

**BGU 361-1-4201**  
**Computer Architecture**  
**Lecture 2: ISA Tradeoffs**  
**Lecturer: Dr. Guy Tel-Zur**

Based on a course by Prof. Onur Mutlu

Carnegie Mellon University

Spring 2015

# Agenda for Today

## ■ Deep dive into ISA and its tradeoffs

### Appendix A

#### The LC-3b ISA

##### A.1 Overview

The Instruction Set Architecture (ISA) of the LC-3b is defined as follows:

- **Memory address space** 16 bits, corresponding to  $2^{16}$  locations, each containing one byte (eight bits). Addresses are numbered from 0 (i.e., x0000) to 65,535 (i.e., xFFFF). Addresses are used to identify memory locations and memory-mapped I/O device registers. Certain regions of memory are reserved for special uses, as described in Figure A.1.
- **Memory addressability** Each memory location contains one byte (eight bits) of information.
- **Bit numbering** Bits of all quantities are numbered, from right to left, starting with bit 0. The left-most bit of the contents of a memory location is bit 7.
- **Memory alignment** Two word-aligned memory locations are required to store one 16-bit word. Two memory locations are word-aligned if their addresses differ only in bit [0]. For example, locations x0006 and x0007 are word-aligned; locations x0007 and x0008 are not.
- **Memory addressing** The address of a byte of information stored in memory is the address of the location containing that byte. The address of a word of information stored in memory is the lower address of the pair of word-aligned memory locations containing the word. For example, a word stored in locations x0006 and x0007 has the address x0006. Since a word can be stored in memory only in two word-aligned locations, the address of a word of memory is always even.
- **Endian-ness** A word stored in memory at word address X has bits [7:0] stored in location X and bits [15:8] stored in location X+1. Since the less significant byte of the word is stored in location X and the more significant byte is stored in location X+1 (that is, the less significant byte "first"), the ordering is called little endian (for the little end first).

1

מחשב חינוכי LC-3b

# Last Lecture Recap

- Levels of Transformation
  - Algorithm, ISA, Microarchitecture
- **Moore's Law**

Read his 1965 paper:

[Cramming more components onto integrated circuits](#)

- What is Computer Architecture
- Why Study Computer Architecture
- Fundamental Concepts
- [Von Neumann Model](#)
- Dataflow Model – data driven
- ISA vs. Microarchitecture

**ארכיטקטורת וון נוימן:**

2 מאפיינים:

- (1) התכנית מאוחסנת בזכרון.
- (2) הרצה סידרתית. ברגע נתון על ערוץ הפנימי (Bus) מועברת או הפקודה או הנתון אבל לא שניהם בו זמנית.

בארכיטקטורה זו התכנית והנתונים מאוחסנים באותו הזיכרון וזאת בניגוד לארכיטקטורת הארווארד בה לתכנית ולנתונים זכרונות נפרדים.

# Review: ISA vs. Microarchitecture

## ■ ISA

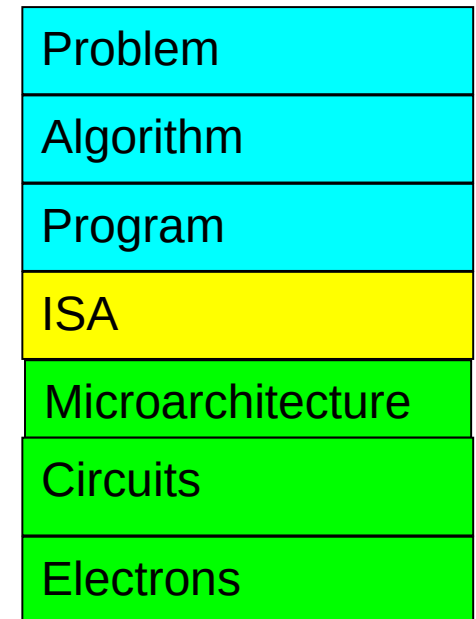
- Agreed upon interface between software and hardware
  - SW/compiler assumes, HW promises
- What the software writer needs to know to write and debug system/user programs

## ■ Microarchitecture

- Specific implementation of an ISA
- Not visible to the software

## ■ Microprocessor

- **ISA, uarch**, circuits
- “Architecture” = ISA + microarchitecture





# Review: ISA



## ■ Instructions

- Opcodes, Addressing Modes, Data Types
- Instruction Types and Formats
- Registers, Condition Codes

## ■ Memory

- Address space, Addressability, Alignment
- Virtual memory management

## ■ Call, Interrupt/Exception Handling

## ■ Access Control, Priority/Privilege

## ■ I/O: memory-mapped vs. instr.

## ■ Task/thread Management

## ■ Power and Thermal Management

## ■ Multi-threading support, Multiprocessor support

Intel® 64 and IA-32 Architectures  
Software Developer's Manual

Volume 1:  
Basic Architecture

ספר של 1000+ עמודים

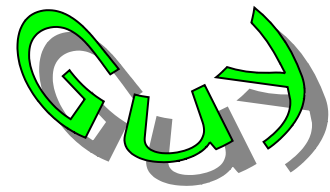
<https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>

# Microarchitecture

- Implementation of the ISA under specific **design constraints and goals**
- Anything done in hardware without exposure to software
  - Pipelining – נחשף לכך – המתכנת לא
  - **In-order** versus **out-of-order** instruction execution
  - Memory access **scheduling** policy
  - **Speculative** execution (make predictions and pre-load data to the cache in order to save memory access latency - גיא)
  - **Superscalar** processing (multiple instruction issue?) - ביצוע פעולות - מרובות במחזור שרון – גם לכך המתכנת לא נחשף
  - Clock gating
  - Caching? Levels, size, associativity, replacement policy
  - בדור"כ המתכנת לא נחשף לשיקולי cache
  - Prefetching? - ראו עוד 2 שקפים
  - Voltage/frequency scaling? - הורדת תדר - אפשרי שיהיה חשוף למשתמש
  - Error correction?

**אפשרי שה-ISA תהיה סדרתית (פון נוימן), אך מתחת,  
המיקרוארכיטקטורה תפעל באופן שונה, לדוגמה OoO, מיקבול...**

# Instruction prefetch



In computer architecture, instruction prefetch is a technique used in microprocessors to speed up the execution of a program by reducing wait states.

Modern microprocessors are much faster than the memory where the program is kept, meaning that the program's instructions cannot be read fast enough to keep the microprocessor busy. Adding a cache can provide faster access to needed instructions. Prefetching occurs when a processor requests an instruction from main memory before it is actually needed. Once the instruction comes back from memory, it is placed in a cache. When an instruction is actually needed, the instruction can be accessed much more quickly from the cache than if it had to make a request from memory. Since programs are generally executed sequentially, performance is likely to be best when instructions are prefetched in program order. Alternatively, the prefetch may be part of a complex branch prediction algorithm, where the processor tries to anticipate the result of a calculation and fetch the right instructions in advance.

Source: <https://www.definitions.net/definition/instruction+prefetch>

# Property of ISA vs. $\mu$ arch?

תרגיל: מה לדעתכם שייך ל-ISA ומה שייך למיקרוארכיטקטורה?

- א) **ADD** instruction's opcode
- ב) Number of general purpose registers
- ג) Number of ports to the register file
- ד) Number of cycles to execute the **MUL** instruction
- ה) Whether or not the machine employs pipelined instruction execution

תשובות: א) ISA ב) ISA וגם מיקרו ג) מיקרו ד) מיקרו ה) מיקרו

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

לכן יכולות להיות מקרוארכיטקטורות רבות עבור אותו ISA

# Design Point

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

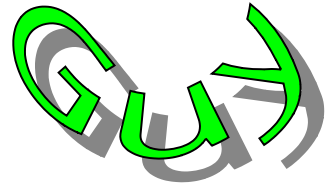
|                   |
|-------------------|
| Problem           |
| Algorithm         |
| Program           |
| ISA               |
| Microarchitecture |
| Circuits          |
| Electrons         |

ש: מה קובע את נקודת התכן? ת: הייעוד/המטרה לשמה נבנה המעבד

# Application Space

Reading assignment: קריאת חובה

"Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution" by Yale Patt



- Dream, and they will appear...

Other examples of the application space that continue to drive the need for unique design points are the following:

- 1) scientific applications such as those whose computations control nuclear power plants, determine where to drill for oil, and predict the weather;
- 2) transaction-based applications such as those that handle ATM transfers and e-commerce business;
- 3) business data processing applications, such as those that handle inventory control, payrolls, IRS activity, and various personnel record keeping, whether the personnel are employees, students, or voters;
- 4) network applications, such as high-speed routing of Internet packets, that enable the connection of your home system to take advantage of the Internet;
- 5) guaranteed delivery (a.k.a. real time) applications that require the result of a computation by a certain critical deadline;
- 6) embedded applications, where the processor is a component of a larger system that is used to solve the (usually) dedicated application;
- 7) media applications such as those that decode video and audio files;
- 8) random software packages that desktop users would like to run on their PCs.

Each of these application areas has a very different set of characteristics. Each application area demands a different set of tradeoffs to be made in specifying the microprocessor to do the job.

חישוב מדעי

כספומט

עיבוד נתונים – מרשם  
האוכלוסין

מעבדי תעבורת רשת

מערכת זמן אמת

מערכות משובצות  
מחשב

עיבוד מדיה (וידאו)  
מחשב אישי לשימוש כללי

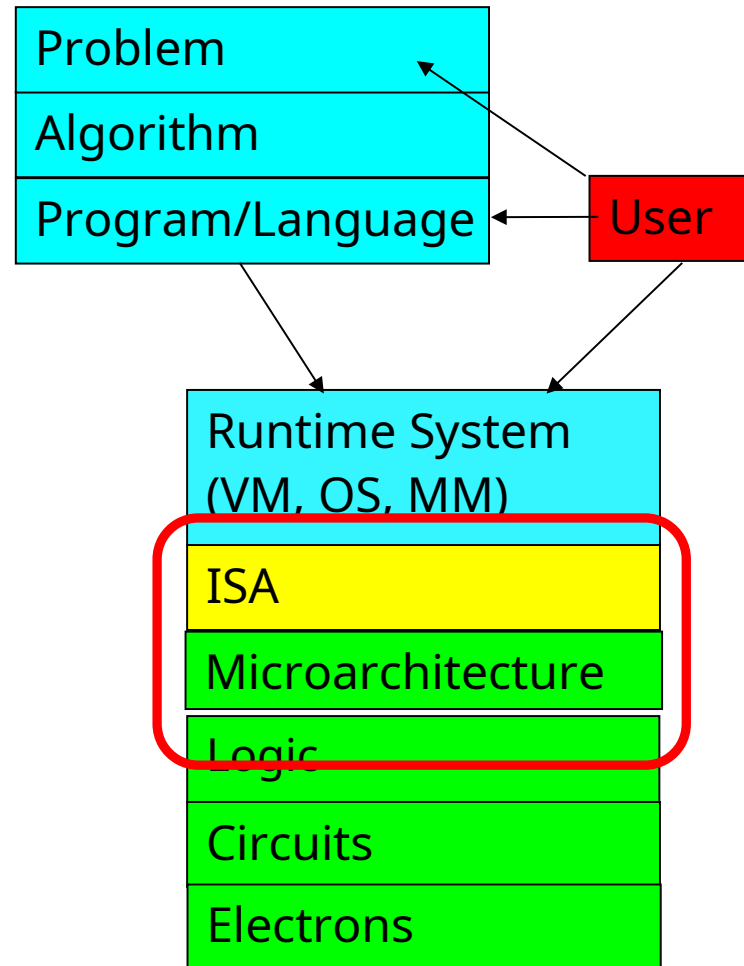
# Tradeoffs: Soul of Computer Architecture

- ISA-level tradeoffs
- Microarchitecture-level tradeoffs
- System and Task-level tradeoffs
  - How to divide the labor between hardware and software
- *Computer architecture is the science and art of making the appropriate trade-offs to meet a design point*
  - **ארכיטקטורת מחשב היא המדע והאמנות של עשיית פשרות נכונות כדי לקיים את נקודת התכן**
    - *Why art? - ראה השקף הבא*



# Why Is It (Somewhat) Art?

New demands  
from the top  
(Look Up)



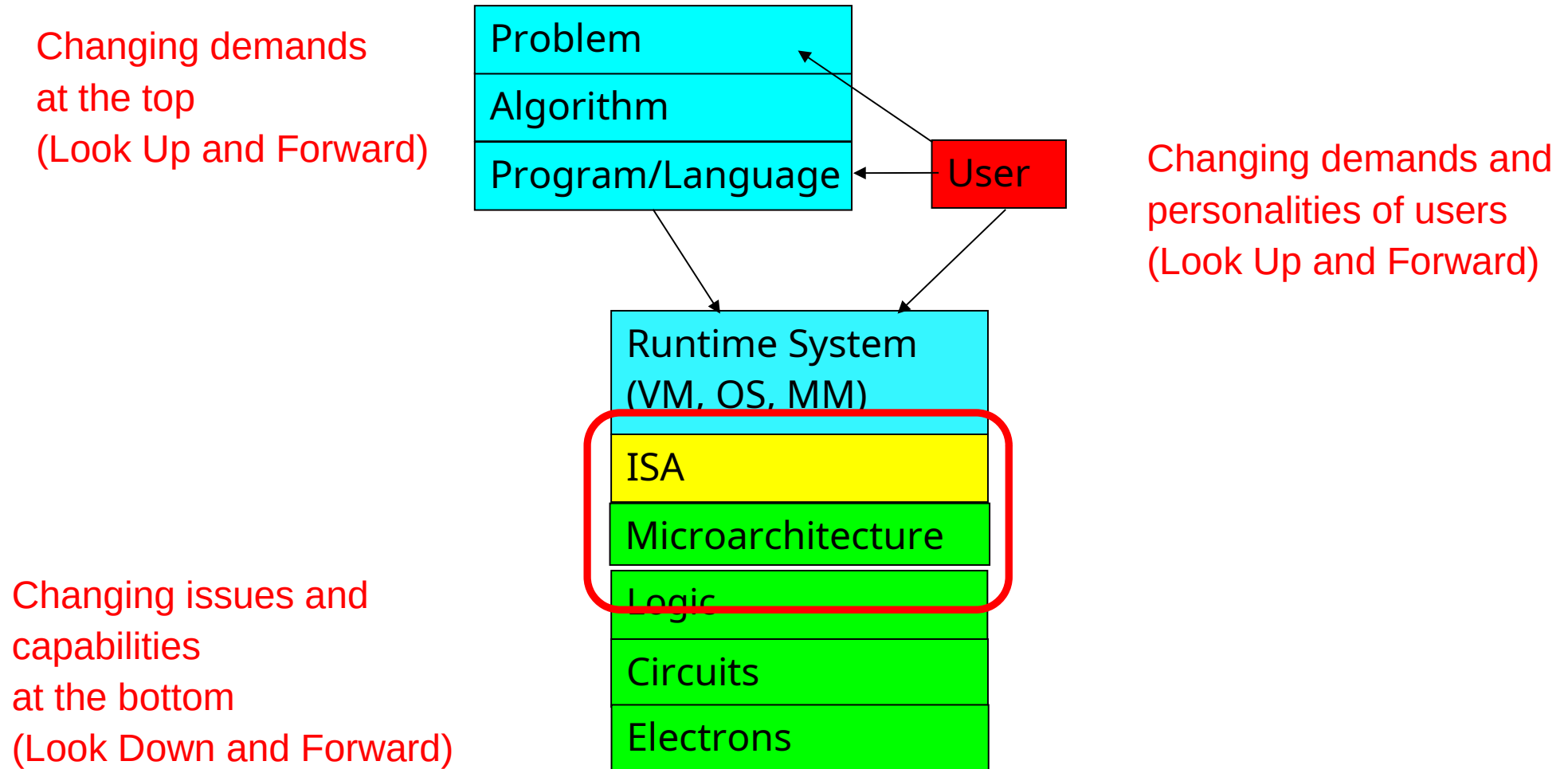
New demands and  
personalities of users  
(Look Up)

New issues and  
capabilities  
at the bottom  
(Look Down)

הכל צפוי להשתנות...

- We do not (fully) know the future (applications, users, market)  
“It’s Difficult to Make Predictions, Especially About the Future”

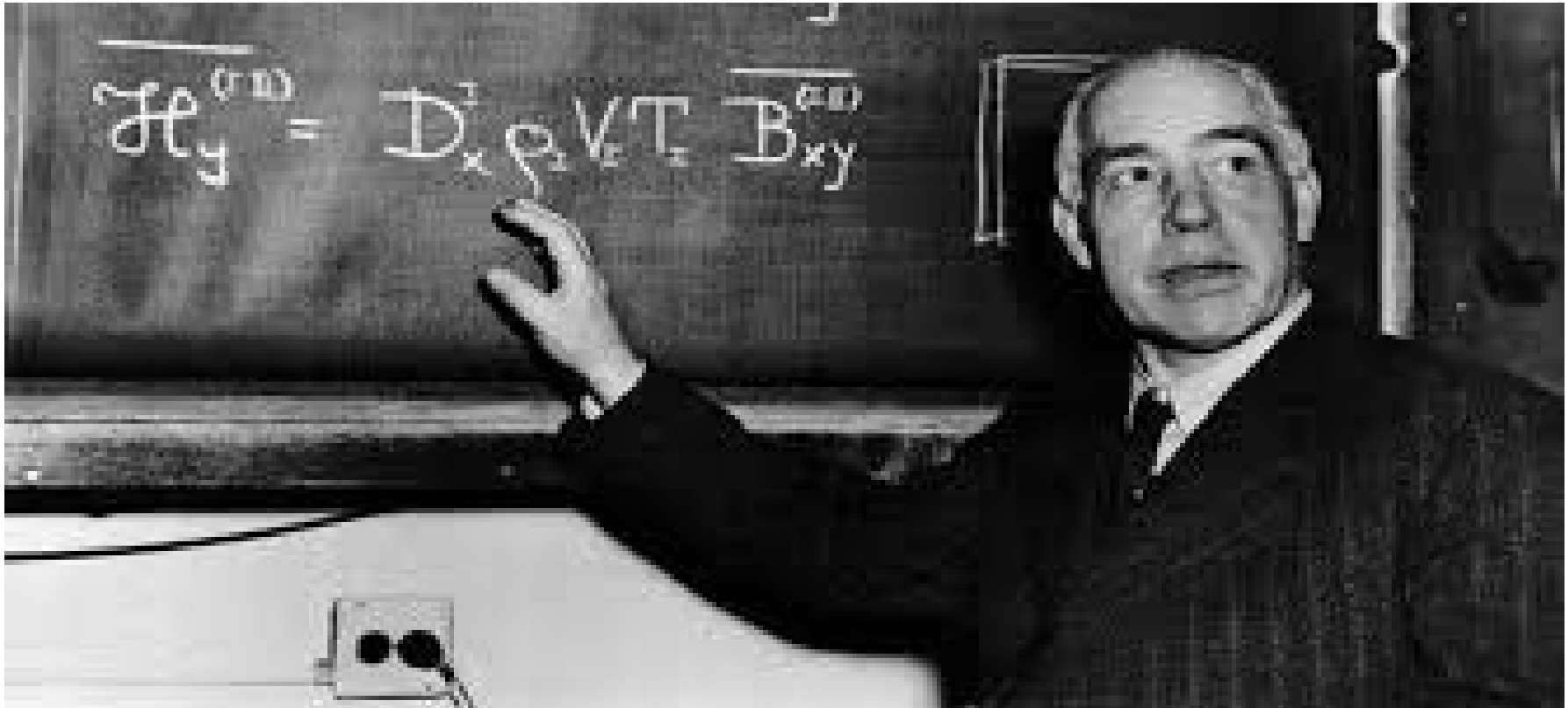
# Why Is It (Somewhat) Art?



- And, the future is not constant (it changes)!

14 לדוגמה חישבו על בניית גשר לעומת בניית מעבד – בראיה של צרכים עתידיים (+10 שנים...)

**“it is difficult to predict, especially the future” - [Niels Bohr](#)**



Niels Bohr (1885-1962)

Was a Danish physicist who made foundational contributions to understanding atomic structure and quantum theory, for which he received the Nobel Prize in Physics in 1922. Bohr was also a philosopher and a promoter of scientific research.

