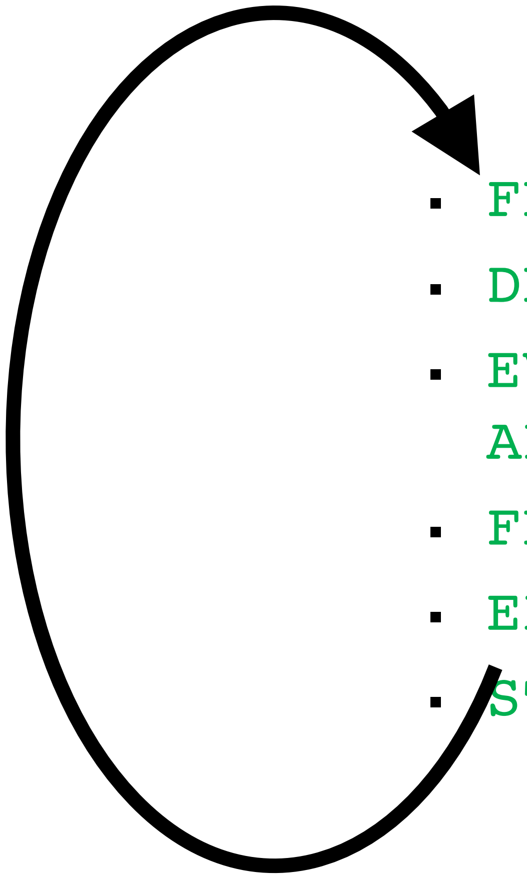
```
a = A[2];
```

LC-3 assembly

```
LDR  R3, R0, #4
```

R3 $\leftarrow$ Memory[R0 + 4]

# After STORE RESULT, a NEW FETCH



- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT

## Instruction (Processing) Cycle

## FETCH

- The FETCH phase obtains the instruction from memory and loads it into the Instruction Register (IR)

- This phase is common to every instruction type

- Complete description
  - Step 1: Load the MAR with the contents of the PC, and simultaneously increment the PC

  - Step 2: Interrogate memory. This results in the instruction being placed in the MDR by memory

  - Step 3: Load the IR with the contents of the MDR

## Machine Cycle

- Each of these steps is under the direction of the **control unit**

- Each step takes one machine cycle
  - Each machine cycle takes one clock cycle (the two are the same)

- Each instruction cycle consists of many machine cycles
  - If each instruction cycle takes one machine cycle, such a simple machine is called a **single-cycle** computer or microarchitecture
  - Single-cycle machines are much simpler to build that what we are discussing here (e.g., the control unit is not an FSM)

## Machine Cycle

- A clock cycle is a small fraction of a second

- 1 GHz Intel CPU completes 1 billion clock cycles in one second
  - One clock cycle takes one billionths of a second
  - Or **1 nanoseconds** (**ns**)

- In **one second, the computer can perform 1 billion machine cycles** where each machine cycle executes an instruction (or part of an instruction)

# FETCH in LC-3

Step 1: Load MAR and increment PC

Step 2: Access memory

Step 3: Load IR with the content of MDR

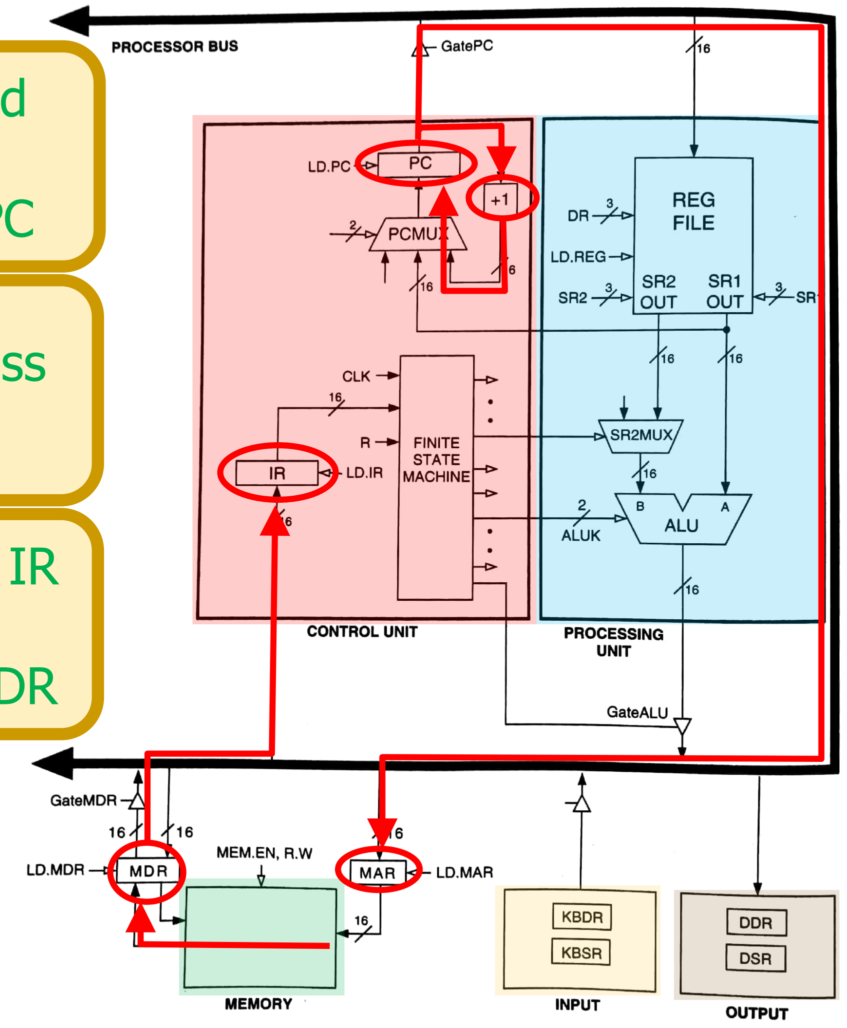


**Figure 4.3** The LC-3 as an example of the von Neumann model

## DECODE

- The DECODE phase identifies the instruction
  - Also generates the set of control signals to process the identified instruction in later phases of the instruction cycle
- Recall the decoder
  - A 4-to-16 decoder identifies which of the 16 opcodes is going to be processed
- The input is the four bits IR[15:12]
- The remaining 12 bits identify what else is needed to process the instruction

# DECODE in LC-3

DECODE identifies the instruction to be processed

Also generates the set of control signals to process the instruction

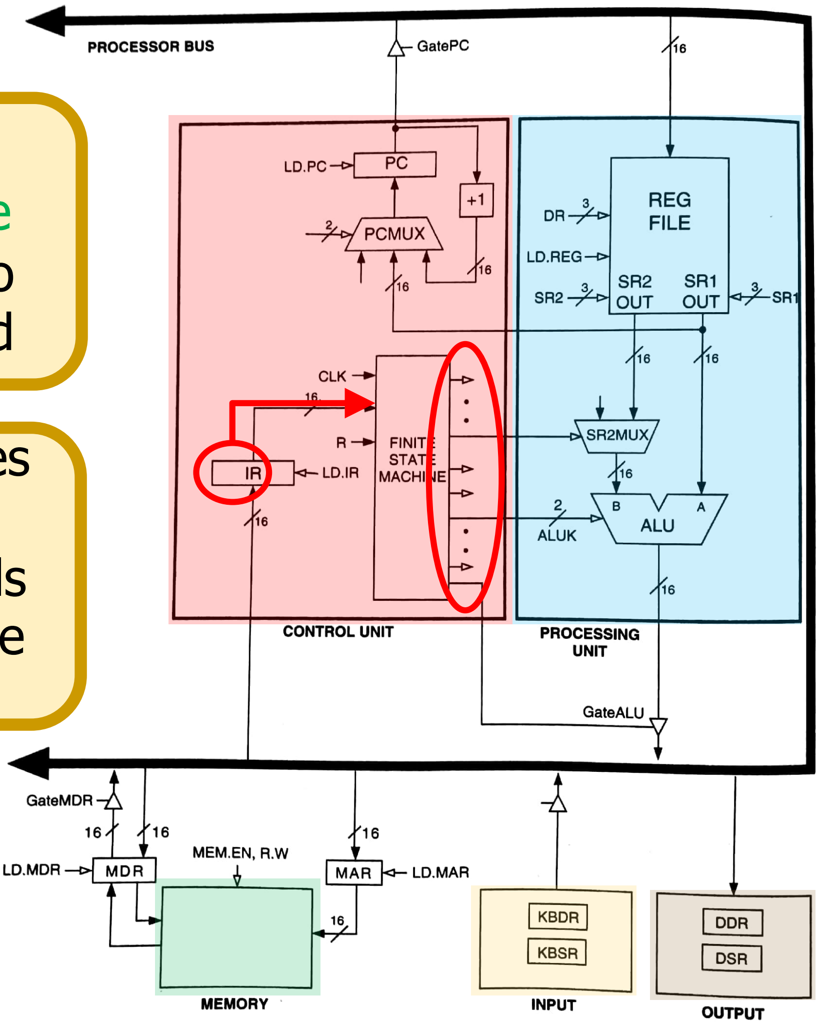


**Figure 4.3** The LC-3 as an example of the von Neumann model

## EVALUATE ADDRESS

- The EVALUATE ADDRESS phase computes the address of the memory location that is needed to process the instruction
- This phase is necessary in LDR
  - It computes the address of the data word that is to be read from memory
  - By adding an offset to the content of a register
- But not necessary in ADD

# EVALUATE ADDRESS in LC-3

LDR calculates the address by adding a register and an immediate

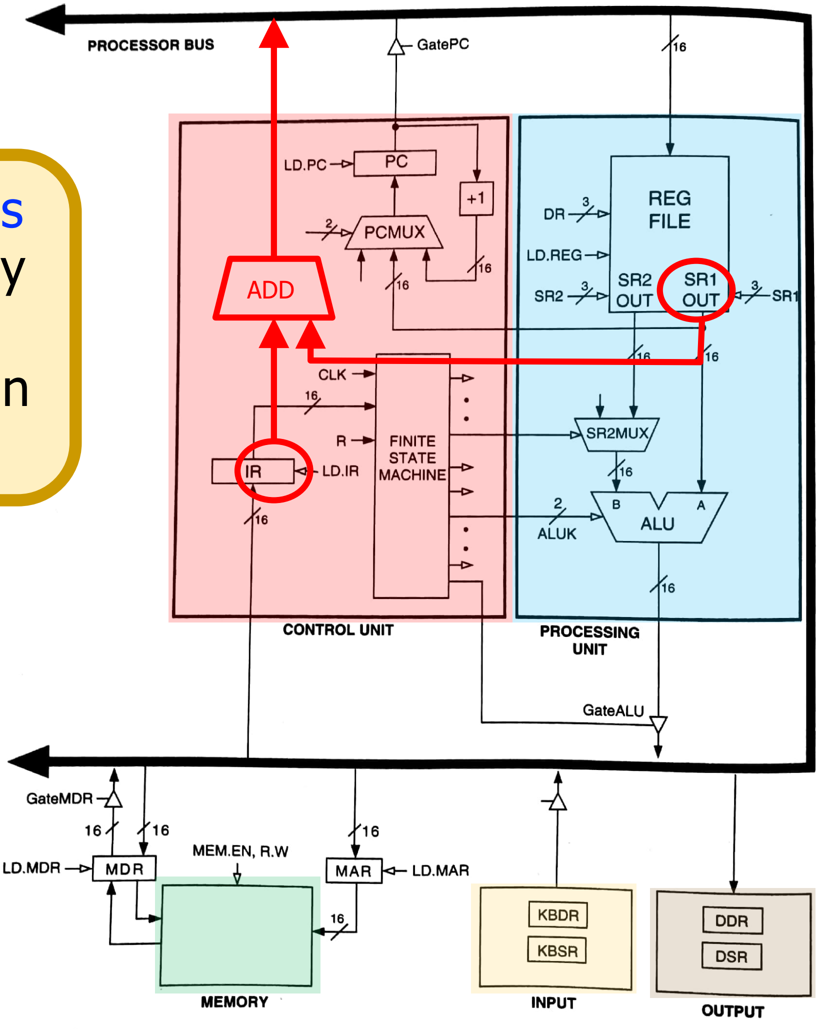


Figure 4.3 The LC-3 as an example of the von Neumann model

## FETCH OPERANDS

- The FETCH OPERANDS phase obtains the source operands needed to process the instruction

- In LDR
  - Step 1: Load MAR with the address calculated in EVALUATE ADDRESS

  - Step 2: Read memory, placing source operand in MDR

- In ADD
  - Obtain the source operands from the register file

  - In some microprocessors, operand fetch from register file can be done at the same time the instruction is being decoded

# FETCH OPERANDS in LC-3

LDR loads MAR (step 1), and places the results in MDR (step 2)

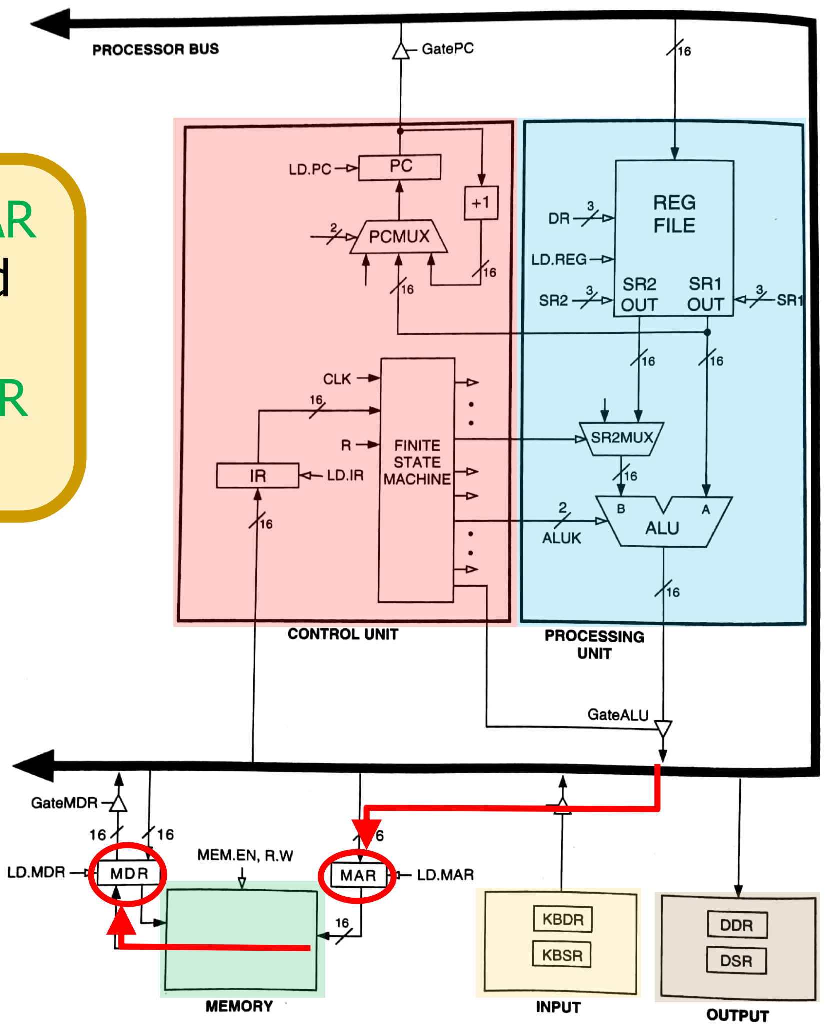


Figure 4.3 The LC-3 as an example of the von Neumann model

## EXECUTE

- The EXECUTE phase executes the instruction
  - In ADD, it performs addition in the ALU
  - In XOR, it performs bitwise XOR in the ALU
  - …

# EXECUTE in LC-3

ADD adds SR1 and SR2

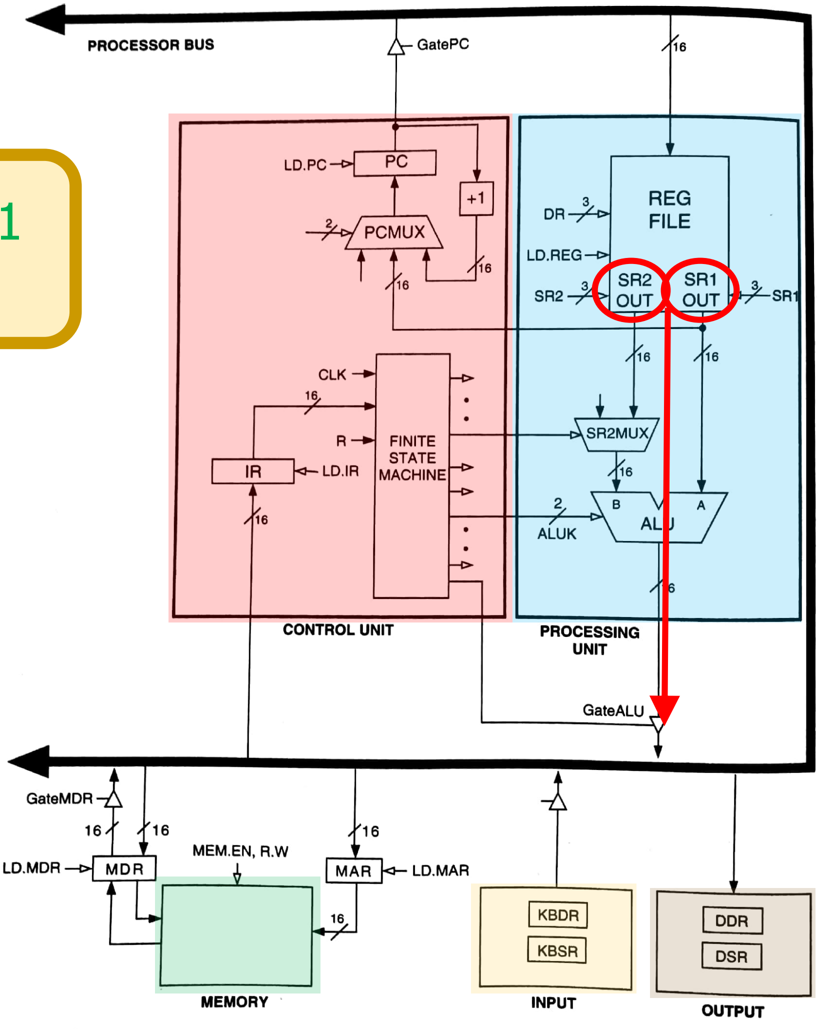


Figure 4.3 The LC-3 as an example of the von Neumann model

## STORE RESULT

- The STORE RESULT phase writes the result to the designated destination
- Once STORE RESULT is completed, a new instruction cycle starts (with the FETCH phase)

# STORE RESULTS in LC-3

ADD loads ALU Result into DR

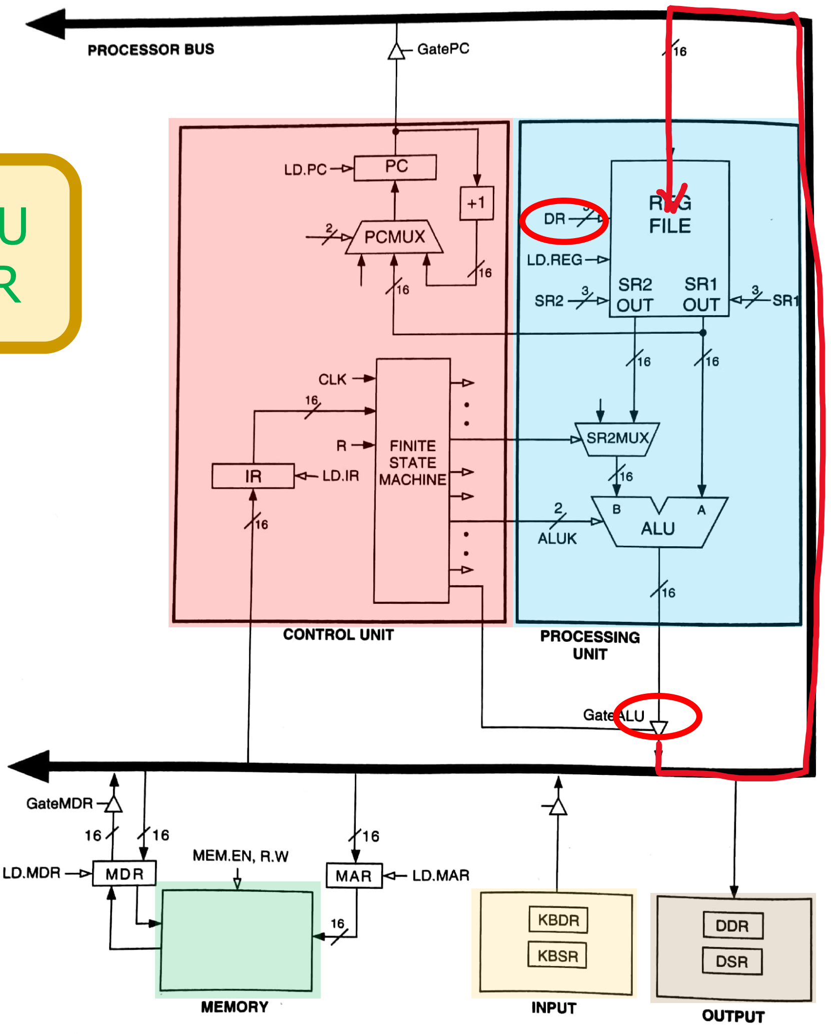


**Figure 4.3** The LC-3 as an example of the von Neumann model

# STORE RESULTS in LC-3

LDR loads
MDR into DR

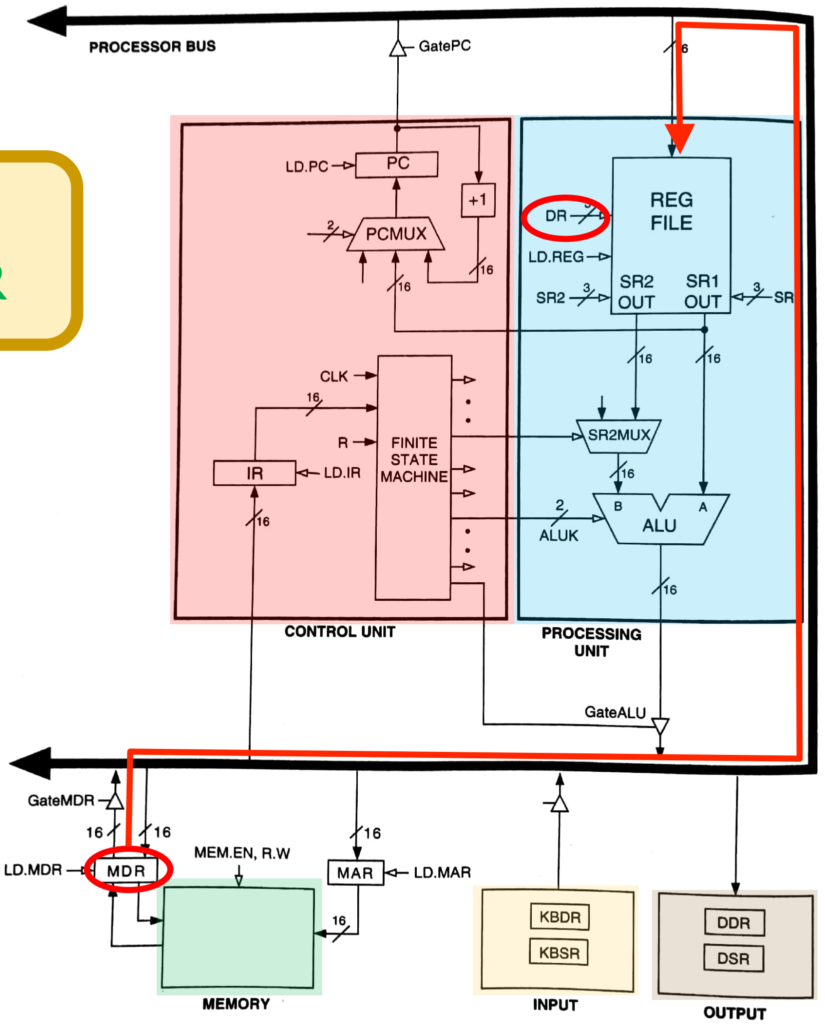
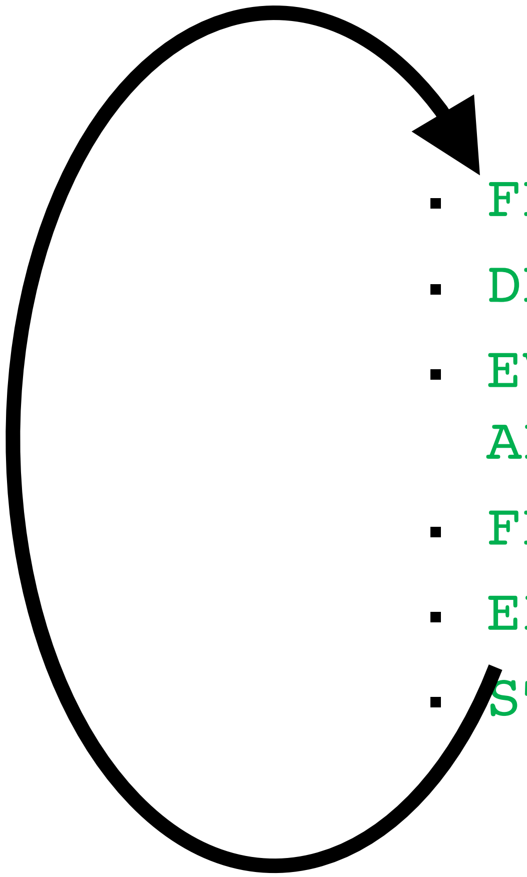


Figure 4.3 The LC-3 as an example of the von Neumann model

## The Instruction Cycle



- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT

## Changing the Sequence of Execution

- A computer program executes in sequence (i.e., in program order)
  - First instruction, second instruction, third instruction and so on

- Unless we change the sequence of execution

- Control instructions allow a program to execute out of sequence
  - They can change the PC by loading it during the EXECUTE phase
  - That wipes out the incremented PC (loaded during the FETCH phase)

## Jump (Branch)

- **Unconditional** branch or jump (ARM)

```
B   TARGET
```

- **Conditional** branch or jump (ARM)

```
BEQ   TARGET
```

```
BNE   TARGET
```

- These instructions are encoded using a special branch format in ARM ISA
- LC-3 has a jump instruction that can load a register into PC
- Let’s see

# PC UPDATE in LC-3

JMP loads
SR1 into PC

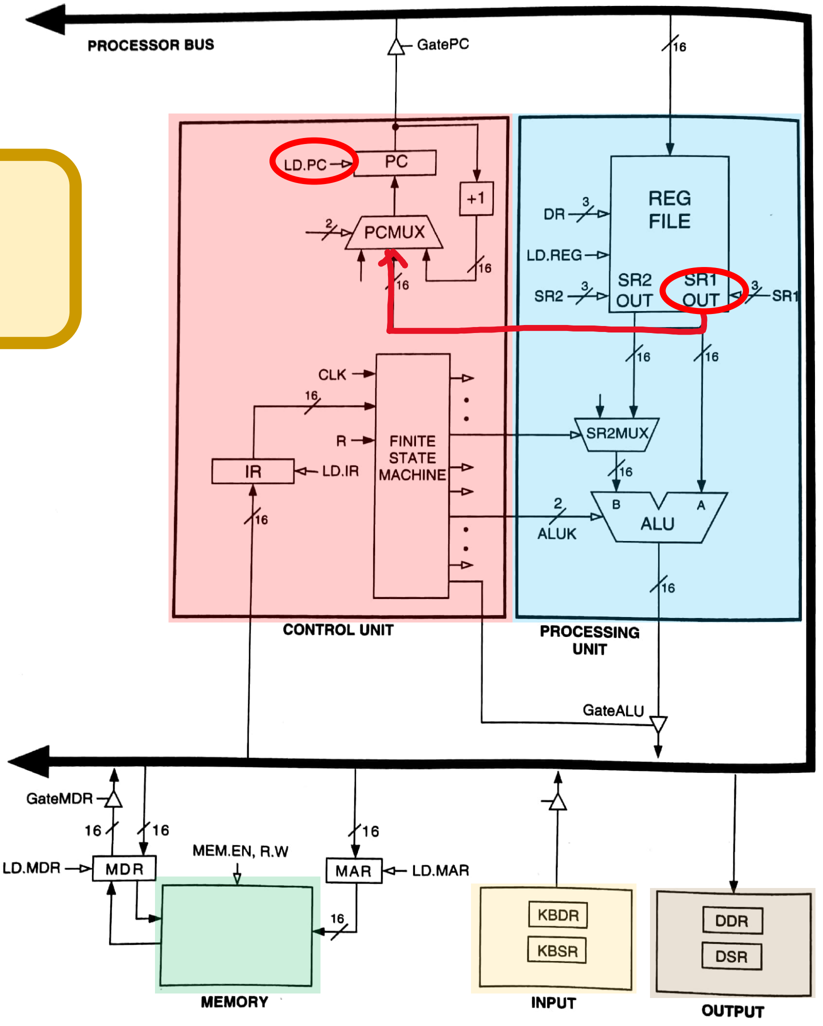
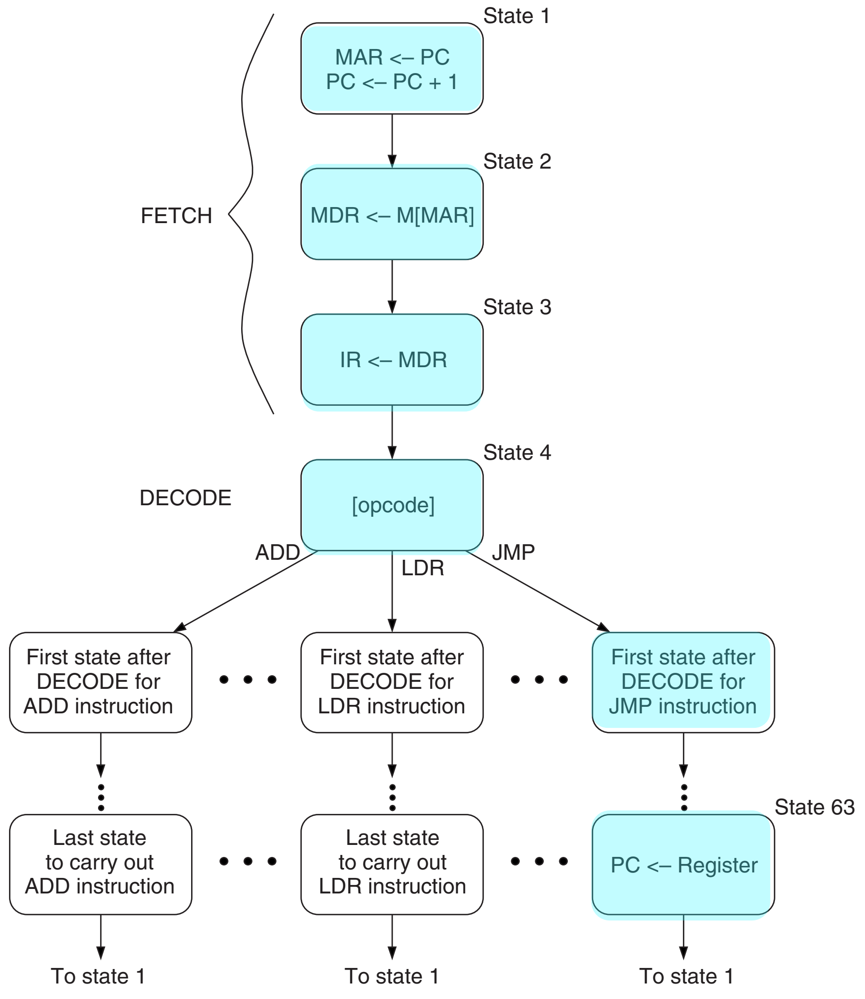


**Figure 4.3** The LC-3 as an example of the von Neumann model

# Control (FSM) of the Instruction Cycle



FETCH
State 1
MAR <– PC
PC <– PC + 1
State 2
MDR <– M[MAR]
State 3
IR <– MDR
DECODE
State 4
[opcode]
ADD
LDR
JMP
First state after DECODE for ADD instruction
First state after DECODE for LDR instruction
First state after DECODE for JMP instruction
Last state to carry out ADD instruction
Last state to carry out LDR instruction
State 63
PC <– Register
To state 1
To state 1
To state 1

Figure 4.4 An abbreviated state diagram of the LC-3

- State 1
  - The FSM asserts GatePC and LD.MAR
  - It selects input (+1) in PCMUX and asserts LD.PC
- State 2
  - MDR is loaded with the instruction
- State 3
  - The FSM asserts GateMDR and LD.IR
- State 4
  - The FSM goes to next state depending on opcode
- State 63
  - JMP loads register into PC
- Full state diagram in Patt&Pattel, Appendix C

**This is an FSM Controlling the LC-3 Processor**

## The Instruction Cycle

- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT

## The Instruction Cycle: Things to Note

- Not all instructions need all phases
- The ordering of phases in not set in stone
- Some phases can be grouped as one
- Some structures may not be needed in a different microarchitecture
- Microarchitecture “style” dictates many details (week 6)

## The Instruction Cycle: Things to Note

- What we have seen is a very general multi-cycle CPU
  - Each instruction takes multiple “machine cycles” to complete
- In Labs 4 – 6 + first assignment you build a single-cycle CPU
  - The entire instruction (all phases) must finish in one cycle
  - Contrast with multi-cycle CPU as you build
  - One clock cycle = One machine cycle = One instruction cycle
- We Will cover both single-cycle and multi-cycle ARM CPUs

## ARM and QuAC Instruction Set Architectures (ISAs)

ARM (**Chapter 6** of H&H + Assignment 2) and QuAC (Assignment 1)

## Von Neumann Model: Two Key Properties

- Von Neumann model is also called ***stored program computer*** (instructions in memory). It has two key properties:

- Stored program
  - Instructions stored in a linear memory array
  - Memory is unified between instructions and data
    - **The interpretation of a stored value depends on the control signals**

- Sequential instruction processing
  - One instruction processed (fetched, executed, completed) at a time
  - Program counter (instruction pointer) identifies the current instruction
  - Program counter is advanced sequentially except for control transfer instructions

## Recall: Instruction Types

- There are three main types of instructions

- Operate (data processing) instructions
  - Execute operations in the ALU

- Data movement (memory) instructions
  - Read from or write to memory

- Control flow (branch/jump) instructions
  - Change the sequence of execution (decision making)

## Data Processing Instructions

# ARM Data Processing (DP) Instructions

- a = b + c – d
  - We can use two ARM instructions to do the computation

```
ADD  t,   b,   c
SUB  a,   t,   d
```

- ADD and SUB are instruction **mnemonics**
- Instructions operate on operands `(a, b, c)`
- Computers operate on binary data not variable names
  - We need to specify the **physical location** of operands
  - We have registers, memory, constants in instructions

## Registers as Operands

- Instructions need **fast access** to operands, but **memory** is **slow**
  - Keep a small set of registers close to the CPU in a register file
  - ARM architecture uses **16 registers**
  - 32-bit architecture means 32-bit registers
- **a = b + c - d**
  - R0 = **a**, R1 = **b**, R2 = **c**, R3 = **d**, R4 = **t**

Mapping is chosen by human, or a tool called **compiler** that translates high-level code to assembly

```
ADD  t,   b,   c
SUB  a,   t,   d
```

```
ADD  R4,  R1,  R2
SUB  R0,  R4,  R3
```

## Aside: Compiler vs. Assembler

- **Compiler translates**
  - high-level language code into
    - assembly code (human readable)

- **Assembler translates**
  - assembly code into
    - machine code **(1s and 0s)**

## Source/Destination Operand

- Instructions operate on one or more **source** operands and store the result after execution in a **destination** operand

```
ADD  R4,  R1,  R2
SUB  R0,  R4,  R3
```

- R1 and R2 are the **source operands** for the `ADD` instruction

- R4 is the **destination operand** for the `ADD` instruction

## Another Example

- **a = b – c**
- **f = (g + h) – (i + j)**
  - Variables a – c are held in registers R0 – R2 and f – j are held in registers R3 – R7

```
SUB  R0,  R1,  R2
ADD  R8,  R4,  R5
ADD  R9,  R6,  R7
SUB  R3,  R8,  R9
```

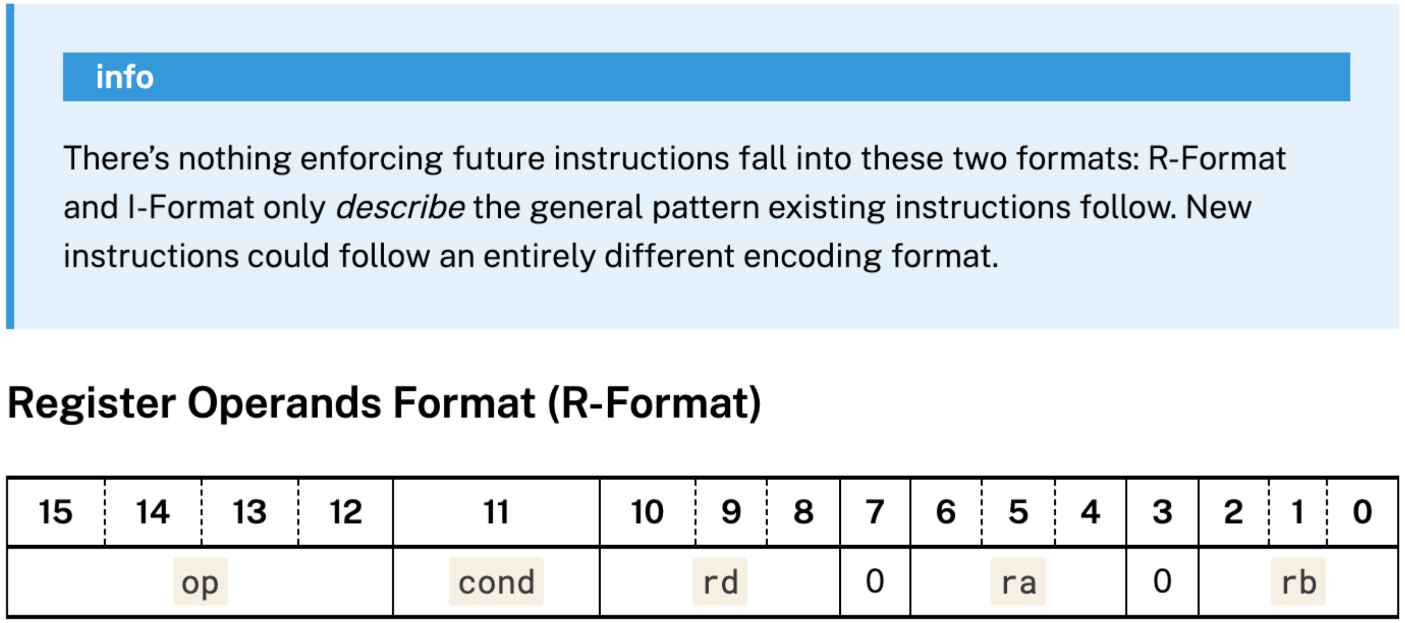
## Design Principle # 1

- Regularity leads to simpler hardware

  - Instructions with a consistent number of operands (2 sources, 1 destination) are easier to encode and handle in hardware

## Design Principle # 1

- Regularity leads to simpler hardware
  - Instru s (2
    source
    handl



info

There's nothing enforcing future instructions fall into these two formats: R-Format and I-Format only *describe* the general pattern existing instructions follow. New instructions could follow an entirely different encoding format.

**Register Operands Format (R-Format)**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| op | | | | cond | rd | | | 0 | ra | | | 0 | rb | | |

  - **QuAC** also follows the same principle!

## The Register Set (File)

- ARM defines 16 *architectural* registers
  - The register set is part of the ISA specification

- R0 – R12 are used for storing variables

- R13 – R15 have **special** uses

## Design Principle # 2

- Smaller is Faster

  - Reading data from a **small** register file is **faster** than reading from a large file

## Constant & Immediate in Instruction

- ARM instructions can use constant or immediate operands

  **Fact:** 98% of all the constants in a program would fit in 13 bits

- The value is available immediately from the instruction
  - Advantage: No register or memory access
  - Disadvantage: Immediate can be 8 – 12 bits because **limited bits in the encoding (instruction format)**

- In the following example, assume R7 = **a**, R8 = **b**

High-Level code

```
a = a + 4
b = a – 12
```

ARM Assembly Code

```
ADD  R7,  R7,  #4
SUB  R8,  R7,  #0xC
```

## Design Principle # 3

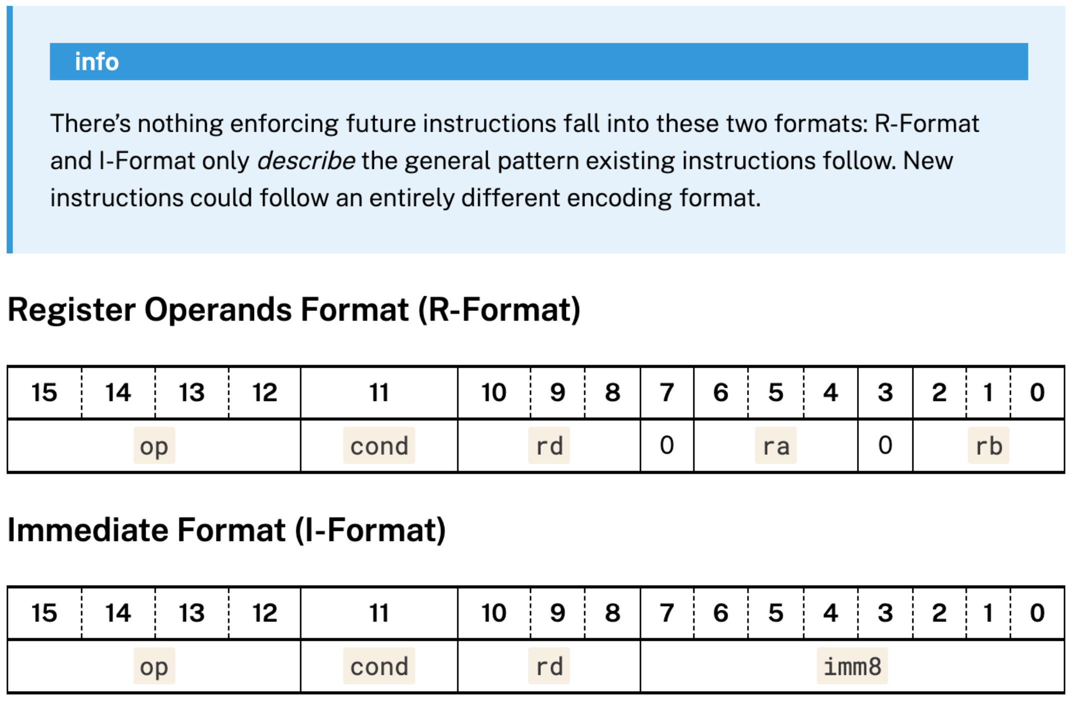
- Good design demands good compromises

  - To encode **immediate** instructions in QuAC, we need a new format

  - Same with ARM although encoding is more complex

# Design Principle # 3

- Good design demands good compr
  - To enc e need to move w format.



**info**

There's nothing enforcing future instructions fall into these two formats: R-Format and I-Format only *describe* the general pattern existing instructions follow. New instructions could follow an entirely different encoding format.

**Register Operands Format (R-Format)**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| op |  |  |  | cond | rd |  |  | 0 | ra |  |  | 0 | rb |  |  |

**Immediate Format (I-Format)**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| op |  |  |  | cond | rd |  |  | imm8 |  |  |  |  |  |  |  |

- **We follow the same principle in QuAC**

## MOV Instruction

- **MOV** is a useful instruction for initializing register values

- MOV can also take a register source operand
  - MOV R1, R7 copies the contents of register R7 into R1

  - In the following example, assume R4 = **i**, R5 = **x**

High-Level code

```
i = 0;
x = 4080;
```

ARM Assembly Code

```
MOV  R4,   #0
MOV  R5,   #0xFF0
```

# Instruction Format – 1: Data Processing

| 31:28 | 27:26 | 25:20 | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|
| cond | op | funct | Rn | Rd | Src2 |

- **Operands**
  - **Rn [19:16]:** first source operand register (0000, 0001, ..., 1111)
  - **Src2 [11:0]:** second source register or **unsigned** immediate
  - **Rd [15:12]:** destination register

- **Control fields**
  - **cond [31:28]:** specifies conditional execution (1110 for unconditional)
  - **op [27:26]:** the operation code or opcode (00 for data processing)
  - **funct [25:20]:** the specific function/operation to perform

# Breaking down funct Field

| 31:28 | 27:26 | 25:20 | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|
| cond | op | funct | Rn | Rd | Src2 |

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|---|---|
| cond | 00 | I | cmd | S | Rn | Rd | Src2 |

- **cmd [24:21]:** specifies the specific DP instruction (0100 for ADD; 0010 for SUB)

- **I-bit [25]:** informs the control unit about Src2
  - I = 0: Src2 is a register
  - I = 1: Src2 is an immediate

- **S-bit [20]:** 1 if the instruction sets the condition flags

# Two DP Formats (Src2 Variations)

## Immediate (assume 11:8 are 0 for now)

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:8 | | | | 7:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 00 | **1** | cmd | S | Rn | Rd | 0 | 0 | 0 | 0 | imm8 |

## Register (assume 11:4 are 0 for now)

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:4 | | | | | | | | 3:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 00 | **0** | cmd | S | Rn | Rd | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rm |

# DP with Src2 as Immediate

- Bit 25 (I) informs the CPU how to interpret Src2
  - I = 1, CPU interprets Src2[7:0] as an **unsigned** 8-bit constant

- Format (Src2 = immediate)

$$\begin{tikzcd}
\texttt{ADD} & \texttt{R0,} \arrow[d] & \texttt{R1,} \arrow[d] & \texttt{\#16} \arrow[d] \\
\texttt{ADD} & \texttt{Rd,} & \texttt{Rn,} & \texttt{\#imm8}
\end{tikzcd}$$

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:8 | | | | 7:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 00 | 1 | cmd | S | Rn | Rd | 0 | 0 | 0 | 0 | imm8 |

## DP with Src2 as Register

- Bit 25 (I) informs the CPU how to interpret Src2
  - I = 0, CPU interprets Src2[3:0] as a register

- Format (Src2 = Register)

$$\begin{tikzcd}
\texttt{ADD} & \texttt{R0,} \arrow[d] & \texttt{R1,} \arrow[d] & \texttt{R3} \arrow[d] \\
\texttt{ADD} & \texttt{Rd,} & \texttt{Rn,} & \texttt{Rm}
\end{tikzcd}$$

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:4 | | | | | | | | 3:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 00 | 0 | cmd | S | Rn | Rd | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rm |

## More Data Processing Insts.

- `AND`
- `ORR (OR)`
- `EOR (XOR)`

- `BIC (Bit Clear)`

- `MVN (MoVe and Not)`

## The Bit Clear Instruction

- `Bit Clear (BIC)`
  - Used for bit masking bits and forcing **unwanted** bits to 0

- `BIC R6, R1, R2`
  - R2 is the mask
    - The bits we want to CLEAR or ZERO in `R1` are set to TRUE in `R2`

  - The instruction stores the result of `R1 AND (NOT R2)` in `R6`

# Example of Data Processing

Source registers

| | | | | |
|---|---|---|---|---|
| R1 | 0100 0110 | 1010 0001 | 1111 0001 | 1011 0111 |
| R2 | 1111 1111 | 1111 1111 | 0000 0000 | 0000 0000 |

| Assembly code | Result | | | | |
|---|---|---|---|---|---|
| `AND R3, R1, R2` | R3 | 0100 0110 | 1010 0001 | 0000 0000 | 0000 0000 |
| `ORR R4, R1, R2` | R4 | 1111 1111 | 1111 1111 | 1111 0001 | 1011 0111 |
| `EOR R5, R1, R2` | R5 | 1011 1001 | 0101 1110 | 1111 0001 | 1011 0111 |
| `BIC R6, R1, R2` | R6 | 0000 0000 | 0000 0000 | 1111 0001 | 1011 0111 |
| `MVN R7, R2` | R7 | 0000 0000 | 0000 0000 | 1111 1111 | 1111 1111 |

## Design Principle # 4

- Make the common case fast
  - ARM architecture includes only **simple,** *commonly used* instructions
  - The number of instructions is kept small, so the hardware required for decoding is **simple**, **small**, and **fast**
  - More elaborate operations are performed using sequences of **multiple simple instructions**

## RISC vs. CISC Architectures

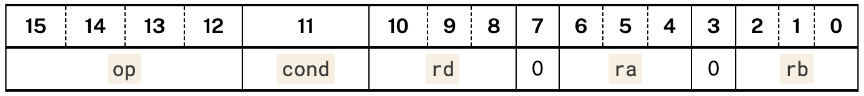
- Reduced Instruction Set Computer (RISC)
  - Provide a **small set** of simple instructions
  - Minimizes hardware complexity (high clock rate, power-efficient)
  - Requires many instructions to solve a complex problem
  - **Examples:** ARM, MIPS, QuAC, RISC-V

- Complex Instruction Set Computer (CISC)
  - Provides many complex instructions
  - Complex hardware (longer critical paths, lower clock frequency)
  - Each instruction is more complex so fewer instructions to solve a problem
  - **Example:** Intel x86

# Another RISC ISA: QuAC

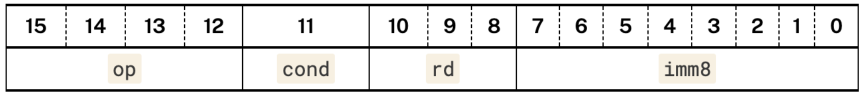
- **Fixed width** instructions make decoding easy and simple
- A small number of **crucial** instructions (fewer opcodes save instruction real-estate)

Register Operands Format (R-Format)



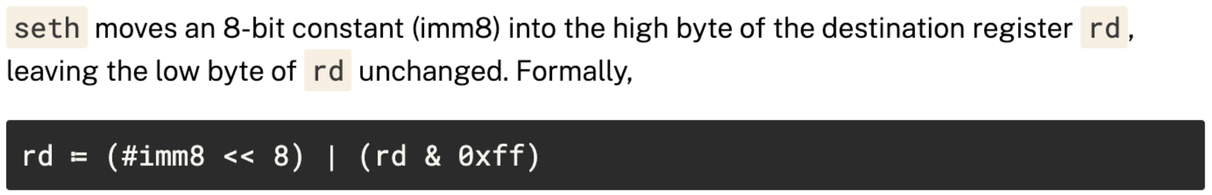
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `op` | | | | `cond` | `rd` | | | 0 | `ra` | | | 0 | `rb` | | |

Immediate Format (I-Format)



| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `op` | | | | `cond` | `rd` | | | `imm8` | | | | | | | |

- Two formats and **regularity** in the ISA (across formats)
  - `rd` in same place ($\texttt{Instr}_{10:8}$)
  - `opcode` in the same place
- `seth`: somewhat complex



`seth` moves an 8-bit constant (imm8) into the high byte of the destination register `rd`, leaving the low byte of `rd` unchanged. Formally,

```
rd = (#imm8 << 8) | (rd & 0xff)
```

- Few general-purpose registers
- Space for constants in the ISA
- Easy to convert to hexadecimal
- The only way to access memory is via a dedicated set of instructions
- Conditional execution + general-purpose PC = Conditional branch instructions

## Data Movement Instructions

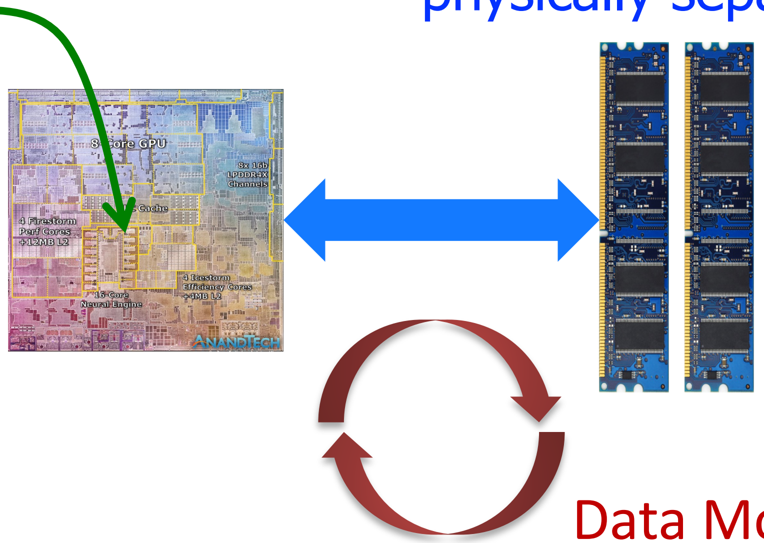
## Data Movement Instructions

- Real programs need to operate on more data than can fit in the register file
    - Most data resides in (slow) memory
    - Fetched from memory into the register file when needed
    - Moved to memory from the register file to free up a register

## Motivation

**Small and Fast Registers** are inside the CPU close to the ALU

**Large and Slow External Main Memory is outside the CPU, and** physically separated from the CPU



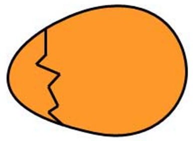
Data Movement Instructions move data to and from registers and memory

## Data Movement Instructions

- **Two instructions to facilitate data movement**
  - **The LDR instruction:** Bring data word from memory into the register file
    - **L**oa**D** **R**egister
  - **The STR instruction:** Store data word from the register file into memory
    - **ST**ore **R**egister

# Memory View (32 bits = 4 bytes)

- Byte-addressable memory (each box is a byte & each row is a word)
- Byte addresses (left) and 8-bit byte data (right, 1 byte = 2 Hex digits)



| Byte Address | | | | Word Address | Data | | | | Word Number |
|---|---|---|---|---|---|---|---|---|---|
| | | ⋮ | | ⋮ | | ⋮ | | | ⋮ |
| 13 | 12 | 11 | 10 | 00000010 | CD | 19 | A6 | 5B | Word 4 |
| F | E | D | C | 0000000C | 40 | F3 | 07 | 88 | Word 3 |
| B | A | 9 | 8 | 00000008 | 01 | EE | 28 | 42 | Word 2 |
| 7 | 6 | 5 | 4 | 00000004 | F2 | F1 | AC | 07 | Word 1 |
| 3 | 2 | 1 | 0 | 00000000 | AB | CD | EF | 78 | Word 0 |

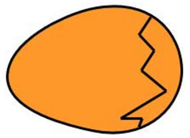
MSB LSB

Little-Endian View

$\longleftrightarrow$ 4 Bytes

# Memory View (32 bits = 4 bytes)

- Byte-addressable memory (each box is a byte & each row is a word)
- Byte addresses (left) and 8-bit byte data (right, 1 byte = 2 Hex digits)



| Byte Address | | | | Word Address | Data | | | | Word Number |
|---|---|---|---|---|---|---|---|---|---|
| | | | ⋮ | ⋮ | | | ⋮ | | ⋮ |
| 10 | 11 | 12 | 13 | 00000010 | C D | 1 9 | A 6 | 5 B | Word 4 |
| C | D | E | F | 0000000C | 4 0 | F 3 | 0 7 | 8 8 | Word 3 |
| 8 | 9 | A | B | 00000008 | 0 1 | E E | 2 8 | 4 2 | Word 2 |
| 4 | 5 | 6 | 7 | 00000004 | F 2 | F 1 | A C | 0 7 | Word 1 |
| 0 | 1 | 2 | 3 | 00000000 | A B | C D | E F | 7 8 | Word 0 |

MSB LSB

Big-Endian View

4 Bytes

# Revision (Start of Week 6/1)

- **Steps of Transformation**
  - From high-level language code to assembly code (compiler or human)
  - From assembly code to machine code (assembler or human)
- **Instruction set architecture**
  - Instruction set
    - Opcodes and operands
    - Data types
    - Addressing modes
    - Instruction formats
  - Architectural state
    - Memory
    - Register set
    - Program counter

## Reading from Memory

- **Format** of **L**oa**D** **R**egister instruction

  `LDR R0, [R1,  #12]`

- **Address calculation (base + offset addressing)**
  - Add **base** address (contents of `R1`) to the **offset** (`#12`)
  - Address = (`R1 + 12`)
  - Use any register for base address
  - `R1` is a source (register) operand

- **Result**
  - `R0` holds the data at memory address [`R1 + 12`] after the instruction is executed
  - `R0` is a destination (register) operand

## LDR Example

- Read a 32-bit word of data at memory (byte) address 8 into R3. Use R2 as the base register. Show the contents of R3.
  - Let's initialize R2 to 0, and add 8 as the offset

```
MOV    R2,  #0
LDR    R3,  [R2, #8]
```

| | |
|---|---|
| R3 | 0x 01 EE 28 42 |

| Word Address | Data | | | | Word Number |
|---|---|---|---|---|---|
| ⋮ | ⋮ | | | | ⋮ |
| 00000010 | CD | 19 | A6 | 5B | Word 4 |
| 0000000C | 40 | F3 | 07 | 88 | Word 3 |
| 00000008 | 01 | EE | 28 | 42 | Word 2 |
| 00000004 | F2 | F1 | AC | 07 | Word 1 |
| 00000000 | AB | CD | EF | 78 | Word 0 |

## Address vs. Value

- Square brackets signify **address** (also called **pointer** in C)

```
LDR    R3,  [R2, #8]
```

  - If you [add the contents of register R2 to constant #8, you will get the **address** with which to **access** memory]

**^ Base + Offset Addressing Mode**

- When presented with an address, memory obliges by returning the value stored at address given (8 in this example)

- In a 32-bit computer
  - Width of address bus = 32 bits (address space = $2^{32}$ locations)
  - Although memory is byte-addressable, it returns a 32-bit word to fill the entire register

## Writing to Memory

- **Format** of **ST**ore **R**egister instruction
  **STR** R0, [R1, #12]

- **Address calculation**
  - Add base address (`R1`) to the offset (`12`)
  - Address = (`R1 + 12`)
  - `R0` and `R1` are both source (register) operands

- **Result**
  - Memory address (`R1 + 12`) will now have the value in `R0` after the instruction is executed
  - Destination operand is memory address computed from source operands

## STR Example

- Store the value held in `R7` into memory word `21`
  - Let's initialize `R5` to `0`, and add `84` (`21 X 4`) as the offset

```
MOV     R5,     #0
STR     R7,     [R5,   #0x54]
```

- The offset can be written in decimal or hexadecimal: `84 (decimal) is 0x54 (Hex)`

## Instruction Format – 2: Memory

| 31:28 | 27:26 | 25:20 | | | | | | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| cond | op | I | P | U | B | W | L | Rn | Rd | Src2 |

- op = 01
- Rn = base register (base address)
- Rd = destination (load), source (store)
- Src2 = offset (register, shifted register, immediate)
- funct [25:20] = 6 control bits
  - I (Bit 25): Encoding of Src2
  - L (Bit 20): Load or Store

## LDR with Src2 as Immediate

- I (Bit 25) = 1: Src2 = imm12 where imm2 is a 12-bit unsigned offset added to the value in the base register (Rn)

- Format of **L**oa**D** **R**egister instruction

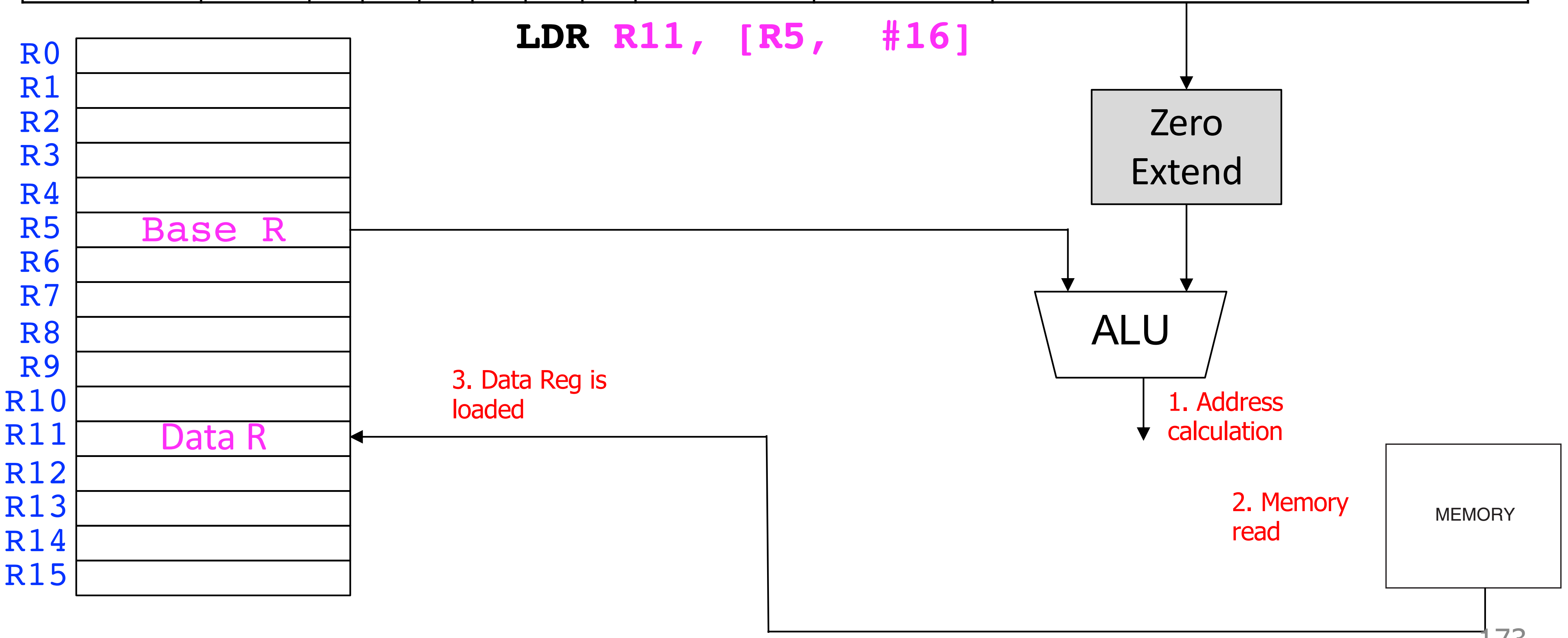
$$\begin{tikzcd} \texttt{LDR} & \texttt{R0,} \arrow[d] & \texttt{[R1,} \arrow[d] & \texttt{\#12]} \arrow[d] \\ \texttt{LDR} & \texttt{Rd,} & \texttt{[Rn,} & \texttt{\#imm12]} \end{tikzcd}$$

- L (Bit 20) = 1: CPU performs an LDR

| 31:28 | 27:26 | 25:20 | | | | | | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| cond | 01 | 1 | 1 | 1 | 0 | 0 | 1 | Rn | Rd | imm12 |

# LDR Datapath

| 31:28 | 27:26 | | | 25:20 | | | | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| cond | 01 | 1 | 1 | 1 | 0 | 0 | 1 | Rn | Rd | imm12 = 16 |



## STR with Src2 as Immediate

- I (Bit 25) = 1: Src2 = imm12 where imm2 is a 12-bit unsigned offset added to the value in the base register (Rn)

- Format of **ST**ore **R**egister instruction

```
STR R0, [R1,  #12]
     ↓     ↓      ↓
STR Rd, [Rn,  #imm12]
```

- L (Bit 20) = 0: CPU performs an STR

| 31:28 | 27:26 | 25:20 | | | | | | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| cond | 01 | 1 | 1 | 1 | 0 | 0 | 0 | Rn | Rd | imm12 |

REGISTER can hold memory **address**

[R1] : R1 is a **pointer** (→) to Data

Memory Load returns Data or Value

Data is Stored in memory. Address is INPUT

**Same Memory Stores Instructions and Data**

[PC] → Instruction

## Conditional Execution

## Conditional Execution

- ALU operations set the condition (status) flags
  - They are contained in a register called the **C**urrent **P**rogram **S**tatus **R**egister (**CPSR**)

- We can execute instructions conditionally based on a specific condition flag being TRUE or FALSE

## Conditional Execution

- ARM allows conditional execution in two steps
  - **Step 1**:  Instruction sets the condition flags  (Negative, Zero, Carry, Overflow)
  - **Step 2**:  Subsequent instructions execute based on the state of the condition flags

## Setting the Condition Flags

- **Method 1:** Use the **COMP**ARE instruction

```
CMP  R5,  R6
```

  - The instruction subtracts the second source operand from the first operand (R5 – R6)

  - The instruction does not save any result

- Flags are set as follows

| | |
|---|---|
| Is 0, | Z = 1 |
| Is negative, | N = 1 |
| Causes a carry out, | C = 1 |
| Causes a signed overflow, | V = 1 |

## Setting the Condition Flags

- **Method 2:** Append the instruction mnemonic with **S**

```
ADDS R1, R2, R3
```

- The instruction adds source operands R2 and R3
- It sets the flags (**S**)
- It saves the result in R1

## Condition Mnemonics

- We can execute instructions conditionally based on the status of the flags register

- Condition for execution is encoded as a *condition mnemonic* appended to the instruction mnemonic

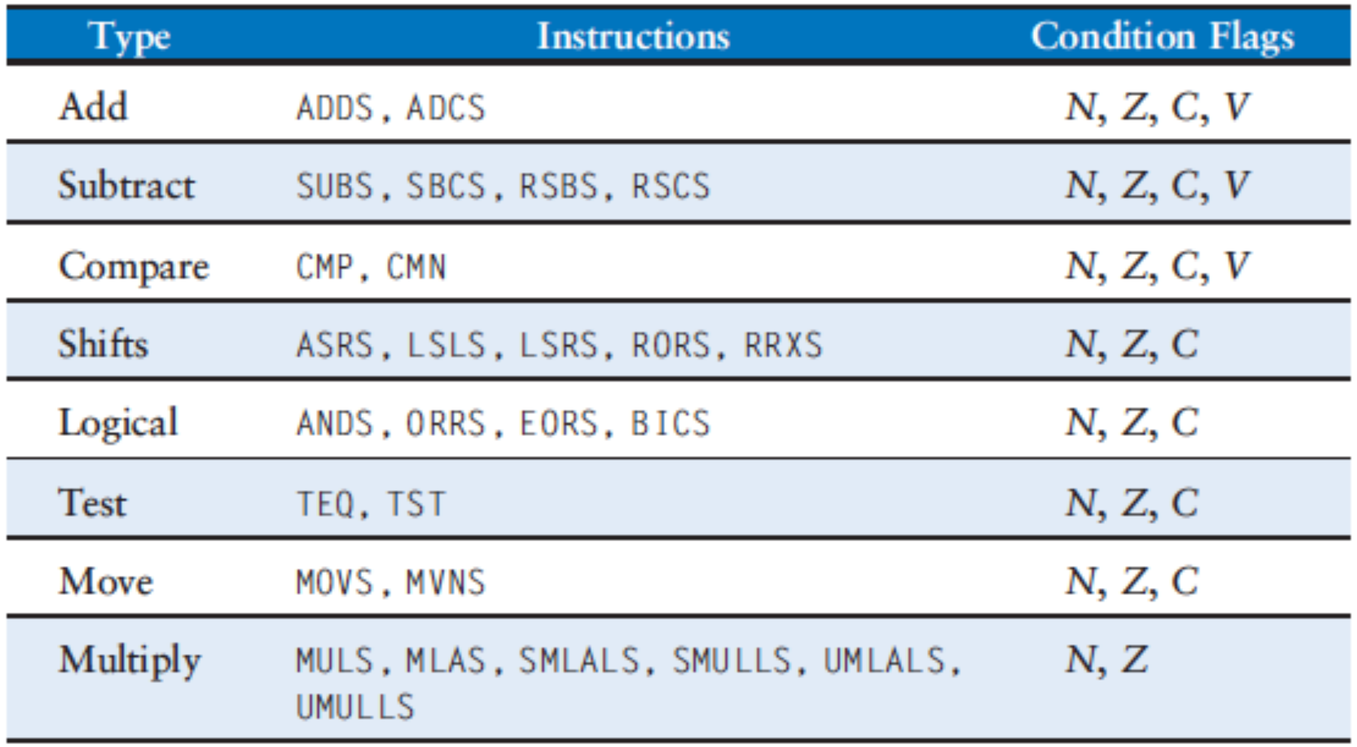
```
CMP   R1, R2
SUBNE R3, R5, R8
ADDEQ R1, R2, R3
```

- **NE** and **EQ** are condition mnemonics
- SUB executes only if `R1` is not equal to `R2` (meaning `Z = 0`)

# Condition Mnemonics

| *cond* | Mnemonic | Name | CondEx |
|---|---|---|---|
| 0000 | EQ | Equal | $Z$ |
| 0001 | NE | Not equal | $\bar{Z}$ |
| 0010 | CS / HS | Carry set / Unsigned higher or same | $C$ |
| 0011 | CC / LO | Carry clear / Unsigned lower | $\bar{C}$ |
| 0100 | MI | Minus / Negative | $N$ |
| 0101 | PL | Plus / Positive of zero | $\bar{N}$ |
| 0110 | VS | Overflow / Overflow set | $V$ |
| 0111 | VC | No overflow / Overflow clear | $\bar{V}$ |
| 1000 | HI | Unsigned higher | $\bar{Z}C$ |
| 1001 | LS | Unsigned lower or same | $Z\ OR\ \bar{C}$ |
| 1010 | GE | Signed greater than or equal | $\overline{N \oplus V}$ |
| 1011 | LT | Signed less than | $N \oplus V$ |
| 1100 | GT | Signed greater than | $\bar{Z}(\overline{N \oplus V})$ |
| 1101 | LE | Signed less than or equal | $Z\ OR\ (N \oplus V)$ |
| 1110 | AL (or none) | Always / unconditional | ignored |

# Instructions that affect condition flags



| Type | Instructions | Condition Flags |
|---|---|---|
| Add | `ADDS, ADCS` | *N, Z, C, V* |
| Subtract | `SUBS, SBCS, RSBS, RSCS` | *N, Z, C, V* |
| Compare | `CMP, CMN` | *N, Z, C, V* |
| Shifts | `ASRS, LSLS, LSRS, RORS, RRXS` | *N, Z, C* |
| Logical | `ANDS, ORRS, EORS, BICS` | *N, Z, C* |
| Test | `TEQ, TST` | *N, Z, C* |
| Move | `MOVS, MVNS` | *N, Z, C* |
| Multiply | `MULS, MLAS, SMLALS, SMULLS, UMLALS, UMULLS` | *N, Z* |

## Example

- `R5 = 17 and R9 = 23`
- Will the **SUBEQ** and **ORRMI** instructions execute?
- **N Z C V** = ?

```
CMP     R5,  R9
SUBEQ   R1,  R2,  R3
ORRMI   R4,  R0,  R9
```

## Another Example (page 307-308 of book)

- `R2 = 0x80000000 and R3 = 0x00000001`

- **Which instructions will execute?**

  - **N Z C V** = ?

```
CMP    R2,  R3
ADDEQ  R4,  R5,  #78
ANDHS  R7,  R8,  R9
ORRMI  R10, R11, R12
EORLT  R12, R7,  R10
```

# Conditional Execution in QuAC

- Bit 11 is associated with a condition code

Register Operands Format (R-Format)

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| op | | | | cond | rd | | | 0 | ra | | | 0 | rb | | |

- ALU instructions set the flags (a.k.a. condition codes). See Flags in QuAC ISA
  - The CPU uses that information to determine whether to execute the current instruction or not (e.g., store result into register file or memory)

| Name | Suffix | Encoding | Condition | Meaning |
|---|---|---|---|---|
| Always | - | `0` | - | Always executes |
| Equals | `eq` | `1` | `Z == 1` | Execute if latest ALU result was zero |

- If **cond field** ($\text{Instr}_{11}$) is TRUE, then
  - Execute the instruction only if he last ALU instruction set the **Z** flag to TRUE
  - Otherwise, do not execute the instruction (depart from the usual control flow)
- The default encoding of the **cond** field is 0 (execute the instruction)
  - `add   r1, r2, r3` (**cond** = FALSE)
  - `addeq r1, r2, r3` (**cond** = TRUE)

# Recall: Conditional Execution in QuAC

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `op` | | | | `cond` 1 | `rd` | | | 0 | `ra` | | | 0 | `rb` | | |

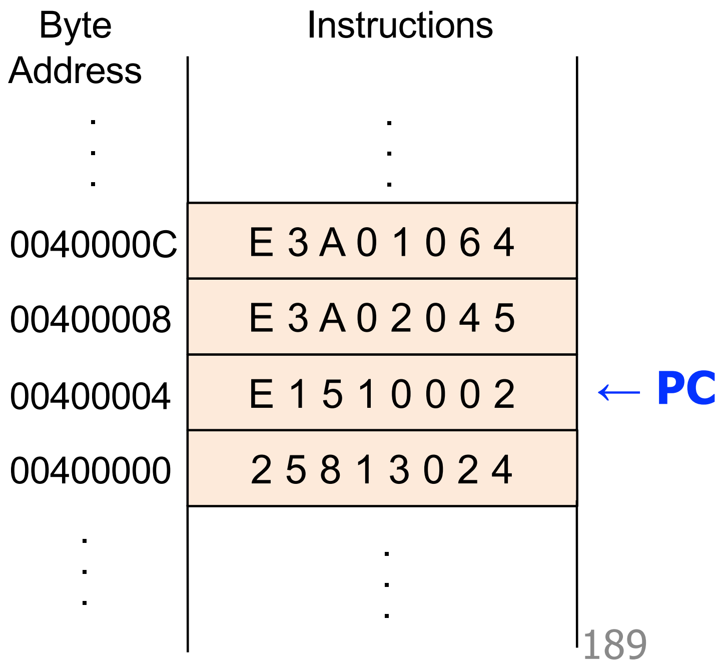
- `add`**`eq`** `r1, r2, r3` (cond = TRUE)

- What is the relationship between **eq** and **Z** flag?
  - A comparison of two registers shows they are equal (i.e., their difference is 0)

| Name | Suffix | Encoding | Condition | Meaning |
|---|---|---|---|---|
| Always | - | `0` | - | Always executes |
| Equals | `eq` | `1` | `Z == 1` | Execute if latest ALU result was zero |

## Branch Instructions

# Program Counter (PC) points to (contains the **address of**) **next instruction** to execute



## Normal (Sequential) Execution

- 32-Bit ISA with Byte-Addressable Memory
  - `PC = PC + 4`

- 64-Bit ISA with Byte-Addressable Memory
  - `PC = PC + 8`

- 32-Bit ISA with Word-Addressable Memory
  - `PC = PC + 1`

## Normal (Sequential) Execution

Increment `PC` during instruction FETCH to prepare to execute the **NEXT** Instruction

**However:** It is often useful to break this sequence

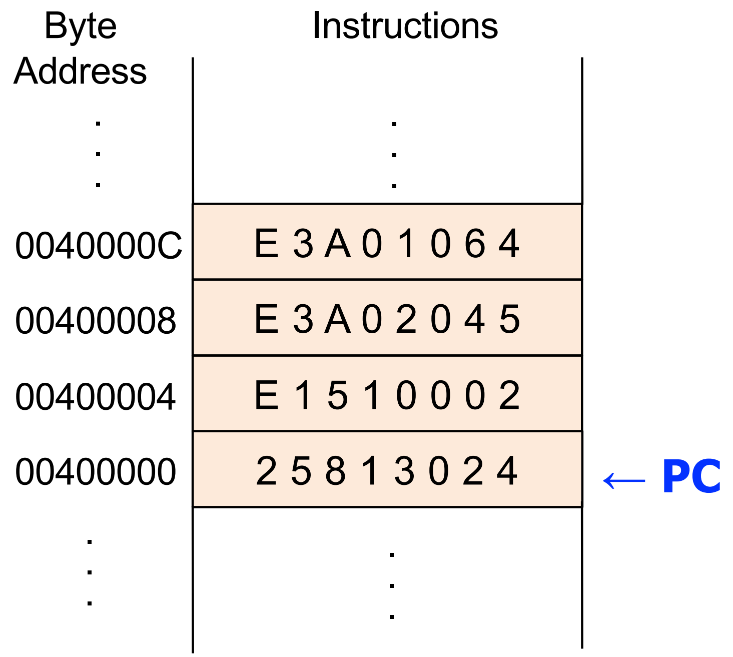
(1) Altering the `PC` **differently** can break the sequential flow of program execution

(2) Branch instructions alter the program counter to break the sequential flow of exeuction

# Program Counter (PC)

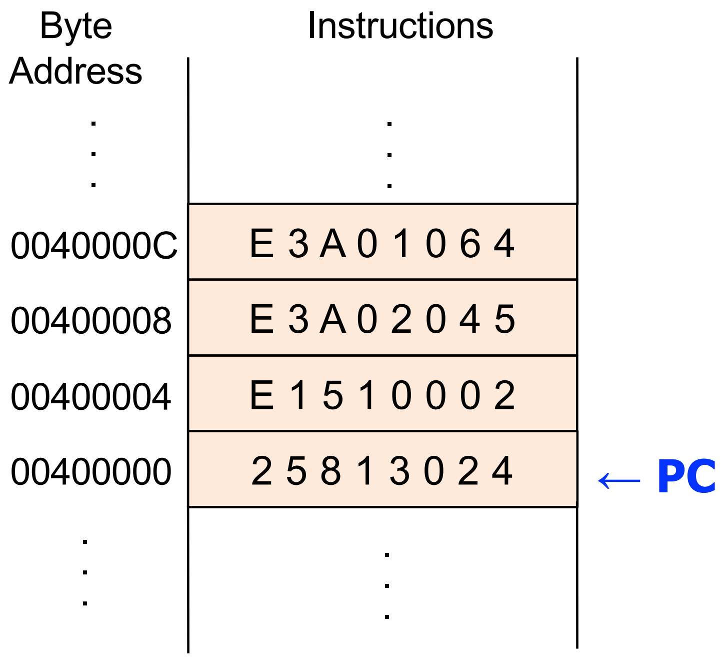
- Program Counter (**PC**): Contains the **address** of (or **points** to) the next instruction to be executed
- Incremented by 4 (= 4 bytes or 32 bits) in the FETCH phase

- PC = PC + 4 to execute the next **sequential instruction** in memory



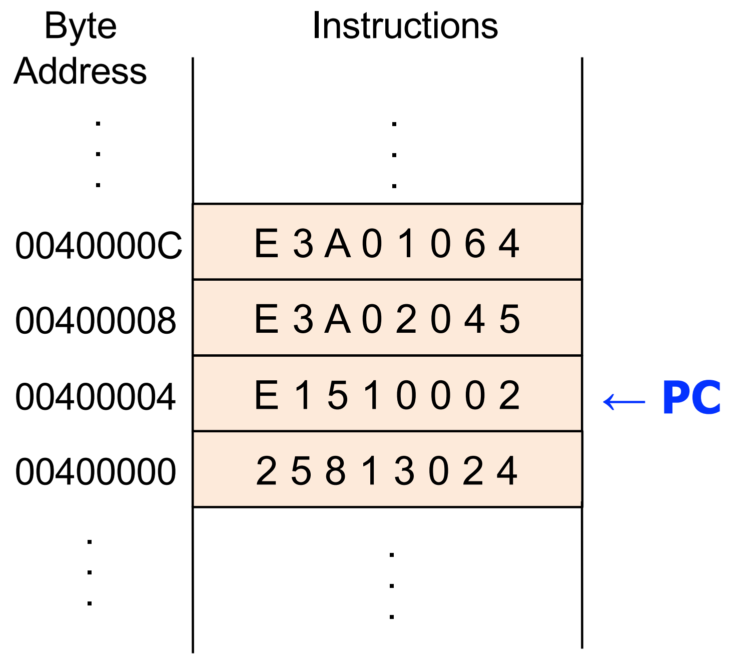
# Program Counter (PC)