# How Can We Adapt to the Future

- This is part of the task of a good computer architect
- Many options (bag of tricks)
  - Keen (נלהב) insight and good design
  - Good use of fundamentals and principles
    - Efficient design
    - Heterogeneity
    - Reconfigurability
    - ...
  - Good use of the underlying technology
  - ...

# ISA Principles and Tradeoffs

# General structure of instructions

# Many Different ISAs Over Decades

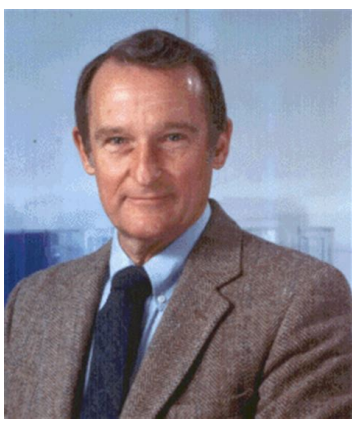
- x86
- PDP-x: Programmed Data Processor (PDP-11)
- VAX
- IBM 360
- CDC 6600
- SIMD ISAs: CRAY-1, Connection Machine
- VLIW ISAs: Multiflow, Cydrome, IA-64 (EP
- PowerPC, POWER

POWER = "*Performance Optimization With Enhanced RISC*"

- RISC ISAs: Alpha, **MIPS**, SPARC, ARM, **RISC-V**
- ⇒What are the fundamental differences?
  - E.g., how instructions are specified and what they do
  - E.g., how complex are the instructions

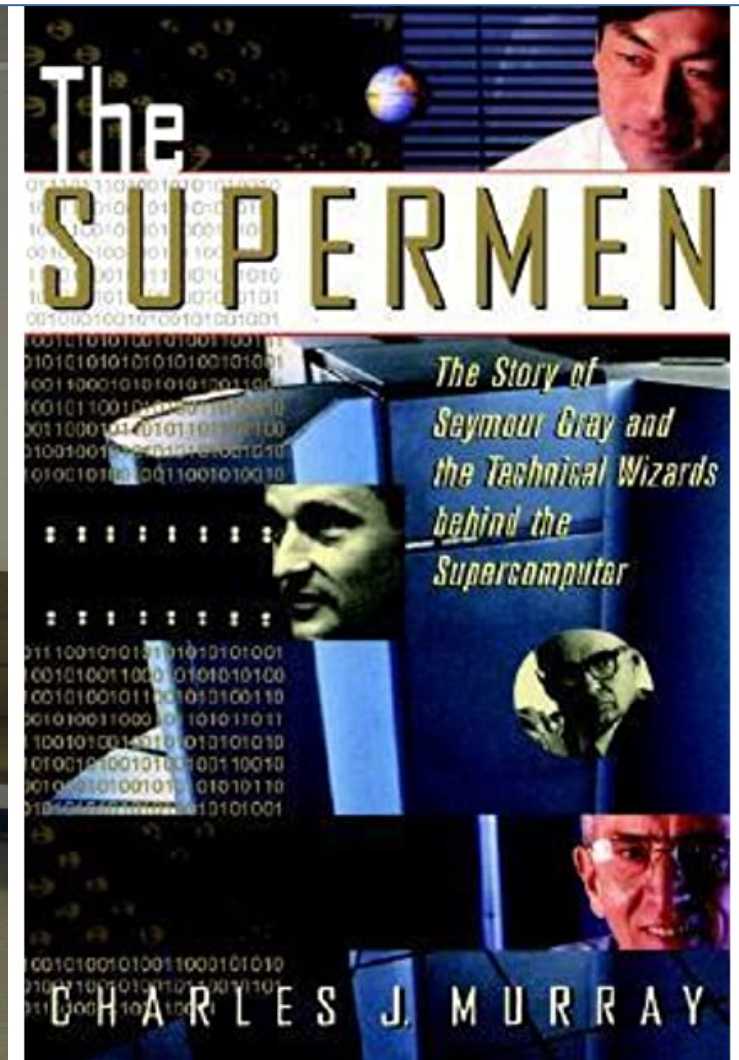
Me by a Cray-1, at the Supercomputing conference, SC18. Dallas, Tx, 2018





Seymour Cray (1925 – 1996) was an American electrical engineer and supercomputer architect who designed a series of computers that were the fastest in the world for decades, and founded Cray Research which built many of these machines. Called "*the father of supercomputing*".

- Wikipedia



צילום: 15/4/2008 במרכז החישובים CESCA בברצלונה



# Instruction

- Basic element of the HW/SW interface
- Consists of
  - **opcode**: what the instruction does
  - **operands**: who it is to do it to
- Example from the Alpha ISA ז"ל:

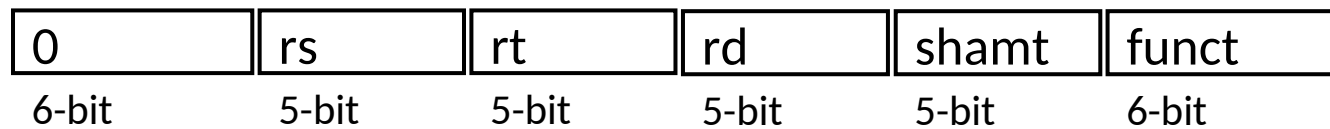


לא ניתן לתרגל אודות אלפא מאחר ולא קיימים יותר קומפיילרים...

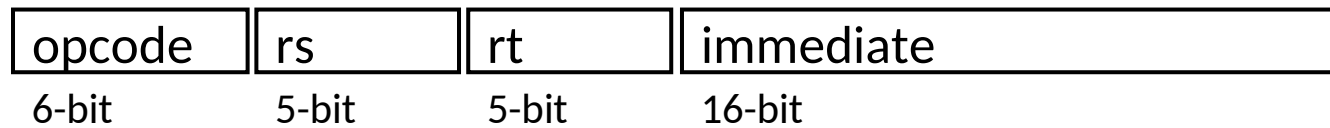
|        |        |       |          |     |   |                |
|--------|--------|-------|----------|-----|---|----------------|
| 31     | 26 25  | 21 20 | 16 15    | 5 4 | 0 |                |
| Opcode | Number |       |          |     |   | PALcode Format |
| Opcode | RA     | Disp  |          |     |   | Branch Format  |
| Opcode | RA     | RB    | Disp     |     |   | Memory Format  |
| Opcode | RA     | RB    | Function | RC  |   | Operate Format |

מבנה סדור: מקומות קבועים. אורך קבוע – 32 ביט.

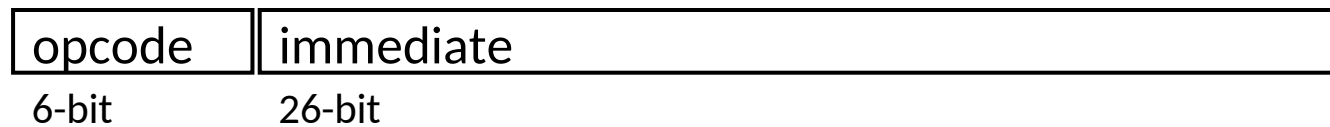
# MIPS



R-type



I-type

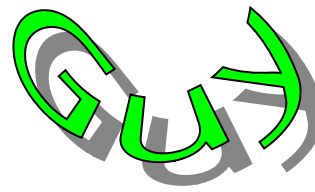


J-type



# ידיעה חדשותית

## 8/3/2021



## Electronic Engineering JOURNAL

/https://www.eejournal.com/article/wait-what-mips-becomes-risc-v

feature article



March 8, 2021

### Wait, What? MIPS Becomes RISC-V Classic CPU Company Exits Bankruptcy, Throws in the Towel

by Jim Turley

What a long, strange trip it's been. MIPS Technologies no longer designs MIPS processors. Instead, it's joined the RISC-V camp, abandoning its eponymous architecture for one that has strong historical and technical ties. The move apparently heralds the end of the road for MIPS as a CPU family, and a further (slight) diminution in the variety of processors available. It's the

final arc of an architecture.

MIPS as a company has passed through a lot of hands, most recently as part of [Wave Computing](#), the ill-fated AI startup. Wave was developing its unique AI acceleration hardware on top of a general-purpose MIPS CPU, and then it [bought the entire MIPS organization](#). The move seemed both unusual and unnecessary, and something of a distraction, given that they'd already licensed the necessary IP. Over time, it started to look like the MIPS side of the business was the most stable part of the company. Wave finally collapsed into bankruptcy last year, emerging just last week under a new name: that of its acquired technology, MIPS. Say hello to the new MIPS.

Development of the MIPS processor architecture has now stopped, and MIPS (the company) will start making chips based on RISC-V. This is a complete change of business model, not just CPU. The old MIPS was in the business of licensing IP, just like [ARM](#) or [Ceva](#) or [Rambus](#). It didn't make anything tangible. Companies like the old Wave Computing were its customers, and processors like ARM and RISC-V were its competitors. Now that equation is inverted.

כדי ללמוד עוד אודות המימוש של ארכיטקטורת MIPS במיקרו-בקר PIC32MZ ראו את הקובץ שבקישור:

<https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/ReferenceManuals/60001192B.pdf>

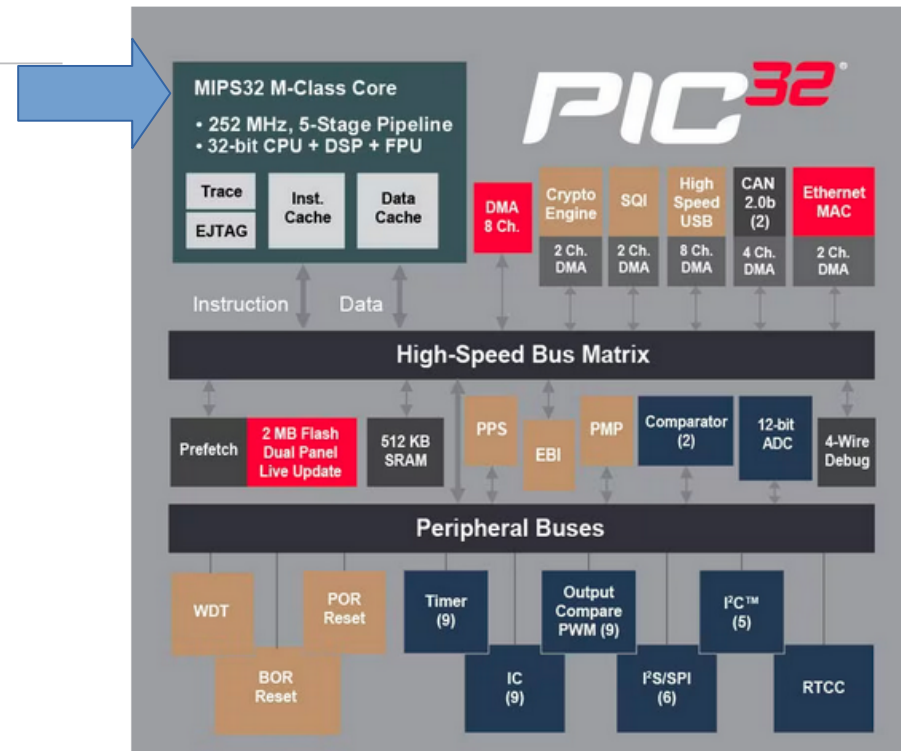
# PIC32MZ EF Family of Microcontrollers

## Highest-Performance PIC32 Microcontrollers

The PIC32MZ EF series of high-performance microcontrollers (MCUs) with industry-leading connectivity and peripheral options has been created to empower you to build complex applications quickly. These devices are fully supported by the MPLAB® Harmony Integrated software framework, which features software solutions to accelerate the development of graphics and connectivity applications.

## Key Features

- 252 MHz/415 DMIPS performance
- Up to 2 MB dual-panel live update Flash and 512 KB RAM
- Excellent connectivity options (Hi-Speed USB, CAN, and 10/100 Ethernet)
- Integrated double-precision FPU accelerating performance in process-intensive applications
- Optional full-featured hardware crypto accelerator
- Rich peripheral set
- Extended temperature and AECQ-100 Grade 1 qualification



# RISC-V

|            |    |           |    |     |     |         |     |            |        |        |          |          |        |         |        |         |  |         |
|------------|----|-----------|----|-----|-----|---------|-----|------------|--------|--------|----------|----------|--------|---------|--------|---------|--|---------|
| 31         | 30 | 25        | 24 | 21  | 20  | 19      | 15  | 14         | 12     | 11     | 8        | 7        | 6      | 0       |        |         |  |         |
| funct7     |    |           |    | rs2 |     |         | rs1 |            | funct3 |        | rd       |          |        | opcode  |        | R-type  |  |         |
| imm[11:0]  |    |           |    |     |     | rs1     |     | funct3     |        | rd     |          |          | opcode |         | I-type |         |  |         |
| imm[11:5]  |    |           |    | rs2 |     |         | rs1 |            | funct3 |        | imm[4:0] |          |        | opcode  |        | S-type  |  |         |
| imm[12]    |    | imm[10:5] |    |     | rs2 |         |     | rs1        |        | funct3 |          | imm[4:1] |        | imm[11] |        | opcode  |  | SB-type |
| imm[31:12] |    |           |    |     |     |         |     |            |        | rd     |          |          | opcode |         | U-type |         |  |         |
| imm[20]    |    | imm[10:1] |    |     |     | imm[11] |     | imm[19:12] |        |        | rd       |          |        | opcode  |        | UJ-type |  |         |

| Format | Purpose           | Example        |
|--------|-------------------|----------------|
| -----  | -----             | -----          |
| R      | Register-Register | ADD, MUL, AND  |
| I      | Immediate/Load    | ADDI, LW, JALR |
| S      | Store             | SW, SB, SH     |
| SB     | Branch            | BEQ, BNE, BLT  |
| U      | Upper Immediate   | LUI, AUIPC     |
| UJ     | Jump              | JAL            |

דומה ל-RISC  
אך קצת יותר  
מורכב

3 3 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0  
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

|      |   |   |   |        |                      |   |   |   |      |     |   |   |               |    |   |   |           |     |   |   |        |    |   |    |                                       |               |                           |                            |                                                 |                                   |                                      |
|------|---|---|---|--------|----------------------|---|---|---|------|-----|---|---|---------------|----|---|---|-----------|-----|---|---|--------|----|---|----|---------------------------------------|---------------|---------------------------|----------------------------|-------------------------------------------------|-----------------------------------|--------------------------------------|
| Cond | 0 | 0 | 1 | Opcode |                      |   |   | S | Rn   |     |   |   | Rd            |    |   |   | Operand 2 |     |   |   |        |    |   |    | <i>Data Processing / PSR Transfer</i> |               |                           |                            |                                                 |                                   |                                      |
| Cond | 0 | 0 | 0 | 0      | 0                    | 0 | A | S | Rd   |     |   |   | Rn            |    |   |   | Rs        |     |   |   | 1      | 0  | 0 | 1  | Rm                                    |               |                           |                            | <i>Multiply</i>                                 |                                   |                                      |
| Cond | 0 | 0 | 0 | 0      | 1                    | U | A | S | RdHi |     |   |   | RdLo          |    |   |   | Rn        |     |   |   | 1      | 0  | 0 | 1  | Rm                                    |               |                           |                            | <i>Multiply Long</i>                            |                                   |                                      |
| Cond | 0 | 0 | 0 | 1      | 0                    | B | 0 | 0 | Rn   |     |   |   | Rd            |    |   |   | 0         | 0   | 0 | 0 | 1      | 0  | 0 | 1  | Rm                                    |               |                           |                            | <i>Single Data Swap</i>                         |                                   |                                      |
| Cond | 0 | 0 | 0 | 1      | 0                    | 0 | 1 | 0 | 1    | 1   | 1 | 1 | 1             | 1  | 1 | 1 | 1         | 1   | 1 | 0 | 0      | 0  | 1 | Rn |                                       |               |                           | <i>Branch and Exchange</i> |                                                 |                                   |                                      |
| Cond | 0 | 0 | 0 | P      | U                    | 0 | W | L | Rn   |     |   |   | Rd            |    |   |   | 0         | 0   | 0 | 0 | 1      | S  | H | 1  | Rm                                    |               |                           |                            | <i>Halfword Data Transfer: register offset</i>  |                                   |                                      |
| Cond | 0 | 0 | 0 | P      | U                    | 1 | W | L | Rn   |     |   |   | Rd            |    |   |   | Offset    |     |   |   | 1      | S  | H | 1  | Offset                                |               |                           |                            | <i>Halfword Data Transfer: immediate offset</i> |                                   |                                      |
| Cond | 0 | 1 | 1 | P      | U                    | B | W | L | Rn   |     |   |   | Rd            |    |   |   | Offset    |     |   |   |        |    |   |    | <i>Single Data Transfer</i>           |               |                           |                            |                                                 |                                   |                                      |
| Cond | 0 | 1 | 1 |        |                      |   |   |   |      |     |   |   |               |    |   |   |           |     |   |   |        | 1  |   |    |                                       |               | <i>Undefined</i>          |                            |                                                 |                                   |                                      |
| Cond | 1 | 0 | 0 | P      | U                    | S | W | L | Rn   |     |   |   | Register List |    |   |   |           |     |   |   |        |    |   |    | <i>Block Data Transfer</i>            |               |                           |                            |                                                 |                                   |                                      |
| Cond | 1 | 0 | 1 | L      | Offset               |   |   |   |      |     |   |   |               |    |   |   |           |     |   |   |        |    |   |    |                                       | <i>Branch</i> |                           |                            |                                                 |                                   |                                      |
| Cond | 1 | 1 | 0 | P      | U                    | N | W | L | Rn   |     |   |   | CRd           |    |   |   | CP#       |     |   |   | Offset |    |   |    |                                       |               |                           |                            | <i>Coprocessor Data Transfer</i>                |                                   |                                      |
| Cond | 1 | 1 | 1 | 0      | CP Opc               |   |   |   | CRn  |     |   |   | CRd           |    |   |   | CP#       |     |   |   | CP     |    |   |    | 0                                     | CRm           |                           |                            |                                                 | <i>Coprocessor Data Operation</i> |                                      |
| Cond | 1 | 1 | 1 | 0      | CP Opc               |   |   |   | L    | CRn |   |   |               | Rd |   |   |           | CP# |   |   |        | CP |   |    |                                       | 1             | CRm                       |                            |                                                 |                                   | <i>Coprocessor Register Transfer</i> |
| Cond | 1 | 1 | 1 | 1      | Ignored by processor |   |   |   |      |     |   |   |               |    |   |   |           |     |   |   |        |    |   |    |                                       |               | <i>Software Interrupt</i> |                            |                                                 |                                   |                                      |

3 3 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0  
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

Figure 4-1: ARM instruction set formats



# Set of Instructions, Encoding, and Spec

|                  | 15   | 14 | 13 | 12 | 11           | 10                | 9 | 8         | 7 | 6 | 5        | 4       | 3       | 2 | 1 | 0 |
|------------------|------|----|----|----|--------------|-------------------|---|-----------|---|---|----------|---------|---------|---|---|---|
| ADD <sup>+</sup> | 0001 |    |    |    | DR           |                   |   | SR1       |   |   | A        | op.spec |         |   |   |   |
| AND <sup>+</sup> | 0101 |    |    |    | DR           |                   |   | SR1       |   |   | A        | op.spec |         |   |   |   |
| BR               | 0000 |    |    |    | n            | z                 | p | PCOffset9 |   |   |          |         |         |   |   |   |
| JMP              | 1100 |    |    |    | 000          |                   |   | BaseR     |   |   | 000000   |         |         |   |   |   |
| JSR(R)           | 0100 |    |    |    | A            | operand.specifier |   |           |   |   |          |         |         |   |   |   |
| LDB <sup>+</sup> | 0010 |    |    |    | DR           |                   |   | BaseR     |   |   | boffset6 |         |         |   |   |   |
| LDW <sup>+</sup> | 0110 |    |    |    | DR           |                   |   | BaseR     |   |   | offset6  |         |         |   |   |   |
| LEA <sup>+</sup> | 1110 |    |    |    | DR           |                   |   | PCOffset9 |   |   |          |         |         |   |   |   |
| RTI              | 1000 |    |    |    | 000000000000 |                   |   |           |   |   |          |         |         |   |   |   |
| SHF <sup>+</sup> | 1101 |    |    |    | DR           |                   |   | SR        |   |   | A        | D       | amount4 |   |   |   |
| STB              | 0011 |    |    |    | SR           |                   |   | BaseR     |   |   | boffset6 |         |         |   |   |   |
| STW              | 0111 |    |    |    | SR           |                   |   | BaseR     |   |   | offset6  |         |         |   |   |   |
| TRAP             | 1111 |    |    |    | 0000         |                   |   | trapvect8 |   |   |          |         |         |   |   |   |
| XOR <sup>+</sup> | 1001 |    |    |    | DR           |                   |   | SR1       |   |   | A        | op.spec |         |   |   |   |
| not used         | 1010 |    |    |    |              |                   |   |           |   |   |          |         |         |   |   |   |
| not used         | 1011 |    |    |    |              |                   |   |           |   |   |          |         |         |   |   |   |

- Example from LC-3b ISA

[http://users.ece.utexas.edu/~patt/09s.360N/handouts/new\\_byte.pdf](http://users.ece.utexas.edu/~patt/09s.360N/handouts/new_byte.pdf)

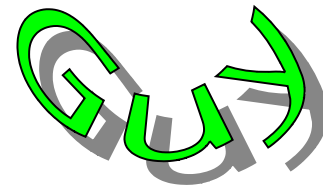
- Why unused instructions?
- Aside: concept of “**bit steering**”
  - A bit in the instruction determines the interpretation of other bits

רק 25 עמודים לעומת 1200 עמודים עבור x86

Bit steering concept – next slide

# ADD

## Addition



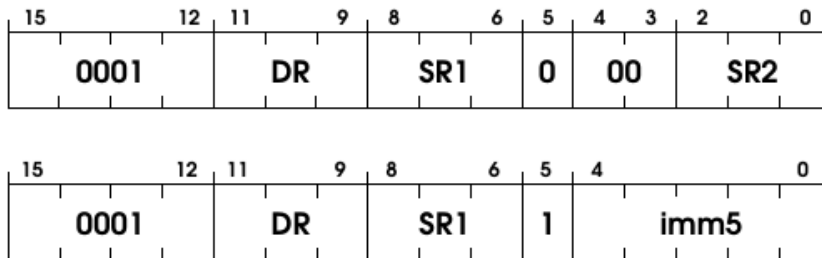
Reference: LC-3b ISA

[http://users.ece.utexas.edu/~patt/09s.360N/handouts/new\\_byte.pdf](http://users.ece.utexas.edu/~patt/09s.360N/handouts/new_byte.pdf)

## Assembler Formats

```
ADD DR, SR1, SR2
ADD DR, SR1, imm5
```

## Encodings



## Operation

```
if (bit[5] == 0)
    DR = SR1 + SR2;
else
    DR = SR1 + SEXT(imm5);
setcc();
```

*DR* = Destination register.  
*Imm5* = A 5-bit immediate value  
*SR* = Source Register

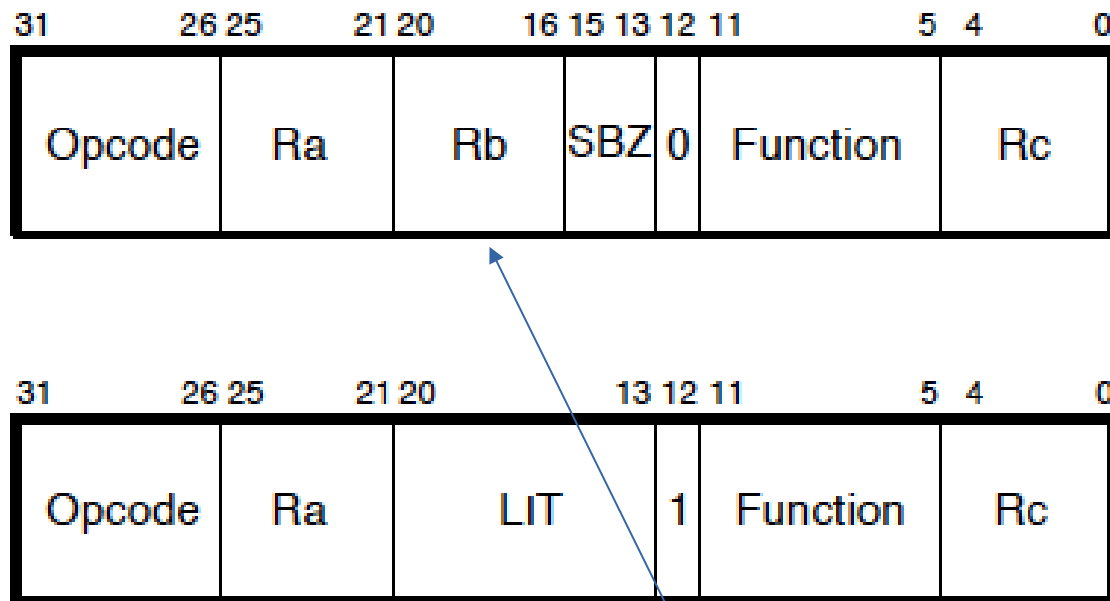
## Description

If bit [5] is 0, the second source operand is obtained from SR2. If bit [5] is 1, the second source operand is obtained by sign-extending the imm5 field to 16 bits. In both cases, the second source operand is added to the contents of SR1, and the result stored in DR. The condition codes are set, based on whether the result is negative, zero, or positive.



# Bit Steering in Alpha

Figure 3-4: Operate Instruction Format



If bit <12> of the instruction is 0, the Rb field specifies a source register operand.

If bit <12> of the instruction is 1, an 8-bit zero-extended literal constant is formed by bits <20:13> of the instruction. The literal is interpreted as a positive integer between 0 and 255 and is zero-extended to 64 bits.

# The Elements of ISA

# What Are the Elements of an ISA?

## ■ על כך כבר דיברנו – Instruction sequencing model

- **Control flow** vs. **data flow**
- Tradeoffs?

## Instruction processing style

- Specifies the number of “operands” an instruction “operates” on and how it does so

- **0, 1, 2, 3 address machines**

- 0-address: **stack machine** (op, push A, pop A)
- 1-address: **accumulator machine** (op ACC, ld A, st A)
- 2-address: 2-operand machine (op S,D; one is both source and dest).  
S=Source, D=Destination
- 3-address: 3-operand machine (op S1,S2,D; source and dest separate)

- Tradeoffs?

- Larger operate instructions vs. more executed operations
- Code size vs. execution time vs. on-chip memory space

מורכבות המעבד VS. מורכבות הגישה לזיכרון

פקודה implicit.

פועלת על 2 המקומות העליונים ב-stack ומחזירה

את התוצאה למקום העליון במחסנית

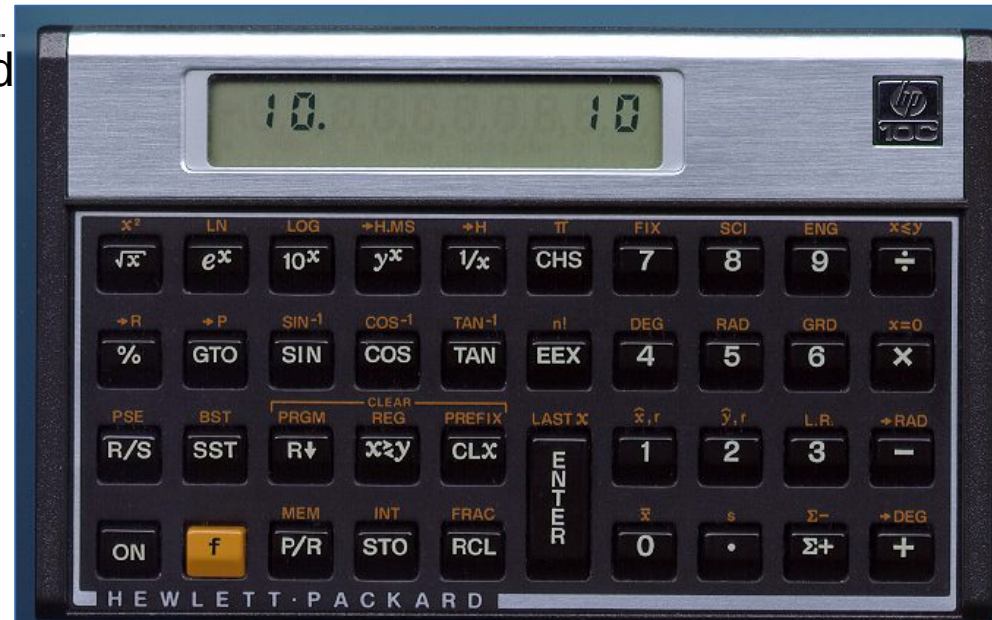
# Compact Code and Stack Architectures



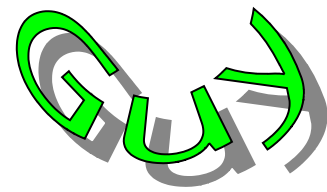
In the 1960s, a few companies followed a radical approach to instruction sets. In the belief that it was too hard for compilers to utilize registers effectively, these companies abandoned registers altogether! Instruction sets were based on a stack model of execution, like that found in the older Hewlett-Packard handheld calculators. Operands are pushed on the stack from memory or popped off the stack into memory. Operations take their operands from the stack and then place the result back onto the stack. In addition to simplifying compilers by eliminating register allocation, stack architectures lent themselves to compact instruction encoding, thereby removing memory size as an excuse not to program in high-level languages.

Reference: H-P Computer Organization, 4<sup>th</sup> Ed  
2.20-3 Historical Perspective and Further Reading

אתם מוזמנים לקרוא בוויקיפדיה אודות  
(RPN) [Reverse Polish Notation](#)



# Algebraic vs. RPN methods



## Example:

$$(3+5) \div (7+6) = x$$

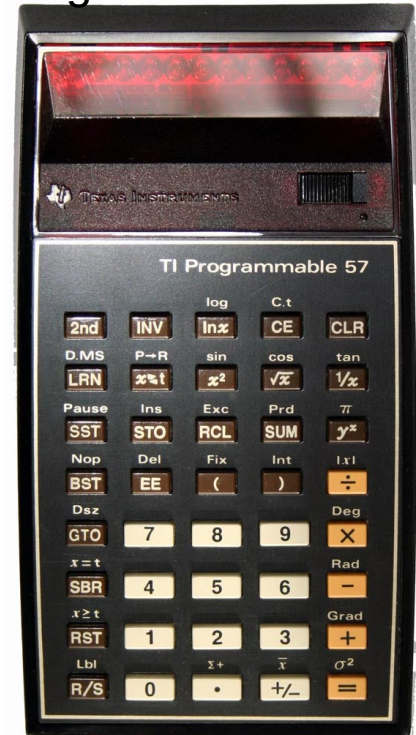
- Algebraic method: Add:  $3 + 5 = 8$  . Write the answer down or store it in the memory. Add:  $7 + 6 = 13$  . Now enter the 8 from the first outcome and share these results through the second answer for the result  $x = 0.62$  .
- RPN method : Type 3 and then press ENTER. Type 5 and then press the plus key (+). Type 7 and then press ENTER. Type 6 and then press the plus key (+). Note that the outcome is displayed , the second sum . Now comes the surprise : press the button for sharing and the calculator returns the result ( 0.62 ) again .

Algebraic : 13 steps , the effort to write results down or remember the first result when you run the second calculation , not even being counted.

RPN: 9 steps and you need to write down anything.

Sharing = divide (÷)

TI57 Algebraic calculator



השווה מול הדגם של HP בשקף הקודם

# השוואה בין 2 שיטות החישוב

## Algebraic

1. (

2. 3

3. +

4. 5

5. )

6. /

7. (

8. 7

9. +

10. 6

11. )

12. =

13. אופציונלי. לאחר שלב 5 רישום

התוצאה וניקוי הצג

## RPN

1. 3

2. ENTER

3. 5

4. +

5. 7

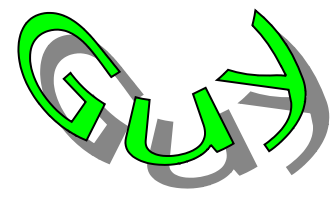
6. ENTER

7. 6

8. +

9. /

# Stack Machine - Example



$$R = ((5 + 7) * 8) * 9$$

LIFO

NER

Stacked  
Architecture

push 9

-||- 8

-||- 5

-||- 7



Stack  
pointer

ADD

MUL

MUL

יפעל על 2 הערכים העליונים במחסנית

L2. March 2020

# An Example: Stack Machine

+ Small instruction size (no operands needed for operate instructions) – אין אופרנדים בהוראה אלא רק האופ-קוד בלבד

- Simpler logic
- Compact code

+ Efficient procedure calls: all parameters on stack

- No additional cycles for parameter passing

-- Computations that are not easily expressible with “**postfix notation**” are difficult to map to stack machines

- Cannot perform operations on many values at the same time (only top N values on the stack at the same time)
- Not flexible

יעילות מצד אחד וחוסר גמישות מהצד השני





# An Example: Stack Machine (II)

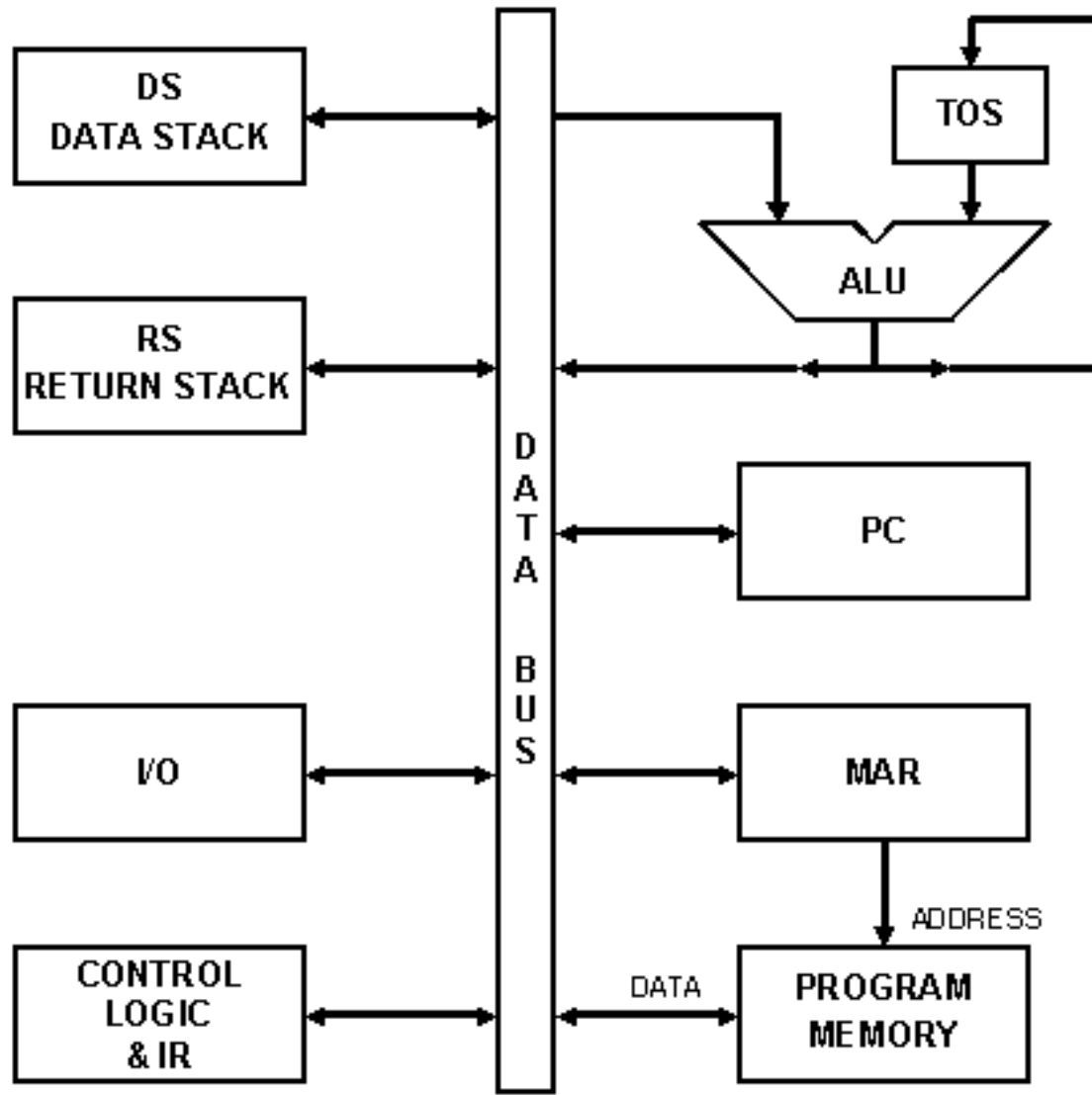


Figure 3.1 -- The canonical stack machine.

רשות:

Koopman, "Stack Computers: The New Wave," 1989.

[http://www.ece.cmu.edu/~koopman/stack\\_computers/sec3\\_2.html](http://www.ece.cmu.edu/~koopman/stack_computers/sec3_2.html)

קישור להורדה חינם של הספר:

[https://users.ece.cmu.edu/~koopman/stack\\_computers/stack\\_computers\\_book.pdf](https://users.ece.cmu.edu/~koopman/stack_computers/stack_computers_book.pdf)

DS = Data Stack

RS = Return Stack

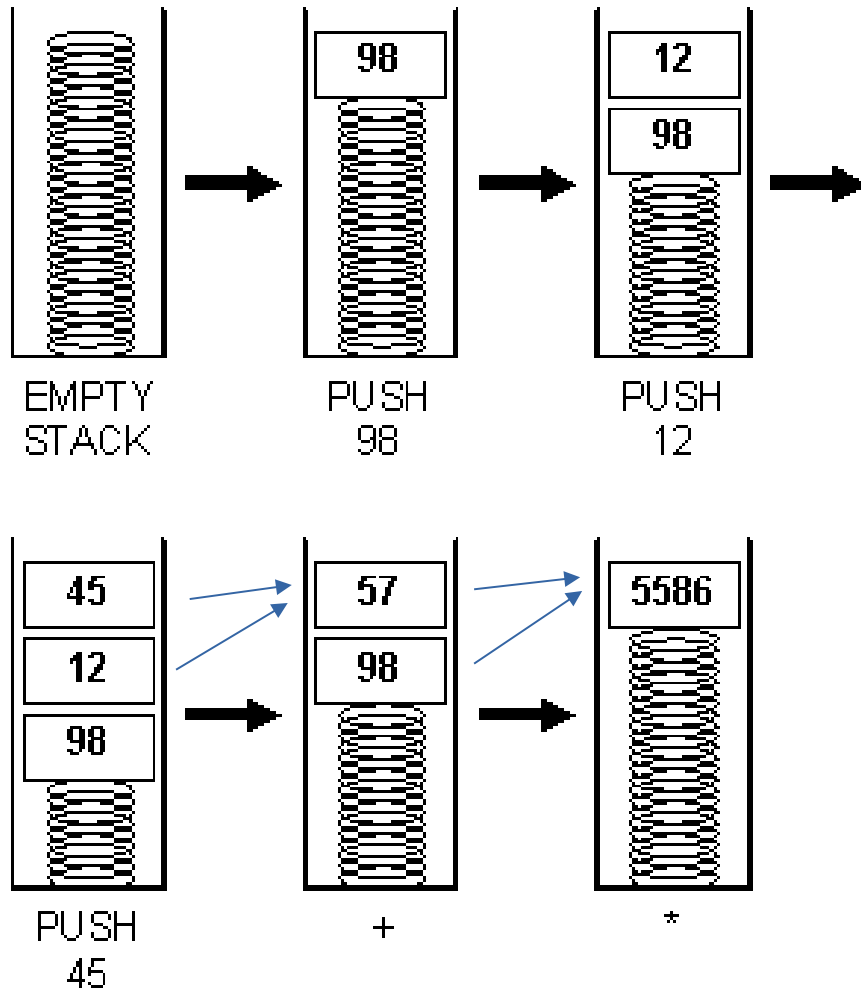
TOS = Top Of Stack register

PC = Program Counter

MAR = Memory Address Register

IR = Instruction Register

# An Example: Stack Machine Operation



איזה תרגיל מתבצע בדוגמה הבאה?