- PC = PC + 4 to execute the next **sequential instruction** in memory



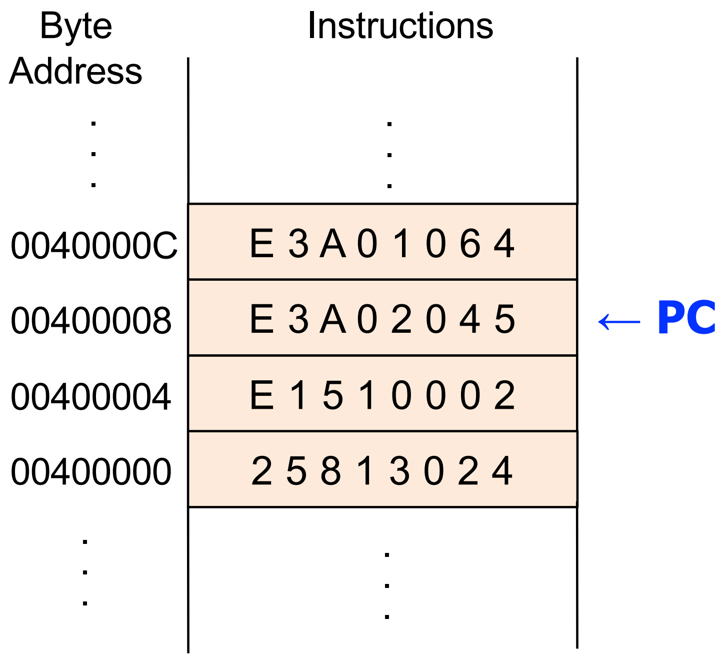
## Program Counter (PC)

- PC = PC + 4 to execute the next **sequential instruction** in memory



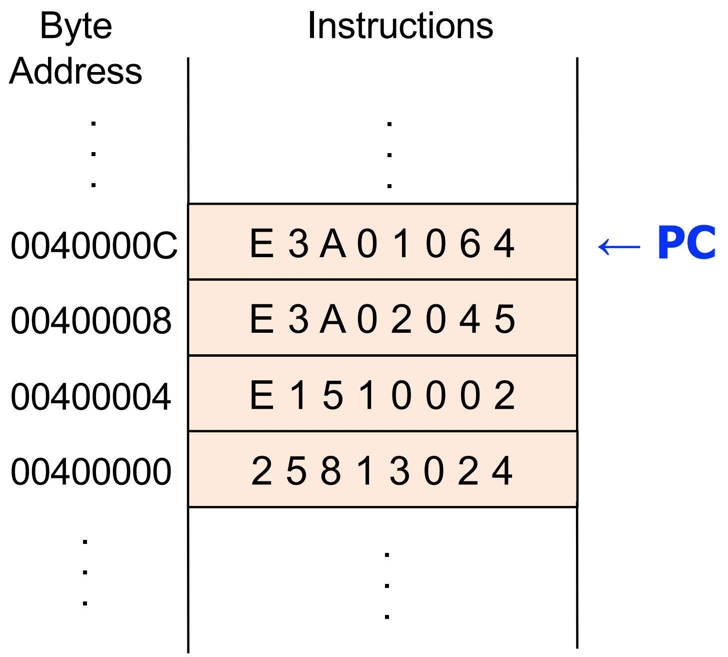
# Program Counter (PC)

- PC = PC + 4 to execute the next **sequential instruction** in memory



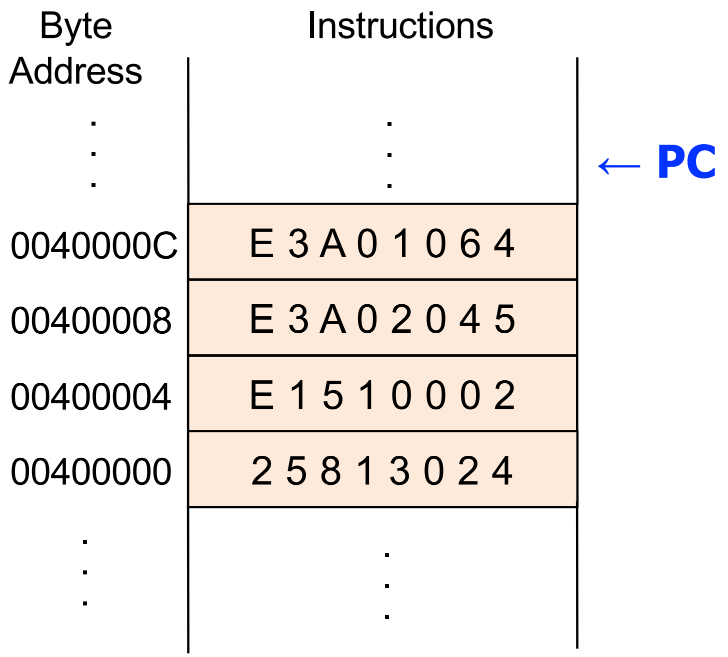
## Program Counter (PC)

- PC = PC + 4 to execute the next **sequential instruction** in memory



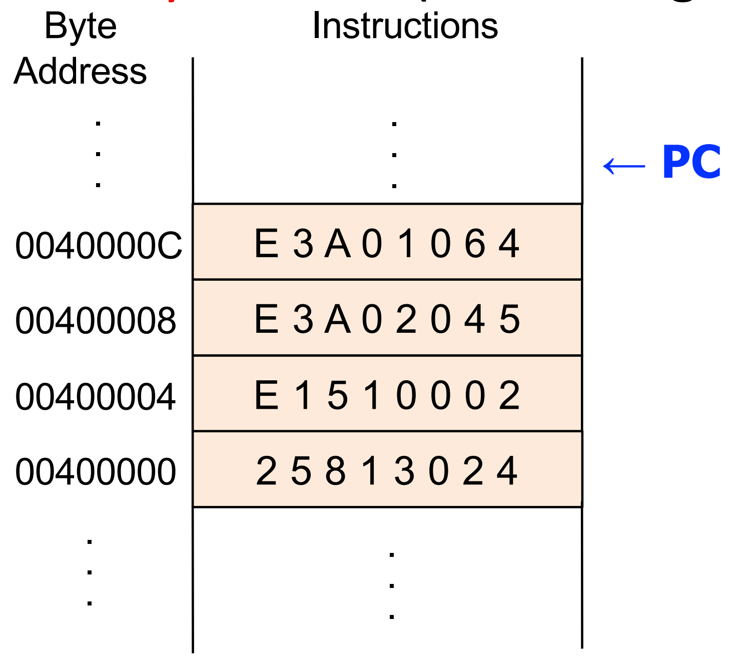
## Program Counter (PC)

- PC = PC + 4 to execute the next **sequential instruction** in memory



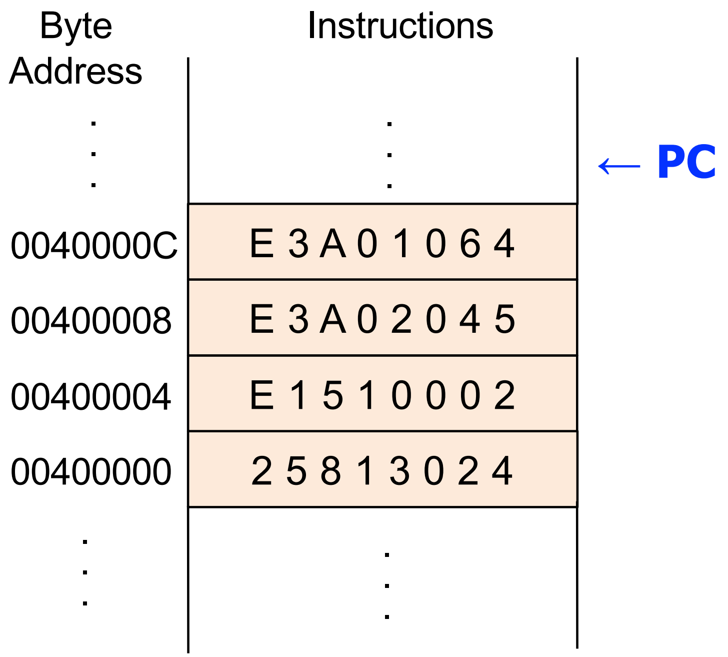
# Branch Instructions and PC

- Branch instructions change the **PC** to point to a different instruction than the next **sequential instruction** in memory
- Updated by a different address in the EXECUTE phase
  - New address PC points to is determined by formula (addressing mode)



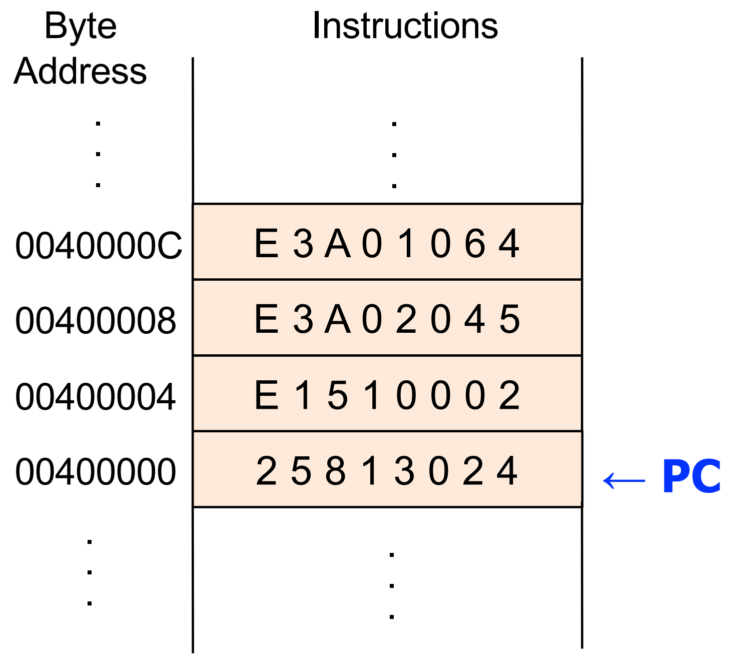
# Branch Instructions and PC

- Update PC to **re-**execute the four instruction sequence again (**for loop**)



## Branch Instructions and PC

- Update PC to **re-**execute the four instruction sequence again (**for loop**)



## Branch Instructions

- Typically, a computer program is executed in sequence
  - First instruction is executed, then the second, then the third, and so on
- **Decision making** is an important advantage of computers
  - `if` and `if-else` statements
  - `for` and `while` loops
  - `switch-case` statements
- ARM provides branch instructions to **skip** and **repeat** code

## Type of Branches

- Branch (B)
  - Branches to another **TARGET** instruction
  - Unconditional branch: always executes the target instruction
  - Conditional branch: either executes the TARGET instruction or the next sequential instruction in memory based on a condition
    - **BEQ** (Branch if the Zero flag is set)
    - **BNE** (Branch if the Zero flag is not set)
- Branch and Link (BL)
  - A special branch to provide support for functions in C++ or Java
    - Architectural support for high-level language needs

## Unconditional Branch

- The **B**ranch in this example is unconditional and **always** TAKEN (T)

```
Assembly code:
   ADD  R1,  R2,  #17
   B    TARGET
   ORR  R1,  R1,  R3
   AND  R3,  R1,  #0xFF
TARGET
   SUB  R1,  R1,  #78
```

- After encountering **B**, the CPU executes SUB instead of ORR

- The **label** TARGET is a **memory address** in human readable form
  - TARGET is transformed into a **memory address** by a tool called **assembler**
  - Assemblers transform assembly code into machine code (**0s and 1s**)

Assembly language let us give meaningful (human-readable and easy to differentiate) symbolic names (labels) to memory locations, such as TARGET, rather than use binary addresses

We call these names **Symbolic Addresses**

## Conditional Branch

- **Conditional** branch uses condition mnemonics

- Recall **conditional** execution and condition mnemonics

# Recall: ARM Condition Mnemonics

| *cond* | Mnemonic | Name | CondEx |
|---|---|---|---|
| 0000 | EQ | Equal | $Z$ |
| 0001 | NE | Not equal | $\bar{Z}$ |
| 0010 | CS / HS | Carry set / Unsigned higher or same | $C$ |
| 0011 | CC / LO | Carry clear / Unsigned lower | $\bar{C}$ |
| 0100 | MI | Minus / Negative | $N$ |
| 0101 | PL | Plus / Positive of zero | $\bar{N}$ |
| 0110 | VS | Overflow / Overflow set | $V$ |
| 0111 | VC | No overflow / Overflow clear | $\bar{V}$ |
| 1000 | HI | Unsigned higher | $\bar{Z}C$ |
| 1001 | LS | Unsigned lower or same | $Z\ OR\ \bar{C}$ |
| 1010 | GE | Signed greater than or equal | $\overline{N \oplus V}$ |
| 1011 | LT | Signed less than | $N \oplus V$ |
| 1100 | GT | Signed greater than | $\bar{Z}(\overline{N \oplus V})$ |
| 1101 | LE | Signed less than or equal | $Z\ OR\ (N \oplus V)$ |
| 1110 | AL (or none) | Always / unconditional | ignored |

## Conditional Branch

- Conditional branch uses condition mnemonics

```
Assembly code:
    MOV  R0,  #4
    ADD  R1,  R0,  R0
    CMP  R0,  R1
    BEQ  THERE
    ORR  R1,  R1,  R1
THERE
    ADD  R1,  R1,  #78
```

- CMP subtracts R1 from R0 and **sets** all flags
  - Z flag is FALSE because R0 – R1 is not 0
- The branch **BEQ** evaluates to FALSE
  - Branch is NOT TAKEN (NT)
  - The next instruction executed is the ORR instruction

# Instruction Format – 3: Branch

| 31:28 | 27:26 | 25:24 | 23:0 |
|---|---|---|---|
| cond | op | 1L | imm24 |

- op = 10
- imm24 = 24-bit **signed** immediate
- The two bits [25:24] form the funct field
  - Bit 25 is always 1
  - **L bit:** L = 0 for **B** (Branch)
  - **L bit:** L = 1 for **BL** (Branch and Link)

- Format

$$\begin{tikzcd}
\mathtt{B} & \mathtt{TARGET} \arrow[d] \\
\mathtt{B} & \mathtt{imm24}
\end{tikzcd}$$

# Branch with L = 0

- Branch with **L** bit (Bit 24) as 0 is a regular branch

| 31:28 | 27:26 | 25:24 | 23:0 |
|---|---|---|---|
| cond | 10 | 10 | imm24 |

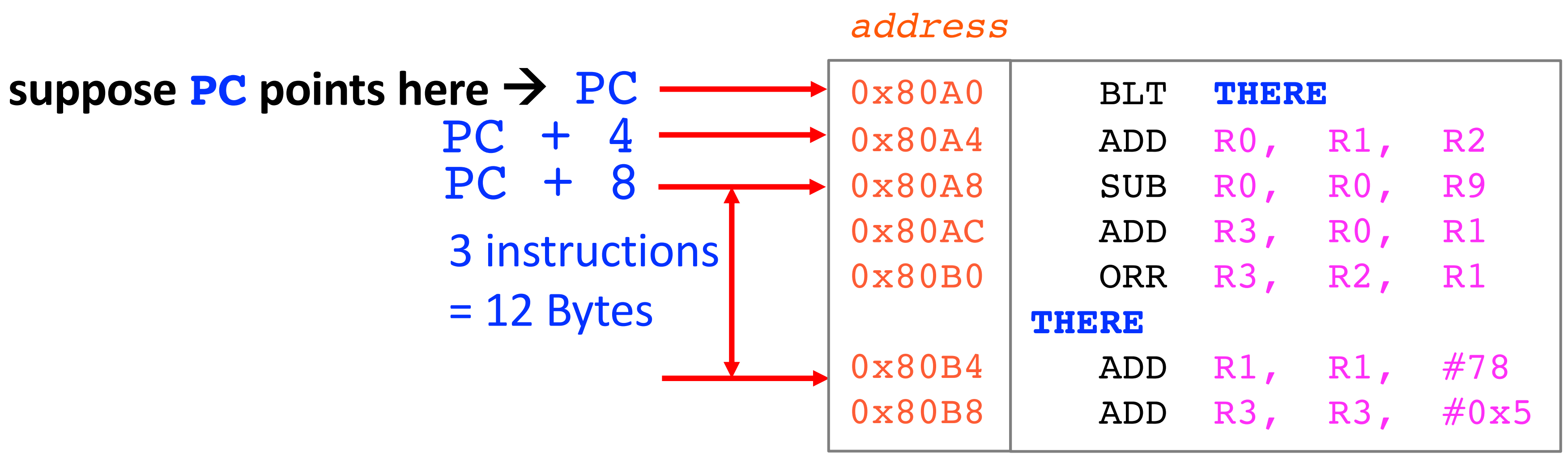
- **Branch Target Address (BTA)**: The address of the next instruction to execute if the branch is taken

- How is **BTA** calculated?
  1. Shift left `imm24` by 2 (to convert **words** to **bytes**)
  2. Sign-extend (copy `Instruction[23]` into `Instruction[24:31]`)
  3. Add `PC + 8`

# BTA Calculation Example

- Instruction encodes the distance from `PC + 8` as 3 32-bit words

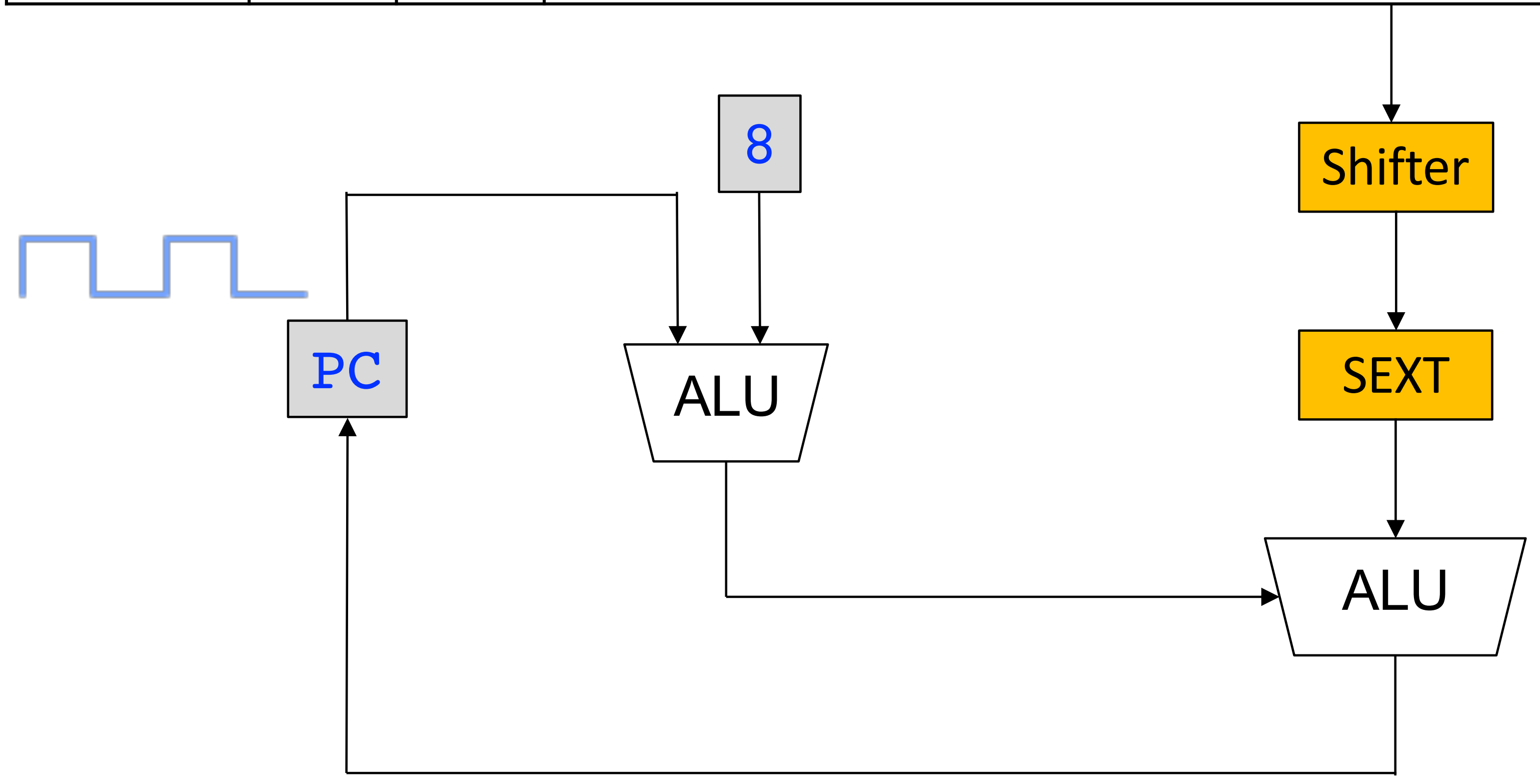
| 31:28 | 27:26 | 25:24 | 23:0 |
|---|---|---|---|
| cond | 10 | 10 | imm24 = 3 (000000000000000000000011) |



```
0x80A0      BLT   THERE
0x80A4      ADD   R0,   R1,   R2
0x80A8      SUB   R0,   R0,   R9
0x80AC      ADD   R3,   R0,   R1
0x80B0      ORR   R3,   R2,   R1
          THERE
0x80B4      ADD   R1,   R1,   #78
0x80B8      ADD   R3,   R3,   #0x5
```

# BTA Calculation DataPath

| 31:28 | 27:26 | 25:24 | 23:0 |
|---|---|---|---|
| cond | 10 | 10 | imm24 = 3 (0000000000000000000000011) |



# BTA Calculation Summary

The processor calculates the **BTA** in three steps

1. Shift left imm24 by 2 (to convert words to bytes)
2. Sign-extend (copy $\text{Instr}_{23}$ into $\text{Instr}_{31:24}$)
3. Add PC + 8

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | **= 3** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | **= 12** |

# Branch-Related Terminology

- **Two main types of branches**
  - Conditional branch: Executes the next sequential instruction or TARGET instruction based on a condition
  - Unconditional branch: Always (unconditionally) executes the TARGET instruction
- **Branch Target**
  - Memory address of the TARGET instruction
- **Branch Condition**
  - Condition which if TRUE branch jumps to the TARGET instruction

- **Branch Resolution/Evaluation**
  - The act of evaluating the branch condition
  - Two outcomes of branch resolution are:
    - Taken Branch (T): branch condition **evaluates** to TRUE
    - Untaken (Not Taken or NT) Branch: branch **evaluates** to FALSE

- **Branch behavior**
  - Strongly (most of times) Taken/Untaken **OR** Weakly (some of the times) Taken/Untaken
  - Always Taken **OR** Always Untaken

- **Branch Prediction**
  - In high-performance CPUs, branches prevent the CPU from doing useful work
  - Modern CPUs use a branch predictor to **predict** the branch **direction** (T/NT) and branch TARGET

## if and if-else

- We will study high-level language (C) to assembly transformation in this course

## The Three Program Constructs

- We will see three basic constructs used in **structured programs** (**construct** comes from **construct**ing a program)

- Sequential ✓
  - One subtask, followed by other, never going back to first

- Conditional
  - One of the two subtasks but not both, depending on some condition

- Iterative
  - Doing a subtask a number of times

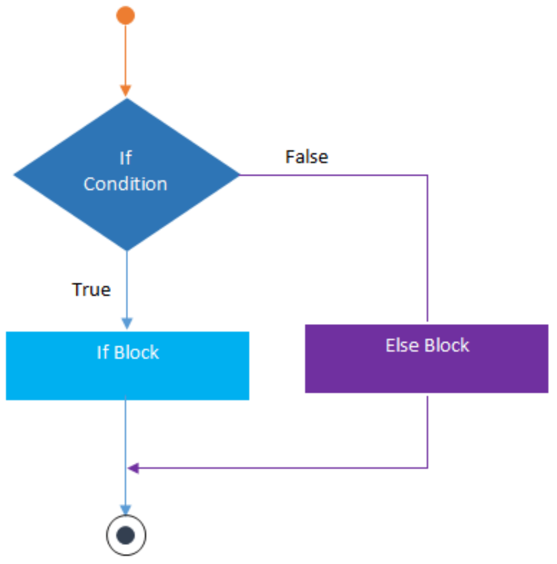
## Conditional Statements

- If the condition is TRUE, do one subtask. Otherwise, do a different subtask

- A subtask or block of code may do nothing

- We call it a **conditional** construct

- All languages provide conditional constructs



If-Else statement

## `if` Statement

```
C code:
   if (apples == oranges)
      f = i + 1;
   f = f – i;
```

```
Assembly code:
; R0 = apples
; R1 = oranges
; R2 = f
; R3 = i
    CMP  R0,  R1              ▪  apples == oranges?
    BNE  L1                   ▪  if yes, branch to L1
    ADD  R2,  R3,  #1         ▪  if no, add 1 to i
L1
    SUB  R2,  R2,  R3         ▪  Subtract i from f
```

- The assembly code checks for the opposite **condition** in C code
- Skips the `if` block when the **condition** is not satisfied
- If the branch is NOT TAKEN, the `if` block is executed

## `if` Statement

- It is very rarely the case that computer programs can be written only one way

  - Use the **BEQ** instruction instead of **BNE**
  - Using conditional execution (next)

## if Statement

```
C code:
   if (apples == oranges)
      f = i + 1;
   f = f – i;
```

```
Assembly code:
; R0 = apples
; R1 = oranges
; R2 = f
; R3 = i
   CMP  R0,  R1          ▪ apples == oranges?
   BEQ  L1               ▪ if yes, branch to L1
   B    L2               ▪ if no, add 1 to i
L1
   ADD  R2,  R3,  #1     ▪ Subtract i from f
L2
   SUB  R2,  R2,  R3
```

- More faithfully translates the high-level code
- If the branch is TAKEN, the `if` block is executed
- There is an extra branch instruction hence worst performance

# if with Conditional Execution

```
C code:
   if (apples == oranges)
      f = i + 1;
   f = f – i;
```

```
Assembly code:
; R0 = apples
; R1 = oranges
; R2 = f
; R3 = i
    CMP     R0,   R1
    ADDEQ   R2,   R3,   #1
    SUB   R2,   R2,   R3
```

- apples == oranges?
- if yes, add 1 to i
- Subtract i from f

- This solution is shorter and faster (one fewer instruction)
- If the if block is long, it is tedious to write conditional mnemonics
- Conditional execution requires NEEDLESS fetching of instructions from memory
- In high-performance CPUs, branch instructions introduce extra delay if the branch predictor makes a mistake (branch misprediction)

## if-else

## `if-else` Statement

```
C code:
   if (apples == oranges)
      f = i + 1;
   else
      f = f – i;
```

```
Assembly code:
; R0 = apples
; R1 = oranges
; R2 = f
; R3 = i
   CMP  R0,  R1
   BNE  L1
   ADD  R2,  R3,  #1
   B    L2
L1
   SUB  R2,  R2,  R3
L2
```

- apples == oranges?
- if yes, branch to L1
- if no, add 1 to i
- Branch to L2
- Subtract i from f

## if-else Statement

```
C code:
  if (apples == oranges)
      f = i + 1;
  else
      f = f – i;
  ...
```

```
Assembly code:
; R0 = apples
; R1 = oranges
; R2 = f
; R3 = i
   CMP   R0,   R1
   BNE   L1
   ADD   R2,   R3,   #1
   B     L2
L1
   SUB   R2,   R2,   R3
L2
```

- apples == oranges?
- if yes, branch to L1
- if no, add 1 to i
- Branch to L2
- Subtract i from f

## if-else Statement

```
C code:
   if (apples == oranges)
      f = i + 1;
   else
      f = f – i;
   ...
```

```
Assembly code:
; R0 = apples
; R1 = oranges
; R2 = f
; R3 = i
   CMP   R0,   R1
   BNE   L1
   ADD   R2,   R3,   #1
   B     L2
L1
   SUB   R2,   R2,   R3
L2
```

- apples == oranges?
- if yes, branch to L1
- if no, add 1 to i
- Branch to L2
- Subtract i from f

## if-else Statement

```
C code:
   if (apples == oranges)
      f = i + 1;
   else
      f = f – i;
   ...
```

```
Assembly code:
; R0 = apples
; R1 = oranges
; R2 = f
; R3 = i
   CMP   R0,   R1
   BNE   L1
   ADD   R2,   R3,   #1
   B     L2
L1
   SUB   R2,   R2,   R3
L2
```

- apples == oranges?
- if yes, branch to L1
- if no, add 1 to i
- Branch to L2
- Subtract i from f

## if-else Statement

```
C code:
   if (apples == oranges)
      f = i + 1;
   else
      f = f – i;
   ...
```

```
Assembly code:
; R0 = apples
; R1 = oranges
; R2 = f
; R3 = i
   CMP  R0,  R1
   BNE  L1
   ADD  R2,  R3,  #1
   B    L2
L1
   SUB  R2,  R2,  R3
L2
```

- apples == oranges?
- if yes, branch to L1
- if no, add 1 to i
- Branch to L2
- Subtract i from f

## if-else Statement

```
C code:
   if (apples == oranges)
      f = i + 1;
   else
      f = f – i;
   ...
```

```
Assembly code:
; R0 = apples
; R1 = oranges
; R2 = f
; R3 = i
   CMP  R0,  R1
   BNE  L1
   ADD  R2,  R3,  #1
   B    L2
L1
   SUB  R2,  R2,  R3
L2
```

- apples == oranges?
- if yes, branch to L1
- if no, add 1 to i
- Branch to L2
- Subtract i from f

## `if-else` Statement

```
C code:
   if (apples == oranges)
      f = i + 1;
   else
      f = f – i;
   ...
```

```
Assembly code:
; R0 = apples
; R1 = oranges
; R2 = f
; R3 = i
   CMP  R0,  R1
   BNE  L1
   ADD  R2,  R3,  #1
   B    L2
L1
   SUB  R2,  R2,  R3
L2
```

- apples == oranges?
- if yes, branch to L1
- if no, add 1 to i
- Branch to L2
- Subtract i from f

## `if-else` Statement

- It is very rarely the case that computer programs can be written only one way

- **Do it yourself:** Find an alternative way to write the `if-else` statement

## `if-else` with Conditional Execution

```
C code:
   if (apples == oranges)
      f = i + 1;
   else
      f = f – i;
```

```
Assembly code:
; R0 = apples
; R1 = oranges
; R2 = f
; R3 = i
   CMP      R0,  R1
   ADDEQ    R2,  R3,  #1
   SUBNE    R2,  R2,  R3
```

- This solution is shorter and faster (one fewer instruction)
- Suppose the `if` block is long, it is then tedious to write conditional mnemonics
- Conditional execution requires NEEDLESS fetching of instructions from memory
- On the other hand, in high-performance CPUs, branch instructions introduce extra delay if the branch predictor makes a mistake (branch misprediction)

## Switch Statement

## switch-case Statement

```
C code:
   switch (button) {
      case 1:  atm = 20;  break;
      case 2:  atm = 50;  break;
      case 3:  atm = 100; break;
      default: atm = 0;   break;
   }
```

- Execute one of several blocks of code (**cases**) depending on the condition
- Break out of the entire **switch** block {...} after executing a specific block
- In the above example condition is the state of variable button
- If no conditions are met, the default block is executed

## switch-case Statement

```
C code:
   switch (button) {
      case 1:  atm = 20;  break;
      case 2:  atm = 50;  break;
      case 3:  atm = 100; break;
      default: atm = 0;   break;
   }
```

```
Assembly code:
; R0 = button
; R1 = atm
   CMP   R0,  #1
   MOVEQ R1,  #20
   BEQ   DONE
   CMP   R0,  #2
   MOVEQ R1,  #50
   BEQ   DONE
   CMP   R0,  #3
   MOVEQ R1,  #100
   BEQ   DONE
   MOV   R1,  #0
DONE
```

- Comment begins with ;
- Another comment
- is button == 1?
- atm = 20
- break out
- is button == 2?
- atm = 50
- break out
- is button == 3?
- atm = 100
- break out
- Execute default case

# switch-case Statement

```
C code:
   switch (button) {
      case 1:   atm = 20;  break;
      case 2:   atm = 50;  break;
      case 3:   atm = 100; break;
      default:  atm = 0;   break;
   }
```

```
Assembly code:
; R0 = button                  ▪ Comment begins with ;
; R1 = atm                     ▪ Another comment
   CMP   R0,  #1               ▪ is button == 1?
   MOVEQ R1,  #20              ▪ atm = 20
   BEQ   DONE                  ▪ break out
   CMP   R0,  #2               ▪ is button == 2?
   MOVEQ R1,  #50              ▪ atm = 50
   BEQ   DONE                  ▪ break out
   CMP   R0,  #3               ▪ is button == 3?
   MOVEQ R1,  #100             ▪ atm = 100
   BEQ   DONE                  ▪ break out
   MOV   R1,  #0               ▪ Execute default case
DONE
```

# switch-case Statement

```
C code:
   switch (button) {
      case 1:   atm = 20;  break;
      case 2:   atm = 50;  break;
      case 3:   atm = 100; break;
      default:  atm = 0;   break;
   }
```

```
Assembly code:
; R0 = button
; R1 = atm
   CMP   R0,  #1
   MOVEQ R1,  #20
   BEQ   DONE
   CMP   R0,  #2
   MOVEQ R1,  #50
   BEQ   DONE
   CMP   R0,  #3
   MOVEQ R1,  #100
   BEQ   DONE
   MOV   R1,  #0
DONE
```

- Comment begins with ;
- Another comment
- is button == 1?
- atm = 20
- break out
- is button == 2?
- atm = 50
- break out
- is button == 3?
- atm = 100
- break out
- Execute default case

## switch-case Statement

```
C code:
   switch (button) {
      case 1:  atm = 20;  break;
      case 2:  atm = 50;  break;
      case 3:  atm = 100; break;
      default: atm = 0;   break;
   }
```

```
Assembly code:
; R0 = button                  ▪ Comment begins with ;
; R1 = atm                     ▪ Another comment
   CMP    R0,  #1              ▪ is button == 1?
   MOVEQ  R1,  #20             ▪ atm = 20
   BEQ    DONE                 ▪ break out
   CMP    R0,  #2              ▪ is button == 2?
   MOVEQ  R1,  #50             ▪ atm = 50
   BEQ    DONE                 ▪ break out
   CMP    R0,  #3              ▪ is button == 3?
   MOVEQ  R1,  #100            ▪ atm = 100
   BEQ    DONE                 ▪ break out
   MOV    R1,  #0              ▪ Execute default case
DONE
```

## switch-case Statement

```
C code:
   switch (button) {
      case 1:  atm = 20;  break;
      case 2:  atm = 50;  break;
      case 3:  atm = 100; break;
      default: atm = 0;   break;
   }
```

```
Assembly code:
; R0 = button                 ▪ Comment begins with ;
; R1 = atm                    ▪ Another comment
   CMP    R0,  #1             ▪ is button == 1?
   MOVEQ  R1,  #20            ▪ atm = 20
   BEQ    DONE                ▪ break out
   CMP    R0,  #2             ▪ is button == 2?
   MOVEQ  R1,  #50            ▪ atm = 50
   BEQ    DONE                ▪ break out
   CMP    R0,  #3             ▪ is button == 3?
   MOVEQ  R1,  #100           ▪ atm = 100
   BEQ    DONE                ▪ break out
   MOV    R1,  #0             ▪ Execute default case
DONE
```

We will cover loops and arrays after the teaching break

Next: Microarchitecture

## For Loop

## Loops

- Life is full of **repetition!**
  - Standard routines **repeat** each day, week, month, ...
  - **Terminating** at some point

- Repetition (iteration) is also the essence of computing!
  - Compute the sum of first one billion numbers
  - Go over each student record and change numerical grade to letter
    - Terminate if no more records are found

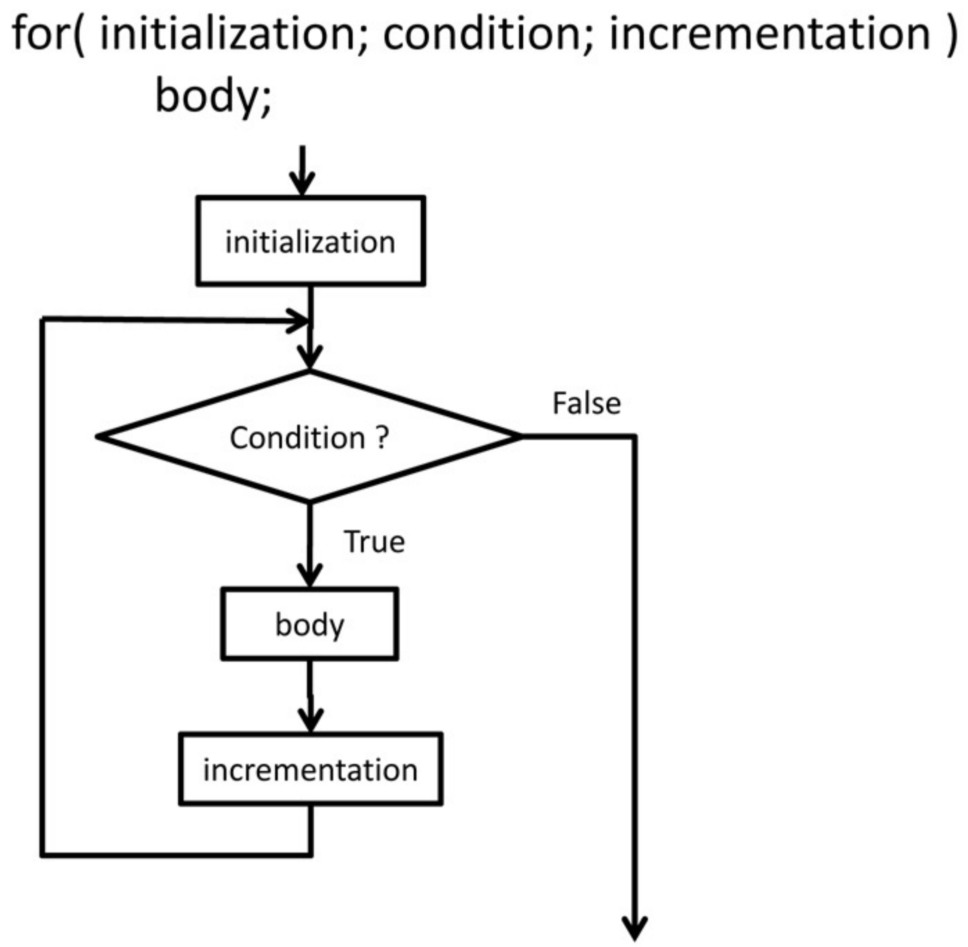
- CPUs are very good at looping sometimes but not always depending on a condition!

## Loops

- Loops are **iterative** constructs that repeat a subtask several times, but only as long as some condition is TRUE (subtask = sequence of instructions)
- If the condition is TRUE, do the **subtask** (also called **loop body**)
- After the subtask is finished, go back and check the condition again
- As long as the result of the condition is TRUE, the program continues to carry out the same subtask again and again
- The first time the test is NOT TRUE, the program proceeds onward

## Loops

- Loops are **iterati** ral times, but only as long as sc
- If the condition i: ody)
- After the subtask :ion again
- As long as the re: ı continues to carry out the san
- The first time the : onward



## Loops

- We will look at
    - For Loop
    - While Loop
- **Our focus**
    - How are loops in high-level languages **transformed** (**translated**) into assembly by human or compiler?

## For Loop in C

```
C code:
   int i;
   int sum = 0;

   for (i = 0; i < 10; i = i + 1)
      sum = sum + i;
   ...
   ...
```

- The variable "**i**" is called the loop index or counter
- The For **statement** has three components
  - i = 0 : index initialization
  - i < 10 : loop termination condition
  - i = i + 1 : loop advancement
- The **body** of the loop can have one or more statements

## For Loop in ARM Assembly

```
C code:
   int i;
   int sum = 0;

   for (i = 0; i < 10; i = i + 1)
      sum = sum + i;
   ...
   ...
```

```
Assembly code:
; R0 = i                   ▪ Comment begins with ;
; R1 = sum                 ▪ Another comment
   MOV  R0,  #0            ▪ Initialize i
   MOV  R1,  #0            ▪ Initialize sum
FOR                        ▪ Label/Address of CMP
   CMP  R0,  #10           ▪ check condition:i<10 ?
   BGE  DONE               ▪ if (i>=10) exit loop
   ADD  R1,  R1,  R0       ▪ sum = sum + i
   ADD  R0,  R0,  #1       ▪ Increment i
   B    FOR                ▪ repeat loop
DONE
```

- **High-level code:** Few lines (statements); **Assembly code:** Many lines (instructions)
- **High-level code:** Variable names; **Assembly code:** Registers & memory addresses
- **High-level code:** Hides machine details (e.g., **MOV**ement); **ASM:** Expose details
- In both C and assembly, the **control flow** (sequential and iterative constructs) are visible
  - Easier to identify in C, more difficult in assembly
- Let’s do a line-by-line comparison of the above code ...