Koopman, "Stack Computers: The New Wave," 1989.

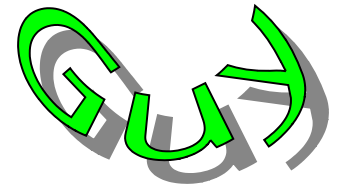
[http://www.ece.cmu.edu/~koopman/stack\\_computers/sec3\\_2.html](http://www.ece.cmu.edu/~koopman/stack_computers/sec3_2.html)

פתרון:

$(45+12)*98$

Figure 3.2 -- An example stack machine.

# Other Examples



- PDP-11: A 2-address machine
  - PDP-11 ADD: 4-bit opcode, 2 6-bit operand specifiers
  - Why? Limited bits to specify an instruction
  - Disadvantage: One source operand is always clobbered with the result of the instruction
    - *How do you ensure you preserve the old value of the source?*
- X86: A 2-address (memory/memory) machine
- Alpha: A 3-address (load/store) machine
- RISC-V, MIPS - 3 address machine
- ARM - 3 address machine

→ S op D → D  
S = source  
D = Destination  
Op = operation

Clobbered = מסורבל

Instead of: Source1 \* Source2 → Destination

We have: Source \* Destination → Destination

התוצאה "דורסת" את המקור

# PDP-11 Addressing modes

| ADDRESSING MODE              | DESCRIPTION                                                                                | FORMAT                                | EXAMPLE INSTRUCTION           |
|------------------------------|--------------------------------------------------------------------------------------------|---------------------------------------|-------------------------------|
| Absolute Mode                | Directly addresses an absolute memory location using a 15-bit address.                     | <code>@xxxxxxx xxxx</code>            | <code>LDAC 7777, R0</code>    |
| Direct Mode                  | Similar to absolute mode but uses a different syntax for clarity.                          | <code>.xxxxxx</code>                  | <code>LDB .1234, R0</code>    |
| Indexed Mode                 | Adds the contents of a base register to an immediate offset to form the effective address. | <code>base + offset</code>            | <code>LDAC @R1+777, R0</code> |
| Base Indexed Mode            | Combines base and indexed addressing; uses both a base register and an index register.     | <code>(base + index)</code>           | <code>LDB @(R2+R3), R0</code> |
| Stack Pointer Indexed Mode   | Uses the Stack Pointer (SP) as a base register to form the effective address.              | <code>(SP + offset)</code>            | <code>LDAC @SP+7, R0</code>   |
| Program Counter Indexed Mode | Uses the Program Counter (PC) as a base register to form the effective address.            | <code>(PC + offset)</code>            | <code>LDB @(PC+10), R0</code> |
| Immediate Mode               | Uses an immediate value directly in the instruction itself.                                | <code>#value</code>                   | <code>ADD #123, R1</code>     |
| Relative Mode                | Used for branching instructions where the offset is relative to the current address.       | <code>address or offset</code>        | <code>BRA +10</code>          |
| Bit Field Mode               | Allows addressing specific bits within a byte.                                             | <code>bitfield within<br/>byte</code> | <code>TEST #5, R0</code>      |
| Register Indirect Mode       | Uses an indirect register to form the effective address.                                   | <code>*(register)</code>              | <code>LDAC *@R2, R0</code>    |

# PDP-11





# X86 addressing modes

| ADDRESSING MODE                         | DESCRIPTION                                                                                               | FORMAT                                                       | EXAMPLE INSTRUCTION                          |
|-----------------------------------------|-----------------------------------------------------------------------------------------------------------|--------------------------------------------------------------|----------------------------------------------|
| Absolute Addressing                     | Directly addresses an absolute memory location using an immediate address.                                | <code>0x1234</code> or <code>0x5678</code>                   | <code>MOV EAX, 0x1234</code>                 |
| Direct Addressing                       | Similar to absolute addressing but uses a different syntax for clarity.                                   | <code>[0x1234]</code>                                        | <code>MOV EBX, [0x1234]</code>               |
| Immediate Addressing                    | Uses an immediate value directly in the instruction itself.                                               | <code>#value</code>                                          | <code>ADD EAX, #5</code>                     |
| Register Addressing                     | Directly uses a register as the operand or address.                                                       | <code>EAX</code> , <code>EBX</code> , etc.                   | <code>MOV EBX, EAX</code>                    |
| Register Indirect Addressing            | Uses a base register with an optional displacement to form the effective address.                         | <code>[EBX]</code> , <code>[EBX + 0x10]</code>               | <code>MOV EAX, [EBX]</code>                  |
| Base-Indexed Addressing                 | Combines a base register and an index register (with or without a displacement).                          | <code>[EBX + EBX*2 + 0x10]</code>                            | <code>MOV EAX, [EBX + ESI*4 + 0x10]</code>   |
| Base-Indexed-Displacement Addressing    | Uses both a base register and an index register with a displacement.                                      | <code>[EBX + ESI*2 + 0x10]</code>                            | <code>MOV EAX, [EBX + ESI*4 + 0x10]</code>   |
| Base-Indexed-Direct Addressing          | Combines a base register and an index register with direct addressing.                                    | <code>[EBX + ESI*2 + 0x1234]</code>                          | <code>MOV EAX, [EBX + ESI*4 + 0x1234]</code> |
| Base-Displacement Addressing            | Uses a base register with an optional displacement.                                                       | <code>[EBX + 0x10]</code>                                    | <code>MOV EAX, [EBX + 0x10]</code>           |
| Segmented Addressing (Real Mode)        | Uses a segment register and an offset to form the effective address.                                      | <code>[DS:0x1234]</code>                                     | <code>MOV EAX, [DS:0x1234]</code>            |
| Stack Pointer Indexed Addressing        | Uses the Stack Pointer (SP) as a base register to form the effective address.                             | <code>[ESP + 0x10]</code>                                    | <code>POP ESI</code>                         |
| Instruction Pointer Relative Addressing | Used for relative branching instructions where the offset is relative to the current instruction pointer. | <code>JMP short 0x1234</code> , <code>JMP near 0x1234</code> | <code>JMP short +0x10</code>                 |

# What Are the Elements of An ISA?

- **Instructions** לא רק סוג הפקודה הוא חלק מה- ISA אלא גם האופן שמביאים את הפקודה
  - Opcode addressing modes קיימים הרבה סוגי
  - Operand specifiers (addressing modes)
    - How to obtain the operand?

Why are there different addressing modes?

כדי לאפשר גמישות באופנים רבים אל הנתונים

- **Data types**

- Definition: Representation of information for which there are instructions that operate on the representation
- **Integer, floating point, character, binary, decimal, BCD**
- Doubly linked list, queue, string, bit vector, stack
  - VAX: INSQUEUE and REMQUEUE instructions on a doubly linked list or queue; FINDFIRST
  - Digital Equipment Corp., “[VAX11 780 Architecture Handbook](#),” 1977.
  - X86: SCAN opcode operates on character strings; PUSH/POP

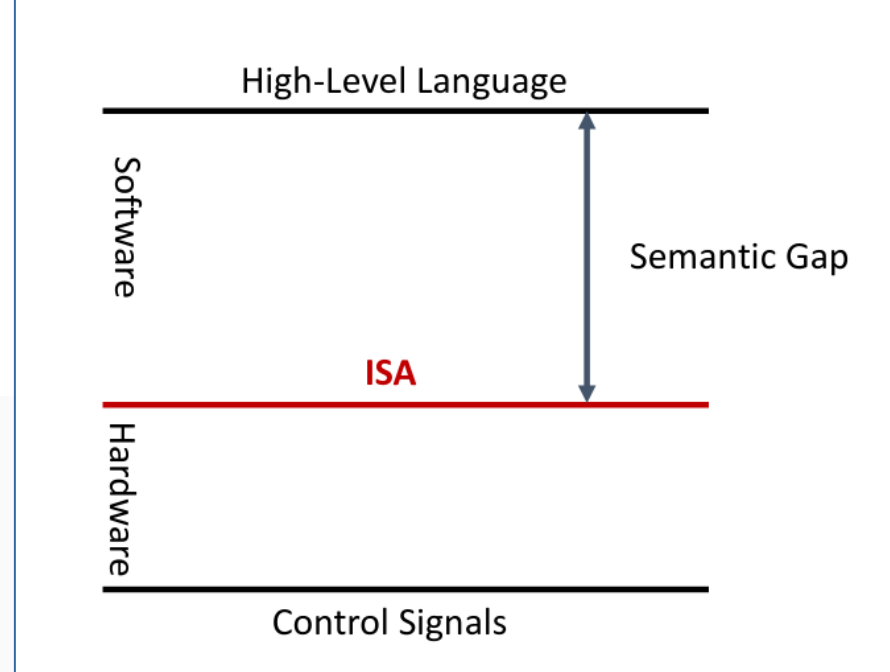
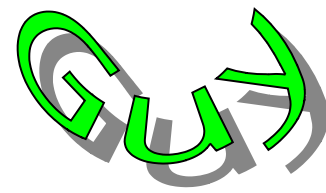
תמיכה בטיפוסי נתונים שונים משליכה על מורכבות ה- ISA

# VAX-11/780





# Semantic Gap



Semantic gap

Semantic gap

HLL (program)

ISA

HLL = High Level Language

ISA

Control Signals

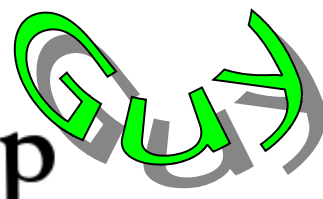
10/3/2020

# Semantic Gap

the "semantic gap" refers to the difference between the high-level language used by programmers and the low-level instructions understood by the hardware. The semantic gap can be eliminated by adding instructions that bridge this gap, but these additions can sometimes introduce what is called a "semantic clash."

According to Wulf et al. (1981), the semantic gap is the difference between high-level language constructs and the lower-level instructions in hardware.

# ISA-level Tradeoffs: Semantic Gap



- Where to place the ISA? Semantic gap
  - Closer to high-level language (HLL) → Small semantic gap, complex instructions
  - Closer to hardware control signals? → Large semantic gap, simple instructions
- RISC vs. CISC machines
  - RISC: Reduced instruction set computer
  - CISC: Complex instruction set computer
    - FFT, QUICKSORT, POLY, FP instructions?
    - VAX INDEX instruction (array access with bounds checking)

# ISA-level Tradeoffs: Semantic Gap

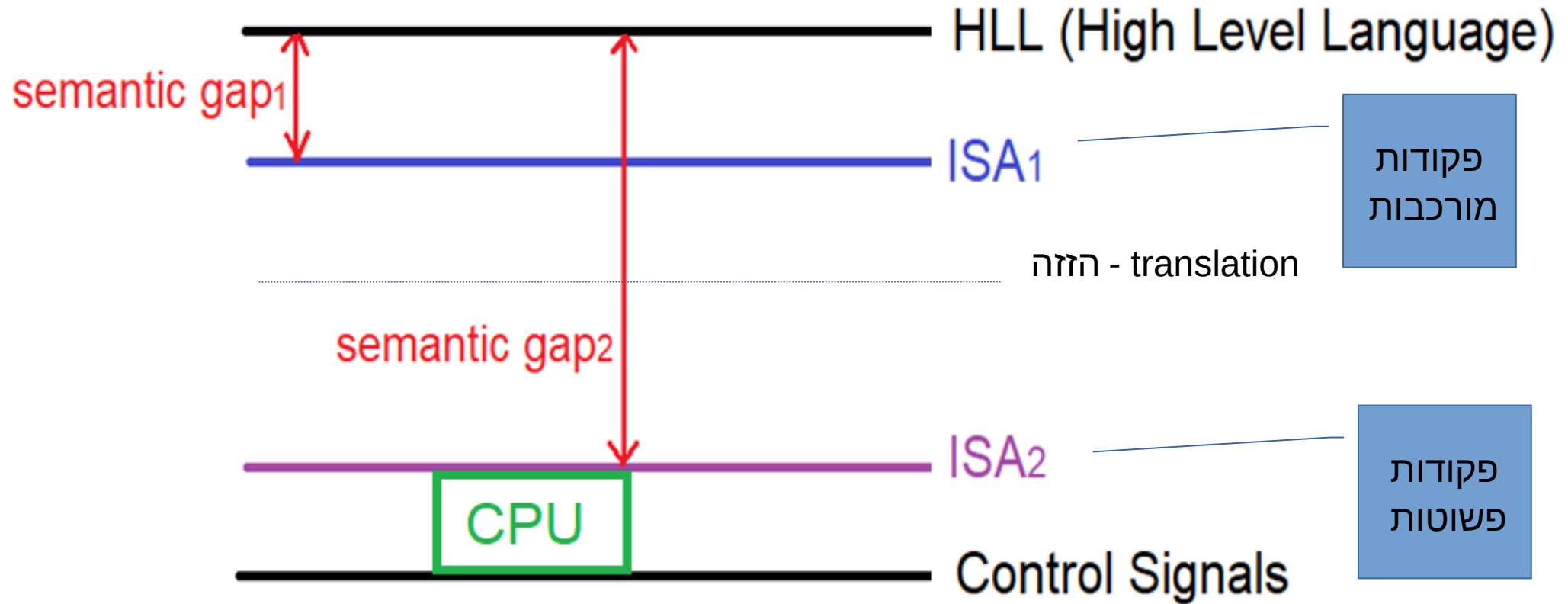


- Some tradeoffs (for you to think about)
- Simple compiler, complex hardware vs. complex compiler, simple hardware
  - Caveat: Translation (indirection) can change the tradeoff!  
אזהרה
- Burden of backward compatibility
- Performance?
  - Optimization opportunity: Example of VAX INDEX instruction: who (compiler vs. hardware) puts more effort into optimization?
  - Instruction size, code size

Source:

[https://passlab.github.io/CSE564/notes/lecture04\\_ISA\\_supplement.pdf](https://passlab.github.io/CSE564/notes/lecture04_ISA_supplement.pdf)

# Semantic Gap



# Data Type Tradeoffs

- What is the benefit of having more or high-level data types in the ISA?
- What is the disadvantage?
- Think compiler/programmer vs. microarchitect

כמה קרובים הטיפוסים הקיימים ב-ISA לטיפוסים הקיימים בשפה העילית
- Concept of **semantic gap**
  - Data types coupled tightly to the **semantic level**, or complexity of instructions

קישור:
- Example: Early RISC architectures vs. [Intel 432](#)
  - Early RISC: Only integer data type (לדוגמה MIPS)
  - Intel 432: Object data type, capability based machine (תכנות מונחה)  
דוגמה נוספת לסוג של נתונים הוא תמיכה בפורמט BCD:  
Binary Coded Decimal – ראו השקף הבא...

# אודות iAPX432, שנת 1981

## Why it failed?

Despite its innovations, the 432 was famously slow—often 5 to 10 times slower than the contemporary 8086 or the Motorola 68000.

The "Semantic Gap": By trying to bridge the gap between high-level languages and hardware so closely, the overhead became unbearable.

Complex Implementation: The design was so complex it had to be spread across multiple chips (the 43201 and 43202), which introduced communication delays.

The Intel 432 serves as a classic "cautionary tale" in computer architecture: a design can be mathematically and logically superior in its security and elegance, yet still fail if it cannot compete on raw execution speed.

see: <https://www.righto.com/2023/07/the-complex-history-of-intel-i960-risc.html>



# An Example: BCD

- Each decimal digit is encoded with a fixed number of bits



"Binary clock" by Alexander Jones & Eric Pierce - Own work, based on Wapcaplet's Binary clock.png on the English Wikipedia. Licensed under CC BY-SA 3.0 via Wikimedia Commons - [http://commons.wikimedia.org/wiki/File:Binary\\_clock.svg#mediaviewer/File:Binary\\_clock.svg](http://commons.wikimedia.org/wiki/File:Binary_clock.svg#mediaviewer/File:Binary_clock.svg)

"Digital-BCD-clock" by Julo - Own work. Licensed under Public Domain via Wikimedia Commons - <http://commons.wikimedia.org/wiki/File:Digital-BCD-clock.jpg#mediaviewer/File:Digital-BCD-clock.jpg>

להמחשה,

BCD הוא חלק מה-ISA  
של X86**FBSTP—Store BCD Integer and Pop**

| Opcode | Instruction  | 64-Bit Mode | Compat/Leg Mode | Description                          |
|--------|--------------|-------------|-----------------|--------------------------------------|
| DF /6  | FBSTP m80bcd | Valid       | Valid           | Store ST(0) in m80bcd and pop ST(0). |

**Description**

Converts the value in the ST(0) register to an 18-digit packed BCD integer, stores the result in the destination operand, and pops the register stack. If the source value is a non-integral value, it is rounded to an integer value, according to rounding mode specified by the RC field of the FPU control word. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

The destination operand specifies the address where the first byte destination value is to be stored. The BCD value (including its sign bit) requires 10 bytes of space in memory.

The following table shows the results obtained when storing various classes of numbers in packed BCD format.

**Table 3-19. FBSTP Results**

| ST(0)                                        | DEST |
|----------------------------------------------|------|
| $-\infty$ or Value Too Large for DEST Format | *    |
| $F \leq -1$                                  | -D   |
| $-1 < F < -0$                                | **   |
| -0                                           | -0   |
| +0                                           | +0   |
| $+0 < F < +1$                                | **   |
| $F \geq +1$                                  | +D   |
| $+\infty$ or Value Too Large for DEST Format | *    |
| NaN                                          | *    |

**NOTES:**

F Means finite floating-point value.

D Means packed-BCD number.

\* Indicates floating-point invalid-operation (#IA) exception.

\*\*  $\pm 0$  or  $\pm 1$ , depending on the rounding mode.

If the converted value is too large for the destination format, or if the source operand is an  $\infty$ , SNaN, QNaN, or is in an unsupported format, an invalid-arithmetic-operand condition is signaled. If the invalid-operation exception is not masked, an invalid-arithmetic-operand exception (#IA) is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the packed BCD indefinite value is stored in memory.

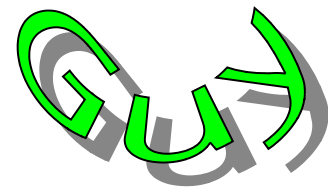
This instruction's operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

DEST  $\leftarrow$  BCD(ST(0));

PopRegisterStack;

# What Are the Elements of An ISA?



## ■ Memory organization

□ **Address space:** How many uniquely identifiable locations in memory

כמות הכתובות  
מרחב הזיכרון

□ **Addressability:** How much data does each uniquely identifiable location store

הכי פשוט זה לתמוך רק בסוג אחד אבל המשתמשים צריכים גמישות. למשל יישומים של עיבוד תמונה יזדקקו לפעולות על בייטים.

■ Byte addressable: most ISAs, characters are 8 bits

■ Bit addressable: Burroughs 1700. Why?

דרישה לגישה להגיע לכל ביט בודד בזיכרון היא נדירה (Burroughs 1700):

■ 64-bit addressable: Some supercomputers. Why?

לדוגמה "טען את הביט הזה..."

חומר  
למחשבה

32-bit addressable: First Alpha(\*)

הסיבה לכך שהמחשב יועד לווירטואליזציה של ISA של כל מערכת (\*\*) ולכן פיסת המידע הקטנה ביותר, ביט, איפשרה את מקסימום הגמישות.

■ Food for thought

□ How do you add 2 32-bit numbers with only byte addressability?

□ How do you add 2 8-bit numbers with only 32-bit addressability?

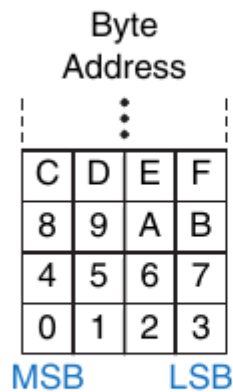
□ **Big endian vs. little endian?** MSB at low or high byte.

□ **Support for virtual memory**

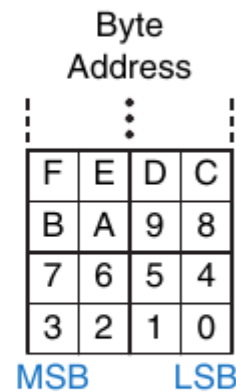
(\*) לא היה ניתן לקרוא בייט בודד, אם רצו לקרוא בייט היה צריך לעשות פעולות הזזה, shift, ו-masking היתה תמיכה רק בפעולות על word ללא תמיכה בפעולות על בייט. בהמשך בעקבות דרישה מהמשתמשים זה שונה.

virtualized ISA (\*\*) 54

### Big-Endian



### Little-Endian



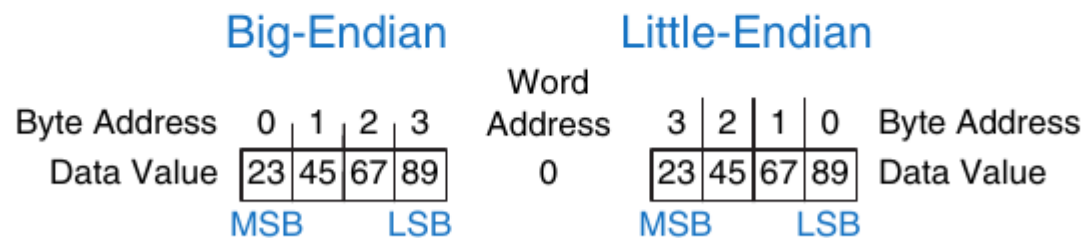
**Figure 6.3 Big- and little-endian memory addressing**

Byte-addressable memories are organized in a big-endian or little-endian fashion, as shown in Figure 6.3. In both formats, the most significant byte (MSB) is on the left and the least significant byte (LSB) is on the right.

In big-endian machines, bytes are numbered starting with 0 at the big (most significant) end. In little-endian machines, bytes are numbered starting with 0 at the little (least significant) end. Word addresses are the same in both formats and refer to the same four bytes. Only the addresses of bytes within a word differ.

**Solution:** Figure 6.4 shows how big- and little-endian machines store the value 0x23456789 in memory word 0. After the load byte instruction, `lb $s0, 1($0)`, `$s0` would contain 0x00000045 on a big-endian system and 0x00000067 on a little-endian system.

הסבר



**Figure 6.4 Big-endian and little-endian data storage**

הדגמה בכיתה:  
הפלט של פקודת `lscpu`

Credit for this slide and the previous slides:  
**Harris & Harris,**  
“*Digital Design and Computer Architecture*”



The terms big-endian and little-endian come from Jonathan Swift's *Gulliver's Travels*, first published in 1726 under the pseudonym of Isaac Bickerstaff. In his stories the Lilliputian king required his citizens (the Little-Endians) to break their eggs on the little end. The Big-Endians were rebels who broke their eggs on the big end.

GUY





# כדי שיהיה קל לזכור...(\*)

## Big-Endian (הסדר ההגיוני/המדעי)

במערכת **Big-Endian**, המיקום של המידע הראשי (המספר הגדול ביותר) מגיע קודם.

- **מה זה אומר?** הבייט בעל הערך הגבוה ביותר (Most Significant Byte – MSB) נכתב או נשמר בכתובת הזיכרון הנמוכה ביותר (כלומר, קודם).
- **האנלוגיה:** זה כמו שאתה כותב מספר רגיל (למשל, 1234). אתה כותב את הספרה הגדולה ביותר (1) ראשונה משמאל. זה הסדר ההגיוני והמקובל לקריאה אנושית.
- **דוגמה:** אם המספר הוא `0x12345678`, הוא יישמר בזיכרון בסדר: `12 ← 34 ← 56 ← 78`.

## Little-Endian (הסדר ההופכי)

במערכת **Little-Endian**, המיקום של המידע המצטבר (המספר הקטן ביותר) מגיע קודם.

- **מה זה אומר?** הבייט בעל הערך הנמוך ביותר (Least Significant Byte – LSB) נכתב או נשמר בכתובת הזיכרון הנמוכה ביותר (כלומר, קודם).
- **האנלוגיה:** זה כמו להפוך את המספר שהיינו רגילים לקרוא אותו. במקום להתחיל במספר הגדול ביותר, מתחילים במספר הקטן ביותר.
- **דוגמה:** אם המספר הוא `0x12345678`, הוא יישמר בזיכרון בסדר: `12 ← 34 ← 56 ← 78`.

(\*) תודה למודל החדש Gemma 4 (2.4.2026)



# B 1700

## CENTRAL SYSTEM

**Burroughs**

FIELD ENGINEERING

# TECHNICAL MANUAL

Copyright © 1972  
Burroughs Corporation  
Detroit, Michigan 48232



INTRODUCTION  
AND  
OPERATION

FUNCTIONAL  
DETAIL

CIRCUIT  
DETAIL

ADJUSTMENTS

MAINTENANCE  
PROCEDURES

INSTALLATION  
PROCEDURES

RELIABILITY  
IMPROVEMENT  
NOTICES

OPTIONAL  
FEATURES



קישורים לסרטונים אודות החברה (רשות):

<https://www.youtube.com/watch?v=eyOpO-040W8>

<https://www.youtube.com/watch?v=QeiuVNDQg4k>



# Some Historical Readings – קריאת רשות

- If you want to dig deeper
- “ ,WilnerDesign of the Burroughs 1700.AFIPS 1972 ”,
- “ ,LevyThe Intel iAPX 432.1981 ”,
  - <http://www.cs.washington.edu/homes/levy/capabook/Chapter9.pdf>

# What Are the Elements of An ISA?

## ■ Registers

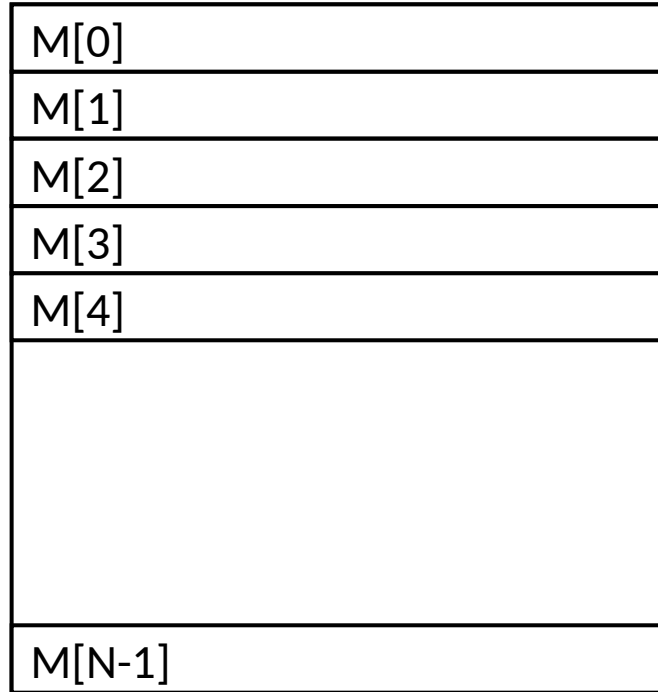
- How many? ( the answer may include 0 )
- Size of each register

## ■ Why is having registers a good idea?

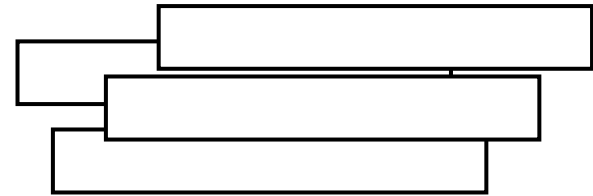
- Because programs exhibit a characteristic called **data locality**
- **A recently produced/accessed value is likely to be used more than once (temporal locality)**
  - Storing that value in a register eliminates the need to go to memory each time that value is needed

*\* Re-use the data!*

# 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 instruction

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

# Aside: Programmer Invisible State

- Microarchitectural state
- Programmer cannot access this directly
  
- e.g. cache state
- e.g. pipeline registers

# Evolution of Register Architecture

ציר הזמן

- Accumulator
  - a legacy from the “adding” machine days
- Accumulator + address registers
  - need register indirection
  - initially address registers were special-purpose, i.e., can only be loaded with an address for indirection
  - eventually arithmetic on addresses became supported
- General purpose registers (GPR)
  - all registers good for all purposes
  - grew from a few registers to 32 (common for RISC) to 128 in Intel IA-64

- Operate instructions – הוראות של ביצוע פעולות
  - Process data: arithmetic and logical operations
  - Fetch operands, compute result, store result
  - Implicit sequential control flow
- Data movement instructions – הוראות של הזזת נתונים
  - Move data between memory, registers, I/O devices
  - Implicit sequential control flow
- Control flow instructions – הוראות של בקרת התקדמות התכנית
  - Change the sequence of instructions that are executed

# What Are the Elements of An ISA?

- Load/store vs. memory/memory architectures
    - Load/store architecture: operate instructions operate only on registers
      - E.g., RISC-V, MIPS, ARM and many RISC ISAs
    - Memory/memory architecture: operate instructions can operate on memory locations
      - E.g., x86, VAX and many CISC ISAs
- אבל לא קיים ב RISC



# What Are the Elements of An ISA?

## ■ Addressing modes specify how to obtain the operands

- ❑ Absolute       $\text{LW rt, 10000}$   
use immediate value as address
- ❑ Register Indirect:       $\text{LW rt, (r}_{\text{base}}\text{)}$   
use  $\text{GPR[r}_{\text{base}}\text{]}$  as address
- ❑ Displaced or based:       $\text{LW rt, offset(r}_{\text{base}}\text{)}$   
use  $\text{offset+GPR[r}_{\text{base}}\text{]}$  as address
- ❑ Indexed:       $\text{LW rt, (r}_{\text{base}}, \text{r}_{\text{index}}\text{)}$   
use  $\text{GPR[r}_{\text{base}}\text{]}+\text{GPR[r}_{\text{index}}\text{]}$  as address
- ❑ Memory Indirect       $\text{LW rt ((r}_{\text{base}}\text{)})$   
use value at  $\text{M[ GPR[ r}_{\text{base}} ]]}$  as address
- ❑ Auto inc/decrement       $\text{LW Rt, (r}_{\text{base}}\text{)}$   
use  $\text{GRP[r}_{\text{base}}\text{]}$  as address, but inc. or dec.  $\text{GPR[r}_{\text{base}}\text{]}$  each time

Load Word

שרשור כפול: הערך ברגיסטר מפנה  
לכתובת בזיכרון שמפנה לכתובת  
הסופית בזיכרון ממנה יש לטעון את  
הנתון

אחרי ביצוע הפעולה קדם או החסר את הערך ברגיסטר. הסגנון הזה יעיל כאשר למשל  
עוברים על פני מערך

# What Are the Benefits of Different Addressing Modes?

- Another example of programmer vs. microarchitect tradeoff
- Advantage of more addressing modes:
  - Enables **better mapping of high-level constructs to the machine**: some accesses are better expressed with a different mode → reduced number of instructions and code size
    - Think array accesses (autoincrement mode)
    - Think indirection (pointer chasing): מצביע למצביע, מצביע של מצביעים...
    - Sparse matrix accesses – מערך של מצביעים לאלמנטים השונים מאפס
- Disadvantage:
  - **More work** for the **compiler** – יש לקומפילר הרבה אפשרויות זה עלול להקשות עליו
  - **More work** for the **microarchitect**

# ISA Orthogonality

- **Orthogonal** ISA: הגדרה
  - **All** addressing modes can be used with **all** instruction types
  - Example: VAX
    - (~13 addressing modes) x (>300 opcodes) x (integer and FP formats)  
גמישות רבה
- Who is this good for?
  - the compiler designer
- Who is this bad for?

ראו

[Orthogonal Instruction Set Wiki](#)

# Is the LC-3b ISA Orthogonal?

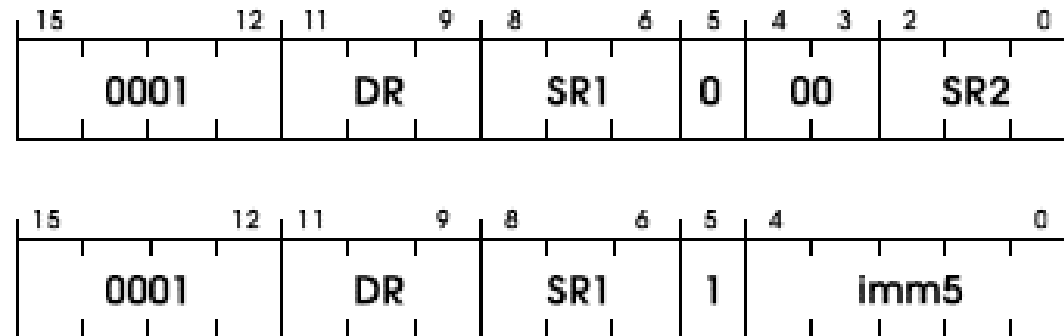
|                  | 15   | 14 | 13 | 12 | 11           | 10                | 9 | 8         | 7 | 6 | 5        | 4       | 3       | 2 | 1 | 0 |
|------------------|------|----|----|----|--------------|-------------------|---|-----------|---|---|----------|---------|---------|---|---|---|
| ADD <sup>+</sup> | 0001 |    |    |    | DR           |                   |   | SR1       |   |   | A        | op.spec |         |   |   |   |
| AND <sup>+</sup> | 0101 |    |    |    | DR           |                   |   | SR1       |   |   | A        | op.spec |         |   |   |   |
| BR               | 0000 |    |    |    | n            | z                 | p | PCOffset9 |   |   |          |         |         |   |   |   |
| JMP              | 1100 |    |    |    | 000          |                   |   | BaseR     |   |   | 000000   |         |         |   |   |   |
| JSR(R)           | 0100 |    |    |    | A            | operand.specifier |   |           |   |   |          |         |         |   |   |   |
| LDB <sup>+</sup> | 0010 |    |    |    | DR           |                   |   | BaseR     |   |   | boffset6 |         |         |   |   |   |
| LDW <sup>+</sup> | 0110 |    |    |    | DR           |                   |   | BaseR     |   |   | offset6  |         |         |   |   |   |
| LEA <sup>+</sup> | 1110 |    |    |    | DR           |                   |   | PCOffset9 |   |   |          |         |         |   |   |   |
| RTI              | 1000 |    |    |    | 000000000000 |                   |   |           |   |   |          |         |         |   |   |   |
| SHF <sup>+</sup> | 1101 |    |    |    | DR           |                   |   | SR        |   |   | A        | D       | amount4 |   |   |   |
| STB              | 0011 |    |    |    | SR           |                   |   | BaseR     |   |   | boffset6 |         |         |   |   |   |
| STW              | 0111 |    |    |    | SR           |                   |   | BaseR     |   |   | offset6  |         |         |   |   |   |
| TRAP             | 1111 |    |    |    | 0000         |                   |   | trapvect8 |   |   |          |         |         |   |   |   |
| XOR <sup>+</sup> | 1001 |    |    |    | DR           |                   |   | SR1       |   |   | A        | op.spec |         |   |   |   |
| not used         | 1010 |    |    |    |              |                   |   |           |   |   |          |         |         |   |   |   |
| not used         | 1011 |    |    |    |              |                   |   |           |   |   |          |         |         |   |   |   |

**תשובה לא.**  
**למשל, לא לכל ההוראות**  
**תמיכה ב-immediate**

# LC-3b: Addressing Modes of ADD

## 2 אופני מיעון

### Encodings



### Operation

```
if (bit[5] == 0)
    DR = SR1 + SR2;
else
    DR = SR1 + SEXT(imm5);
setcc();
```

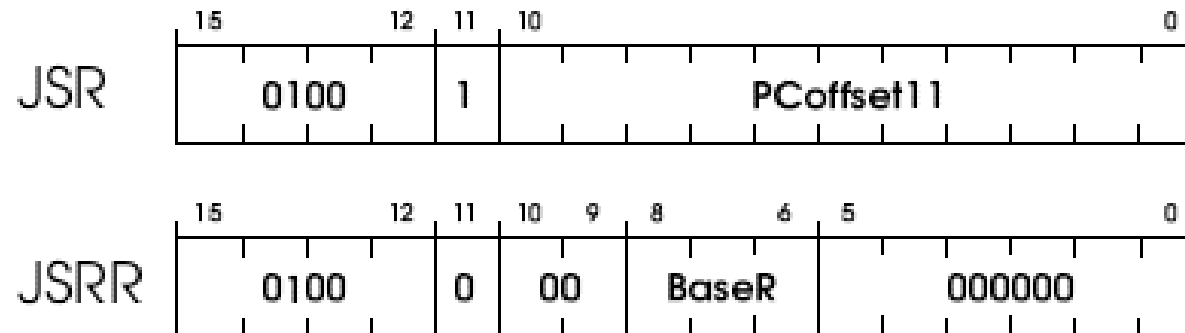
setcc()

Indicates that condition codes N, Z, and P are set based on the value of the result written to DR. If the value is negative, N = 1, Z = 0, P = 0. If the value is zero, N = 0, Z = 1, P = 0. If the value is positive, N = 0, Z = 0, P = 1.

# LC-3b: Addressing Modes of JSR(R)

JSR = Jump to Subroutine

## Encodings



## Operation

```
R7 = PC† ;  
if (bit[11] == 0)  
    PC = BaseR;  
else  
    PC = PC† + LSHF(SEXT(PCOffset11), 1);
```

Relative addressing

**LC-3b is not  
Orthogonal**

## Description

Left shift

Sign extend

First, the incremented PC is saved in R7. This is the linkage back to the calling routine. Then, the PC is loaded with the address of the first instruction of the subroutine, causing an unconditional jump to that address. The address of the subroutine is obtained from the base register (if bit[11] is 0), or the address is computed by sign-extending bits [10:0] to 16 bits, left-shifting the result one bit, and then adding this value to the incremented PC (if bit[11] is 1).

# What Are the Elements of an ISA?

- How to interface with I/O devices

- Memory mapped I/O

- A region of memory is mapped to I/O devices
    - I/O operations are loads and stores to those locations

- Special I/O instructions

- IN and OUT instructions in x86 deal with ports of the chip
      - You can LOAD from the keyboard. You can STORE to the monitor

- Tradeoffs?

- Which one is more general purpose?
      - גיא: Memory mapped I/O is more general purpose. Any kind of device can be accessed as long as it can be mapped to a memory location.



# What Are the Elements of An ISA?

## ■ Privilege modes – לא נעסוק בזה

- User vs supervisor
- Who can execute what instructions?

## ■ Exception and interrupt handling

- What procedure is followed when something goes wrong with an instruction?
- What procedure is followed when an external device requests the processor?
- **Vectored vs. non-vectored interrupts** (early MIPS)
  - Vectored – כאשר יש את המידע מי הפריע למעבד? איזה התקן ביצע את הפסיקה
  - Non vectored interrupts - המידע לא קיים, המעבד צריך לנחש מי הפריע לו - פשטות לעומת מורכבות

## ■ Virtual memory

- Each program has the illusion of the entire memory space, which is greater than physical memory (Guy: "swap space" in Linux)

# Another Question or Two

- Q: Does the LC-3b ISA contain complex instructions?
  - A: No
- How complex can an instruction be?