# For Loop in ARM Assembly

```
C code:
   int i;
   int sum = 0;

   for (i = 0; i < 10; i = i + 1)
      sum = sum + i;
   ...
   ...
```

```
Assembly code:
; R0 = i
; R1 = sum
   MOV   R0,  #0          ▪ Comment begins with ;
   MOV   R1,  #0          ▪ Another comment
FOR                       ▪ Initialize i
   CMP   R0,  #10         ▪ Initialize sum
   BGE   DONE             ▪ Label/Address of CMP
   ADD   R1,  R1,  R0     ▪ check condition:i<10 ?
   ADD   R0,  R0,  #1     ▪ if (i>=10) exit loop
   B     FOR              ▪ sum = sum + i
DONE                      ▪ Increment i
                          ▪ repeat loop
```

- In high-level language programs, we initialize variables
  - In assembly initializing variables translates to initializing registers

# For Loop in ARM Assembly

```
C code:
   int i;
   int sum = 0;

   for (i = 0; i < 10; i = i + 1)
      sum = sum + i;
   ...
   ...
```

```
Assembly code:
; R0 = i                      ▪ Comment begins with ;
; R1 = sum                    ▪ Another comment
   MOV  R0,  #0               ▪ Initialize i
   MOV  R1,  #0               ▪ Initialize sum
FOR                           ▪ Label/Address of CMP
   CMP  R0,  #10              ▪ check condition:i<10 ?
   BGE  DONE                  ▪ if (i>=10) exit loop
   ADD  R1,  R1,  R0          ▪ sum = sum + i
   ADD  R0,  R0,  #1          ▪ Increment i
   B    FOR                   ▪ repeat loop
DONE
```

## For Loop in ARM Assembly

```
C code:
   int i;
   int sum = 0;

   for (i = 0; i < 10; i = i + 1)
      sum = sum + i;
   ...
   ...
```

```
Assembly code:
; R0 = i
; R1 = sum
   MOV   R0,  #0          ▪ Comment begins with ;
   MOV   R1,  #0          ▪ Another comment
FOR                       ▪ Initialize i
   CMP   R0,  #10         ▪ Initialize sum
   BGE   DONE             ▪ Label/Address of CMP
   ADD   R1,  R1,  R0     ▪ check condition:i<10 ?
   ADD   R0,  R0,  #1     ▪ if (i>=10) exit loop
   B     FOR              ▪ sum = sum + i
DONE                      ▪ Increment i
                          ▪ repeat loop
```

- Check termination condition to break out of the loop if condition is met

## For Loop in ARM Assembly

```
C code:
   int i;
   int sum = 0;

   for (i = 0; i < 10; i = i + 1)
      sum = sum + i;
   ...
   ...
```

```
Assembly code:
; R0 = i                       ▪ Comment begins with ;
; R1 = sum                     ▪ Another comment
   MOV   R0,   #0              ▪ Initialize i
   MOV   R1,   #0              ▪ Initialize sum
FOR                            ▪ Label/Address of CMP
   CMP   R0,   #10             ▪ check condition:i<10 ?
   BGE   DONE                  ▪ if (i>=10) exit loop
   ADD   R1,   R1,   R0        ▪ sum = sum + i
   ADD   R0,   R0,   #1        ▪ Increment i
   B     FOR                   ▪ repeat loop
DONE
```

- Add the loop counter `i` to the variable `sum`

## For Loop in ARM Assembly

```
C code:
   int i;
   int sum = 0;

   for (i = 0; i < 10; i = i + 1)
      sum = sum + i;
   ...
   ...
```

```
Assembly code:
; R0 = i                    ▪ Comment begins with ;
; R1 = sum                  ▪ Another comment
   MOV  R0,  #0             ▪ Initialize i
   MOV  R1,  #0             ▪ Initialize sum
FOR                         ▪ Label/Address of CMP
   CMP  R0,  #10            ▪ check condition:i<10 ?
   BGE  DONE                ▪ if (i>=10) exit loop
   ADD  R1,  R1,  R0        ▪ sum = sum + i
   ADD  R0,  R0,  #1        ▪ Increment i
   B    FOR                 ▪ repeat loop
DONE
```

- Increment the loop counter

## For Loop in ARM Assembly

```
C code:
   int i;
   int sum = 0;

   for (i = 0; i < 10; i = i + 1)
      sum = sum + i;
   ...
   ...
```

```
Assembly code:
; R0 = i
; R1 = sum
   MOV  R0,  #0              ▪ Comment begins with ;
   MOV  R1,  #0              ▪ Another comment
FOR                          ▪ Initialize i
   CMP  R0,  #10             ▪ Initialize sum
   BGE  DONE                 ▪ Label/Address of CMP
   ADD  R1,  R1,  R0         ▪ check condition:i<10 ?
   ADD  R0,  R0,  #1         ▪ if (i>=10) exit loop
   B    FOR                  ▪ sum = sum + i
DONE                         ▪ Increment i
                             ▪ repeat loop
```

- Keep iterating by branching back to the CMP instruction

## For Loop in ARM Assembly

```
C code:
   int i;
   int sum = 0;

   for (i = 0; i < 10; i = i + 1)
      sum = sum + i;
   ...
   ...
```

```
Assembly code:
; R0 = i                          ▪ Comment begins with ;
; R1 = sum                        ▪ Another comment
   MOV  R0,  #0                   ▪ Initialize i
   MOV  R1,  #0                   ▪ Initialize sum
FOR                               ▪ Label/Address of CMP
   CMP  R0,  #10                  ▪ check condition:i<10 ?
   BGE  DONE                      ▪ if (i>=10) exit loop
   ADD  R1,  R1,  R0              ▪ sum = sum + i
   ADD  R0,  R0,  #1              ▪ Increment i
   B    FOR                       ▪ repeat loop
DONE
```

- Keep iterating by branching back to the CMP instruction

## Same `For` Loop in a Different Style

- Let’s see the same for loop translated using a different style

## Same For Loop in a Different Style

```
C code:
   int i;
   int sum = 0;

   for (i = 0; i < 10; i = i + 1)
      sum = sum + i;
```

```
Assembly code:
; R0 = i
; R1 = sum
   MOV  R0,  #0
   MOV  R1,  #0
COND
   CMP  R0,  #10       ▪ check condition
   BLT  FOR            ▪ if i<10 repeat
   B    DONE           ▪ if i>=10, leave for
FOR
   ADD  R1,  R1,  R0   ▪ add sum to i
   ADD  R0,  R0,  #1   ▪ Increment i
   B    COND           ▪ Iterate again
DONE
```

- More faithfully follows the for loop semantics in C
- Use **BLT** instead of **BGE**
- Different ways to translate a high-level statement into ASM

# Aside: Syntax versus Semantics

- **Syntax:** **Arrangement** of keywords in a statement
  - There is a ; after a statement
  - The loop statement uses parentheses

```
C code:
   int i;
   int sum = 0;

   for (i = 0; i < 10; i = i + 1)
      sum = sum + i;
```

- **Semantics:** **Meaning** of keywords and their arrangement
  - Repeat the instructions in the loop body until condition is not met
  - Add **`sum`** to **`i`**
  - What the CPU does depends on statement and instruction semantics

- Without rules of syntax, it would be tedious to understand programmer's intention
- Without clearly defined **instruction semantics**: difficult to write programs to solve specific problems & to build CPUs that do "right" thing

## Different way to solve the same problem, more efficient translation

- Let’s sum numbers from 0 – 9 in a different way

- And see if it helps **reducing the number of instructions** required for translation

# Decremented Loop

```
C code:
   int i;
   int sum = 0;

   for (i = 9; i >= 0; i = i - 1)
      sum = sum + i;
```

```
Assembly code:
; R0 = i
; R1 = sum
   MOV  R0,  #9
   MOV  R1,  #0
FOR
   ADD  R1,  R1,  R0     ▪ add sum to i
   SUBS R0,  R0,  #1     ▪ i-- and set flags
   BNE  FOR              ▪ if i!=0 keep looping
DONE
```

- **Saves 2 instructions per iteration compared to optimized (increment) version**
  - Decrement loop variable & compare: **`SUBS R0, R0, #1`**
  - Only 1 branch instead of 2
- MANY ways to solve (transform) a high-level problem into assembly
  - **Code Optimization:** A sub-field of Compilers
    - Aims to minimize total instruction count, branch instruction count, and maximize register utilization (to avoid frequent trips to memory)

## For Loop

- Repeat **TEN** times: **add** `10` to `R1`
  - What is wrong with the code below (one way to think of a FOR loop)?

```
ADD   R1,   R1,   #10
ADD   R1,   R1,   #10
ADD   R1,   R1,   #10
ADD   R1,   R1,   #10
ADD   R1,   R1,   #10
ADD   R1,   R1,   #10
ADD   R1,   R1,   #10
ADD   R1,   R1,   #10
ADD   R1,   R1,   #10
ADD   R1,   R1,   #10
```

- Poor practice
- Code is not reusable
  - Next time it may be 20 not 10
- Instructions cost Memory!!
  - Each instruction is stored in memory and has an address
  - Memory is expensive!
  - Fast Instruction Cache built out of SRAM inside CPU is very premium
- How many instructions for above with a For loop using branch instruction?

## While Loop

## While Loop in C

- While loops **iterate** a number of times until the **“controlling condition”** or **sentinel** is NOT met (FALSE)

```
C code:
   while (CONDITION) {
      ...
      ...
   }
```

- Special cases of while loops: execute forever (left) and never (right)

```
C code:
   while (TRUE) {
      ...
      ...
   }
```

```
C code:
   while (FALSE) {
      ...
      ...
   }
```

## Example While Loop

- Determine X such that $2^X = 128$

```
C code:
    int POW = 1;
    int X = 0;

    while (POW != 128) {
        POW = POW * 2;
        X = X + 1;
    }
```

```
Assembly code:
; R0 = POW
; R1 = X
    MOV  R0,  #1              ▪  loop initialization
    MOV  R1,  #0              ▪  POW = 1
WHILE                         ▪  X = 0
    CMP  R0,  #128
    BEQ  DONE                 ▪  POW != 128?
    LSL  R0,  R0,  #1         ▪  if POW == 128, exit loop
    ADD  R1,  R1,  #1         ▪  POW = POW * 2
    B    WHILE                ▪  X = X + 1
DONE                          ▪  repeat loop
```

Sometimes a QUEUE makes sense...
quack add!
quack remove!
...sometimes just a SET is enough.
quack remove!
Eventually a MAP is necessary...
nick
bob
rick
pingo
cris
juan
quack! put
...but sooner or later we all need to deal with a STACK.
BILLS
POP!

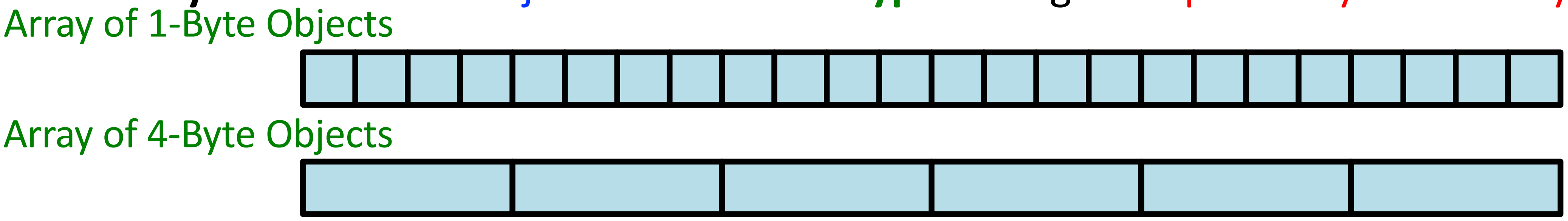
# Arrays

**Data Structure:** Collection of data values organized in a particular way in memory for ease of storage and access. Two aspects: organization and functions to read and update values

Examples: Array, Linked List, Stack, Queue

# What is an Array?

- **Array:** A list of data objects of the same **type** arranged sequentially in memory



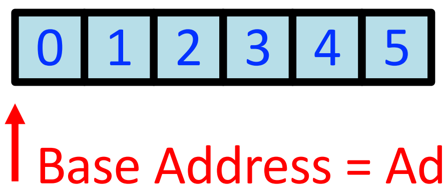
- A data object is a memory location whose content represent “some” value

  - Post office box can store letters, Amazon gifts, pamphlets (all these are pkgs. **types**)
  - How do we know *interpret* the **type** of what is stored in the box?
    - Either we know what we placed there, or we know how to look up the **type**

- The interpretation of the value in memory depends on its **type**
  - 8-Byte Unsigned Integers (`unsigned int`)
  - 4-Byte 2’s Complement Integers (`int`)
  - A 12-Byte student record with `{uint student_Id, int grade}`

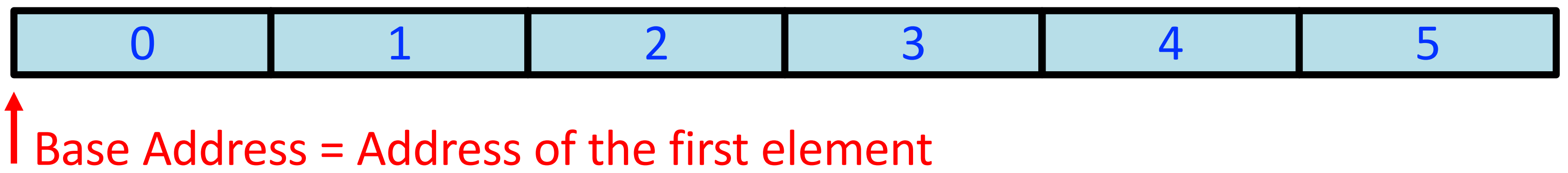
# Array in Memory

- The array below has six elements and each element in a single `byte`
  - The **index** of the first element (`byte`) is 0, then 1, then 2, ....
- It's **base** (starting) address in memory is 0
  - The address of the first element is 0, second element is 1, last element is 5



Base Address = Address of the first element

- Another array with six elements



Base Address = Address of the first element

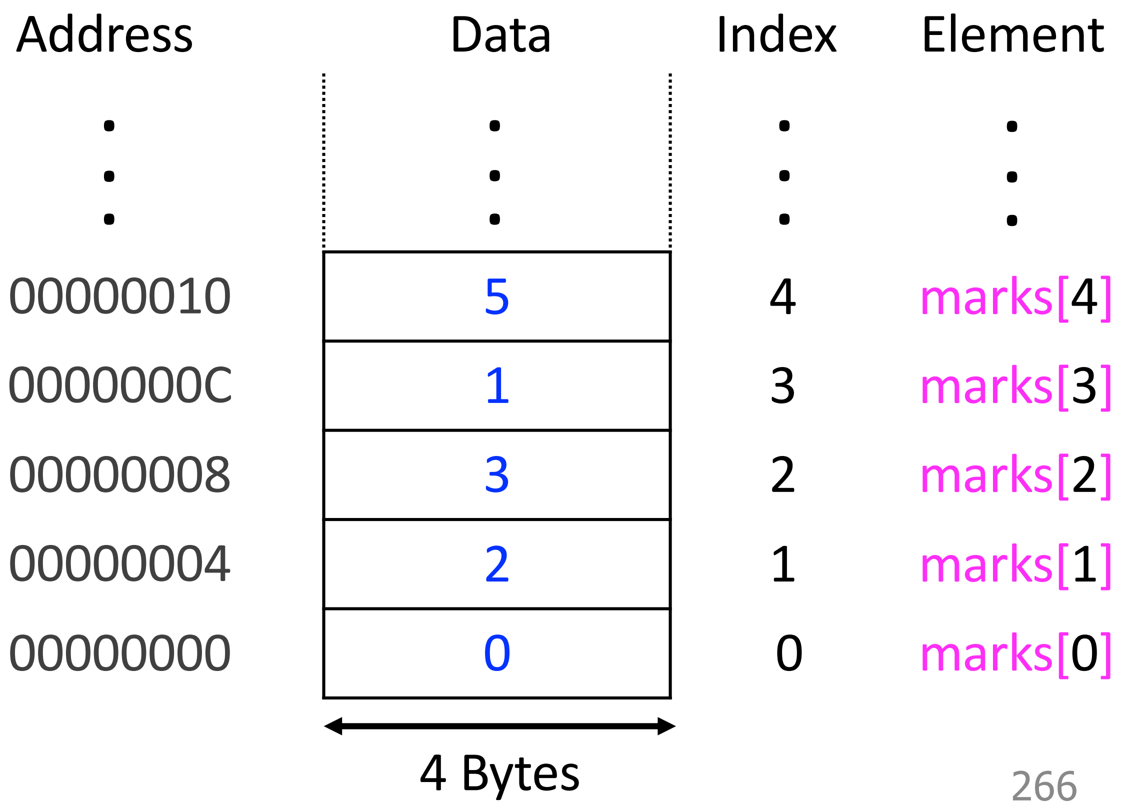
- Same starting address as the first array and same indexing scheme (0, 1, 2, ...)
- Addresses of array elements in memory are different
  - Second element is at an offset 4, last one at 20. Offsets are in bytes

## Array Syntax in C

- Arrays contain a collection of similarly typed elements
- Elements are stored contiguously in memory

**int** is 4 bytes on most architectures

```
C code:
   int marks[5] = {0, 2, 3, 1, 5};
   int a = marks[0];
   marks[3] = 10;
```



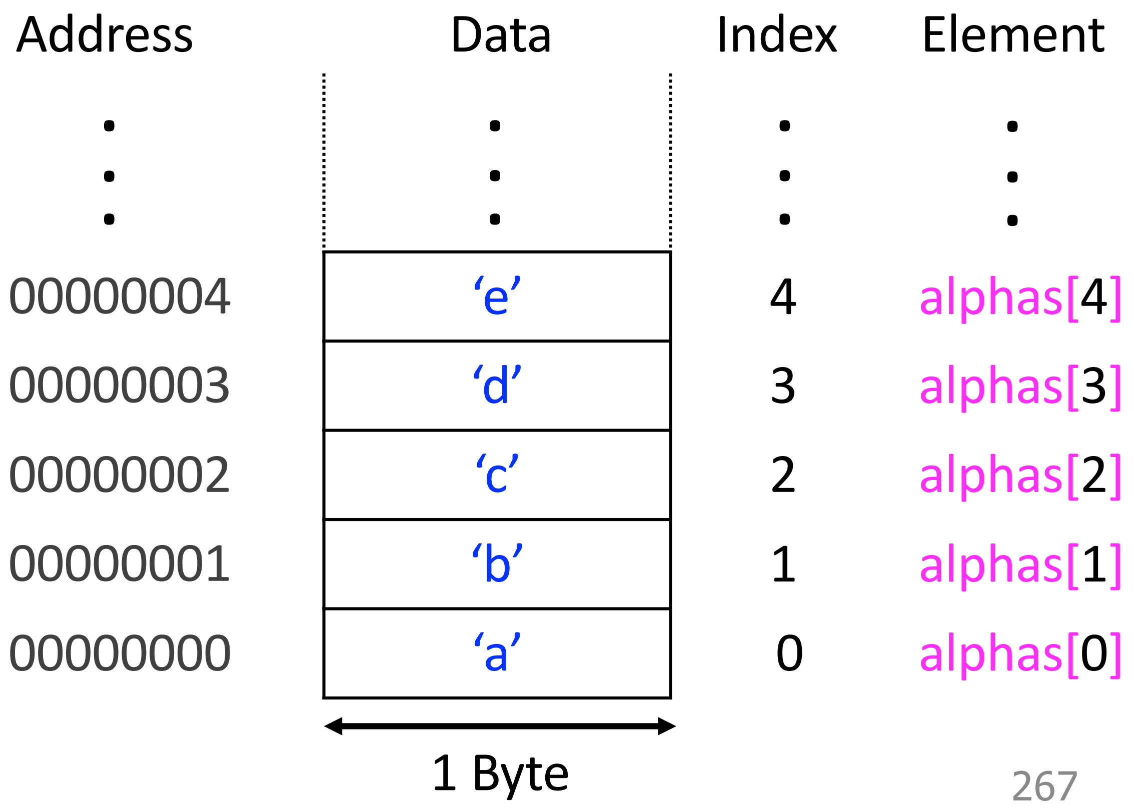
# Array of Characters

- Array of **char**acters (**char** is a data type in C)
- **char** is used for representing characters

**char** is always 1 byte

C code:

```
char alphas[5] = {'a', 'b', 'c', 'd', 'e'};
```



## Example Array in C

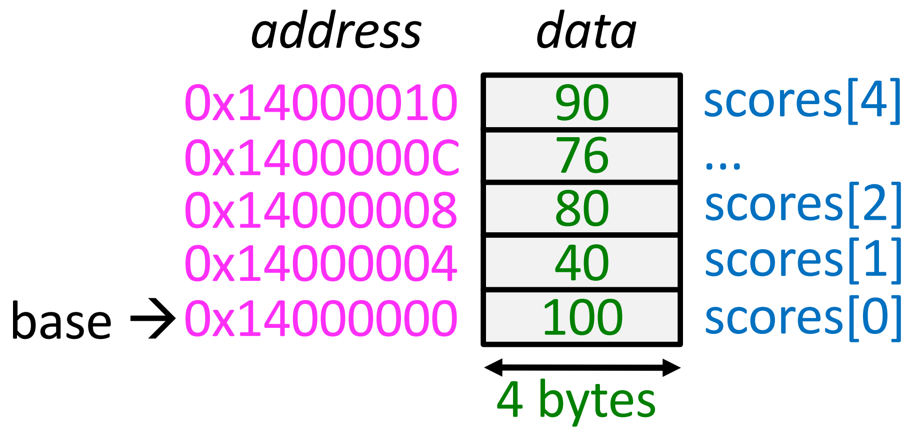
Add 10 to each element of the 200-element scores array

```
C code:
   int i;
   int scores[200];
   // initialization code not
   //shown
   ...
   for (i = 0; i<200; i++)
      scores[i] = scores[i] + 10;
```

## Array Sum

Add 10 to each element of the 200-element scores array

```
C code:
   int i;
   int scores[200];
   // initialization code not
   //shown
   ...
   for (i = 0; i<200; i++)
      scores[i] = scores[i] + 10;
```

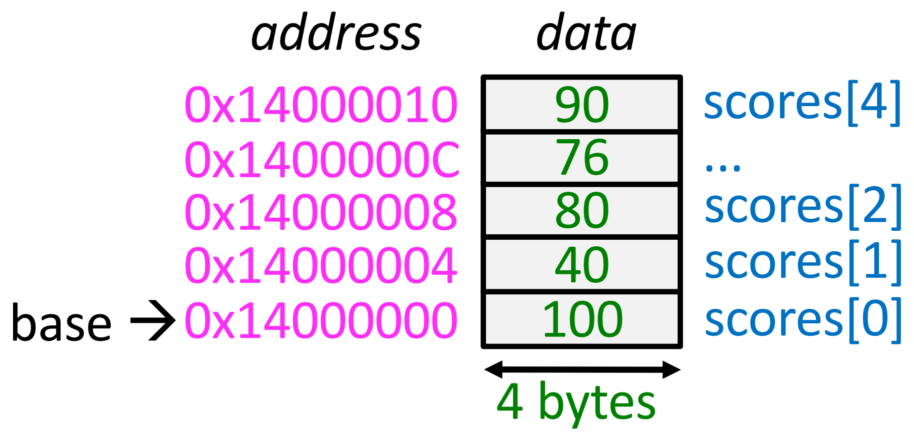


*Showing the scores array in memory*

# Array Sum

Add 10 to each element of the 200-element scores array

```
C code:
   int i;
   int scores[200];
   // initialization code not
   //shown
   ...
   for (i = 0; i<200; i++)
      scores[i] = scores[i] + 10;
```



*Showing the scores array in memory*

```
Assembly code:
; R0 = array base address
; R1 = i
   MOV  R0,  #0x14000000          ▪ R0 = base addr
   MOV  R1,  0                    ▪ i = 0
LOOP
   CMP  R1,  #200                 ▪ i < 200?
   BGE  L3                        ▪ no? exit loop
   LSL  R2,  R1,  #2              ▪ word to byte
   LDR  R3,  [R0, R2]             ▪ R3 = scores[i]
   ADD  R3,  R3,  #10             ▪ R3 = R3 + 10
   STR  R3,  [R0, R2]             ▪ scores[i] += 10
   ADD  R1,  R1,  #1              ▪ i = i + 1
   B    LOOP                      ▪ repeat
L3
```

## LDR with Offset in Register

- New **LDR variant**

$$\begin{tikzcd} \texttt{LDR} & \texttt{R3,} \arrow[d, "\text{dest}"] & \texttt{[R0,} \arrow[d, "\text{base}"] & \texttt{R2]} \arrow[d, "\text{offset}"] \\ \texttt{LDR} & \texttt{Rd,} & \texttt{[Rn,} & \texttt{Rm]} \end{tikzcd}$$

- It is common to load from memory with **[base + offset] addressing mode**, where offset increments by “some” value during each loop iteration

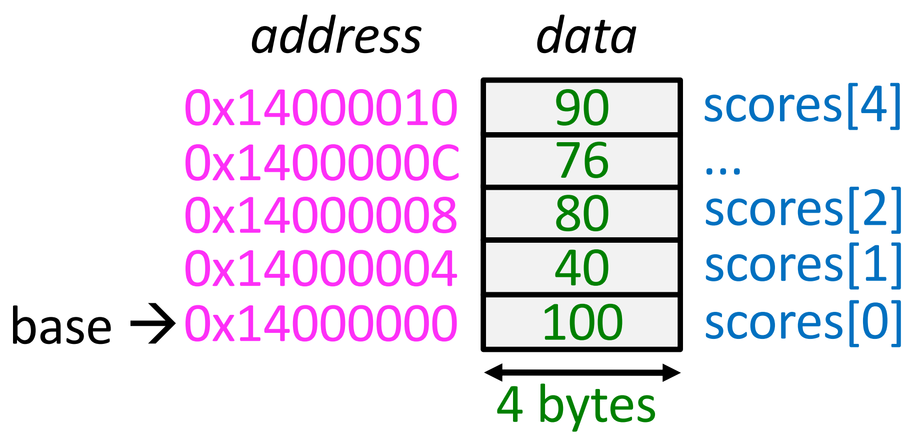
- ISA provides **support** for such scenarios to bridge the semantic gap b/w high-level code and assembly code
  - ISA evolution eases the software “burden”
  - On the other hand, ISA implementation (i.e., microarchitecture) becomes more involved (recall the **RISC vs. CISC** debate)

# Array Sum

Add 10 to each element of the 200-element scores array

```
C code:
   int i;
   int scores[200];
   // initialization code not
   //shown
   ...
   for (i = 0; i<200; i++)
      scores[i] = scores[i] + 10;
```

```
Assembly code:
; R0 = array base address
; R1 = i
   MOV   R0,   #0x14000000         ▪  R0 = base addr
   MOV   R1,   #0                  ▪  i = 0
LOOP
   CMP   R1,   #200                ▪  i < 200?
   BGE   L3                        ▪  no? exit loop
   LSL   R2,   R1,   #2            ▪  word to byte
   LDR   R3,   [R0,  R2]           ▪  R3 = scores[i]
   ADD   R3,   R3,   #10           ▪  R3 = R3 + 10
   STR   R3,   [R0,  R2]           ▪  scores[i] += 10
   ADD   R1,   R1,   #1            ▪  i = i + 1
   B     LOOP                      ▪  repeat
L3
```



*Showing the scores array in memory*

# Another LDR Variant

- We have seen two LDR variants
  - `LDR Rd, [Rn,  #imm]`
  - `LDR Rd, [Rn,  Rm]`

- `LSL` and `LDR` are often used together in array-related code (array traversals)
- ISA provides support for eliminating the extra `LSL` instruction

$$\texttt{LDR R3, [R0, R1, }\underbrace{\texttt{LSL \#2]}}_{\textbf{Left shift is the same as multiplying by 2}}$$

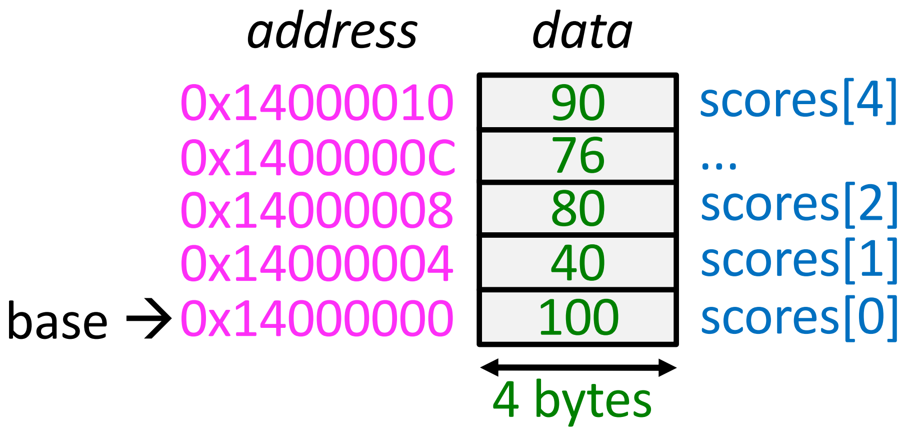
- **Memory address**
  - Left shift `R1` by 2 (scaling `R1`)
  - Add `R1` to `R0`
  - Address = `R0 + (R1 * 4)`

## Condensing Array Sum – 1

Add 10 to each element of the 200-element scores array

```
C code:
   int i;
   int scores[200];
   // initialization code not
   //shown
   ...
   for (i = 0; i<200; i++)
      scores[i] = scores[i] + 10;
```

```
Assembly code:
; R0 = array base address
; R1 = i
   MOV   R0,   #0x14000000
   MOV   R1,   #0
LOOP
   CMP   R1,   #200
   BGE   L3
   LDR   R3,   [R0, R1, LSL, #2]
   ADD   R3,   R3,   #10
   STR   R3,   [R0, R2]
   ADD   R1,   R1,   #1
   B     LOOP
L3
```



*Showing the scores array in memory*

## ARM Indexing Modes

- **Offset Addressing**
  - Address is the sum of base register and offset (`#20, #–20, –R2`)
  - Base register is unchanged
  - `LDR R0, [R1, R2]`
- **Pre-indexed Addressing**
  - Address is the sum of base register and offset
  - Base register is **updated** with the new address **before** the memory access
  - `LDR R0, [R1, R2]!`
- **Post-index Addressing**
  - Address is the base register
  - Base register is updated with the new address **after** the memory access
  - `LDR R0, [R1], R2`

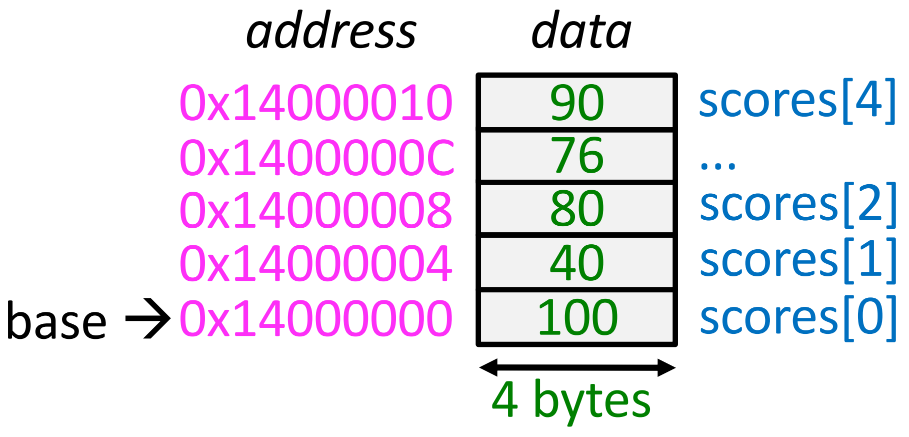
## Examples: ARM Indexing Modes

- **Offset Addressing**
  - `LDR R0, [R1, R2]`
    - **Address:** `R1 + R2` and `R1` does not change
- **Pre-indexed Addressing**
  - `LDR R0, [R1, R2]!`
    - **Address:** `R1 + R2` and `R1 = R1 + R2`
- **Post-index Addressing**
  - `LDR R0, [R1], R2`
    - **Address:** `R1` and `R1 = R1 + R2`
- **Note:** In all cases, offset can be an immediate

# Condensing Array Sum – 2

Add 10 to each element of the 200-element scores array

```
C code:
   int i;
   int scores[200];
   // initialization code not
   //shown
   ...
   for (i = 0; i<200; i++)
      scores[i] = scores[i] + 10;
```



*Showing the scores array in memory*

```
Assembly code:
; R0 = array base address
; R1 = i
   MOV  R0,  #0x14000000      ▪ R0 = base addr
   ADD  R1,  R0,  #800        ▪ R1 = base + 800
LOOP
   CMP  R0,  R1               ▪ end of array?
   BGE  L3                    ▪ yes? exit loop
   LDR  R2,  [R0]             ▪ R2 = scores[i]
   ADD  R2,  R2,  #10         ▪ scores[i] + 10
   STR  R2,  [R0], #4         ▪ store scores[i]
                              ▪ and R0 = R0 + 4
   B    LOOP                  ▪ repeat loop
L3
```

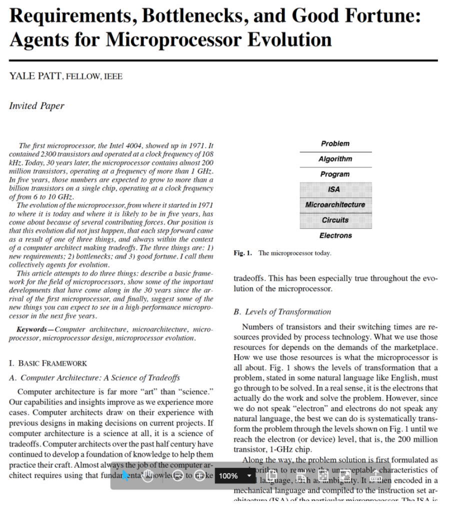
## Condensing Array Sum – 2

Add 10 to each element of the 200-element scores array

```
Assembly code:
; R0 = array base address
; R1 = i
   MOV  R0,  #0x14000000
   ADD  R1,  R0,  #800
LOOP
   CMP  R0,  R1
   BGE  L3
   LDR  R2,  [R0]
   ADD  R2,  R2,  #10
   STR  R2,  [R0], #4
   B    LOOP
L3
```

- This version of `Array Sum` first computes the address of the last byte of the array (`#0x14000800`)
- Each iteration of `LOOP` checks if `R0` is `greater than or equal to #0x14000800`
- If so, we are done, so step out of `LOOP`
- `STR R2, [R0], #4`
  - Stores `R2` at `[R0]`, and after that, adds 4 to `R0`

# Microarchitecture



**Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution**

YALE PATT, FELLOW, IEEE

*Invited Paper*

*The first microprocessor, the Intel 4004, showed up in 1971. It contained 2300 transistors and operated at a clock frequency of 108 kHz. Today, 30 years later, the microprocessor contains almost 200 million transistors, operating at a frequency of more than 1 GHz. In five years, those numbers are expected to grow to more than a billion transistors on a single chip, operating at a clock frequency of from 6 to 10 GHz.*

*The evolution of the microprocessor, from where it started in 1971 to where it is today and where it is likely to be in five years, has come about because of several contributing forces. Our position is that this evolution did not just happen, that each step forward came as a result of one of three things, and always within the context of a computer architect making tradeoffs. The three things are: 1) new requirements; 2) bottlenecks; and 3) good fortune. I call them collectively agents for evolution.*

*This article attempts to do three things: describe a basic framework for the field of microprocessors, show some of the important developments that have come along in the 30 years since the arrival of the first microprocessor, and finally, suggest some of the new things you can expect to see in a high-performance microprocessor in the next five years.*

**Keywords**—*Computer architecture, microarchitecture, microprocessor, microprocessor design, microprocessor evolution.*

I. BASIC FRAMEWORK

*A. Computer Architecture: A Science of Tradeoffs*

Computer architecture is far more "art" than "science." Our capabilities and insights improve as we experience more cases. Computer architects draw on their experience with previous designs in making decisions on current projects. If computer architecture is a science at all, it is a science of tradeoffs. Computer architects over the past half century have continued to develop a foundation of knowledge to help them practice their craft. Almost always the job of the computer architect requires using that fundamental knowledge to make tradeoffs. This has been especially true throughout the evolution of the microprocessor.

Problem

Algorithm

Program

ISA

Microarchitecture

Circuits

Electrons

**Fig. 1.** The microprocessor today.

*B. Levels of Transformation*

Numbers of transistors and their switching times are resources provided by process technology. What we use those resources for depends on the demands of the marketplace. How we use those resources is what the microprocessor is all about. Fig. 1 shows the levels of transformation that a problem, stated in some natural language like English, must go through to be solved. In a real sense, it is the electrons that actually do the work and solve the problem. However, since we do not speak "electron" and electrons do not speak any natural language, the best we can do is systematically transform the problem through the levels shown on Fig. 1 until we reach the electron (or device) level, that is, the 200 million transistor, 1-GHz chip.

Along the way, the problem solution is first formulated as [illegible] to remove the unacceptable characteristics of [illegible] language, such as ambiguity. It is then encoded in a mechanical language and compiled to the instruction set architecture (ISA) of the particular microprocessor. The ISA is

<span style="color:red">**Suggested Reading:**</span> Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution

**Link:** `https://course.ece.cmu.edu/~ece740/f13/lib/exe/fetch.php?media=r0_patt.pdf`

## Recall: Instruction Types

- There are three main types of instructions

- Operate (data processing) instructions
  - Execute operations in the ALU

- Data movement (memory) instructions
  - Read from or write to memory

- Control flow (branch/jump) instructions
  - Change the sequence of execution (decision making)

# ARM Instruction Formats

**DP-I**

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:8 | | | | 7:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 00 | 1 | cmd | **S** | Rn | Rd | 0 | 0 | 0 | 0 | imm8 |

**DP-R**

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:4 | | | | | | | | 3:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 00 | 0 | cmd | **S** | Rn | Rd | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rm |

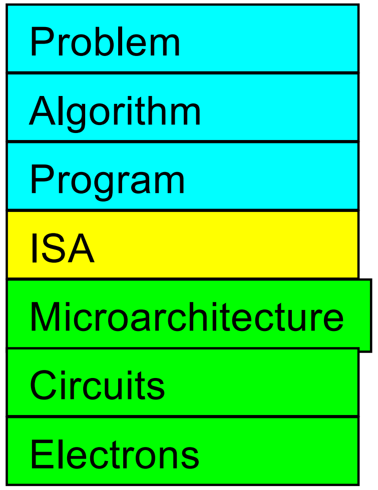
**Mem**

| 31:28 | 27:26 | 25:20 | | | | | | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| cond | 01 | 1 | 1 | 1 | 0 | 0 | **L** | Rn | Rd | imm12 |

**BR**

| 31:28 | 27:26 | 25:24 | 23:0 |
|---|---|---|---|
| cond | 10 | 10 | imm24 |

# Today's Lecture



- Last few lectures
  - Instruction Set Architectures (**ISAs**): ARM and QuAC
  - Assembly programming: ARM
- **Today: Microarchitecture**
  - **Implementation** of the **ISA** (arrangement of registers, memories, ALU, other blocks)
  - Many different microarchitectures for one ISA are possible
    - **Design Point:** Set of considerations for a given problem space (ML, automotive)
    - **Requires making tradeoffs:** Performance, power, reliability, cost, complexity
- **Today:** Design process and principles, single-cycle microarchitecture, and performance analysis
- Other **microarchitectures** we will cover
  - Multi-cycle, pipelined, and out-of-order

## Many ISAs, Many Microarchitectures

- There can be many implementations of the same ISA

  - **MIPS** R2000, R3000, R4000, R6000, R8000, R10000, …
  - **x86**: Intel 80486, Pentium, Pentium Pro, Pentium 4, Kaby Lake, Coffee Lake, Comet Lake, Ice Lake, Golden Cove, Sapphire Rapids, …, AMD K5, K7, K9, Bulldozer, BobCat, Ryzen X, …
  - **POWER** 4, 5, 6, 7, 8, 9, 10 (IBM), …, **PowerPC** 604, 605, 620, …
  - **ARM** Cortex-M*, ARM Cortex-A*, NVIDIA Denver, Apple A*, M1, …
  - **Alpha** 21064, 21164, 21264, 21364, …
  - **RISC-V** …

## How do we implement an ISA?

In other words, how do we design a system that **obeys** the hardware/software interface?