# Complex vs. Simple Instructions

- Complex instruction: An instruction does a lot of work, e.g. many operations
  - Insert in a doubly linked list
  - Compute FFT
  - String copy
- Simple instruction: An instruction does small amount of work, it is a **primitive** using which complex operations can be built
  - Add
  - XOR
  - Multiply

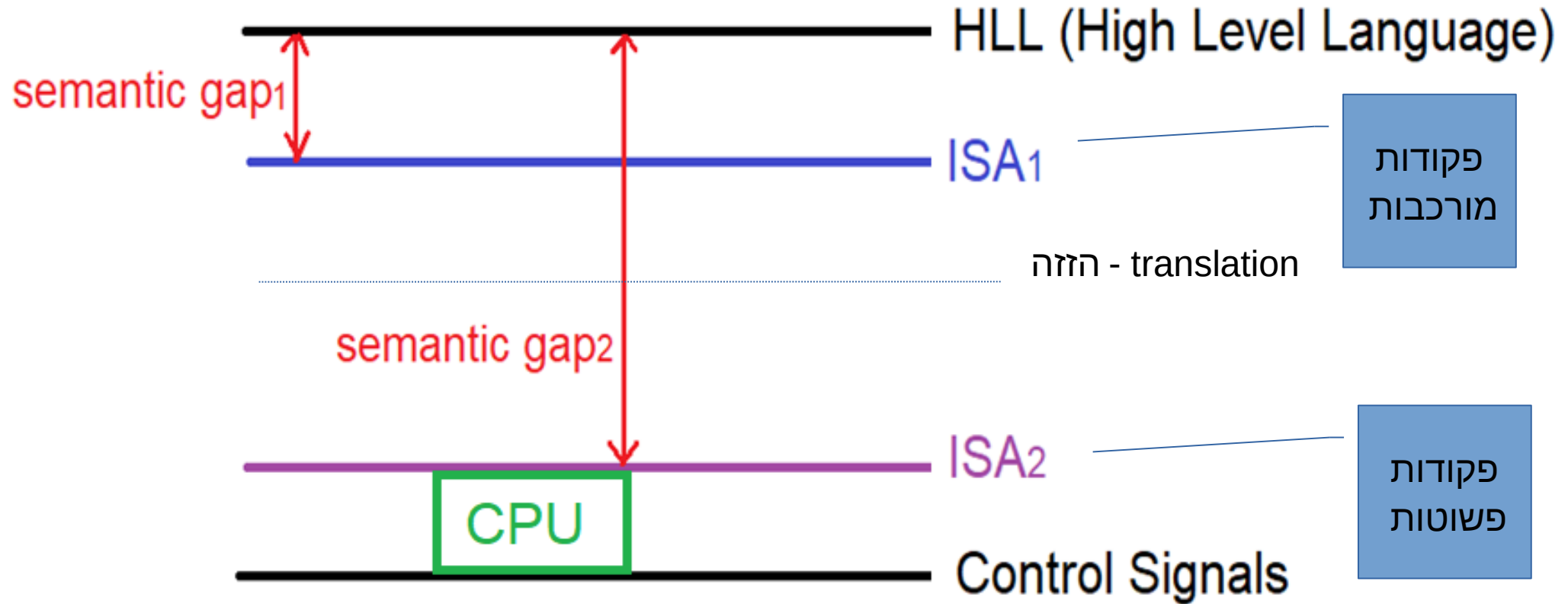
# Complex vs. Simple Instructions

- Advantages of Complex instructions
  - + Denser encoding → smaller code size → better memory utilization, saves off-chip bandwidth, better cache hit rate (better packing of instructions)
  - + Simpler compiler: no need to optimize small instructions as much
- Disadvantages of Complex Instructions
  - Larger chunks of work → compiler has less opportunity to optimize (limited in fine-grained optimizations it can do)
  - More complex hardware → translation from a high level to control signals and optimization needs to be done by hardware

# ISA-level Tradeoffs: Semantic Gap

- Where to place the ISA? Semantic gap
  - Closer to high-level language (HLL) → Small semantic gap, complex instructions
  - Closer to hardware control signals? → Large semantic gap, simple instructions
- RISC vs. CISC machines
  - RISC: Reduced instruction set computer
  - CISC: Complex instruction set computer
    - FFT, QUICKSORT, POLY, FP instructions?
    - VAX INDEX instruction (array access with bounds checking)

# Semantic Gap



# ISA-level Tradeoffs: Semantic Gap

- Some tradeoffs (for you to think about)
- Simple compiler, complex hardware vs. complex compiler, simple hardware
  - Caveat אזהרה: **Translation** (indirection) can change the trade-off!
- Burden of backward compatibility
- Performance? Energy Consumption?
  - Optimization opportunity: Example of VAX INDEX instruction: who (compiler vs. hardware) puts more effort into optimization?
  - Instruction size, code size

ראו  
השקף הבא

## 5.7

## Fallacies and Pitfalls



*Pitfall: Microcode implementing a complex instruction may not be faster than a sequence using simpler instructions.*

Most machines with a large and complex instruction set are implemented using a microcode stored in ROM. Surprisingly, on such machines, sequences of individual simpler instructions are sometimes as fast as or even faster than the custom microcode sequence for a particular instruction. How can this possibly be true? At one time, microcode had the advantage of being fetched from a much faster memory than instructions in the program. Since caches came into use in 1968, microcode no longer has such a consistent edge in fetch time. Microcode does, however, still have the advantage of using internal temporary registers in the computation, which can be helpful on machines with few general-purpose registers. The disadvantage of microcode is that the algorithms must be selected before the machine is announced and can't be changed until the next model of the architecture. The instructions in a program, on the other hand, can utilize improvements in its algorithms at any time during the life of the machine. Along the same lines, the microcode sequence is probably not optimal for all possible combinations of operands.

The **VAX Index** instruction provides an example of the above phenomenon. This instruction checks to see if the **index** is between two bounds, one of which is usually 0. The **VAX-11/780** microcode uses two compares and two branches to do this, while the standard instructions can perform the same check in one compare and one branch. The program can check the **index** against the upper limit using *unsigned* comparisons rather than two's complement comparisons. This treats a negative **index** (less than zero and so failing the comparison) as if it were a very large number, thus exceeding the upper limit. (The algorithm can be used with nonzero lower bounds by first subtracting the lower bound from the **index**.) Replacing the **Index** instruction by a sequence of **VAX** instructions to perform the necessary function always improves performance on the **VAX-11/780**.

*Fallacy: If there is space in control store, new instructions are free of cost.*



# X86: Small Semantic Gap: String Operations

- An instruction operates on a string
  - Move one string of arbitrary length to another location
  - Compare two strings
- Enabled by the ability to specify repeated execution of an instruction (in the ISA)
  - Using a “prefix” called REP prefix (REP = Repeat)
- Example: REP MOVS instruction
  - Only two bytes: REP prefix byte and MOVS opcode byte (F2 A4)
  - Implicit source and destination registers pointing to the two strings (ESI, EDI) – (=Extended Source/Destination Index)
  - Implicit count register (ECX) specifies how long the string is

```
cld      ; Clear direction flag (forward copy)
mov ecx, 100 ; Copy 100 bytes
mov esi, source_address
mov edi, destination_address
rep movsb ; Repeat move byte
```

cld sets the direction flag to forward, so esi and edi will be incremented.

mov ecx, 100 tells the rep movsb to move 100 bytes.

mov esi, source\_address sets the source address.

mov edi, destination\_address sets the destination address.

rep movsb then performs the byte copy.

The CPU checks if ecx is zero. If ecx is not zero, the process repeats.

# X86: Small Semantic Gap: String Operations

לכל טיפוס נתונים...

## REP MOVS (DEST SRC)

כפקודה אחת לעומת קוד שקול:

```
IF AddressSize = 16
  THEN
    Use CX for CountReg;
  ELSE IF AddressSize = 64 and REX.W used
    THEN Use RCX for CountReg; FI;
  ELSE
    Use ECX for CountReg;
FI;
WHILE CountReg ≠ 0 ←
DO
  Service pending interrupts (if any);
  Execute associated string instruction;
  CountReg ← (CountReg - 1);
  IF CountReg = 0
    THEN exit WHILE loop; FI;
  IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
  or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
    THEN exit WHILE loop; FI;
OD;
```

חומר למחשבה

*How many instructions does this take in MIPS?*

DEST ← SRC;

```
IF (Byte move)
  THEN IF DF = 0
    THEN
      (R|E)SI ← (R|E)SI + 1;
      (R|E)DI ← (R|E)DI + 1;
    ELSE
      (R|E)SI ← (R|E)SI - 1;
      (R|E)DI ← (R|E)DI - 1;
    FI;
  ELSE IF (Word move)
    THEN IF DF = 0
      (R|E)SI ← (R|E)SI + 2;
      (R|E)DI ← (R|E)DI + 2;
      FI;
    ELSE
      (R|E)SI ← (R|E)SI - 2;
      (R|E)DI ← (R|E)DI - 2;
    FI;
  ELSE IF (Doubleword move)
    THEN IF DF = 0
      (R|E)SI ← (R|E)SI + 4;
      (R|E)DI ← (R|E)DI + 4;
      FI;
    ELSE
      (R|E)SI ← (R|E)SI - 4;
      (R|E)DI ← (R|E)DI - 4;
    FI;
  ELSE IF (Quadword move)
    THEN IF DF = 0
      (R|E)SI ← (R|E)SI + 8;
      (R|E)DI ← (R|E)DI + 8;
      FI;
    ELSE
      (R|E)SI ← (R|E)SI - 8;
      (R|E)DI ← (R|E)DI - 8;
    FI;
  FI;
```

# Small Semantic Gap Examples in VAX

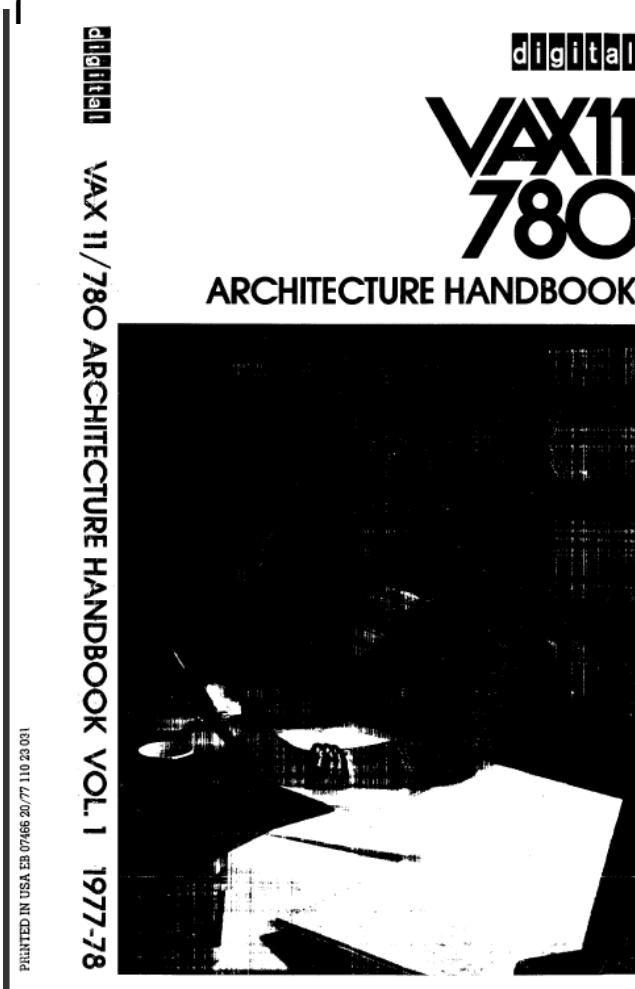
דוגמאות נוספות למרחק סמנטי קצר (<-- פקודות מורכבות

- FIND FIRST
  - Find the first set bit in a bit field
  - Helps OS resource allocation operations
- SAVE CONTEXT, LOAD CONTEXT
  - Special context switching instructions
- INSQUEUE, REMQUEUE
  - Operations on doubly linked list
- INDEX
  - Array access with bounds checking
- STRING Operations
  - Compare strings, find substrings, ...
- Cyclic Redundancy Check Instruction
- EDITPC
  - Implements editing functions to display fixed format output

Edit Packed to Character String

זוהי פקודת מאקרו מורכבת שמתורגמת למיקרו הוראות. למשל המרת מספר מ-BCD ל ASCII

- Digital Equipment Corp., "[VAX11 780 Architecture Handbook](#)," 1977-78.



# Small versus Large Semantic Gap



## ■ CISC vs. RISC

- Complex instruction set computer → complex instructions
  - Initially motivated by “not good enough” code generation(\*)
- Reduced instruction set computer → simple instructions
  - **John Cocke**, mid 1970s, IBM 801(\*\*)
    - Goal: enable better compiler control and optimization

## ■ RISC motivated by

- Memory stalls (no work done in a complex instruction when there is a memory *stall*?)
- Simplifying the hardware → lower cost, higher frequency
- Enabling the compiler to optimize the code better
  - Find fine-grained parallelism to reduce *stalls*(\*\*\*)

(\*) הקומפילרים הראשונים לא היו יעילים מספיק

(\*\*) חתן פרס טיורינג. מתכנן מחשבי RISC הראשונים

(\*\*\*) עיכובים עקב ריבוי גישות לזיכרון הפוגם ביעילות 86



MORE ACM AWARDS



## A.M. TURING AWARD LAUREATES BY...

ALPHABETICAL LISTING

YEAR OF THE AWARD

RESEARCH SUBJECT



## PHOTOGRAPHS

## BIRTH:

Charlotte, North Carolina, May 30, 1925

## DEATH:

Valhalla, New York, July 16, 2002

## EDUCATION:

BSE (Bachelor of Science in Engineering, Duke University, mechanical engineering, 1946); Ph.D. (Duke University, Mathematics, 1953).

## JOHN COCKE



United States – 1987

## CITATION

For significant contributions in the design and theory of compilers, the architecture of large systems and the development of reduced instruction set computers (RISC); for discovering and systematizing many fundamental transformations now used in optimizing compilers including reduction of operator strength, elimination of common subexpressions, register allocation, constant propagation, and dead code elimination.



SHORT  
ANNOTATED  
BIBLIOGRAPHY



ACM TURING  
AWARD  
LECTURE



RESEARCH  
SUBJECTS

John Cocke was born in Charlotte, North Carolina, in 1925. His father Norman was the president of [Duke Power Company](#) (later Duke Energy) and a member of the Board of Trustees of Duke University. That connection made it natural for John to attend college there. He earned his BSE (Bachelor of Science in Engineering) in Mechanical Engineering in 1946 and his PhD in mathematics in 1956. After graduation he joined IBM and worked at the T.J. Watson Research Center until his retirement 37 years later.

# An Aside

- An Historical Perspective on RISC Development at IBM
  - <http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/risc/>

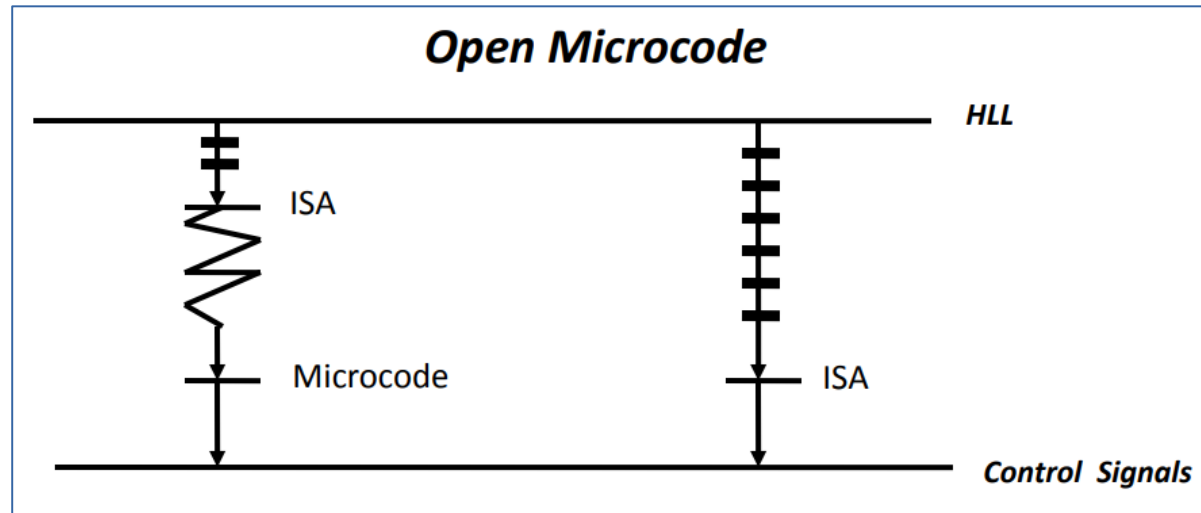
Guy> IBM's "**POWER**" standing for "**P**erformance **O**ptimized **W**ith **E**nhanced **R**ISC,"

# How High or Low Can You Go?

הדגמה אפשרית (ללא מסקנות):  
/lecture02/code קמפל לאסמבלי את hello.c עבור x86 ועבור RISC-V והשווה את גודל הקבצים

## ■ Very large semantic gap

- Each instruction specifies the complete set of control signals in the machine
- Compiler generates control signals
- Microcode
  - Gave way to optimizing compilers



## ■ Very small semantic gap

- ISA is (almost) the same as high-level language
- Java machines, LISP machines, object-oriented machines, capability-based machines – אלה שפות איטיות, כדי להאיץ אותן נדרש תכנון מתאים בחומרה

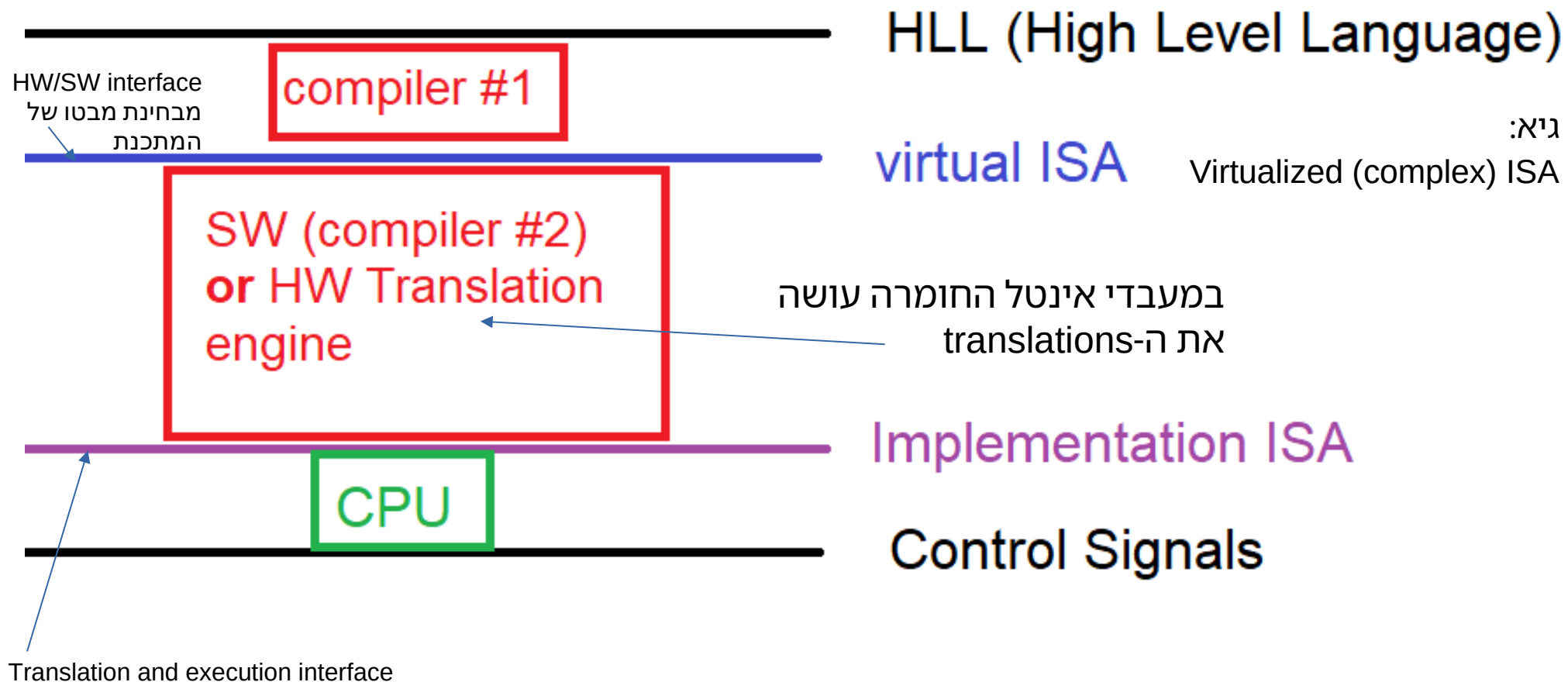


# A Note on ISA Evolution

- ISAs have evolved to reflect/satisfy the concerns of the day
- Examples: If you have...
  - Limited on-chip and off-chip memory size → אולי פקודות מורכבות יוכלו לעזור
  - Limited compiler optimization technology → שוב, יתרון לפקודות שימוש ב translation "שובר" את כללי המשחק של פשרות: מורכבות
  - Limited memory bandwidth → יתרון לפקודות מורכבות
  - Need for specialization in important applications (e.g., MMX)
- Use of **translation** (in HW and SW) enabled underlying implementations to be similar, regardless of the ISA
  - Concept of dynamic/static interface: translation/interpretation
  - Contrast it with hardware/software interface

מעבר מ-ISA אחד ל-ISA אחר --> ראו השקף הבא כדי לשנות את הפשרות של ה-semantic gap

# ISA (virtual ISA) → Implementation ISA



אנחנו מרוויחים יכולת שימוש בפקודות מורכבות בעוד שהמעבד עובד עם פקודות פשוטות

# Effect of Translation

- One can translate from one ISA to another ISA to change the semantic gap tradeoffs
  - ISA (virtual ISA המתכנת שרואה) → Implementation ISA
- Examples
  - Intel's and AMD's x86 implementations translate x86 instructions into programmer-invisible micro-operations (simple instructions) in hardware
  - Rosetta 2 (Apple): Apple's Rosetta 2 technology dynamically translates x86\_64 applications to run on Apple Silicon (ARM) Macs.
  - QEMU: is an open-source emulator and virtualizer that can perform binary translation.
  - Transmeta's x86 implementations translated x86 instructions into "secret" VLIW instructions in software (**code morphing software** - CMS)

לב המעבדים של אינטל דומה ל RISC בזכות ה HW translator

# Hardware-Based Translation

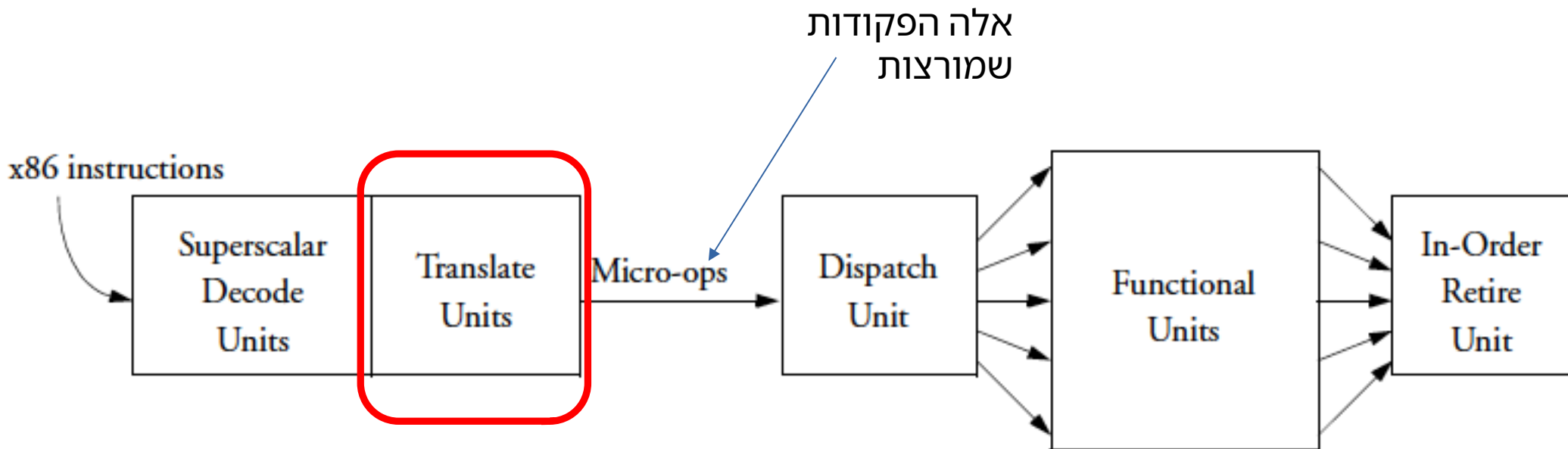


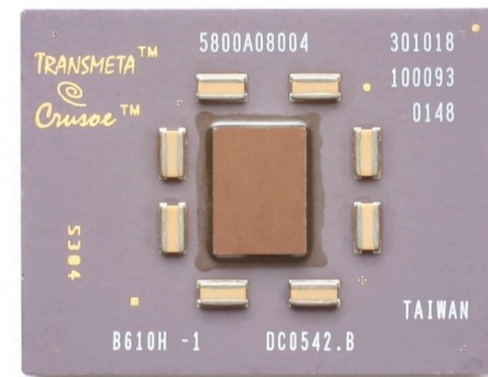
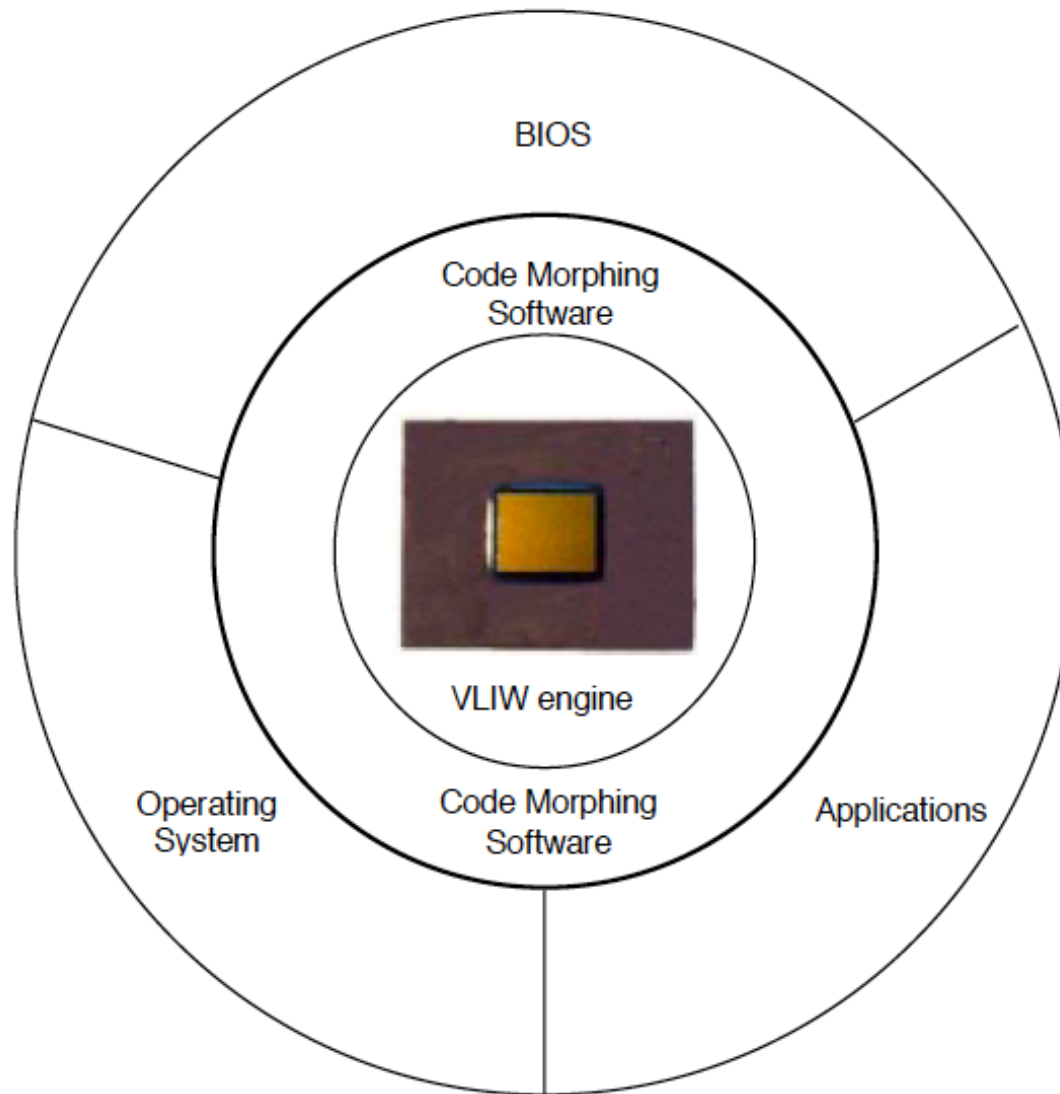
Figure 2. Conventional superscalar out-of-order CPUs use hardware to create and dispatch micro-ops that can execute in parallel.

קריאה מומלצת

Klaiber, "The Technology Behind Crusoe Processors," Transmeta White Paper 2000.

<https://web.stanford.edu/class/cs343/resources/crusoe.pdf> : גיא

# Software-Based Translation



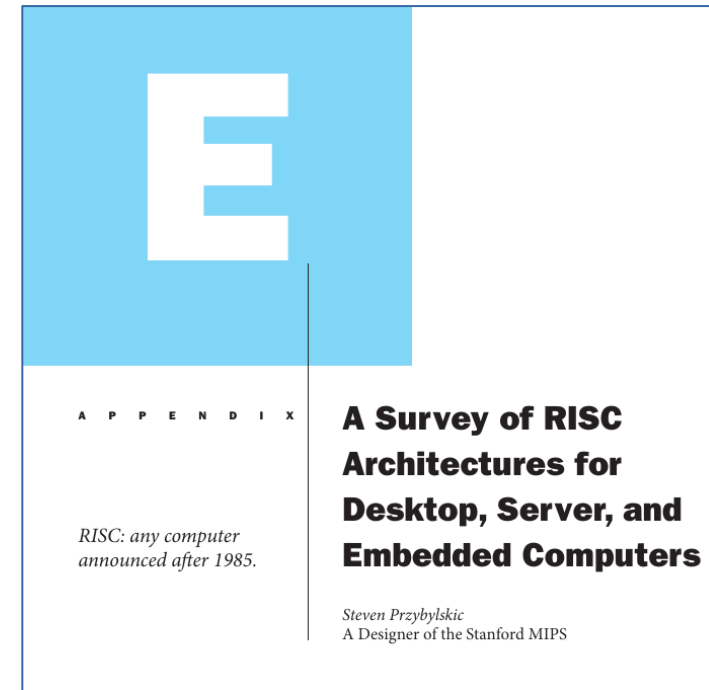
Dynamic  
compiler

Figure 5. The Code Morphing software mediates between x86 software and the Crusoe processor.

Klaiber, "[The Technology Behind Crusoe Processors](#)," Transmeta White Paper 2000.

# לקריאה נוספת

H&P CO&D Appendix E: A Survey of RISC Architectures for Desktop, Server and Embedded Computers.



Another Approach to Instruction Set Architecture—VAX

<https://minnie.tuhs.org/CompArch/Resources/webext3.pdf>



תלמידים: לעיין לאחר השיעור ביתר השקפים

עבורי: עד כאן מצג זו

# יתרונות וחסרונות Instruction Length ISA-level Tradeoffs

- **Fixed length:** Length of all instructions the same
  - + Easier to decode single instruction in hardware
  - + Easier to decode multiple instructions concurrently
  - Wasted bits in instructions (Why is this bad?)
  - Harder-to-extend ISA (how to add new instructions?)
- **Variable length:** Length of instructions different (determined by opcode and sub-opcode)
  - + Compact encoding (Why is this good?)
    - Intel 432: Huffman encoding (sort of). 6 to 321 bit instructions. **ראה שקף הבא - How?**
  - More logic to decode a single instruction
  - Harder to decode multiple instructions concurrently
- **Tradeoffs**
  - Code size (memory space, bandwidth, latency) vs. hardware complexity
  - ISA extensibility and expressiveness vs. hardware complexity
  - Performance? Smaller code vs. ease of decode





GDP = General Data Processor

**iAPX 432 GDP**

**Program Organization**

### **INSTRUCTION OBJECTS**

The GDP represents a procedure as one or more instruction objects (see Figure 2-1). When a procedure (an instruction object) is called, the GDP requires certain information for the context object that represents the call. This information is contained in an Instruction Object Header in the first eight bytes of the instruction object's data part. The remainder of the data part can contain instructions.

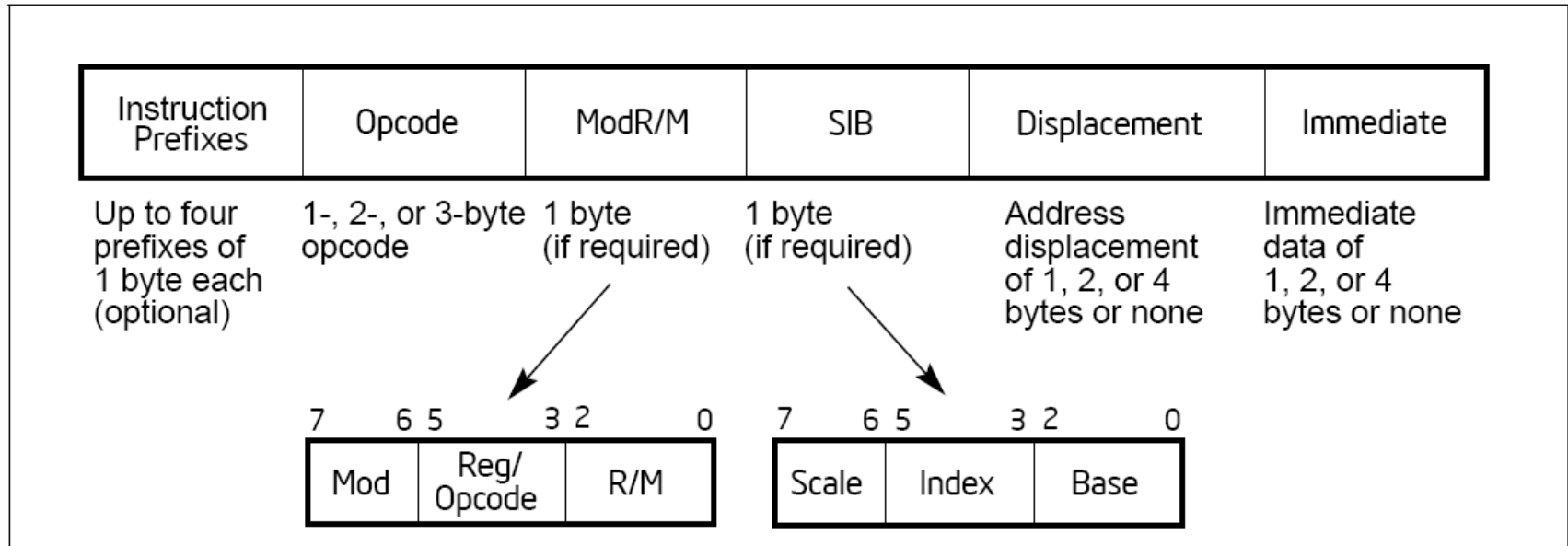
GDP instructions are not aligned on byte or word boundaries and are varying-length sequences of bits. Instruction fields are frequency encoded (Huffman encoded) so that more frequent operation codes, formats, classes, and addressing modes are encoded in fewer bits.

# ISA-level Tradeoffs: Uniform Decode

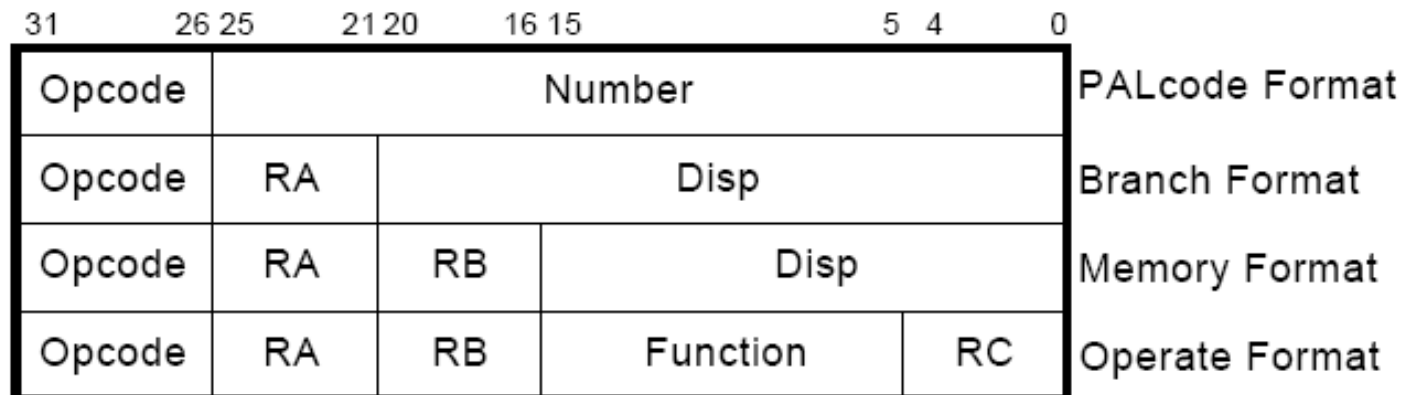
- **Uniform decode:** Same bits in each instruction correspond to the same meaning
  - Opcode is always in the same location
  - Ditto operand specifiers, immediate values, ...
  - Many “RISC” ISAs: Alpha, RISC-V, MIPS, SPARC
  - + Easier decode, simpler hardware
  - + Enables parallelism: generate target address before knowing the instruction is a branch
  - Restricts instruction format (fewer instructions?) or wastes space
  
- **Non-uniform decode**
  - E.g., opcode can be the 1st-7th byte in x86
  - + More compact and powerful instruction format
  - More complex decode logic

# x86 vs. Alpha Instruction Formats

## ■ x86:

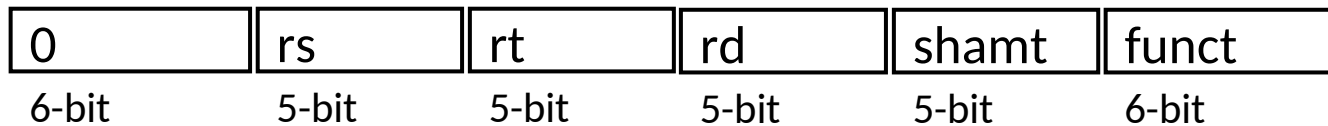


## ■ Alpha:



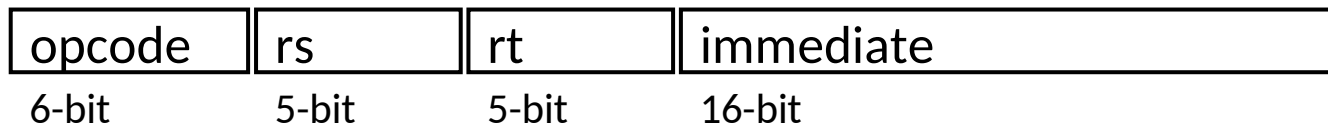
# MIPS Instruction Format

- R-type, 3 register operands



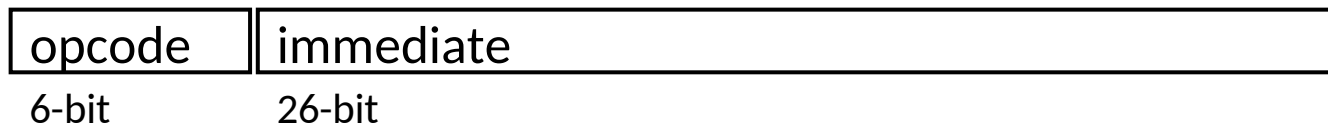
R-type

- I-type, 2 register operands and 16-bit immediate operand



I-type

- J-type, 26-bit immediate operand



J-type

- Simple Decoding

- 4 bytes per instruction, regardless of format
- must be 4-byte aligned (2 lsb of PC must be 2b'00)
- format and fields easy to extract in hardware