"Form follows function."

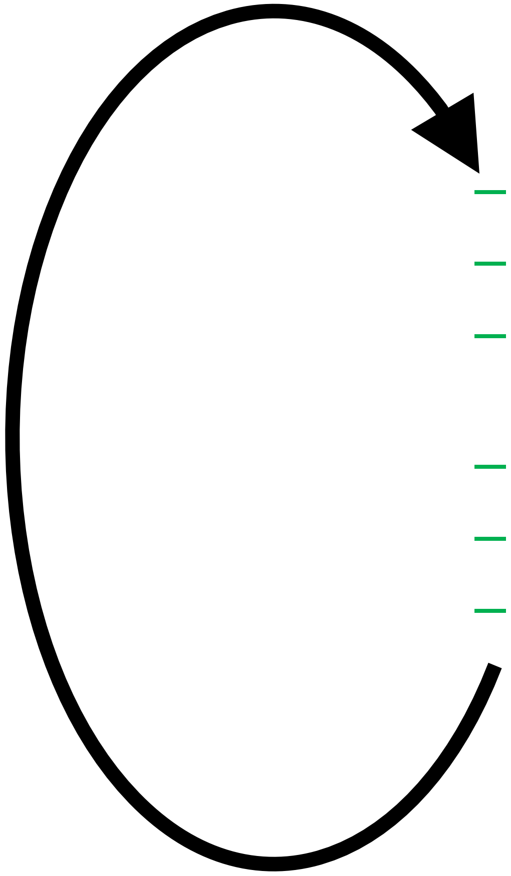
Louis Sullivan

**Before we begin construction, let's pause and ask: what is the purpose of this computer?**

## Purpose: To Process Instructions

One way to process an instruction

Six phases

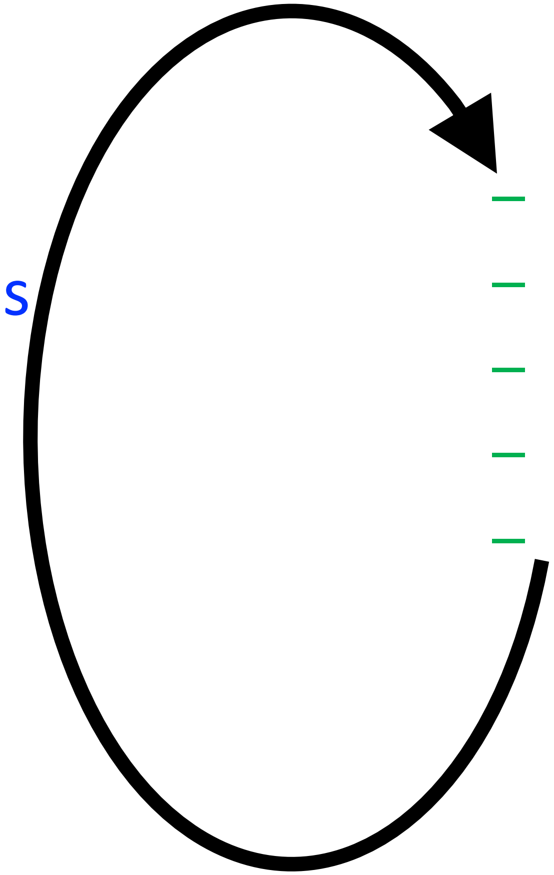


- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT

## Purpose: To Process Instructions

Another way to process an instructions

Five phases



- FETCH
- DECODE/RF READ
- EXECUTE
- MEMORY ACCESS
- WRITEBACK

# How does a machine process insts?

- What does processing an instruction mean in von Neumann model?

AS = Architectural (programmer visible) state before an instruction is processed

$\Downarrow$

**Process Instruction**

$\Downarrow$

AS' = Architectural (programmer visible) state after an instruction is processed

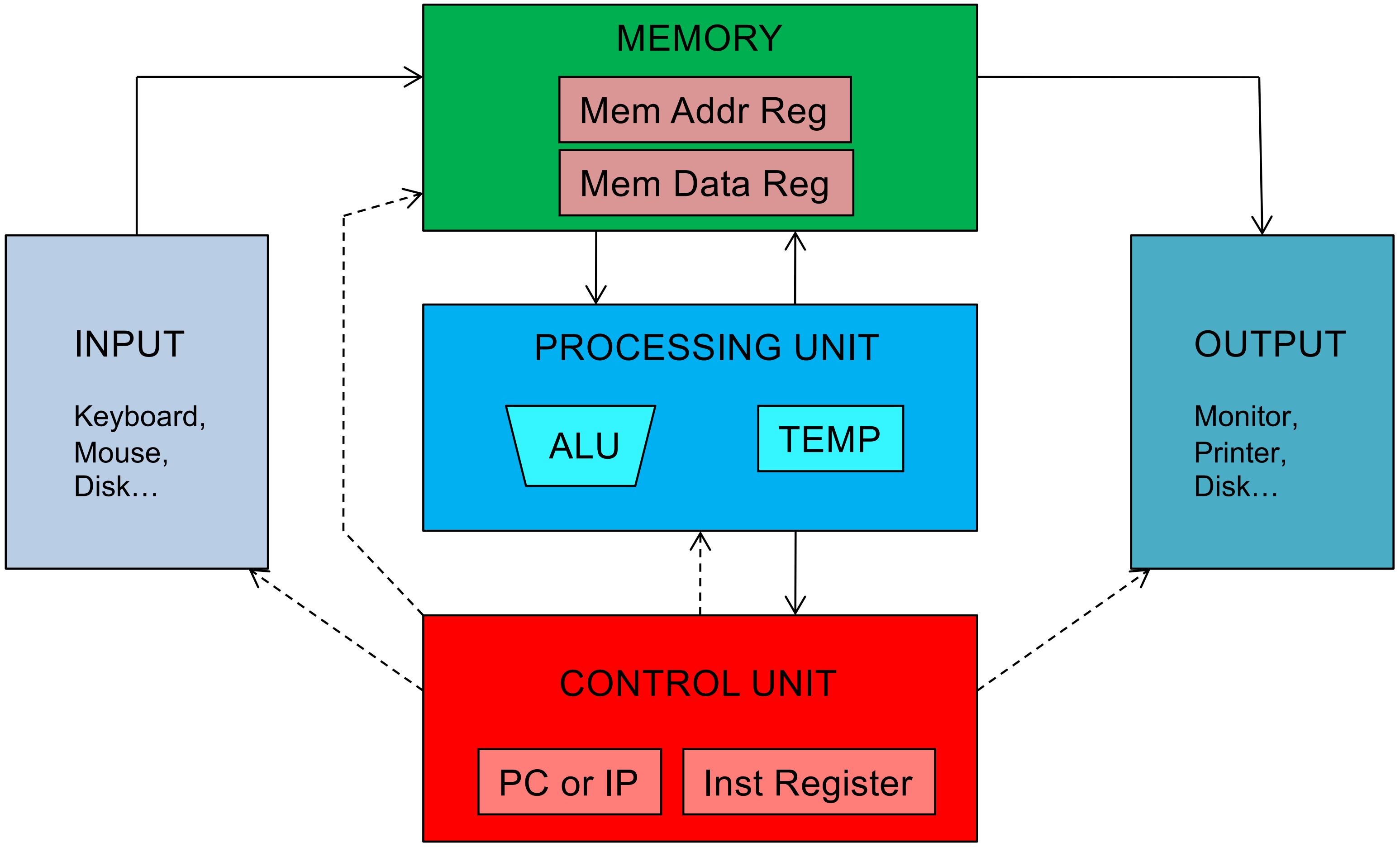
- **Processing an instruction:** Transforming AS to AS' according to the ISA specification of the instruction

## The Von Neumann Model/Architecture

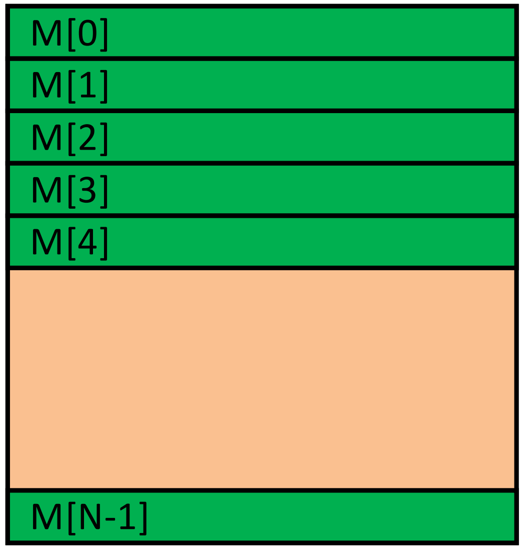
**Stored program**

**Sequential instruction processing**

# The Von Neumann Model/Architecture



# Recall: Programmer Visible (Architectural) State



**Memory**
array of storage locations
indexed by an address

**Registers**
- given special names in the ISA
  (as opposed to addresses)
- general vs. special purpose

memory address
of the current (or next) instruction

Instructions (and programs) specify how to transform
the values of programmer visible state

# ISA = Instruction Set Architecture

- Instruction Set Architecture = Instruction Set **+** Architectural State

  - Instruction Set
    - Opcodes
    - Operands
    - Data types (e.g., 2’s complement)
    - Addressing modes (e.g., base + offset)
    - Instruction formats (Data processing, Immediate, Memory)

  - Architectural state
    - Memory
    - Register set
    - Program counter

## The "Process Instruction" Step

- ISA specifies abstractly what AS' should be, given an instruction and AS
  - It defines an **abstract finite state machine** where
    - State = programmer-visible state
    - Next-state logic = instruction execution specification
  - From ISA point of view, there are no "intermediate states" between AS and AS' during instruction execution
    - One state transition per instruction

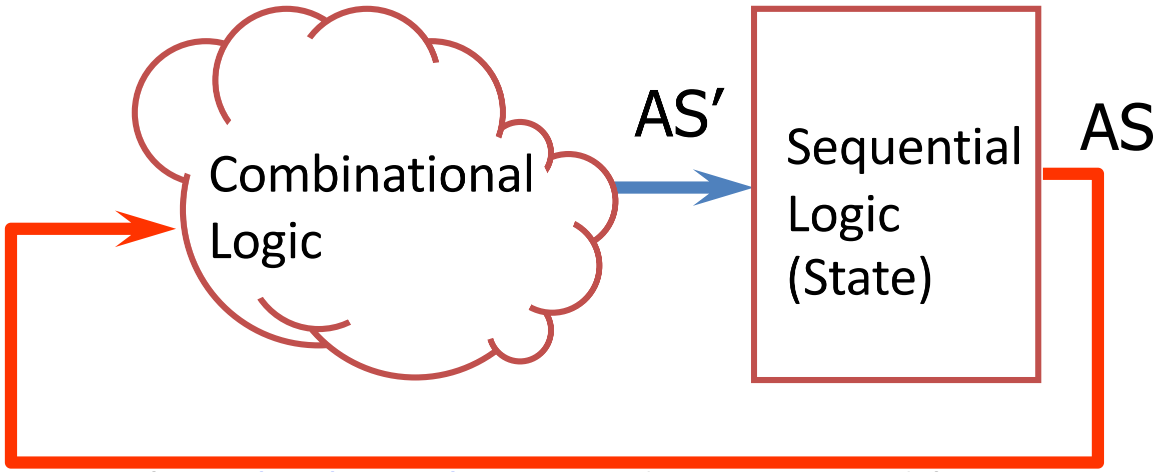
- Microarchitecture implements how AS is transformed to AS'
  - There are many choices in implementation
  - We can have programmer-invisible state to optimize the speed of instruction execution: **multiple** state transitions per instruction
    - Choice 1: AS → AS' (transform AS to AS' in a single clock cycle)
    - Choice 2: AS → AS+MS1 → AS+MS2 → AS+MS3 → AS' (take multiple clock cycles to transform AS to AS')

## Very Basic Instruction Processing Engine

- Each instruction takes a single clock cycle to execute
- Only combinational logic is used to implement instruction execution
  - *No intermediate, programmer-invisible state updates*
- *Easy to explain and a simple control unit!*

## Basic Instruction Processing Engine

- Single-cycle machine



- What is the *clock cycle time* determined by?
- What is the *critical path* (i.e., longest delay path) of the combinational logic determined by?

AS: Architectural State

## Single-Cycle vs. Multi-Cycle Machines

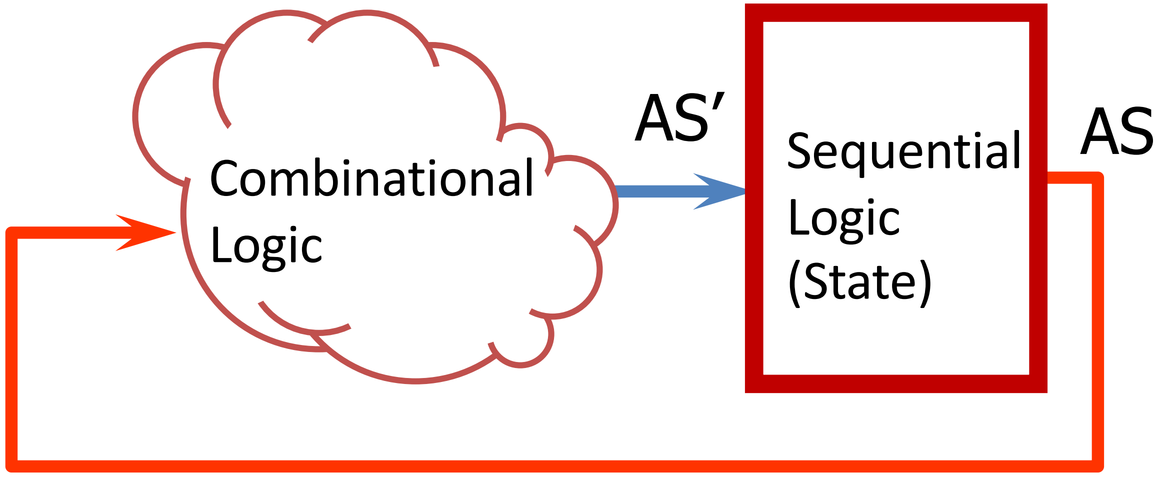
- Single-cycle machines
  - Each instruction takes a single clock cycle
  - All state updates made at the end of an instruction's execution
  - Big disadvantage: The slowest instruction determines cycle time → long clock cycle time

- Multi-cycle machines
  - Instruction processing broken into multiple cycles/stages
  - State updates can be made during an instruction's execution
  - Architectural state updates made at the end of an instruction's execution
  - Advantage over single-cycle: The slowest "stage" determines cycle time

- Both single-cycle and multi-cycle machines literally follow the von Neumann model at the microarchitecture level

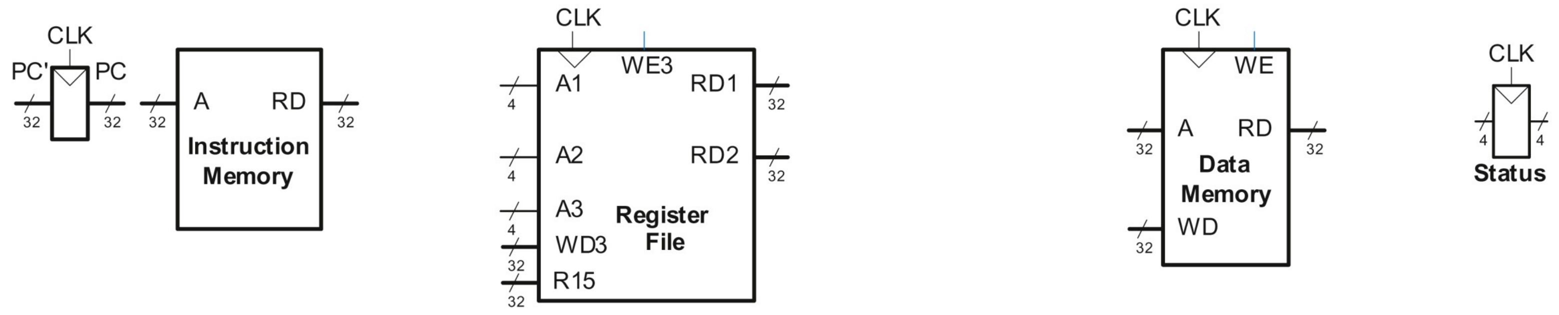
## Basic Instruction Processing Engine

- Single-cycle machine



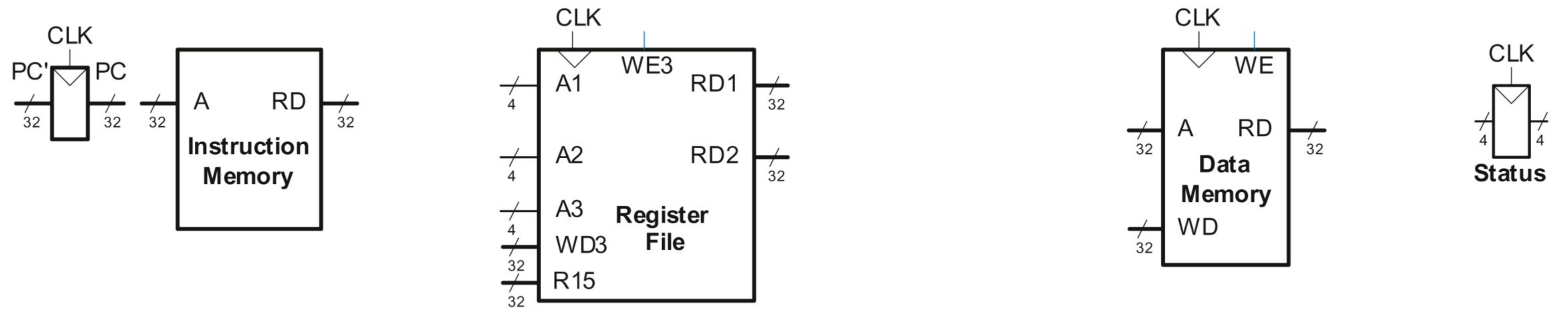
AS: Architectural State

## ARM State (AS) Elements



- PC: Logically part of the register file
  - Read and written every cycle, independently of the normal register file operation. Should it be **"physically"** part of the register file?
- **Instruction memory** has a single read port. One 32-bit address input. One 32-bit instruction (RD) output.
- **Register file:** 15 registers (`R0` to `R14`) + additional input to receive `R15` from PC
  - Two read ports 4-bit A1 and A2 and 32-bit RD1 and RD2
  - One write port A3 (and WD3) and a write enable input
  - **Read of R15 returns PC + 8**
  - **Write of R15 must be handled properly if PC is outside the register file**
  - Reads are combinational and writes happen on the rising edge of the clock

# ARM State (AS) Elements



- Data Memory: Single read/write port
  - If write enable (WE) is TRUE then it writes data WD into address A on the rising edge of the clock
  - If the write enable is FALSE, then it reads value at address A onto RD

- All reads are combinational and constant time (not realistic but Ok for now)
- All writes and state updates happen on the rising edge of the clock
  - Synchronous sequential circuit

# Microarchitecture Division

- **Two interacting parts**
  - Datapath (32-bit in our case)
  - Control unit

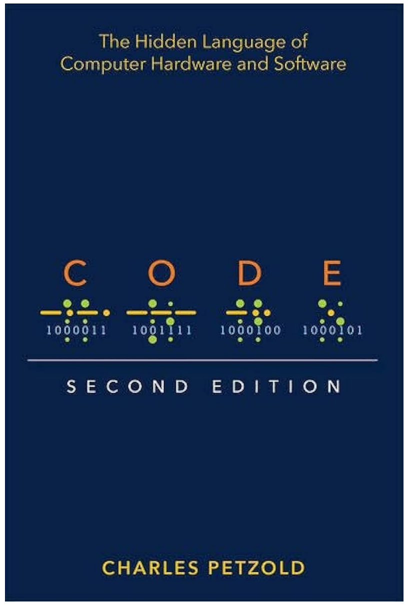
- Datapath operates on words of data
  - Memories, registers, ALUs, and multiplexers

- **Control** unit informs the datapath how to execute an instruction
  - Receives the current instruction from the datapath
  - Produces multiplexer selects, ALU control, register enable, and memory write signals to **control** the operation of the datapath

## Role of Control Unit

Codes stored in memory control the hardware of the computer ... As a puppeteer controlling a troupe of marionettes in an exquisitively choreographed dance of arithmetic and logic. **The CPU control signals are the strings.**

CODE, Charles Petzold



# Design Process/1

- We will add the logic for one instruction at a time
  - LDR (**L**oa**D** **R**egister)
  - STR (**ST**ore Register)
  - **D**ata **P**rocessing (**DP**) instructions with 2nd source operand as an immediate
  - DP with 2nd source operand as a register
  - Branch instruction
- Then build the "Control Unit"

## Design Process/2

- We limit ourselves to a subset of instructions
  - Data-processing instructions: ADD, SUB, AND, ORR (with register and immediate offsets)
  - Memory instructions: LDR, STR (with positive immediate offset)
  - Branches: B
- Once you understand these you can expand the hardware to handle others

## Design Process/3

- New connections are emphasized in black
- Hardware already studied in gray
- Control signals in blue

# LDR with Src2 as Immediate

- I (Bit 25) = 1: Src2 = imm12 where imm12 is a **12-bit unsigned offset** added to the value in the base register (Rn)

- Format of **L**oa**D** **R**egister instruction

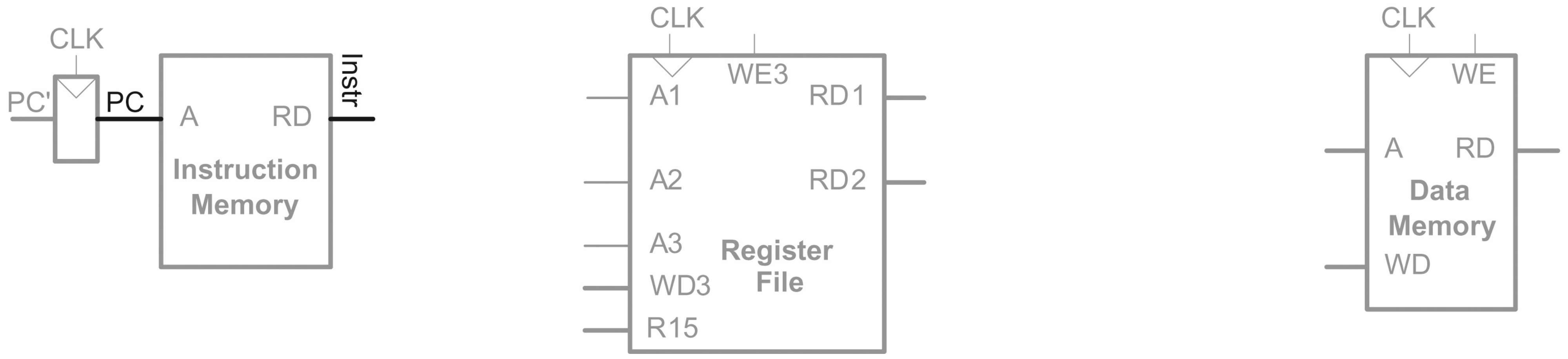
$$\begin{tikzcd} \texttt{LDR} & \texttt{R0,} \arrow[d] & \texttt{[R1,} \arrow[d] & \texttt{\#12]} \arrow[d] \\ \texttt{LDR} & \texttt{Rd,} & \texttt{[Rn,} & \texttt{\#imm12]} \end{tikzcd}$$

- L (Bit 20) = 1: CPU performs an **LDR**

| 31:28 | 27:26 | 25:20 | | | | | | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| cond | 01 | 1 | 1 | 1 | 0 | 0 | 1 | Rn | Rd | imm12 |

# The LDR Datapath

Step 1: Read (Fetch) instruction from memory

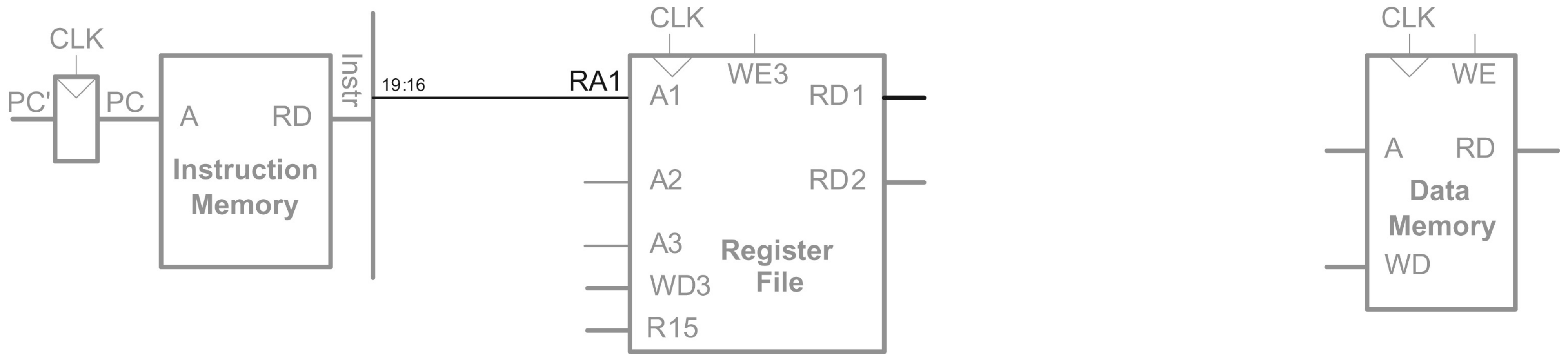


- Remember the distinction between PC (current state) and PC' (next state)
- From this point on, CPU actions depend on the instruction fetched

| 31:28 | 27:26 | | | 25:20 | | | | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| cond | 01 | 1 | 1 | 1 | 0 | 0 | L | Rn | Rd | imm12 |

# The LDR Datapath

Step 2: Read source operand (base register, Rn) from register file

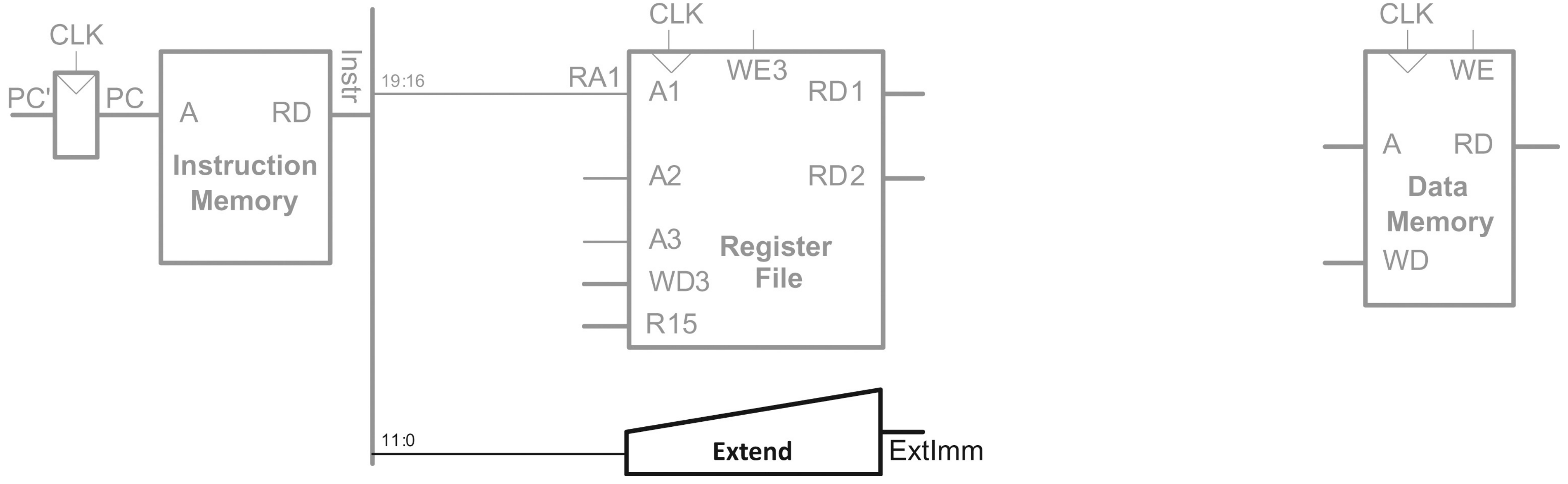


- Data is read onto RD1

| 31:28 | 27:26 | 25:20 | | | | | | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| cond | 01 | 1 | 1 | 1 | 0 | 0 | L | Rn | Rd | imm12 |

# The LDR Datapath

Step 3: Zero-extend the immediate field stored in Instr$_{11:0}$



| 31:28 | 27:26 | 25:20 | | | | | | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| cond | 01 | 1 | 1 | 1 | 0 | 0 | L | Rn | Rd | imm12 |

## Zero Extension

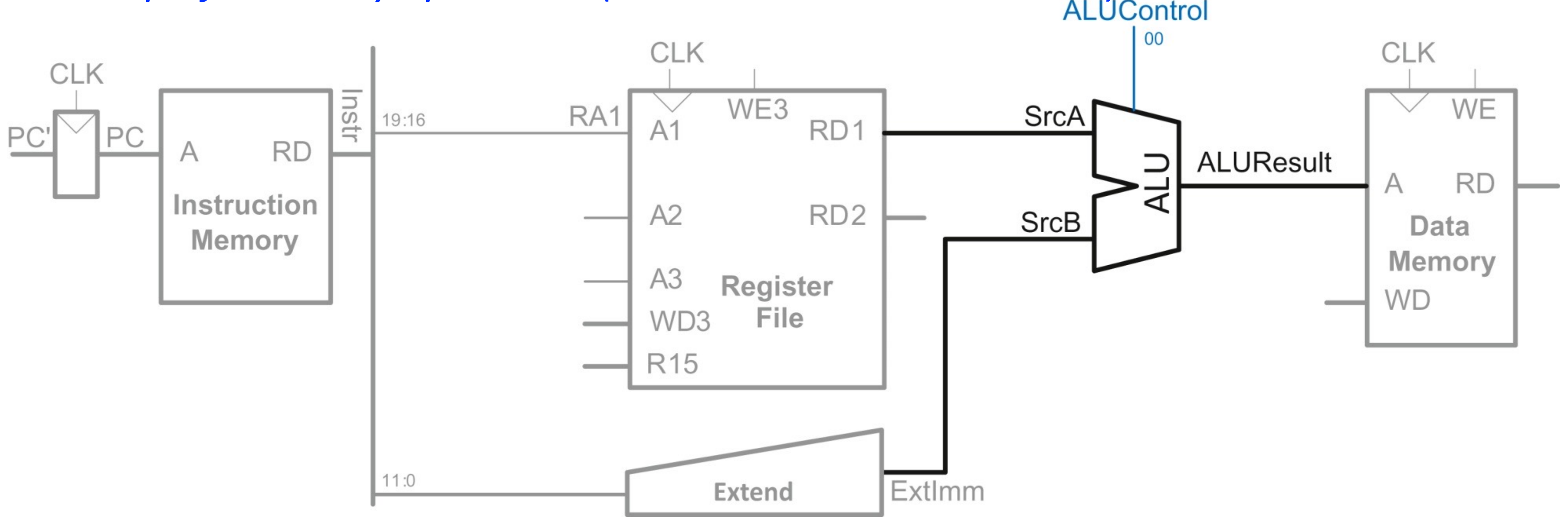
- Appending leading zeros to make a smaller quantity equal to a bigger quantity

- $ImmExt_{31:12}$ = 0 and $ImmExt_{11:0}$ = $Instr_{11:0}$

# The LDR Datapath

Step 4: Compute memory address (ALUControl = 00)

*ALU can perform many operations (which one do we want: ADD)*

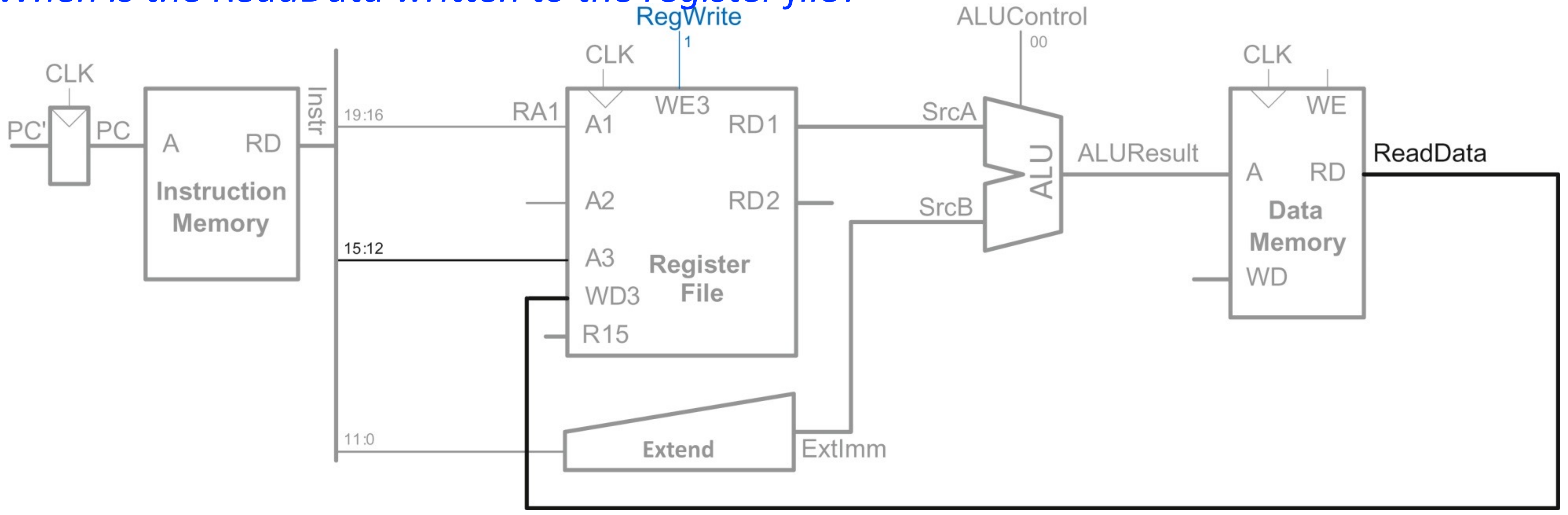


| 31:28 | 27:26 | 25:20 | | | | | | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| cond | 01 | 1 | 1 | 1 | 0 | 0 | L | Rn | Rd | imm12 |

# The LDR Datapath

Step 5: Write back data from read by data memory to Rd in Reg File

*When is the ReadData written to the register file?*

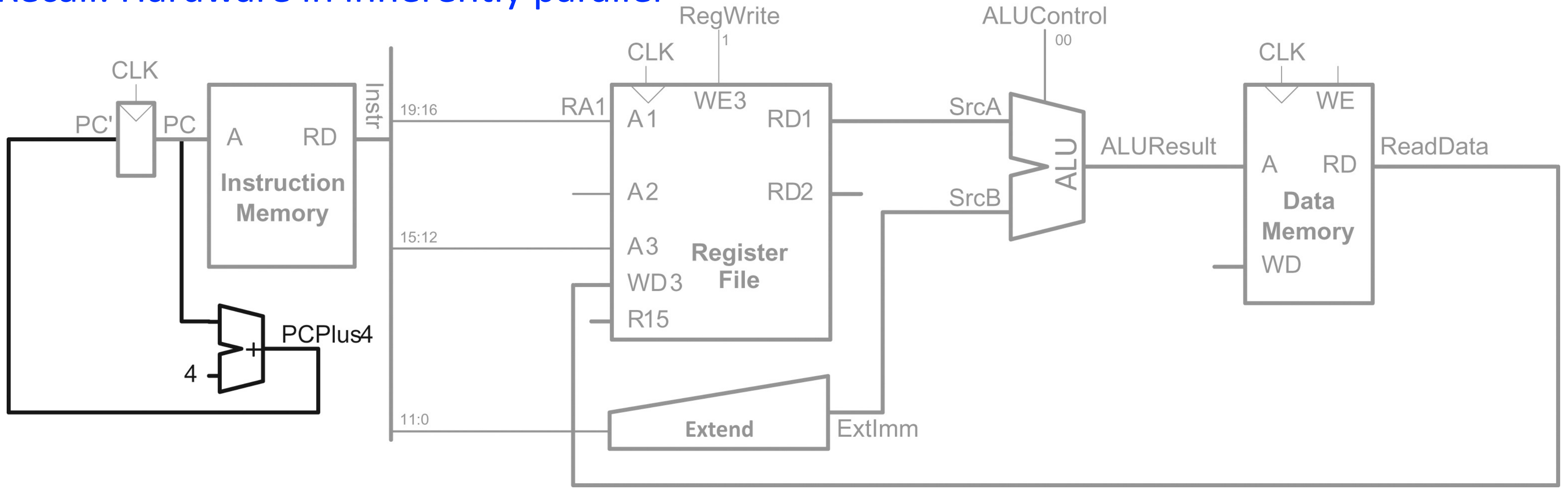


| 31:28 | 27:26 | 25:20 | | | | | | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| cond | 01 | 1 | 1 | 1 | 0 | 0 | L | Rn | Rd | imm12 |

# The LDR Datapath

Step 6: Compute address of next instruction (PC' = PC + 4)

Recall: Hardware in inherently parallel

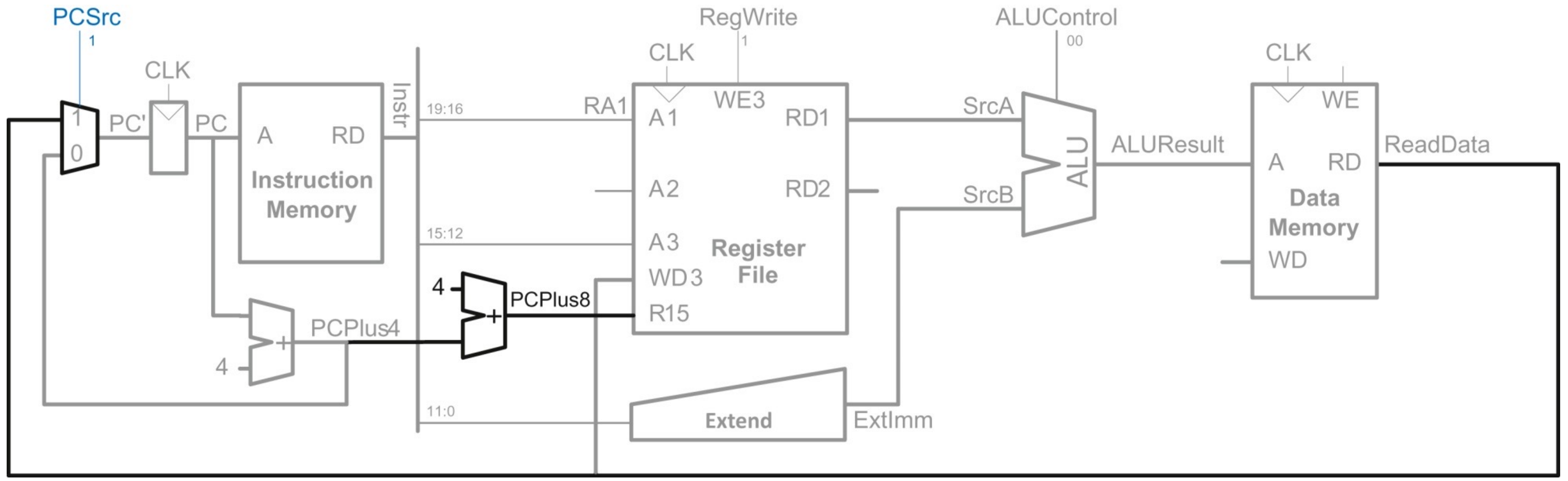


*PC will become PC' the following cycle (recall photography example)*

| 31:28 | 27:26 | 25:20 | | | | | | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| cond | 01 | 1 | 1 | 1 | 0 | 0 | L | Rn | Rd | imm12 |

# The LDR Datapath

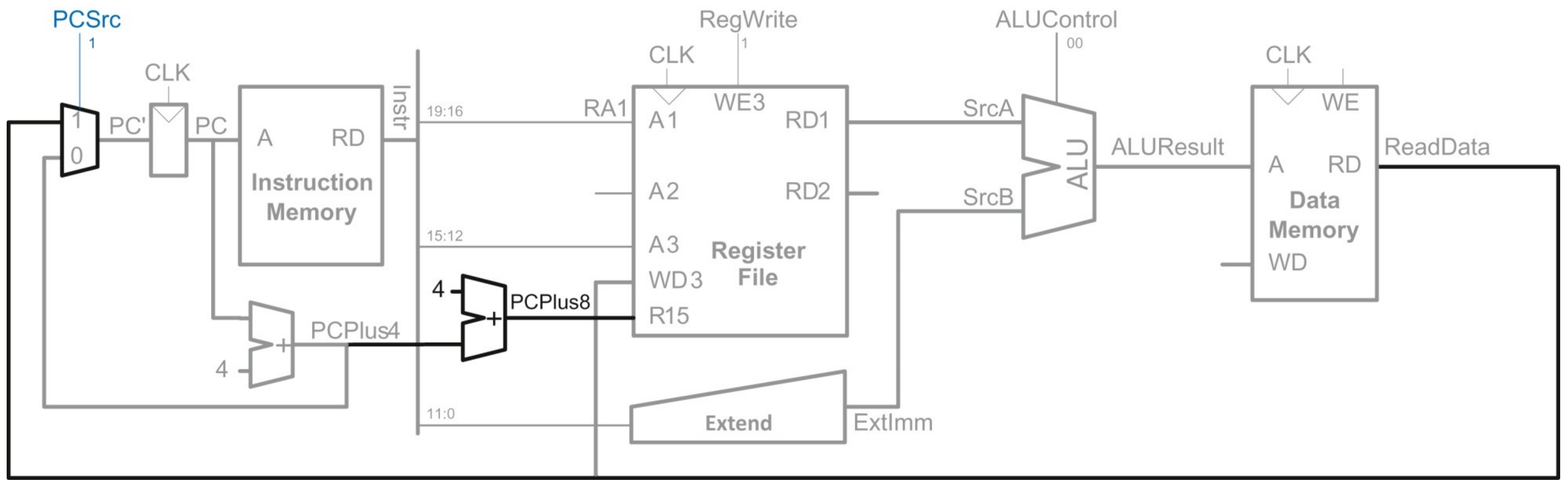
Step 7/a: Reading register R15 returns PC + 8



| 31:28 | 27:26 | 25:20 | | | | | | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| cond | 01 | 1 | 1 | 1 | 0 | 0 | L | Rn | Rd | imm12 |

# The LDR Datapath

Step 7/b: Writing register R15 (PC may be an instruction's result)



| 31:28 | 27:26 | 25:20 | | | | | | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| cond | 01 | 1 | 1 | 1 | 0 | 0 | L | Rn | Rd | imm12 |

# STR Instruction

- STR instruction uses the same instruction format
  - `LDR` and `STR` behave differently at the machine level
- `Rd` is a source operand (specifies the register to store to mem)

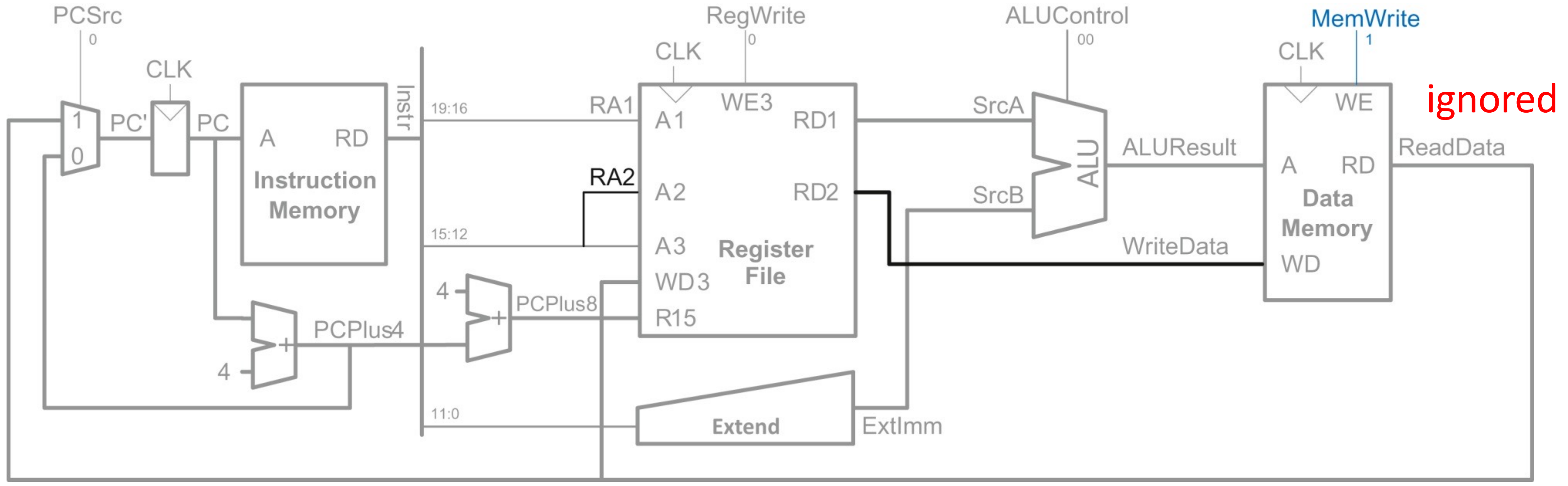
- Format of **ST**ore **R**egister instruction

$$\begin{tikzcd}[column sep=small]
\texttt{STR} & \texttt{R0,} \arrow[d] & \texttt{[R1,} \arrow[d] & \texttt{\#12]} \arrow[d] \\
\texttt{STR} & \texttt{Rd,} & \texttt{[Rn,} & \texttt{\#imm12]}
\end{tikzcd}$$

| 31:28 | 27:26 | 25:20 | | | | | | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| cond | 01 | 1 | 1 | 1 | 0 | 0 | L | Rn | Rd | imm12 |

# The STR Datapath

Step 8: Read a second register (Rd) and write its value to memory



- **ReadData is ignored because RegWrite is FALSE**

| 31:28 | 27:26 | 25:20 | | | | | | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| cond | 01 | 1 | 1 | 1 | 0 | 0 | L | Rn | Rd | imm12 |

# DP Instructions: Immediate

- Like the LDR instruction, but two important differences
  - imm8 instead of imm12
  - The destination register stores the result of the ALU operation instead of memory access
- Format

$$\begin{tikzcd}[column sep=small]
\texttt{ADD} & \texttt{R0,} \arrow[d] & \texttt{R1,} \arrow[d] & \texttt{\#16} \arrow[d] \\
\texttt{ADD} & \texttt{Rd,} & \texttt{Rn,} & \texttt{\#imm8}
\end{tikzcd}$$

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:8 | | | | 7:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 00 | 1 | cmd | S | Rn | Rd | 0 | 0 | 0 | 0 | imm8 |

## Adding Support for DP Instructions

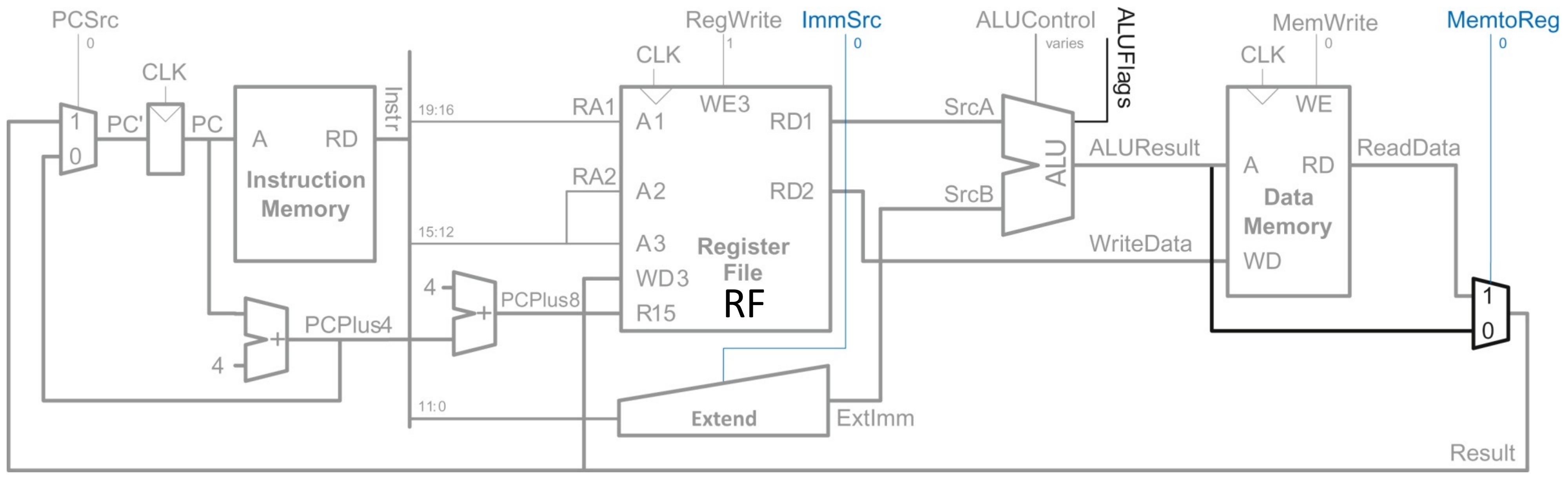
- The ALU functions and encoding

| ALUControl | Function |
|---|---|
| 00 | ADD |
| 01 | SUB |
| 10 | AND |
| 11 | ORR |

- The ALU also produces four **flags** that are sent to the control unit
- *Register file either receives data from the data memory or the ALU*
  - *Add a multiplexer to choose between ReadData and ALUResult*
  - *This multiplexer is controlled by MemtoReg*
  - *MemtoReg = 1 for LDR and 0 for data processing instructions*

# DP-Immediate Datapath

Step 9: Change extend block, and add signal to write ALU result to RF



| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:8 | | | | 7:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 00 | 1 | cmd | S | Rn | Rd | 0 | 0 | 0 | 0 | imm8 |

# DP Instructions: Register

- The second source operand is Rm instead of an immediate
- Place **Rm** on the **A2** port of the register file for DP instructions with register as the second operand

- Format

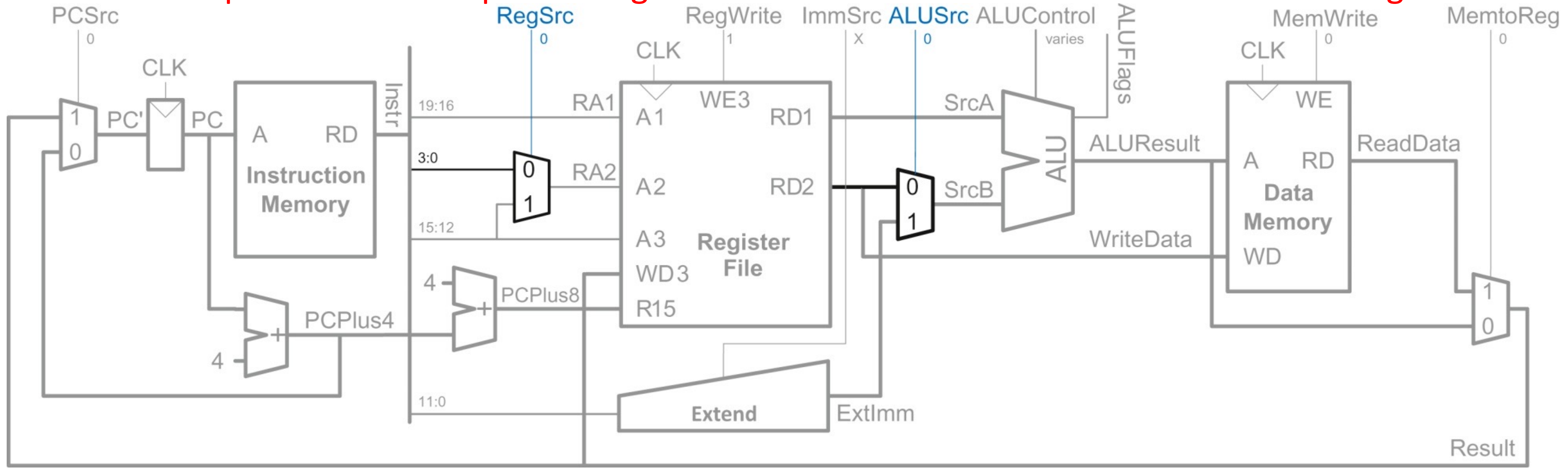
$$\begin{tikzcd}
\texttt{ADD} & \texttt{R0,} \arrow[d] & \texttt{R1,} \arrow[d] & \texttt{R3} \arrow[d] \\
\texttt{ADD} & \texttt{Rd,} & \texttt{Rn,} & \texttt{Rm}
\end{tikzcd}$$

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:4 | | | | | | | | 3:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 00 | 0 | cmd | S | Rn | Rd | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rm |

# DP-Register Datapath

**Step 10:** Read 2nd register (Rm) from Reg File and send RD2 to ALU

We need multiplexers on the inputs of register file and ALU to select the second source register



| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:4 | | | | | | | | 3:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 00 | 0 | cmd | S | Rn | Rd | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rm |

## Branch Instruction: Unconditional

- The second source operand is Rm instead of an immediate
- Place Rm on the A2 port of the register file for DP instructions with register as the second operand

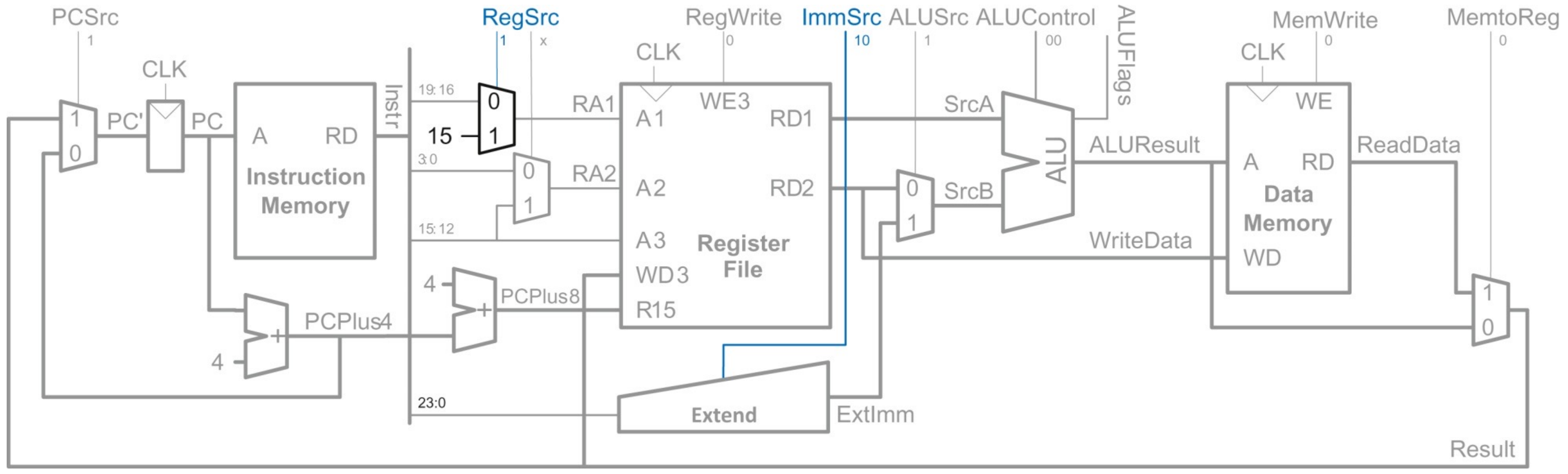
- Format

$$\begin{tikzcd} \texttt{B TARGET} \arrow[d] \\ \texttt{B imm24} \end{tikzcd}$$

| 31:28 | 27:26 | 25:24 | 23:0 |
|---|---|---|---|
| 1110 | 10 | 10 | imm24 |

# Branch Datapath

Step 11: Change extend block, and add a bit to RegSrc for branch



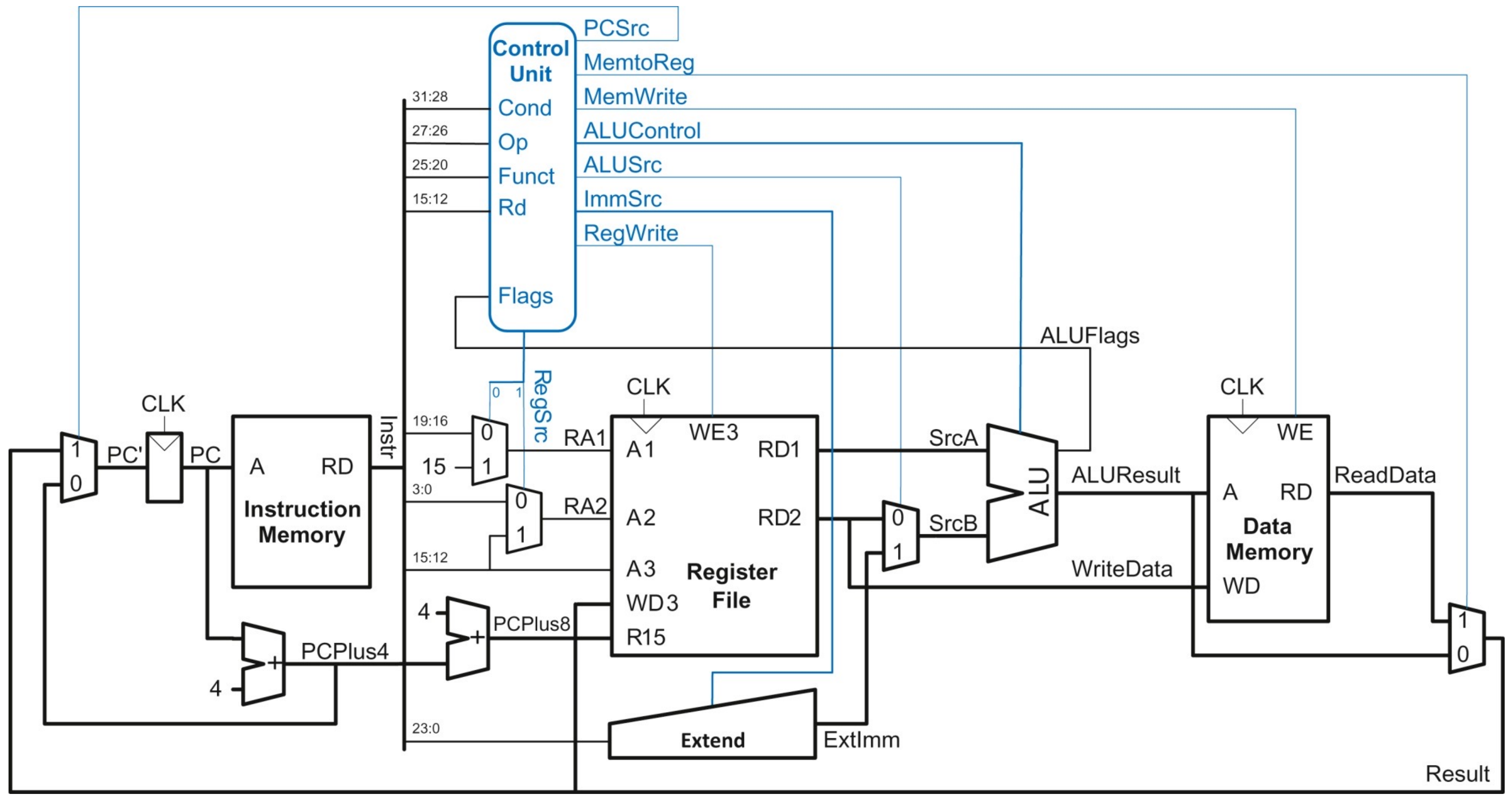
| 31:28 | 27:26 | 25:24 | 23:0 |
|---|---|---|---|
| 1110 | 10 | 10 | imm24 |

## Operation of the Extend Block

- **Each of the three instruction formats interpret the immediate field differently**
  - ImmSrc$_{1:0}$ is the 2-bit control signal input to the extend block

| ImmSrc$_{1:0}$ | ExtImm | Description |
|---|---|---|
| 00 | {24'b0, Instr$_{7:0}$} | Zero-extended *imm8* |
| 01 | {20'b0, Instr$_{11:0}$} | Zero-extended *imm12* |
| 10 | {6{Instr$_{23}$}, Instr$_{23:0}$}00 | Sign-extended *imm24* |

# Datapath with Control



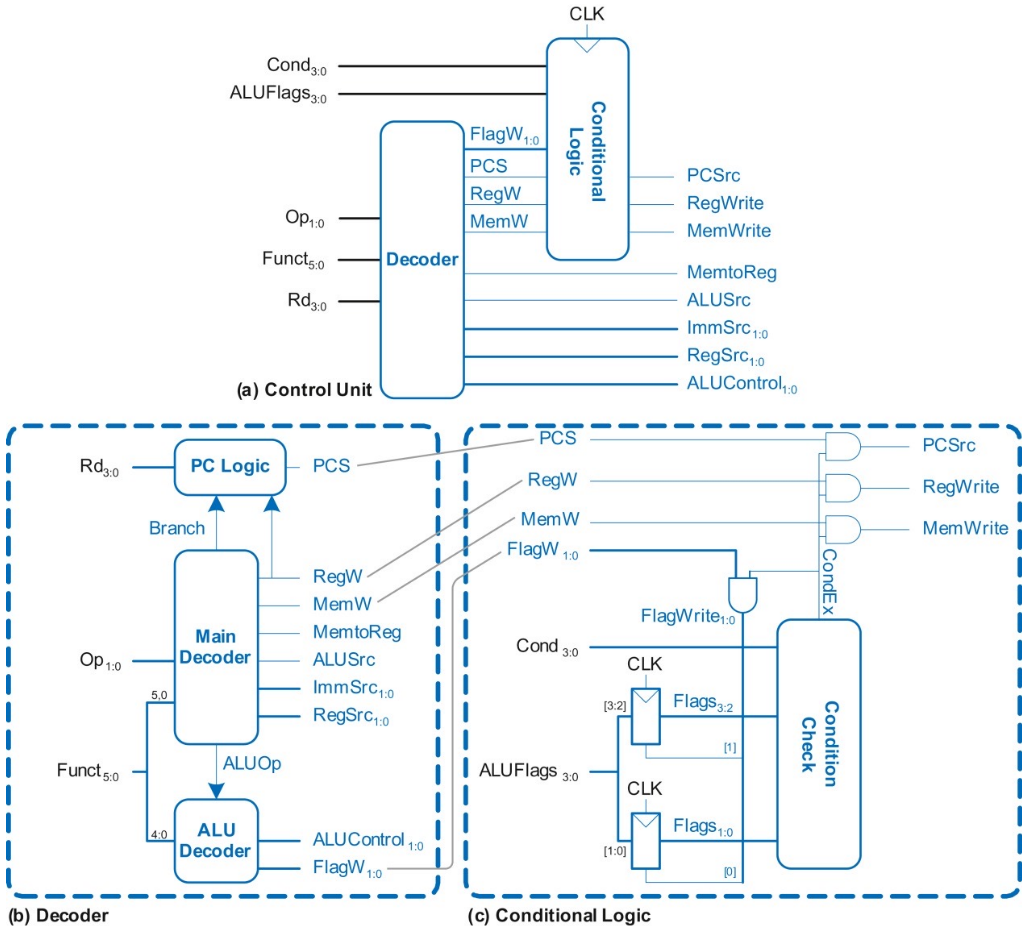
# Control Unit

- Generate control signals based on instruction fields
  - $Instr_{31:20}$ **(cond)**
  - $Instr_{27:26}$ **(opcode)**
  - $Instr_{25:20}$ **(funct)**
  - Flags (needed for conditional execution)
  - Destination register (to update PC properly)

- **Controller for single-cycle microarchitecture is purely combinational**

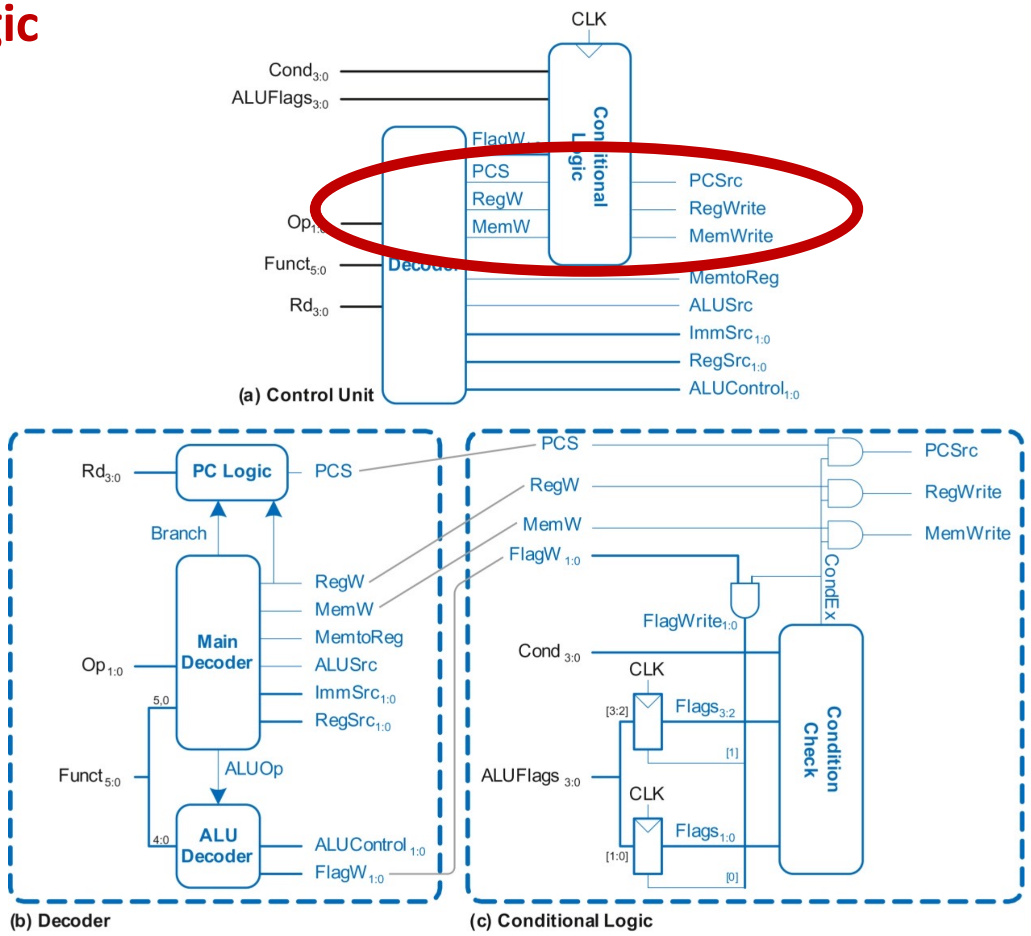
- Conditional logic must enable updates to the **architectural state** when the instruction should be conditionally executed
  - Write enables must be TRUE only if conditional instruction is in fact executed

# One way to build the control unit



# One way to build the control unit

The **write enable lines** that update the **architectural state** could be **“killed”** by the **conditional logic**



# Decoder Truth Table

- Only selected signals are shown in the truth table

| Op | $Funct_5$ | $Funct_0$ | Type | Branch | MemtoReg | MemW | ALUSrc | ImmSrc | RegW | RegSrc | ALUOp |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 0 | X | DP Reg | 0 | 0 | 0 | 0 | XX | 1 | 00 | 1 |
| 00 | 1 | X | DP Imm | 0 | 0 | 0 | 1 | 00 | 1 | X0 | 1 |
| 01 | X | 0 | STR | 0 | X | 1 | 1 | 01 | 0 | 10 | 0 |
| 01 | X | 1 | LDR | 0 | 1 | 0 | 1 | 01 | 1 | X0 | 0 |
| 11 | X | X | B | 1 | 0 | 0 | 1 | 10 | 0 | X1 | 0 |

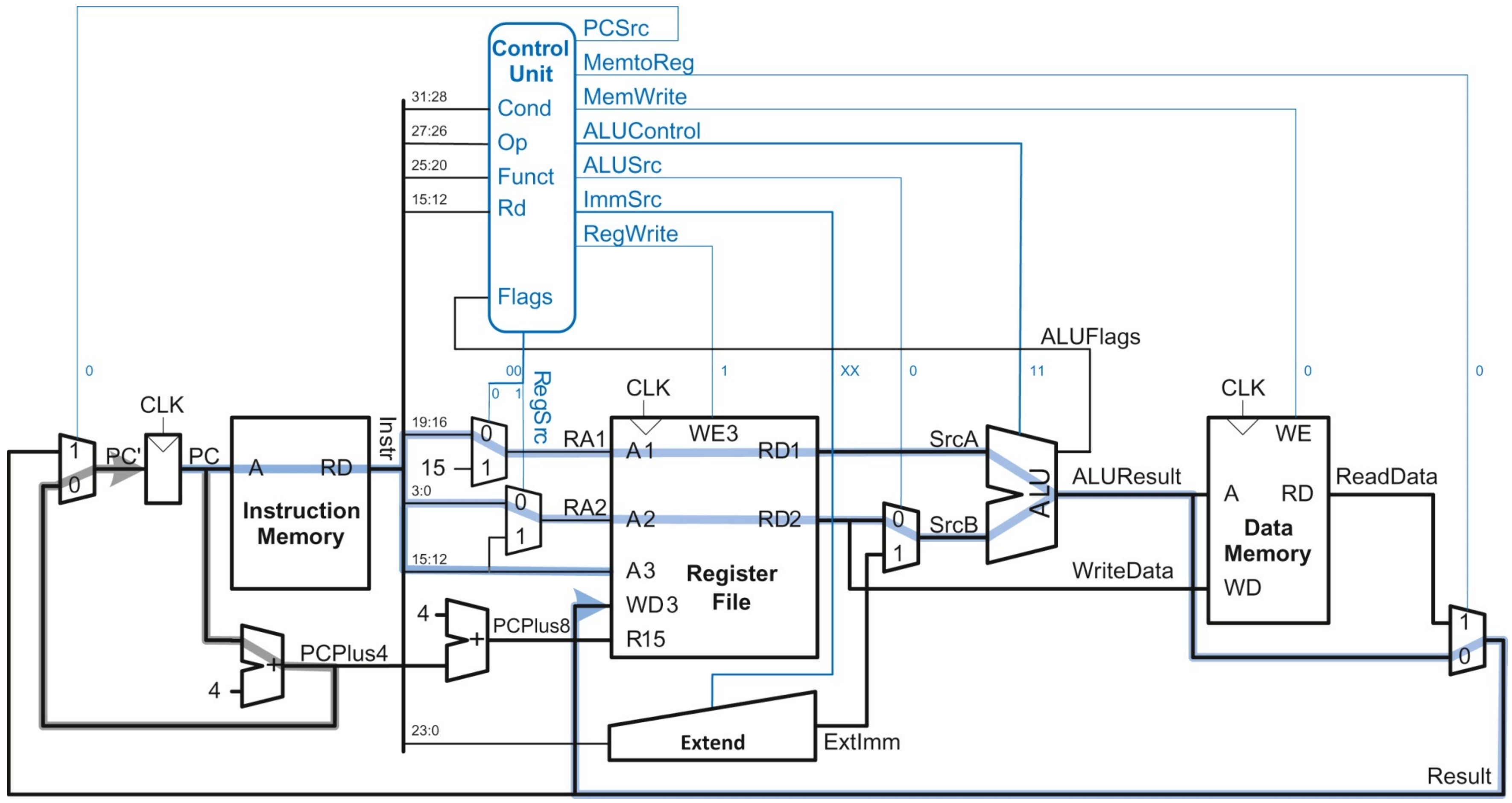
## Example: Generating PCSrc Signal

- **PCSrc is 1 when**
  - Destination register (Rd) is R15
  - RegW is 1 (ADD/SUB or LDR)
  - Instruction is a branch

- **PCSrc = ((Rd == 15) & RegW) | Branch**
  - Assuming the control unit generates a signal called **Branch** when opcode is **10** (B or BL)

- **Important:** Be careful to take conditional execution into account for the assignment!

# Processor Operation: ORR

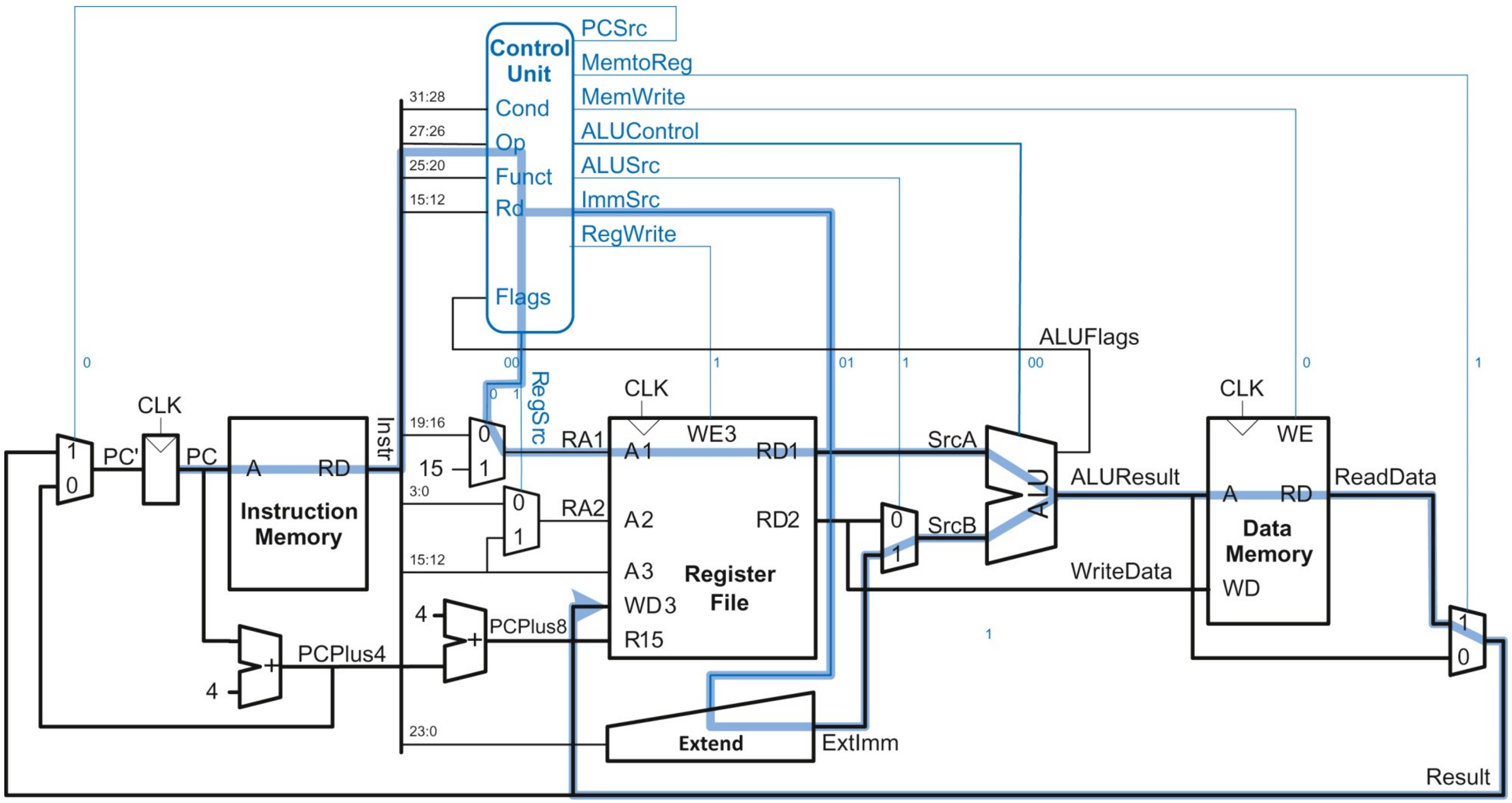


# Processor Operation: ORR

| | |
|---|---|
| **PCSrc** | 0 |
| **MemtoReg** | 0 |
| **MemWrite** | 0 |
| **ALUControl** | **11** |
| **ALUSrc** | 0 |
| **ImmSrc$_{0:1}$** | XX |
| **RegWrite** | 1 |
| **RegSrc$_{0:1}$** | 00 |

| ALUControl | Function |
|---|---|
| 00 | ADD |
| 01 | SUB |
| 10 | AND |
| **11** | ORR |

# Processor Operation: LDR



# Processor Operation: LDR

| | |
|---|---|
| PCSrc | 0 |
| MemtoReg | 1 |
| MemWrite | 0 |
| ALUControl | 00 |
| ALUSrc | 1 |
| $ImmSrc_{0:1}$ | 01 |
| RegWrite | 1 |
| $RegSrc_{0:1}$ | 00 |

| ALUControl | Function |
|---|---|
| 00 | ADD |
| 01 | SUB |
| 10 | AND |
| 11 | ORR |

| $ImmSrc_{1:0}$ | ExtImm | Description |
|---|---|---|
| 00 | {24'b0, $Instr_{7:0}$} | Zero-extended *imm8* |
| 01 | {20'b0, $Instr_{11:0}$} | Zero-extended *imm12* |
| 10 | {6{$Instr_{23}$}, $Instr_{23:0}$}00 | Sign-extended *imm24* |

## Drawback of Single-Cycle CPU

- Is this the best way to build a CPU?

- What are the critical issues?

  - Next: performance analysis basics

## Performance Analysis

## Processor Performance

- Performance is **quantified** by the execution time

- The time it takes for a program to execute from start to finish

- For example, for a given **ISA** and technology, how long does it take to run a program on the single-cycle CPU?

## Processor Performance

- **How fast is my program?**
  - Every program consists of a series of instructions
  - Each instruction needs to be executed

- **So how fast are my instructions?**
  - Instructions are realized on the hardware
  - They can take one or more clock cycles to complete
  - ***Cycles per Instruction = CPI***

- **How much time is one clock cycle?**
  - The critical path determines how much time one cycle requires = ***clock period***
  - 1/clock period = ***clock frequency*** = how many cycles can be done each second

## Execution Time

$$\text{Execution time} = (\#\text{instructions})\left(\frac{cycles}{instruction}\right)\left(\frac{seconds}{cycle}\right)$$

- # instructions (**N**)
  - Depends on the ISA, skill of programmer, compiler, algorithm
- cycles per instruction (**CPI**)
  - Depends on the microarchitecture
- seconds per cycle (clock period, inverse is clock frequency, **f**)
  - critical path, circuit technology, type of adders, gate-level details

## How Can I Make the Program Run Faster?

- **N x CPI x (1/f)**

- **Reduce the number of instructions (N)**
  - Make instructions that ‘do’ more (CISC)
  - Use better compilers

- **Use fewer cycles to perform the instruction (CPI)**
  - Simpler instructions (RISC)
  - Use multiple units/ALUs/cores in parallel

- **Increase the clock frequency (f)**
  - Find a ‘newer’ technology to manufacture
  - Redesign time-critical components
  - Adopt pipelining

## Execution Time (Single-Cycle CPU)

$$\text{Execution time} = (\#\text{instructions})\left(\frac{cycles}{instruction}\right)\left(\frac{seconds}{cycle}\right)$$

- # instructions (**ARM is a RISC ISA**)

- cycles per instruction (**= One, fixed, bad idea!**)

- seconds per cycle (**critical path of the CPU circuit**)

# Critical Path Analysis

- Each instruction in single-cycle CPU takes one clock cycle
- Determining the cycle time requires finding the critical path
- **Different instructions use different resources**
  - `LDR` uses instruction and data memory
  - `ADD` does not use data memory
  - `STR` does not write anything back to the register file
- **Which instruction is the slowest?**
  - Let us revisit the schematics and find out

## Elements of Critical Path

| Parameter | Description |
|---|---|
| $t_{pcq\_PC}$ | PC clock-to-Q delay |
| $t_{mem}$ | Memory read |
| $t_{dec}$ | Decoder propagation delay |
| $t_{mux}$ | Multiplexer delay |
| $t_{RFread}$ | Register file read |
| $t_{ext}$ | Extension block delay |
| $t_{ALU}$ | ALU delay |
| $t_{RFsetup}$ | Set up RF for write (next cycle) |

## Critical Path: LDR

$$T_c = t_{pcq\_PC} + t_{mem} + t_{dec} + \max[t_{mux} + t_{RFread}, t_{ext} + t_{mux}] + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$$

- Memories & register files slower than combinational logic
  - Therefore, $t_{mux} + t_{RFread} >> t_{ext} + t_{mux}$

**Final Equation**

$$\boxed{T_c = t_{pcq\_PC} + 2t_{mem} + t_{dec} + t_{RFread} + t_{ALU} + 2t_{mux} + t_{RFsetup}}$$

## Critical Path: DP-R

$$T_c = t_{pcq\_PC} + t_{mem} + t_{dec} + t_{mux} + t_{RFread} + t_{ALU} + t_{mux} + t_{RFsetup}$$

**Final Equation**

$$\boxed{T_c = t_{pcq\_PC} + t_{mem} + t_{dec} + t_{RFread} + t_{ALU} + 2t_{mux} + t_{RFsetup}}$$

## Critical Path Analysis

- Different instructions have different critical paths
  - **`LDR`** is the slowest instruction
  - **`DP-R`** and **`B`** have shorter critical paths because they do not need to access data memory (Memory is slow!)

- Single-cycle processor is a synchronous sequential circuit
  - Clock period must be **constant** and **long enough** to accommodate the slowest instruction

- The numerical values of different variables in the critical path equation depend on the specific manufacturing technology

## Exercise 1: Performance Analysis

- Find the time it takes to execute a program with 100 billion instructions on a single-cycle CPU in 16 nm CMOS manufacturing process. See the table for delays of logic elements.

| Parameter | Delay (ps) |
|---|---|
| $t_{pcq\_PC}$ | 40 |
| $t_{mem}$ | 200 |
| $t_{dec}$ | 70 |
| $t_{mux}$ | 25 |
| $t_{RFread}$ | 100 |
| $t_{ALU}$ | 120 |
| $t_{RFsetup}$ | 60 |

$$\boxed{\begin{aligned}T_c = {} & t_{pcq\_PC} + 2{*}t_{mem} + t_{dec} + t_{RFread} + t_{ALU} + \\ & 2{*}t_{mux} + t_{RFsetup}\end{aligned}}$$

## Exercise 2: Performance Analysis

```
C code:
   int i;
   int sum = 0;

   for (i = 0; i < 10; i = i + 1)
      sum = sum + i;
```

```
Assembly code:
; R0 = i
; R1 = sum
   MOV  R0,  #0
   MOV  R1,  #0
COND
   CMP  R0,  #10
   BLT  FOR
   B    DONE
FOR
   ADD  R1,  R1,  R0
   ADD  R0,  R0,  #1
   B    COND
DONE
```

```
Assembly code:
; R0 = i
; R1 = sum
   MOV  R0,  #0
   MOV  R1,  #0
FOR
   CMP  R0,  #10
   BGE  DONE
   ADD  R1,  R1,  R0
   ADD  R0,  R0,  #1
   B    FOR
DONE
```

- Find the execution time for each of the two implementations of the **`for`** loop. Use CPU parameters from next slide.

## Drawbacks of Single-Cycle CPU

- Requires two memories (no reuse)
- Requires three adders (no reuse)
- Clock period is dictated by the slowest instruction
    - No way to make the *common case* **fast** (e.g., DP instructions)

## Coming Attractions

# Multi-Cycle CPU

- Divide each instruction into a number of steps

- Perform one step in one clock cycle (instead of an entire instruction)

- Need non-architectural (microarchitectural) registers to store intermediate state

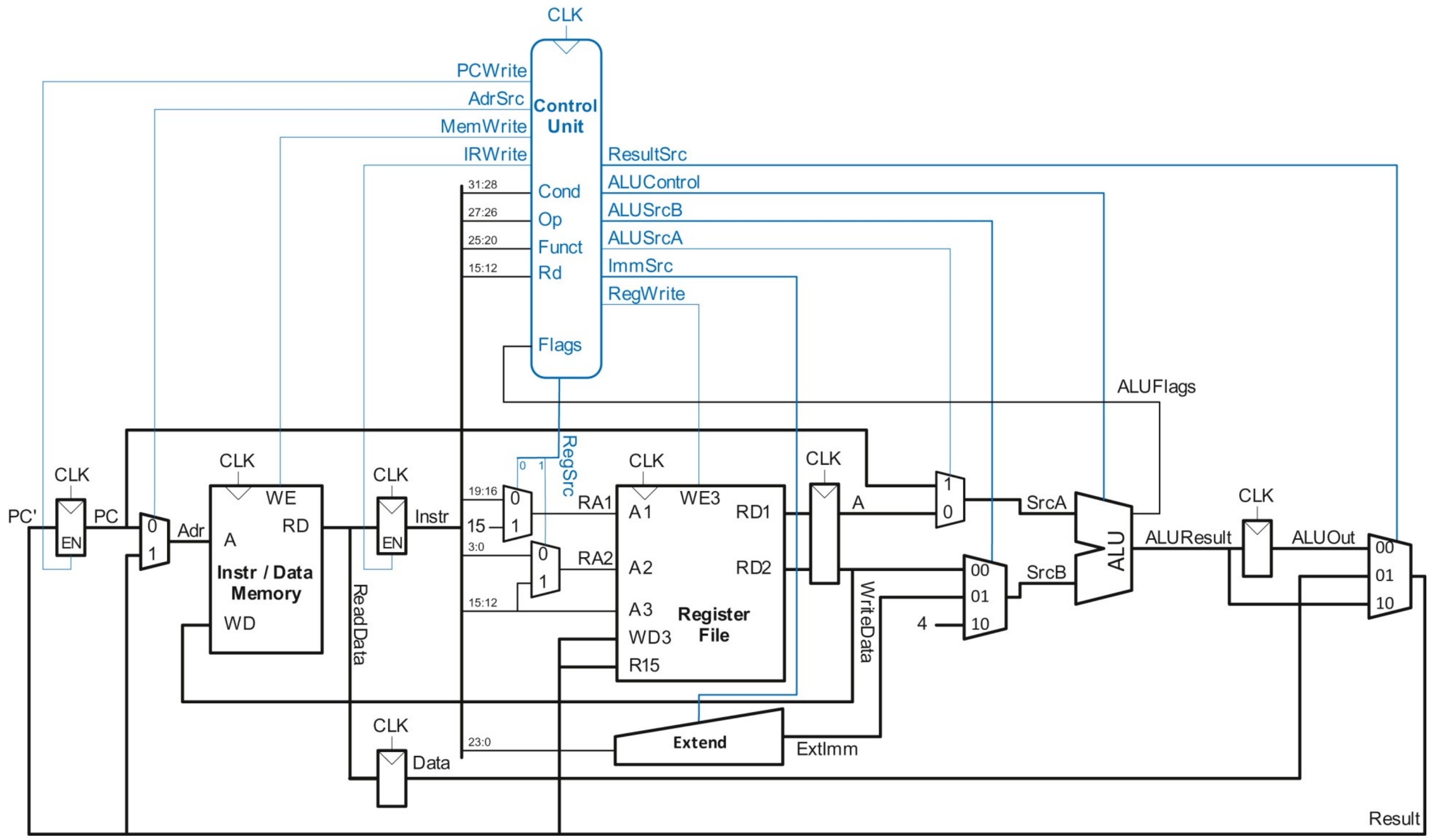
- Need an FSM-based controller to transition between steps
  - Different control signals on different steps

Section 7.4 of H&H

- **After the teaching break: Possible ext. for assignment 1**

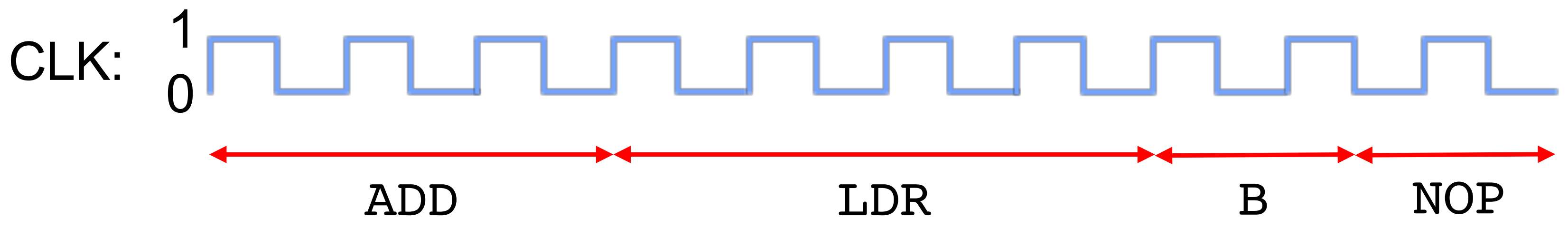
# Multi-Cycle CPU Sneak Peek (Week 7)

- Can you spot the non-architectural state (registers)?



Section 7.4 of H&H

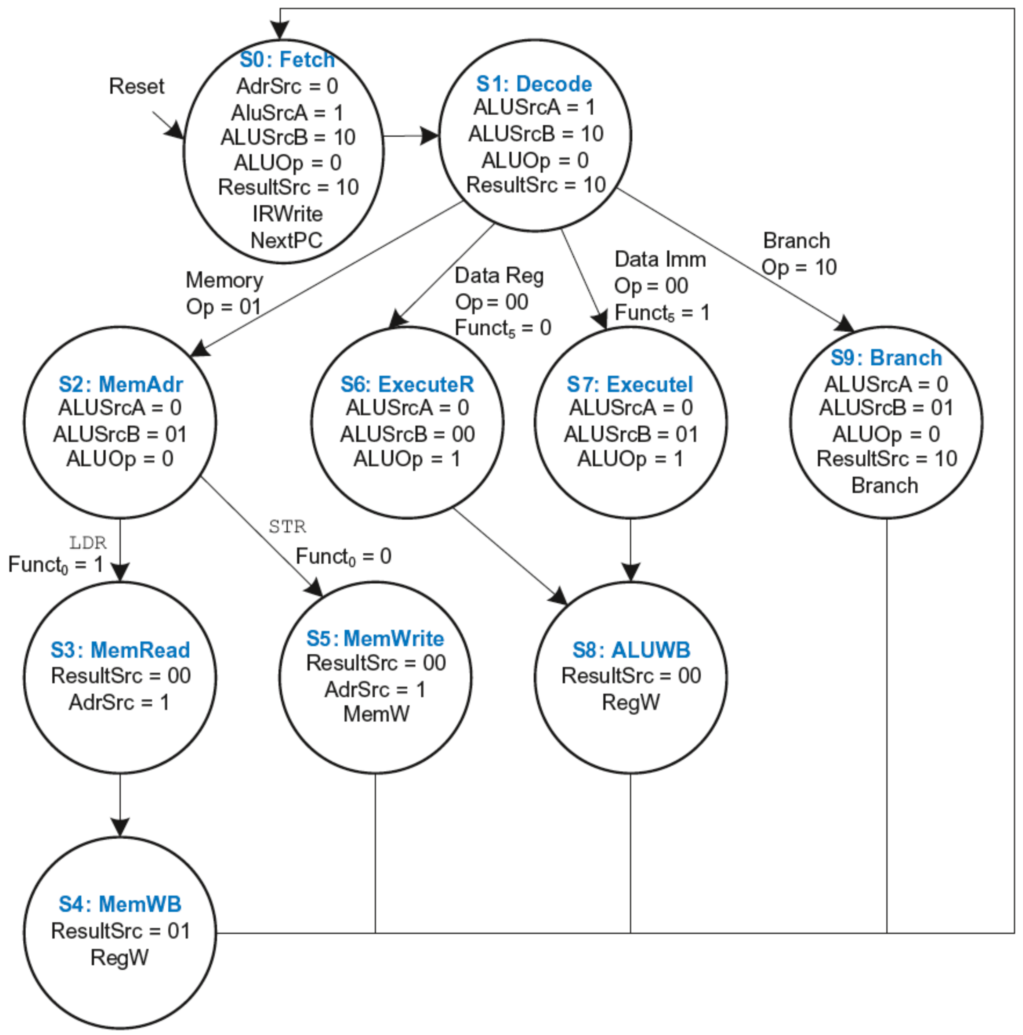
## Multi-Cycle CPU Cycle by Cycle (Week 7)



- Hypothetical multi-cycle CPU
  - ADD and SUB takes 3 cycles
  - LDR and STR take 4 cycles
  - Unconditional branch takes 1 cycle

## Multi-Cycle Control Unit FSM (Week 7)

| State | Datapath µOp |
|---|---|
| Fetch | Instr ← Mem[PC]; PC ← PC+4 |
| Decode | ALUOut ← PC+4 |
| MemAdr | ALUOut ← Rn + Imm |
| MemRead | Data ← Mem[ALUOut] |
| MemWB | Rd ← Data |
| MemWrite | Mem[ALUOut] ← Rd |
| ExecuteR | ALUOut ← Rn op Rm |
| ExecuteI | ALUOut ← Rn op Imm |
| ALUWB | Rd ← ALUOut |
| Branch | PC ← R15 + offset |



## ISA Tradeoffs

# ISA Impacts Software and Hardware

- **Complex instructions**
  - (Upside) Dense and efficient code
  - (Downside) Complex circuits with longer critical paths
  - Example: x86 operate instructions can have both register and memory operands
    - Register-Memory architecture

- **Simple instructions**
  - (Upside) Simple circuits (microarchitecture)
  - (Downside) Large instruction footprint (many instruction to solve the same problem)
  - (Downside) Big semantic gap between high-level code and assembly code
  - Example: ARM allows accessing memory only via LDR/STR
    - Load-Store architecture

- **Number of Registers (tradeoff)**
  - Large register file demands more space in the ISA for encoding
  - But, more registers reduce trips to memory (memory references)

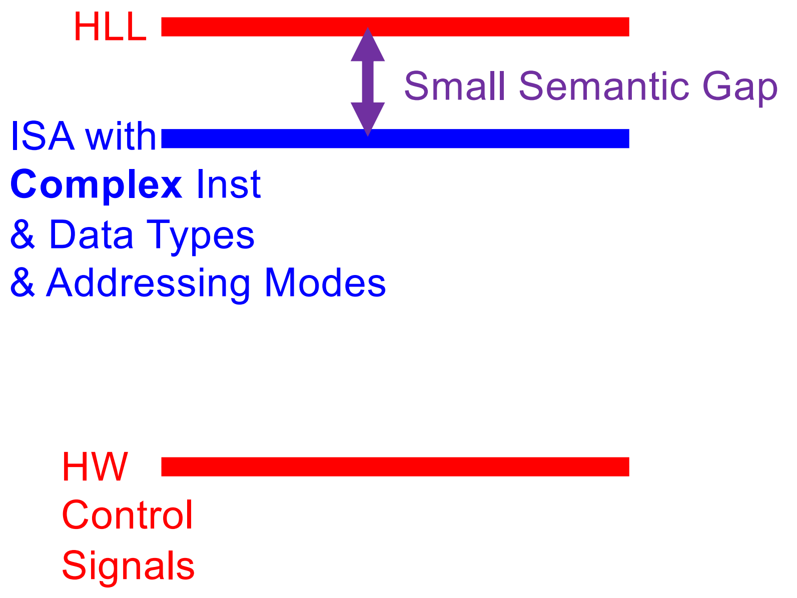
## ISA Impacts Software and Hardware

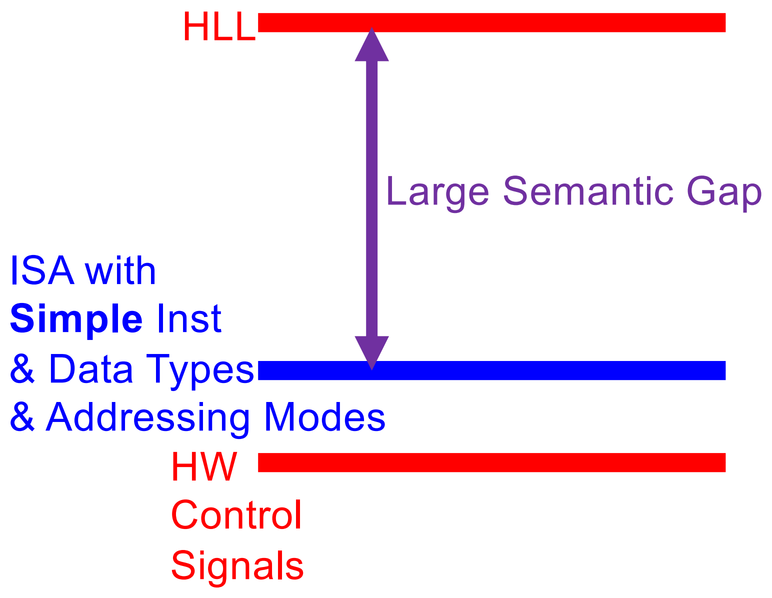
- ISA impacts
  - Performance
  - Power and energy
  - Code size and instruction footprint
  - Circuit cost and complexity (chip area)
  - Future growth (ISA evolution)

## Semantic Gap

- How close instructions & data types & addressing modes are to high-level language (HLL)

## Semantic Gap





Easier mapping of HLL to ISA
**Less work for software designer**
**More work for hardware designer**
Optimization burden on HW

Harder mapping of HLL to ISA
**More work for software designer**
**Less work for hardware designer**
Optimization burden on SW

## Addressing Mode Tradeoffs

## Addressing Modes

- Addressing mode **specifies** how instruction operands are **addressed**
  - Source and destination registers
  - Target address of a memory reference
  - Target address that a branch will jump to

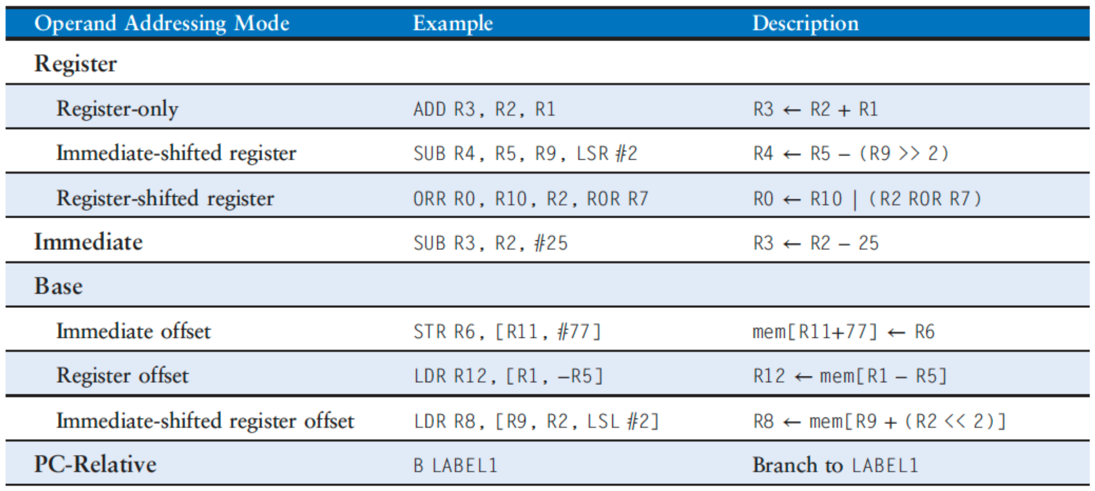
- ARM uses four main modes
  - Register
  - Immediate
  - Base
  - PC-relative

- First three modes for reading/writing operands
- Last mode is for writing the program counter

# ARM Addressing Modes

- Some of the addressing modes allow the second source operand to be shifted
  - Check your references for details

**Table 6.12** ARM operand addressing modes



| Operand Addressing Mode | Example | Description |
|---|---|---|
| Register | | |
| Register-only | `ADD R3, R2, R1` | R3 ← R2 + R1 |
| Immediate-shifted register | `SUB R4, R5, R9, LSR #2` | R4 ← R5 − (R9 >> 2) |
| Register-shifted register | `ORR R0, R10, R2, ROR R7` | R0 ← R10 \| (R2 ROR R7) |
| Immediate | `SUB R3, R2, #25` | R3 ← R2 − 25 |
| Base | | |
| Immediate offset | `STR R6, [R11, #77]` | mem[R11+77] ← R6 |
| Register offset | `LDR R12, [R1, −R5]` | R12 ← mem[R1 − R5] |
| Immediate-shifted register offset | `LDR R8, [R9, R2, LSL #2]` | R8 ← mem[R9 + (R2 << 2)] |
| PC-Relative | `B LABEL1` | Branch to LABEL1 |

Section 6.4.4 of H&H

## Addressing Mode Tradeoffs

- Complex addressing modes simplify high-level code to assembly translation

- But they result in more complex circuits (microarchitecture)
  - ALU to add base and offset
  - Shifter in front of ALU

- Where to place the burden of optimization? Software or Hardware
  - Many simple instructions + Simple microarchitecture
  - Few complex instructions + Complex microarchitecture

## Aside: Data Dependences

- In Von Neumann model, instructions depend on each other for data
- **Data (True) Dependence:** One instruction **produces** a result that the subsequent instruction **consumes**

## Aside: Data Dependences

- One can visualize a **sequential program** as an instruction flow or **data flow**

## Aside: Data Dependences

- Data dependence implies the two instructions must execute in program order
  - They cannot be executed simultaneously **(in parallel at the same time)**
- There are also **control dependences** due to branches as instruction can only execute if a branch evaluates to TRUE
- **And false dependences** (we will see the details later)

# Implication for microarchitecture

- In the end we care about the **correctness of the program**
  - From the **initial** architectural state to the **final** architectural state

- Preserving data flow (not instruction flow) is critical for program correctness

- Single-cycle CPU is one way to satisfy the program correctness criteria
  - Very strict and highly constrained. And hence, poor performance

- High performance requires out of the box thinking
  - **Key technique is parallelism:** we must execute several **(independent)** instructions at the same time

- Understanding dependences is the key to unlocking parallelism

## Aside: What if a machine processes instructions out of program order?

- What does the programmer care about?

```
i1: CMP  R0,  #10
i2: BGE  DONE
i3: ADD  R1,  R1,  R0
i4: ADD  R0,  R0,  #1
```

- Does the programmer care if `i3` executed before `i4`?
  - No: Programmer only cares R1 was updated before R0
  - Can update AS in program order and process instructions out of order (OOO)

- Why would a machine ever do that?
  - Fact: Almost EVERY high-performance computer does that!
  - In-program-order instruction processing (execution) is an illusion in high-end computers

**We will meet after two weeks**

**Revise the lecture content and do the quiz**

**Finish assignment 1**

## Shift Instructions

*Category: Data Processing*

# Shift Instructions

- Shift the value in a register left or right, drop bits off the end
  - **L**ogical **S**hift **L**eft (**LSL**)
  - **L**ogical **S**hift **R**ight (**LSR**)
  - **A**rithmetic **S**hift **R**ight (**ASR**)
  - **R**otate **R**ight (**ROR**)

- **Logical Shift:** shifts the number to the left/right and fills empty slots with zero

- **Arithmetic Shift:** On right shifts fill the most significant bits with the sign bit

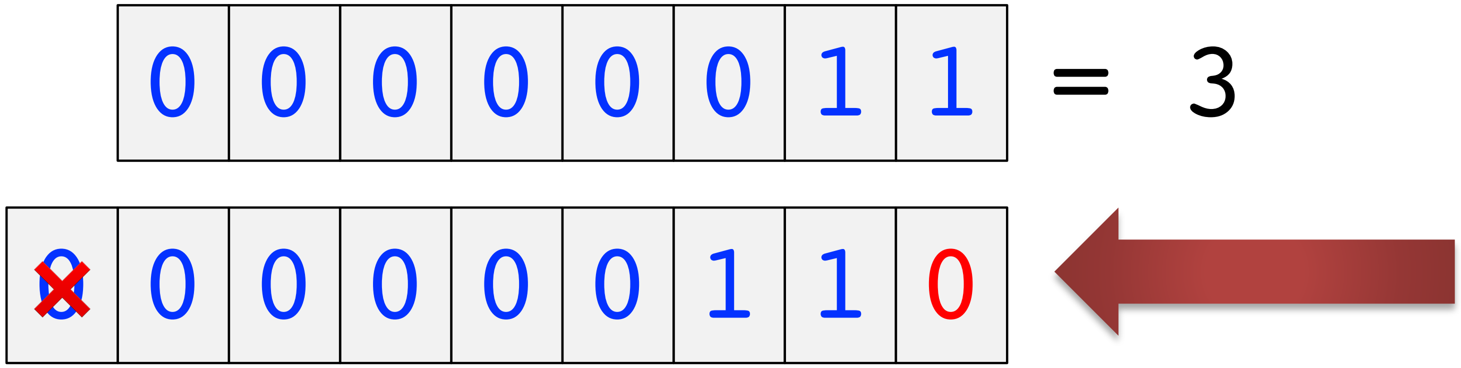
- **Rotate:** rotates number in a circle such that empty spots are filled with bits shifted off the other end

## Logical Shift Left (LSL)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |

- Binary Number in Decimal = 3

# Logical Shift Left (LSL)



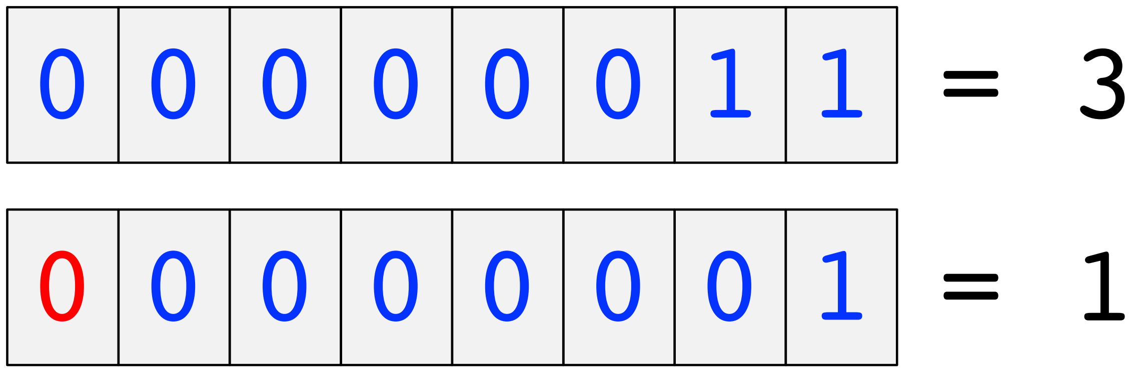
- Shift the number LEFT by ONE BIT
- INSERT 0 in Least Significant Position
- Get **RID** of the Most Significant BIT

## Logical Shift Left (LSL)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | = | 3 |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | = | 6 |

- Binary Number after shift in Decimal = 6
- SHIFT LEFT = MULTIPLY BY 2

## Logical Shift Right (LSR)

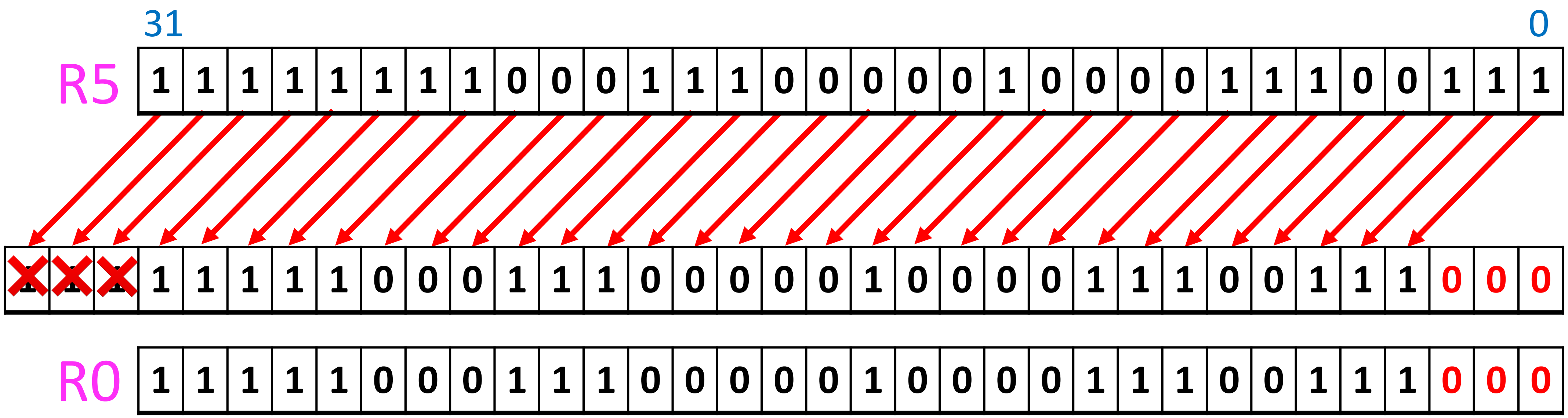


- Binary Number after right shift in Decimal = 1
- SHIFT RIGHT = DIVIDE BY 2

# Logical Shift Left (LSL)

ARM Instruction

```
LSL  R0,  R5,  #3
```

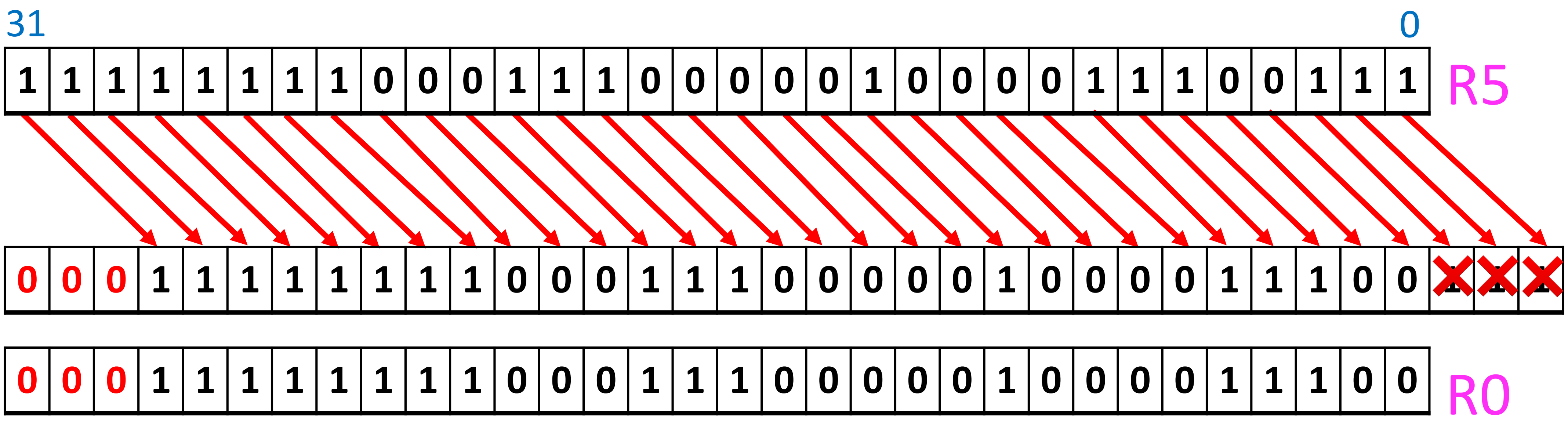


- Shift all bits left **3** positions, fill **3** least significant bits with 0's
- Drop the 3 bits off the end

# Logical Shift Right (LSR)

ARM Instruction

```
LSR   R0,   R5,   #3
```

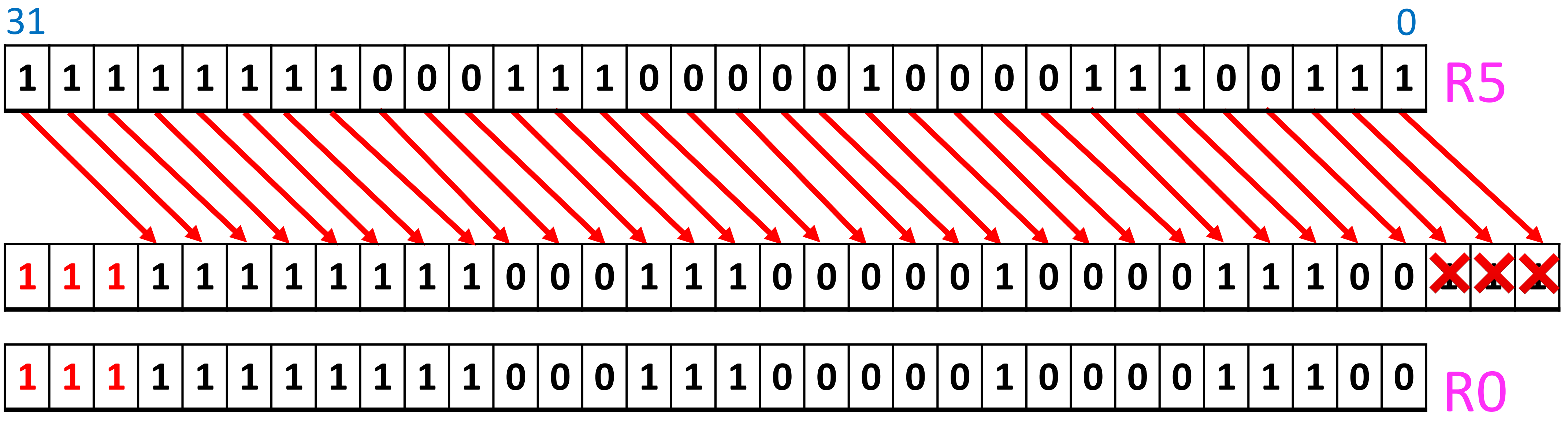


- Shift all bits right 3 positions, insert three 0’s from the right
- Drop the 3 bits from the left

# Arithmetic Shift Right (ASR)

ARM Instruction

```
ASR  R0,  R5,  #3
```



- Shift all bits right 3 positions, insert three 1's from the right
- Drop the 3 bits from the left

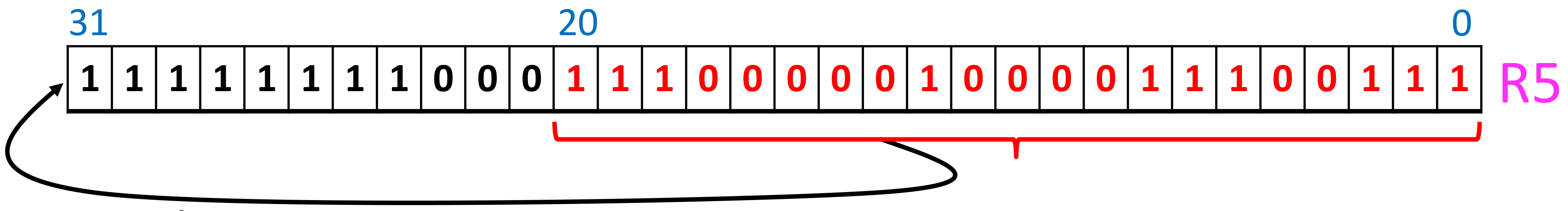
# Rotate Right (ROR)

ARM Instruction

```
ROR  R0,  R5,  #21
```

| 31 | | | | | | | | | | | 20 | | | | | | | | | | | | | | | | | | | | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | R5 |

- Do a circular shift
- Right shift by 21 and put back bits that fall off at left end



*Result*

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | R0 |

## Binary Encoding of Shift Instructions

- See Chapter 6 for encoding of all instructions

- Section 6.4 of H&H

# Shifts: Machine Representation

| | 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:4 | | | | | | | | 3:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **DP-R** | cond | 00 | 0 | cmd | S | Rn | Rd | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rm |

**Shift Instructions**

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:7 | 6:5 | 4 | 3:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| cond | 00 | 0 | 1101 | S | 0000 | Rd | shamt5 | sh | 0 | Rm |

- cmd = 1101
- sh = 00 (LSL), 01 (LSR), 10 (ASR), 11 (ROR)
- Rn = 0
- shamt5 = 5-bit shift amount

# Shifts: Machine Representation

- Format (Src2 = Register)

```
LSL R0,  R5,  #3
    ↓    ↓    ↓
LSL Rd,  Rm,  shamt5
```

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:7 | 6:5 | 4 | 3:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| cond | 00 | 0 | cmd | S | Rn | Rd | shamt5 | sh | 0 | Rm |

## Shifts: Machine Representation

- ARM also has instructions with shift amount held in a register

```
LSL  R4,  R8,  R6
```

```
ROR  R5,  R8,  R6
```

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:8 | 7 | 6:5 | 4 | 3:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 00 | 0 | cmd | S | Rn | Rd | Rs | 0 | sh | 1 | Rm |

## Exercise: Machine Representation

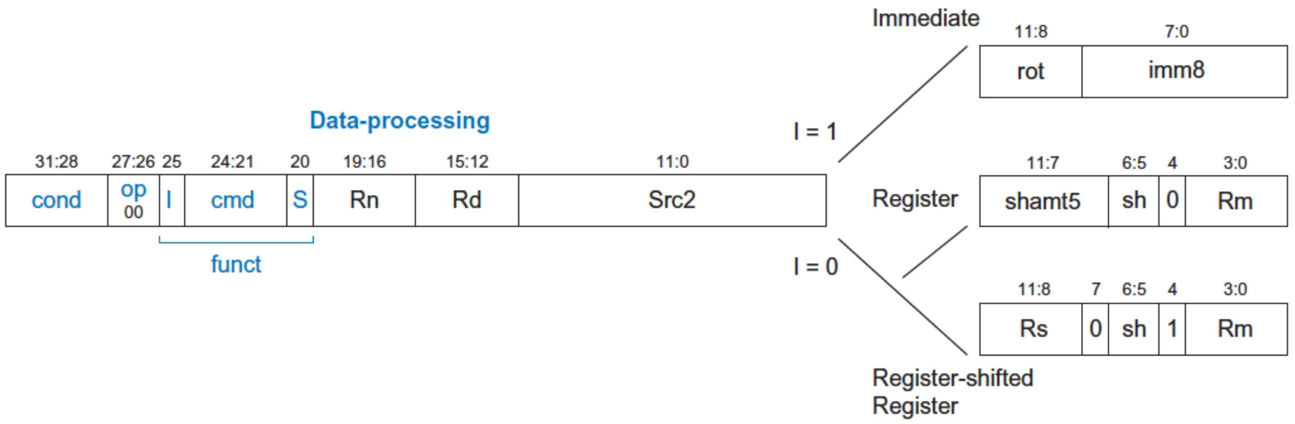
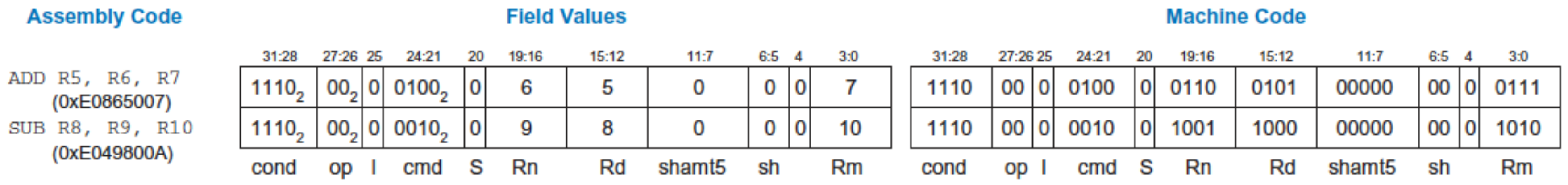


**Figure 6.17** Data-processing instruction format showing the *funct* field and *Src2* variations

**Table 6.8** *sh* field encodings

| Instruction | sh | Operation |
|---|---|---|
| LSL | $00_2$ | Logical shift left |
| LSR | $01_2$ | Logical shift right |
| ASR | $10_2$ | Arithmetic shift right |
| ROR | $11_2$ | Rotate right |

# Exercise: Machine Representation



**Assembly Code**

```
ADD R5, R6, R7
    (0xE0865007)
SUB R8, R9, R10
    (0xE049800A)
```

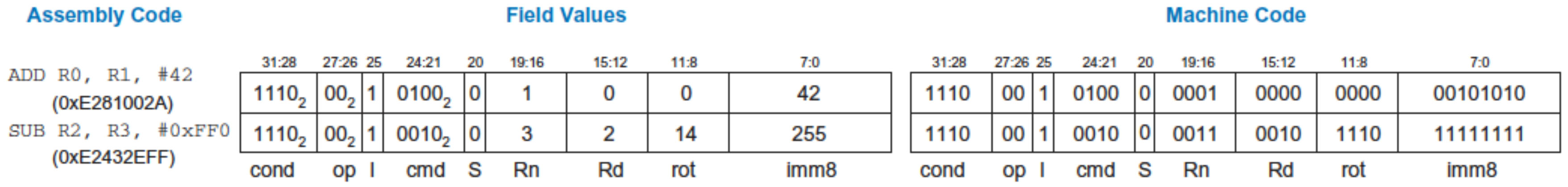
**Field Values**

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:7 | 6:5 | 4 | 3:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| $1110_2$ | $00_2$ | 0 | $0100_2$ | 0 | 6 | 5 | 0 | 0 | 0 | 7 |
| $1110_2$ | $00_2$ | 0 | $0010_2$ | 0 | 9 | 8 | 0 | 0 | 0 | 10 |
| cond | op | I | cmd | S | Rn | Rd | shamt5 | sh | | Rm |

**Machine Code**

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:7 | 6:5 | 4 | 3:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1110 | 00 | 0 | 0100 | 0 | 0110 | 0101 | 00000 | 00 | 0 | 0111 |
| 1110 | 00 | 0 | 0010 | 0 | 1001 | 1000 | 00000 | 00 | 0 | 1010 |
| cond | op | I | cmd | S | Rn | Rd | shamt5 | sh | | Rm |

**Figure 6.18** Data-processing instructions with three register operands



**Assembly Code**

```
ADD R0, R1, #42
    (0xE281002A)
SUB R2, R3, #0xFF0
    (0xE2432EFF)
```

**Field Values**

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:8 | 7:0 |
|---|---|---|---|---|---|---|---|---|
| $1110_2$ | $00_2$ | 1 | $0100_2$ | 0 | 1 | 0 | 0 | 42 |
| $1110_2$ | $00_2$ | 1 | $0010_2$ | 0 | 3 | 2 | 14 | 255 |
| cond | op | I | cmd | S | Rn | Rd | rot | imm8 |

**Machine Code**

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:8 | 7:0 |
|---|---|---|---|---|---|---|---|---|
| 1110 | 00 | 1 | 0100 | 0 | 0001 | 0000 | 0000 | 00101010 |
| 1110 | 00 | 1 | 0010 | 0 | 0011 | 0010 | 1110 | 11111111 |
| cond | op | I | cmd | S | Rn | Rd | rot | imm8 |

**Figure 6.19** Data-processing instructions with an immediate and two register operands

# Recall: DP with Src2 as Register

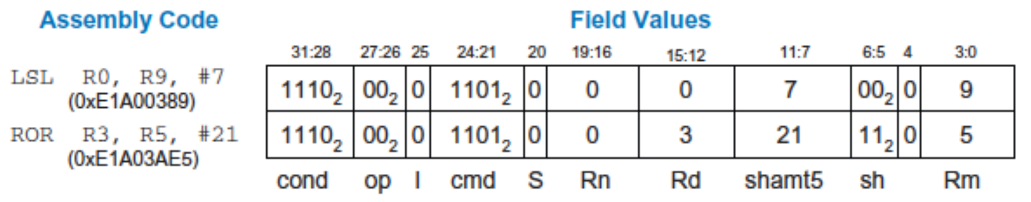
- Bit 25 (I) informs the CPU how to interpret Src2
  - I = 0, CPU interprets Src2[3:0] as a register

- Format (Src2 = Register)

$$\begin{tikzcd} \texttt{ADD} & \texttt{R0,} \arrow[d] & \texttt{R1,} \arrow[d] & \texttt{R3} \arrow[d] \\ \texttt{ADD} & \texttt{Rd,} & \texttt{Rn,} & \texttt{Rm} \end{tikzcd}$$

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:4 | | | | | | | | 3:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 00 | 0 | cmd | S | Rn | Rd | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rm |

# Exercise: Machine Representation

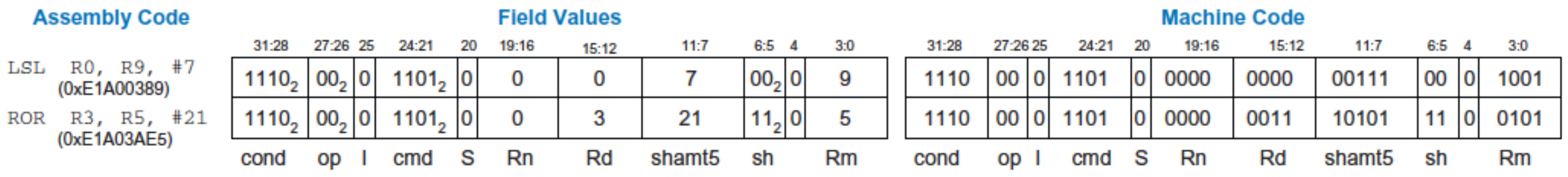


Assembly Code | Field Values

| Assembly Code | 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:7 | 6:5 | 4 | 3:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| `LSL R0, R9, #7` (0xE1A00389) | $1110_2$ | $00_2$ | 0 | $1101_2$ | 0 | 0 | 0 | 7 | $00_2$ | 0 | 9 |
| `ROR R3, R5, #21` (0xE1A03AE5) | $1110_2$ | $00_2$ | 0 | $1101_2$ | 0 | 0 | 3 | 21 | $11_2$ | 0 | 5 |
| | cond | op | I | cmd | S | Rn | Rd | shamt5 | sh | | Rm |

**Figure 6.20** **Shift instructions with imm**

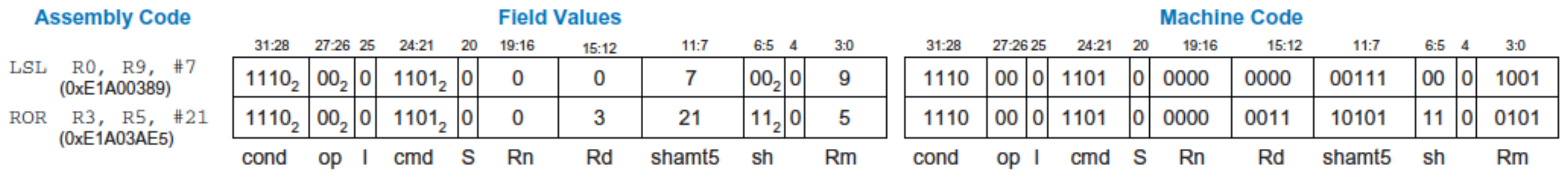
# Exercise: Machine Representation



| Assembly Code | Field Values | | | | | | | | | | | Machine Code | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:7 | 6:5 | 4 | 3:0 | 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:7 | 6:5 | 4 | 3:0 |
| LSL R0, R9, #7 (0xE1A00389) | $1110_2$ | $00_2$ | 0 | $1101_2$ | 0 | 0 | 0 | 7 | $00_2$ | 0 | 9 | 1110 | 00 | 0 | 1101 | 0 | 0000 | 0000 | 00111 | 00 | 0 | 1001 |
| ROR R3, R5, #21 (0xE1A03AE5) | $1110_2$ | $00_2$ | 0 | $1101_2$ | 0 | 0 | 3 | 21 | $11_2$ | 0 | 5 | 1110 | 00 | 0 | 1101 | 0 | 0000 | 0011 | 10101 | 11 | 0 | 0101 |
| | cond | op | I | cmd | S | Rn | Rd | shamt5 | sh | | Rm | cond | op | I | cmd | S | Rn | Rd | shamt5 | sh | | Rm |

**Figure 6.20** **Shift instructions with immediate shift amounts**

# Exercise: Machine Representation



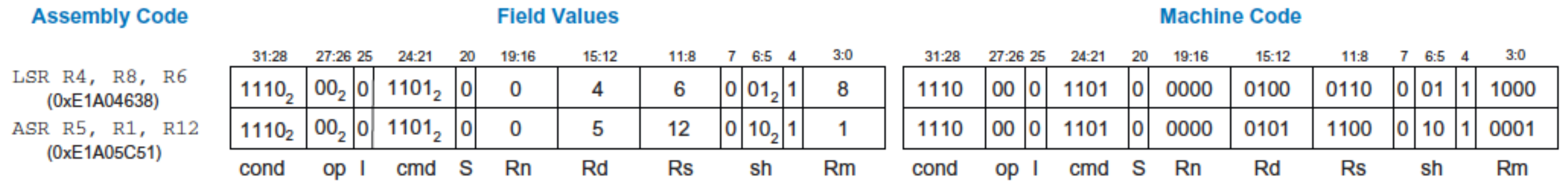
**Field Values**

| Assembly Code | 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:7 | 6:5 | 4 | 3:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| `LSL R0, R9, #7` (0xE1A00389) | $1110_2$ | $00_2$ | 0 | $1101_2$ | 0 | 0 | 0 | 7 | $00_2$ | 0 | 9 |
| `ROR R3, R5, #21` (0xE1A03AE5) | $1110_2$ | $00_2$ | 0 | $1101_2$ | 0 | 0 | 3 | 21 | $11_2$ | 0 | 5 |
| | cond | op | I | cmd | S | Rn | Rd | shamt5 | sh | | Rm |

**Machine Code**

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:7 | 6:5 | 4 | 3:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1110 | 00 | 0 | 1101 | 0 | 0000 | 0000 | 00111 | 00 | 0 | 1001 |
| 1110 | 00 | 0 | 1101 | 0 | 0000 | 0011 | 10101 | 11 | 0 | 0101 |
| cond | op | I | cmd | S | Rn | Rd | shamt5 | sh | | Rm |

**Figure 6.20** Shift instructions with immediate shift amounts



**Field Values**

| Assembly Code | 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:8 | 7 | 6:5 | 4 | 3:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `LSR R4, R8, R6` (0xE1A04638) | $1110_2$ | $00_2$ | 0 | $1101_2$ | 0 | 0 | 4 | 6 | 0 | $01_2$ | 1 | 8 |
| `ASR R5, R1, R12` (0xE1A05C51) | $1110_2$ | $00_2$ | 0 | $1101_2$ | 0 | 0 | 5 | 12 | 0 | $10_2$ | 1 | 1 |
| | cond | op | I | cmd | S | Rn | Rd | Rs | | sh | | Rm |

**Machine Code**

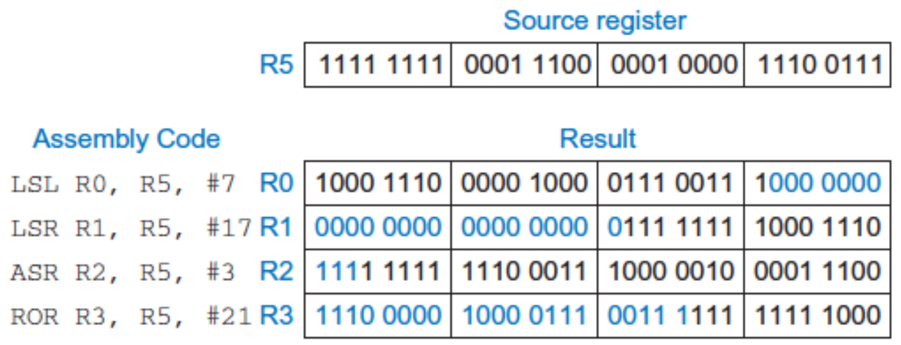
| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:8 | 7 | 6:5 | 4 | 3:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1110 | 00 | 0 | 1101 | 0 | 0000 | 0100 | 0110 | 0 | 01 | 1 | 1000 |
| 1110 | 00 | 0 | 1101 | 0 | 0000 | 0101 | 1100 | 0 | 10 | 1 | 0001 |
| cond | op | I | cmd | S | Rn | Rd | Rs | | sh | | Rm |

**Figure 6.21** Shift instructions with register shift amounts

## Use of Shift Instructions

- **Left shift** by N = Multiplication by $2^N$
- **Arithmetic right shift** by N = Division by $2^N$
- **Extract bits or assemble new bit patterns**
  - Network programming
  - Cryptography
  - Compression of data

# Examples of Shift Instructions

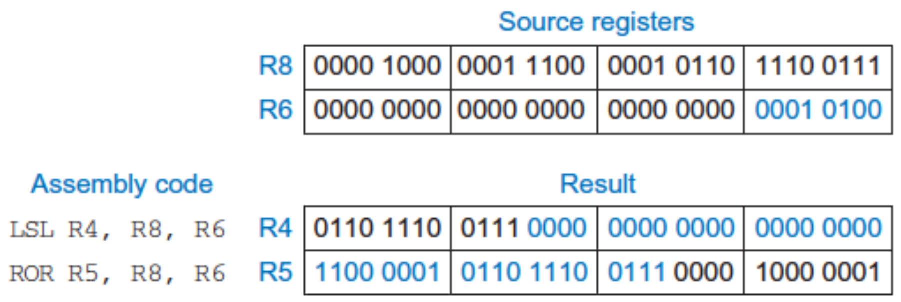


Source register

|  |  |  |  |  |
|---|---|---|---|---|
| R5 | 1111 1111 | 0001 1100 | 0001 0000 | 1110 0111 |

| Assembly Code | Result | | | | |
|---|---|---|---|---|---|
| `LSL R0, R5, #7` | R0 | 1000 1110 | 0000 1000 | 0111 0011 | 1000 0000 |
| `LSR R1, R5, #17` | R1 | 0000 0000 | 0000 0000 | 0111 1111 | 1000 1110 |
| `ASR R2, R5, #3` | R2 | 1111 1111 | 1110 0011 | 1000 0010 | 0001 1100 |
| `ROR R3, R5, #21` | R3 | 1110 0000 | 1000 0111 | 0011 1111 | 1111 1000 |

**Figure 6.4** Shift instructions with immediate shift amounts



Source registers

|  |  |  |  |  |
|---|---|---|---|---|
| R8 | 0000 1000 | 0001 1100 | 0001 0110 | 1110 0111 |
| R6 | 0000 0000 | 0000 0000 | 0000 0000 | 0001 0100 |

| Assembly code | Result | | | | |
|---|---|---|---|---|---|
| `LSL R4, R8, R6` | R4 | 0110 1110 | 0111 0000 | 0000 0000 | 0000 0000 |
| `ROR R5, R8, R6` | R5 | 1100 0001 | 0110 1110 | 0111 0000 | 1000 0001 |

**Figure 6.5** Shift instructions with register shift amounts

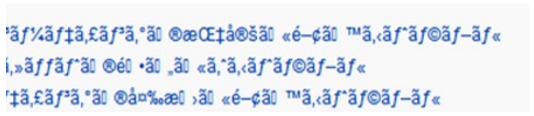
Shift amount can be in a register

Page 305 of H&H

## Manipulating Characters & Bytes

# Characters & Encoding

- Reading and writing text is ubiquitous
  - Different devices (tablet, laptop, desktop, mobile)
  - Different applications (word, whatsapp, email)
  - Different manufactures (Apple, Intel, Samsung)
- Need a **standardized way** to represent characters that make up text
  - Use standardized **byte codes** *to* **represent characters**
  - **Things still go wrong!**



³ãƒ¼ãƒ‡ã‚£ãƒ³ã‚°ã® ®æŒ‡å®šã® «é–¢ã® ™ã‚‹ãƒˆãƒ©ãƒ–ãƒ«
ã‚»ãƒƒãƒˆã® ®é® •ã® „ã® «ã‚ˆã‚‹ãƒˆãƒ©ãƒ–ãƒ«
ƒ‡ã‚£ãƒ³ã‚°ã® ®å¤‰æ® ›ã® «é–¢ã® ™ã‚‹ãƒˆãƒ©ãƒ–ãƒ«

## Thinking about Character Input/Output

- Keyboard data is captured in a register

- Some binary data is sent to a special memory associated with graphics chip to display the character

## Manipulating Characters

- Manipulating characters is common

- We need architectural support for manipulating characters

- Character is the same as a byte
    - So, we need architectural support for manipulating bytes
    - Regular LDR/STR deal with words (not bytes)

## ASCII Encoding

- English characters can be encoded in a single byte (< 256)
- **1963:** `ASCII` was developed
  - American Standard Code for Information & Interchange
  - Assigns each text character a **unique** byte
  - **Information exchange** became feasible across **manufactures** and **geographical boundaries**
- The C language uses the type `char` to represent byte or character
- **Optimize the common case of manipulating bytes**

## Other Encodings

- Other programming languages such as Java, use different character encodings

- Unicode is the most well-known

- 16 bits to represent accents, Asian languages, and more

    - `www.unicode.org`

# Decimal - Binary - Octal - Hex – ASCII Conversion Chart

| Decimal | Binary | Octal | Hex | ASCII | Decimal | Binary | Octal | Hex | ASCII | Decimal | Binary | Octal | Hex | ASCII | Decimal | Binary | Octal | Hex | ASCII |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 00000000 | 000 | 00 | NUL | 32 | 00100000 | 040 | 20 | SP | 64 | 01000000 | 100 | 40 | @ | 96 | 01100000 | 140 | 60 | ` |
| 1 | 00000001 | 001 | 01 | SOH | 33 | 00100001 | 041 | 21 | ! | 65 | 01000001 | 101 | 41 | A | 97 | 01100001 | 141 | 61 | a |
| 2 | 00000010 | 002 | 02 | STX | 34 | 00100010 | 042 | 22 | “ | 66 | 01000010 | 102 | 42 | B | 98 | 01100010 | 142 | 62 | b |
| 3 | 00000011 | 003 | 03 | ETX | 35 | 00100011 | 043 | 23 | # | 67 | 01000011 | 103 | 43 | C | 99 | 01100011 | 143 | 63 | c |
| 4 | 00000100 | 004 | 04 | EOT | 36 | 00100100 | 044 | 24 | $ | 68 | 01000100 | 104 | 44 | D | 100 | 01100100 | 144 | 64 | d |
| 5 | 00000101 | 005 | 05 | ENQ | 37 | 00100101 | 045 | 25 | % | 69 | 01000101 | 105 | 45 | E | 101 | 01100101 | 145 | 65 | e |
| 6 | 00000110 | 006 | 06 | ACK | 38 | 00100110 | 046 | 26 | & | 70 | 01000110 | 106 | 46 | F | 102 | 01100110 | 146 | 66 | f |
| 7 | 00000111 | 007 | 07 | BEL | 39 | 00100111 | 047 | 27 | ‘ | 71 | 01000111 | 107 | 47 | G | 103 | 01100111 | 147 | 67 | g |
| 8 | 00001000 | 010 | 08 | BS | 40 | 00101000 | 050 | 28 | ( | 72 | 01001000 | 110 | 48 | H | 104 | 01101000 | 150 | 68 | h |
| 9 | 00001001 | 011 | 09 | HT | 41 | 00101001 | 051 | 29 | ) | 73 | 01001001 | 111 | 49 | I | 105 | 01101001 | 151 | 69 | i |
| 10 | 00001010 | 012 | 0A | LF | 42 | 00101010 | 052 | 2A | * | 74 | 01001010 | 112 | 4A | J | 106 | 01101010 | 152 | 6A | j |
| 11 | 00001011 | 013 | 0B | VT | 43 | 00101011 | 053 | 2B | + | 75 | 01001011 | 113 | 4B | K | 107 | 01101011 | 153 | 6B | k |
| 12 | 00001100 | 014 | 0C | FF | 44 | 00101100 | 054 | 2C | , | 76 | 01001100 | 114 | 4C | L | 108 | 01101100 | 154 | 6C | l |
| 13 | 00001101 | 015 | 0D | CR | 45 | 00101101 | 055 | 2D | - | 77 | 01001101 | 115 | 4D | M | 109 | 01101101 | 155 | 6D | m |
| 14 | 00001110 | 016 | 0E | SO | 46 | 00101110 | 056 | 2E | . | 78 | 01001110 | 116 | 4E | N | 110 | 01101110 | 156 | 6E | n |
| 15 | 00001111 | 017 | 0F | SI | 47 | 00101111 | 057 | 2F | / | 79 | 01001111 | 117 | 4F | O | 111 | 01101111 | 157 | 6F | o |
| 16 | 00010000 | 020 | 10 | DLE | 48 | 00110000 | 060 | 30 | 0 | 80 | 01010000 | 120 | 50 | P | 112 | 01110000 | 160 | 70 | p |
| 17 | 00010001 | 021 | 11 | DC1 | 49 | 00110001 | 061 | 31 | 1 | 81 | 01010001 | 121 | 51 | Q | 113 | 01110001 | 161 | 71 | q |
| 18 | 00010010 | 022 | 12 | DC2 | 50 | 00110010 | 062 | 32 | 2 | 82 | 01010010 | 122 | 52 | R | 114 | 01110010 | 162 | 72 | r |
| 19 | 00010011 | 023 | 13 | DC3 | 51 | 00110011 | 063 | 33 | 3 | 83 | 01010011 | 123 | 53 | S | 115 | 01110011 | 163 | 73 | s |
| 20 | 00010100 | 024 | 14 | DC4 | 52 | 00110100 | 064 | 34 | 4 | 84 | 01010100 | 124 | 54 | T | 116 | 01110100 | 164 | 74 | t |
| 21 | 00010101 | 025 | 15 | NAK | 53 | 00110101 | 065 | 35 | 5 | 85 | 01010101 | 125 | 55 | U | 117 | 01110101 | 165 | 75 | u |
| 22 | 00010110 | 026 | 16 | SYN | 54 | 00110110 | 066 | 36 | 6 | 86 | 01010110 | 126 | 56 | V | 118 | 01110110 | 166 | 76 | v |
| 23 | 00010111 | 027 | 17 | ETB | 55 | 00110111 | 067 | 37 | 7 | 87 | 01010111 | 127 | 57 | W | 119 | 01110111 | 167 | 77 | w |
| 24 | 00011000 | 030 | 18 | CAN | 56 | 00111000 | 070 | 38 | 8 | 88 | 01011000 | 130 | 58 | X | 120 | 01111000 | 170 | 78 | x |
| 25 | 00011001 | 031 | 19 | EM | 57 | 00111001 | 071 | 39 | 9 | 89 | 01011001 | 131 | 59 | Y | 121 | 01111001 | 171 | 79 | y |
| 26 | 00011010 | 032 | 1A | SUB | 58 | 00111010 | 072 | 3A | : | 90 | 01011010 | 132 | 5A | Z | 122 | 01111010 | 172 | 7A | z |
| 27 | 00011011 | 033 | 1B | ESC | 59 | 00111011 | 073 | 3B | ; | 91 | 01011011 | 133 | 5B | [ | 123 | 01111011 | 173 | 7B | { |
| 28 | 00011100 | 034 | 1C | FS | 60 | 00111100 | 074 | 3C | < | 92 | 01011100 | 134 | 5C | \ | 124 | 01111100 | 174 | 7C | \| |
| 29 | 00011101 | 035 | 1D | GS | 61 | 00111101 | 075 | 3D | = | 93 | 01011101 | 135 | 5D | ] | 125 | 01111101 | 175 | 7D | } |
| 30 | 00011110 | 036 | 1E | RS | 62 | 00111110 | 076 | 3E | > | 94 | 01011110 | 136 | 5E | ^ | 126 | 01111110 | 176 | 7E | ~ |
| 31 | 00011111 | 037 | 1F | US | 63 | 00111111 | 077 | 3F | ? | 95 | 01011111 | 137 | 5F | _ | 127 | 01111111 | 177 | 7F | DEL |

Lower case and upper case differ by 0x20 (32)

## Instructions for Loading/Storing Bytes

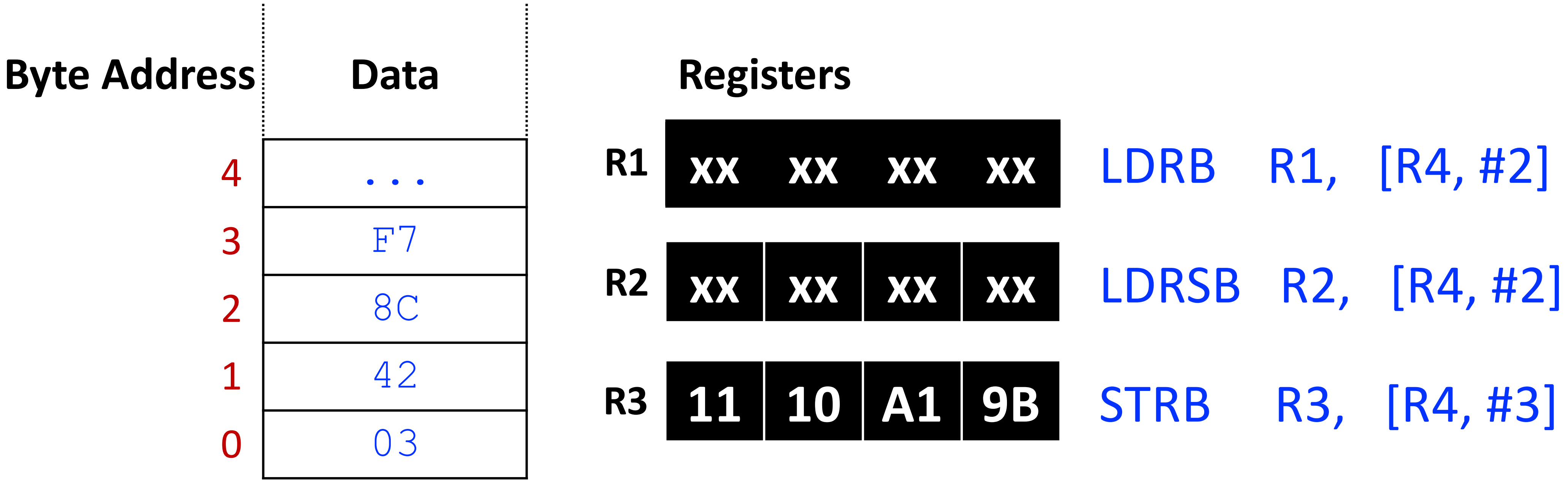
- `LDRB`
  - Load byte in register, and **zero-extend** to fill the 32 bits

- `LDRSB`
  - Load byte in register, and **sign-extend** to fill the 32 bits

- `STRB`
  - Store the **LSB** of the **32-bit integer** into the **specified** byte in memory
  - More significant bits of the register are ignored

# Loading/Storing Bytes

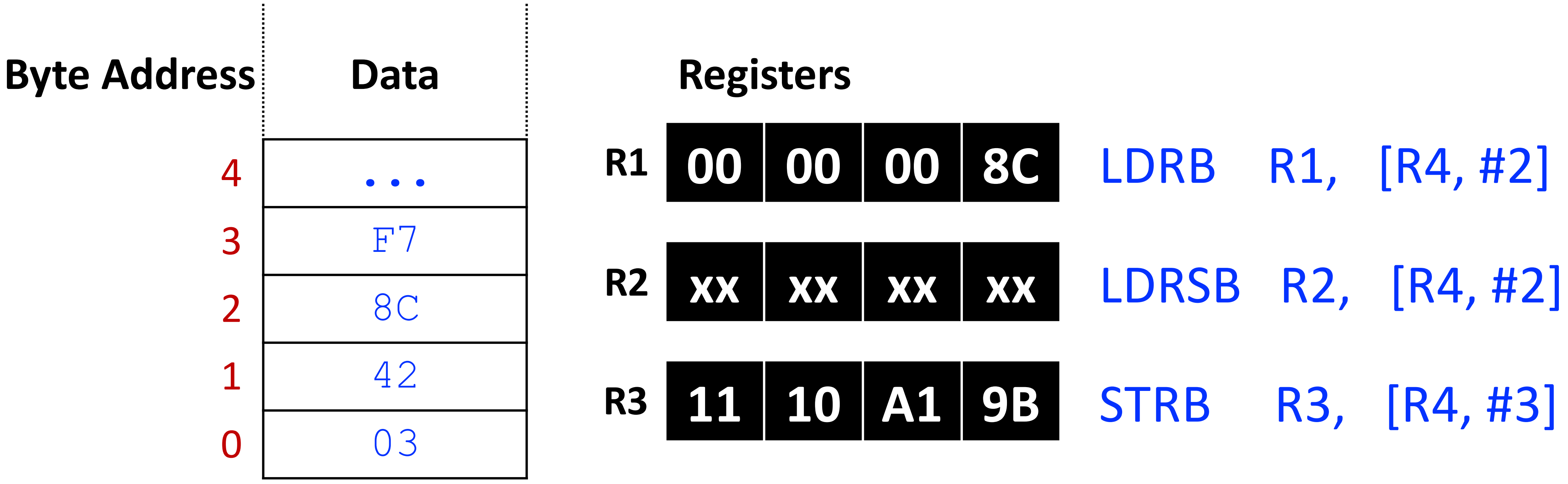
- What is in R1, R2, and memory after each of the instruction has executed? Assume R4 = 0



```
LDRB   R1, [R4, #2]
LDRSB  R2, [R4, #2]
STRB   R3, [R4, #3]
```

## Loading/Storing Bytes

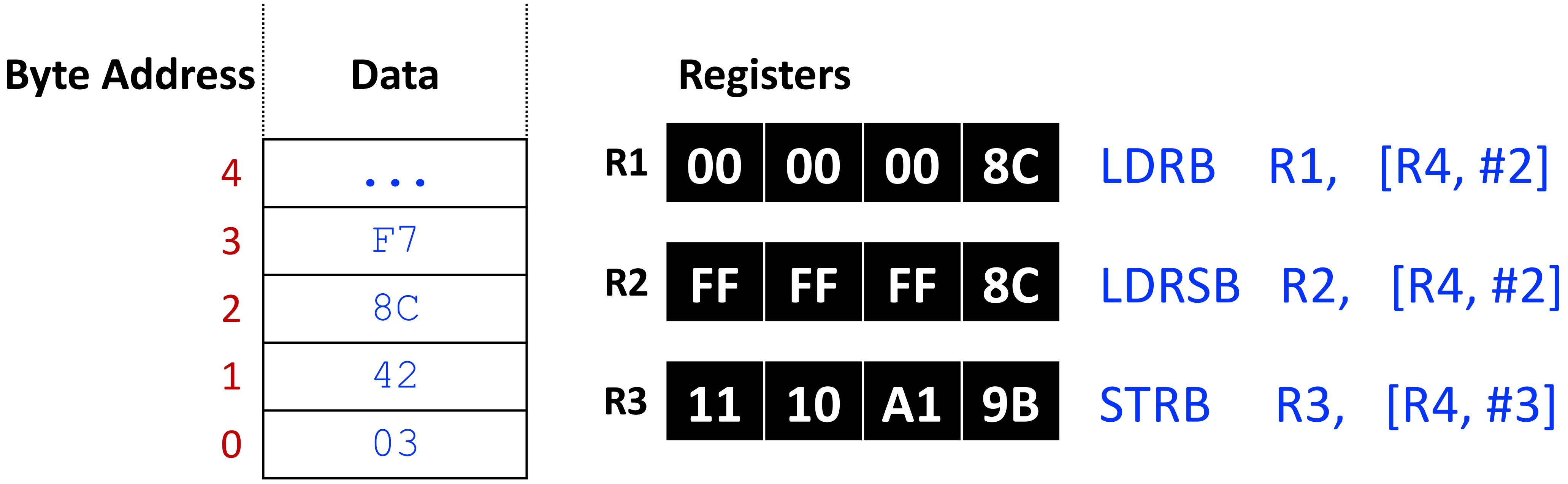
- What is in R1, R2, and memory after each of the instruction has executed? Assume R4 = 0



```
LDRB    R1,  [R4, #2]
LDRSB   R2,  [R4, #2]
STRB    R3,  [R4, #3]
```

# Loading/Storing Bytes

- What is in R1, R2, and memory after each of the instruction has executed? Assume R4 = 0



```
LDRB   R1, [R4, #2]
LDRSB  R2, [R4, #2]
STRB   R3, [R4, #3]
```

# Loading/Storing Bytes

- What is in R1, R2, and memory after each of the instruction has executed? Assume R4 = 0

| Byte Address | Data |
|---|---|
| 4 | ... |
| 3 | 9B |
| 2 | 8C |
| 1 | 42 |
| 0 | 03 |

**Registers**

| | | | | | |
|---|---|---|---|---|---|
| R1 | 00 | 00 | 00 | 8C | LDRB R1, [R4, #2] |
| R2 | FF | FF | FF | 8C | LDRSB R2, [R4, #2] |
| R3 | 11 | 10 | A1 | 9B | STRB R3, [R4, #3] |

## Strings in C

- A series of characters is a string
- Two ways to create strings in C
  - `char welcome[6] = {'H', 'E', 'L', 'L', 'O', '\0'};`
  - `char welcome[]  = "HELLO";`
- Different strings have different number of characters
  - We need to know the end of the string to write correct programs that manipulate strings
  - The null terminator `'\0'` marks the end of the string

# Strings in C

- `char welcome[6] = {'H', 'E', 'L', 'L', 'O', '\0'};`
- `char welcome[]  = "HELLO";`

- Compiler figures out the length
- 5 + 1 for '\0'
- Manually track length (unlike Python)

- Compiler inserts a **null terminator** '\0' automatically

- Need a way to know the end of the string
  - C strings are **null-terminated**

## Exercise: Manipulating Char Array

```
C code:

   char array[11] = “anthonymay”;
   int i;

   for (i = 0; i < 10; i = i + 1)
      array[i] = array[i] – 32;
```

# Exercise: Manipulating Char Array

- Transform the 10-character ASCII string, namely `array`, from lower case to upper case

```
C code:

   char array[11] = “anthonymay”;
   int i;

   for (i = 0; i < 10; i = i + 1)
      array[i] = array[i] – 32;
```

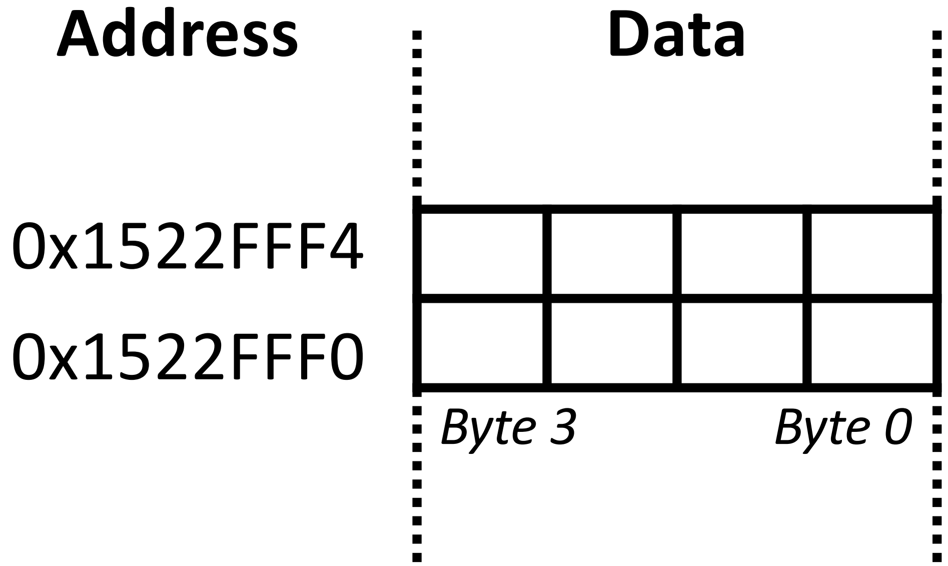
```
Assembly code:
; R0 = base addr, R1 = i
   MOV    R1,  #0                    ▪ i = 0
LOOP
   CMP    R1,  #10                   ▪ i < 10?
   BGE    DONE                       ▪ if i >= 10, exit
   LDRB   R2,  [R0, R1]              ▪ R2 = array[i]
   SUB    R2,  R2,  #32              ▪ subtract 32
   STRB   R2,  [R0, R1]              ▪ store array[i]
   ADD    R1,  R1,  #1               ▪ i = i + 1
   B      LOOP                       ▪ repeat loop
DONE
```

# Exercise: Strings in Memory

- Show how “HELLO!” is stored in memory below at address `0x1522FFF0`.

**ASCII Encoding**

| | |
|---|---|
| H | 0x48 |
| E | 0x65 |
| L | 0x6C |
| O | 0x6F |
| ! | 0x21 |
| Null | 0x00 |

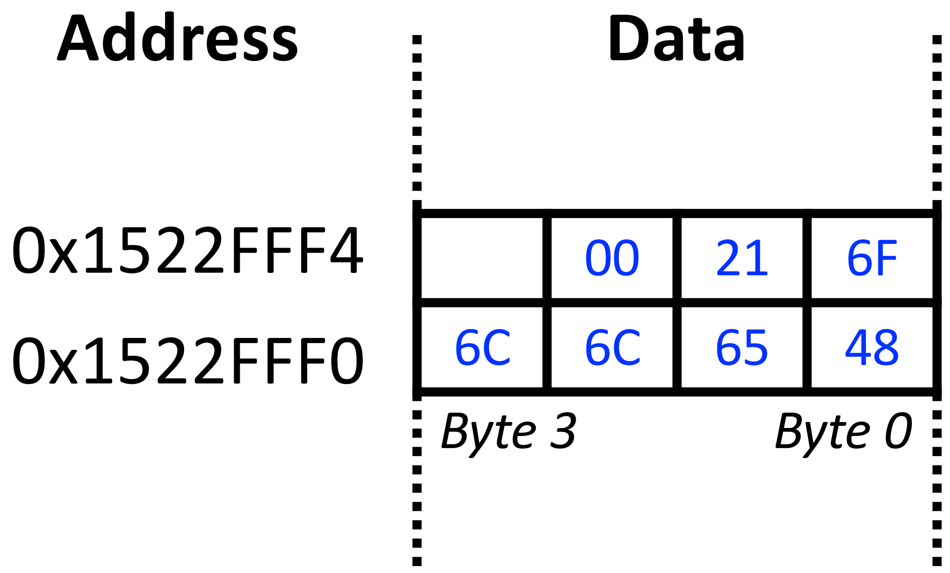


# Exercise: Strings in Memory

- Show how "HELLO!" is stored in memory below at address `0x1522FFF0`.

**ASCII Encoding**

| | |
|---|---|
| H | 0x48 |
| E | 0x65 |
| L | 0x6C |
| O | 0x6F |
| ! | 0x21 |
| Null | 0x00 |



## Some Assembly Practice

## More Assembly Practice

**C Code**

```
    int array[5];
    array[0] = array[0] * 8;
    array[1] = array[1] * 8;
```

**ARM Assembly Code**

```
; R0 = array base address
  MOV R0, #0x60000000        ; R0 = 0x60000000

  LDR R1, [R0]               ; R1 = array[0]
  LSL R1, R1, #3             ; R1 = R1 << 3 = R1*8
  STR R1, [R0]               ; array[0] = R1

  LDR R1, [R0, #4]           ; R1 = array[1]
  LSL R1, R1, #3             ; R1 = R1 << 3 = R1*8
  STR R1, [R0, #4]           ; array[1] = R1
```

## More Assembly Practice

**C Code**

```
int array[200];
int i;
for (i=199; i >= 0; i = i - 1)
   array[i] = array[i] * 8;
```

**ARM Assembly Code**

```
; R0 = array base address, R1 = i
  MOV R0, 0x60000000
  MOV R1, #199

FOR
  LDR  R2, [R0, R1, LSL #2]   ; R2 = array(i)
  LSL  R2, R2, #3             ; R2 = R2<<3 = R3*8
  STR  R2, [R0, R1, LSL #2]   ; array(i) = R2
  SUBS R1, R1, #1             ; i = i - 1
                              ; and set flags
  BPL  FOR                    ; if (i>=0) repeat
loop
```