# RISC-V

יילמד בנפרד

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0  
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

|      |   |   |   |        |                      |   |   |   |      |     |   |               |    |   |           |     |   |        |    |   |     |        |    |                                   |                                                 |                                                |                                       |
|------|---|---|---|--------|----------------------|---|---|---|------|-----|---|---------------|----|---|-----------|-----|---|--------|----|---|-----|--------|----|-----------------------------------|-------------------------------------------------|------------------------------------------------|---------------------------------------|
| Cond | 0 | 0 | 1 | Opcode |                      |   |   | S | Rn   |     |   | Rd            |    |   | Operand 2 |     |   |        |    |   |     |        |    |                                   |                                                 |                                                | <i>Data Processing / PSR Transfer</i> |
| Cond | 0 | 0 | 0 | 0      | 0                    | 0 | A | S | Rd   |     |   | Rn            |    |   | Rs        |     |   | 1      | 0  | 0 | 1   | Rm     |    |                                   | <i>Multiply</i>                                 |                                                |                                       |
| Cond | 0 | 0 | 0 | 0      | 1                    | U | A | S | RdHi |     |   | RdLo          |    |   | Rn        |     |   | 1      | 0  | 0 | 1   | Rm     |    |                                   | <i>Multiply Long</i>                            |                                                |                                       |
| Cond | 0 | 0 | 0 | 1      | 0                    | B | 0 | 0 | Rn   |     |   | Rd            |    |   | 0         | 0   | 0 | 0      | 1  | 0 | 0   | 1      | Rm |                                   |                                                 | <i>Single Data Swap</i>                        |                                       |
| Cond | 0 | 0 | 0 | 1      | 0                    | 0 | 1 | 0 | 1    | 1   | 1 | 1             | 1  | 1 | 1         | 1   | 1 | 1      | 1  | 0 | 0   | 0      | 1  | Rn                                |                                                 |                                                | <i>Branch and Exchange</i>            |
| Cond | 0 | 0 | 0 | P      | U                    | 0 | W | L | Rn   |     |   | Rd            |    |   | 0         | 0   | 0 | 0      | 1  | S | H   | 1      | Rm |                                   |                                                 | <i>Halfword Data Transfer: register offset</i> |                                       |
| Cond | 0 | 0 | 0 | P      | U                    | 1 | W | L | Rn   |     |   | Rd            |    |   | Offset    |     |   | 1      | S  | H | 1   | Offset |    |                                   | <i>Halfword Data Transfer: immediate offset</i> |                                                |                                       |
| Cond | 0 | 1 | 1 | P      | U                    | B | W | L | Rn   |     |   | Rd            |    |   | Offset    |     |   |        |    |   |     |        |    |                                   |                                                 |                                                | <i>Single Data Transfer</i>           |
| Cond | 0 | 1 | 1 |        |                      |   |   |   |      |     |   |               |    |   |           |     |   |        |    | 1 |     |        |    |                                   |                                                 | <i>Undefined</i>                               |                                       |
| Cond | 1 | 0 | 0 | P      | U                    | S | W | L | Rn   |     |   | Register List |    |   |           |     |   |        |    |   |     |        |    |                                   |                                                 |                                                | <i>Block Data Transfer</i>            |
| Cond | 1 | 0 | 1 | L      | Offset               |   |   |   |      |     |   |               |    |   |           |     |   |        |    |   |     |        |    |                                   |                                                 | <i>Branch</i>                                  |                                       |
| Cond | 1 | 1 | 0 | P      | U                    | N | W | L | Rn   |     |   | CRd           |    |   | CP#       |     |   | Offset |    |   |     |        |    |                                   | <i>Coprocessor Data Transfer</i>                |                                                |                                       |
| Cond | 1 | 1 | 1 | 0      | CP Opc               |   |   |   | CRn  |     |   | CRd           |    |   | CP#       |     |   | CP     |    | 0 | CRm |        |    | <i>Coprocessor Data Operation</i> |                                                 |                                                |                                       |
| Cond | 1 | 1 | 1 | 0      | CP Opc               |   |   |   | L    | CRn |   |               | Rd |   |           | CP# |   |        | CP |   | 1   | CRm    |    |                                   | <i>Coprocessor Register Transfer</i>            |                                                |                                       |
| Cond | 1 | 1 | 1 | 1      | Ignored by processor |   |   |   |      |     |   |               |    |   |           |     |   |        |    |   |     |        |    |                                   |                                                 |                                                | <i>Software Interrupt</i>             |

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0  
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

Figure 4-1: ARM instruction set formats

# A Note on Length and Uniformity

- Uniform decode usually goes with fixed length
- In a variable length ISA, uniform decode can be a property of instructions of the same length
  - It is hard to think of it as a property of instructions of different lengths

# A Note on RISC vs. CISC

- Usually, ...
- RISC
  - Simple instructions
  - Fixed length
  - Uniform decode
  - Few addressing modes
- CISC
  - Complex instructions
  - Variable length
  - Non-uniform decode
  - Many addressing modes



# השוואה בין RISC ל-CISC



| Characteristic                      | CISC       |           | RISC       |
|-------------------------------------|------------|-----------|------------|
|                                     | VAX 11/780 | Intel 486 | MIPS R4000 |
| Number of instructions              | 303        | 235       | 94         |
| Addressing modes                    | 22         | 11        | 1          |
| Instructions size (bytes)           | 2–57       | 1–12      | 4          |
| Number of general-purpose registers | 16         | 8         | 32         |

Credit: Sivarama P. Dandamudi, "Introduction to Assembly Language Programming For Pentium and RISC Processors"

# RISC vs. CISC

| RISC                                                                   | CISC                                                                    |
|------------------------------------------------------------------------|-------------------------------------------------------------------------|
| Multiple register sets, often consisting of more than 256 registers    | Single register set, typically 6 to 16 registers total                  |
| Three register operands allowed per instruction (e.g., add R1, R2, R3) | One or two register operands allowed per instruction (e.g., add R1, R2) |
| Parameter passing through efficient on-chip register windows           | Parameter passing through inefficient off-chip memory                   |
| Single-cycle instructions (except for load and store)                  | Multiple-cycle instructions                                             |
| Hardwired control                                                      | Microprogrammed control                                                 |
| Highly pipelined                                                       | Less pipelined                                                          |
| Simple instructions that are few in number                             | Many complex instructions                                               |
| Fixed-length instructions                                              | Variable-length instructions                                            |
| Complexity in compiler                                                 | Complexity in microcode                                                 |
| Only load and store instructions can access memory                     | Many instructions can access memory                                     |
| Few addressing modes                                                   | Many addressing modes                                                   |

Credit:

“The Essentials of Computer Organization and Architecture”,  
5<sup>th</sup> edition by Linda Null

# ISA-level Tradeoffs: Number of Registers

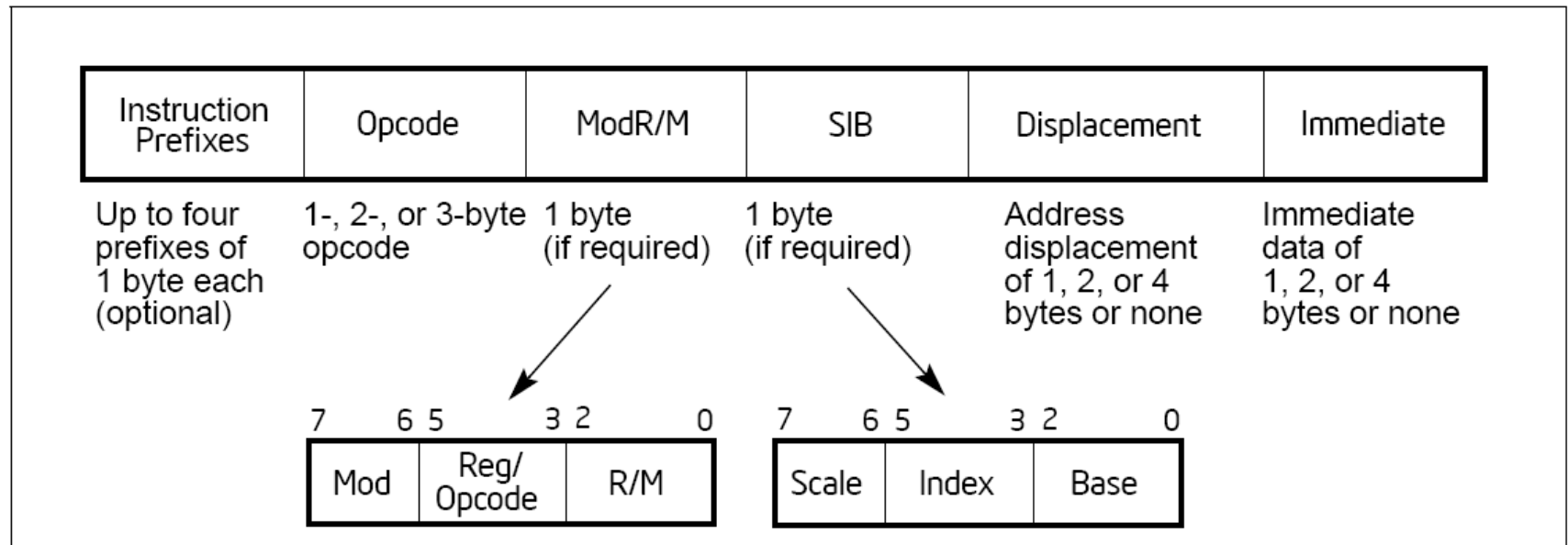
- Affects:
  - Number of bits used for encoding register address
  - Number of values kept in fast storage (register file)
  - (uarch) Size, access time, power consumption of register file
- Large number of registers:
  - + Enables better register allocation (and optimizations) by compiler
    - fewer saves/restores
  - Larger instruction size
  - Larger register file size

# ISA-level Tradeoffs: Addressing Modes

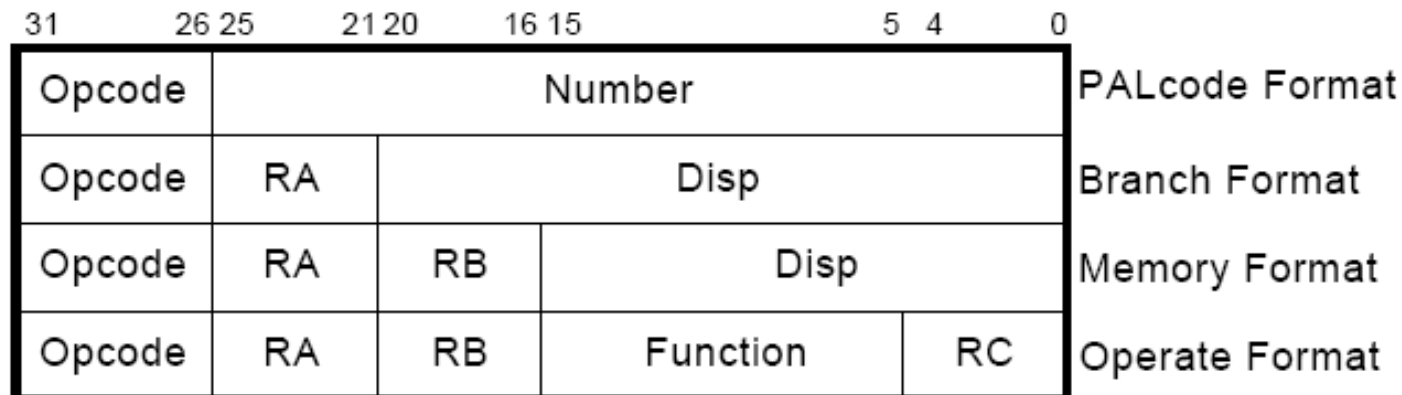
- Addressing mode specifies how to obtain an operand of an instruction
  - Register
  - Immediate
  - Memory (displacement, register indirect, indexed, absolute, memory indirect, autoincrement, autodecrement, ...)
- More modes:
  - + help better support programming constructs (arrays, pointer-based accesses)
  - make it harder for the architect to design
  - too many choices for the compiler?
    - Many ways to do the same thing complicates compiler design
    - Wulf, "*Compilers and Computer Architecture*," *IEEE Computer* 1981

# x86 vs. Alpha Instruction Formats

## ■ x86:



## ■ Alpha:



# x86

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

| r8(/r)<br>r16(/r)<br>r32(/r)<br>mm(/r)<br>xmm(/r)<br>(in decimal) /digit (Opcode)<br>(in binary) REG =                                                                       |     |                                                      | AL<br>AX<br>EAX<br>MM0<br>XMM0<br>0<br>000   | CL<br>CX<br>ECX<br>MM1<br>XMM1<br>1<br>001   | DL<br>DX<br>EDX<br>MM2<br>XMM2<br>2<br>010   | BL<br>BX<br>EBX<br>MM3<br>XMM3<br>3<br>011   | AH<br>SP<br>ESP<br>MM4<br>XMM4<br>4<br>100   | CH<br>BP<br>EBP<br>MM5<br>XMM5<br>5<br>101   | DH<br>SI<br>ESI<br>MM6<br>XMM6<br>6<br>110   | BH<br>DI<br>EDI<br>MM7<br>XMM7<br>7<br>111   |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|------------------------------------------------------|----------------------------------------------|----------------------------------------------|----------------------------------------------|----------------------------------------------|----------------------------------------------|----------------------------------------------|----------------------------------------------|----------------------------------------------|
| Effective Address                                                                                                                                                            | Mod | R/M                                                  | Value of ModR/M Byte (in Hexadecimal)        |                                              |                                              |                                              |                                              |                                              |                                              |                                              |
| [EAX]<br>[ECX]<br>[EDX]<br>[EBX]<br>[--][--] <sup>1</sup><br>disp32 <sup>2</sup><br>[ESI]<br>[EDI]                                                                           | 00  | 000<br>001<br>010<br>011<br>100<br>101<br>110<br>111 | 00<br>01<br>02<br>03<br>04<br>05<br>06<br>07 | 08<br>09<br>0A<br>0B<br>0C<br>0D<br>0E<br>0F | 10<br>11<br>12<br>13<br>14<br>15<br>16<br>17 | 18<br>19<br>1A<br>1B<br>1C<br>1D<br>1E<br>1F | 20<br>21<br>22<br>23<br>24<br>25<br>26<br>27 | 28<br>29<br>2A<br>2B<br>2C<br>2D<br>2E<br>2F | 30<br>31<br>32<br>33<br>34<br>35<br>36<br>37 | 38<br>39<br>3A<br>3B<br>3C<br>3D<br>3E<br>3F |
| [EAX]+disp8 <sup>3</sup><br>[ECX]+disp8<br>[EDX]+disp8<br>[EBX]+disp8<br>[--][--]+disp8<br>[EBP]+disp8<br>[ESI]+disp8<br>[EDI]+disp8                                         | 01  | 000<br>001<br>010<br>011<br>100<br>101<br>110<br>111 | 40<br>41<br>42<br>43<br>44<br>45<br>46<br>47 | 48<br>49<br>4A<br>4B<br>4C<br>4D<br>4E<br>4F | 50<br>51<br>52<br>53<br>54<br>55<br>56<br>57 | 58<br>59<br>5A<br>5B<br>5C<br>5D<br>5E<br>5F | 60<br>61<br>62<br>63<br>64<br>65<br>66<br>67 | 68<br>69<br>6A<br>6B<br>6C<br>6D<br>6E<br>6F | 70<br>71<br>72<br>73<br>74<br>75<br>76<br>77 | 78<br>79<br>7A<br>7B<br>7C<br>7D<br>7E<br>7F |
| [EAX]+disp32<br>[ECX]+disp32<br>[EDX]+disp32<br>[EBX]+disp32<br>[--][--]+disp32<br>[EBP]+disp32<br>[ESI]+disp32<br>[EDI]+disp32                                              | 10  | 000<br>001<br>010<br>011<br>100<br>101<br>110<br>111 | 80<br>81<br>82<br>83<br>84<br>85<br>86<br>87 | 88<br>89<br>8A<br>8B<br>8C<br>8D<br>8E<br>8F | 90<br>91<br>92<br>93<br>94<br>95<br>96<br>97 | 98<br>99<br>9A<br>9B<br>9C<br>9D<br>9E<br>9F | A0<br>A1<br>A2<br>A3<br>A4<br>A5<br>A6<br>A7 | A8<br>A9<br>AA<br>AB<br>AC<br>AD<br>AE<br>AF | B0<br>B1<br>B2<br>B3<br>B4<br>B5<br>B6<br>B7 | B8<br>B9<br>BA<br>BB<br>BC<br>BD<br>BE<br>BF |
| EAX/AX/AL/MM0/XMM0<br>ECX/CX/CL/MM1/XMM1<br>EDX/DX/DL/MM2/XMM2<br>EBX/BX/BL/MM3/XMM3<br>ESP/SP/AH/MM4/XMM4<br>EBP/BP/CH/MM5/XMM5<br>ESI/SI/DH/MM6/XMM6<br>EDI/DI/BH/MM7/XMM7 | 11  | 000<br>001<br>010<br>011<br>100<br>101<br>110<br>111 | C0<br>C1<br>C2<br>C3<br>C4<br>C5<br>C6<br>C7 | C8<br>C9<br>CA<br>CB<br>CC<br>CD<br>CE<br>CF | D0<br>D1<br>D2<br>D3<br>D4<br>D5<br>D6<br>D7 | D8<br>D9<br>DA<br>DB<br>DC<br>DD<br>DE<br>DF | E0<br>E1<br>E2<br>E3<br>E4<br>E5<br>E6<br>E7 | E8<br>E9<br>EA<br>EB<br>EC<br>ED<br>EE<br>EF | F0<br>F1<br>F2<br>F3<br>F4<br>F5<br>F6<br>F7 | F8<br>F9<br>FA<br>FB<br>FC<br>FD<br>FE<br>FF |

register

indirect

absolute

register +  
displacement

register

## NOTES:

1. The [--][--] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

SIB byte → (scale \* index) + base

Table 2-3 is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table.

# x86

Table 2-3. 32-Bit Addressing Forms with the SIB Byte

| r32<br>(in decimal) Base =<br>(in binary) Base =                                  |    |                                                      | EAX<br>0<br>000                              | ECX<br>1<br>001                              | EDX<br>2<br>010                              | EBX<br>3<br>011                              | ESP<br>4<br>100                              | [*]<br>5<br>101                              | ESI<br>6<br>110                              | EDI<br>7<br>111                              |
|-----------------------------------------------------------------------------------|----|------------------------------------------------------|----------------------------------------------|----------------------------------------------|----------------------------------------------|----------------------------------------------|----------------------------------------------|----------------------------------------------|----------------------------------------------|----------------------------------------------|
| Scaled Index                                                                      | SS | Index                                                | Value of SIB Byte (in Hexadecimal)           |                                              |                                              |                                              |                                              |                                              |                                              |                                              |
| [EAX]<br>[ECX]<br>[EDX]<br>[EBX]<br>none<br>[EBP]<br>[ESI]<br>[EDI]               | 00 | 000<br>001<br>010<br>011<br>100<br>101<br>110<br>111 | 00<br>08<br>10<br>18<br>20<br>28<br>30<br>38 | 01<br>09<br>11<br>19<br>21<br>29<br>31<br>39 | 02<br>0A<br>12<br>1A<br>22<br>2A<br>32<br>3A | 03<br>0B<br>13<br>1B<br>23<br>2B<br>33<br>3B | 04<br>0C<br>14<br>1C<br>24<br>2C<br>34<br>3C | 05<br>0D<br>15<br>1D<br>25<br>2D<br>35<br>3D | 06<br>0E<br>16<br>1E<br>26<br>2E<br>36<br>3E | 07<br>0F<br>17<br>1F<br>27<br>2F<br>37<br>3F |
| [EAX*2]<br>[ECX*2]<br>[EDX*2]<br>[EBX*2]<br>none<br>[EBP*2]<br>[ESI*2]<br>[EDI*2] | 01 | 000<br>001<br>010<br>011<br>100<br>101<br>110<br>111 | 40<br>48<br>50<br>58<br>60<br>68<br>70<br>78 | 41<br>49<br>51<br>59<br>61<br>69<br>71<br>79 | 42<br>4A<br>52<br>5A<br>62<br>6A<br>72<br>7A | 43<br>4B<br>53<br>5B<br>63<br>6B<br>73<br>7B | 44<br>4C<br>54<br>5C<br>64<br>6C<br>74<br>7C | 45<br>4D<br>55<br>5D<br>65<br>6D<br>75<br>7D | 46<br>4E<br>56<br>5E<br>66<br>6E<br>76<br>7E | 47<br>4F<br>57<br>5F<br>67<br>6F<br>77<br>7F |
| [EAX*4]<br>[ECX*4]<br>[EDX*4]<br>[EBX*4]<br>none<br>[EBP*4]<br>[ESI*4]<br>[EDI*4] | 10 | 000<br>001<br>010<br>011<br>100<br>101<br>110<br>111 | 80<br>88<br>90<br>98<br>A0<br>A8<br>B0<br>B8 | 81<br>89<br>91<br>99<br>A1<br>A9<br>B1<br>B9 | 82<br>8A<br>92<br>9A<br>A2<br>AA<br>B2<br>BA | 83<br>8B<br>93<br>9B<br>A3<br>AB<br>B3<br>BB | 84<br>8C<br>94<br>9C<br>A4<br>AC<br>B4<br>BC | 85<br>8D<br>95<br>9D<br>A5<br>AD<br>B5<br>BD | 86<br>8E<br>96<br>9E<br>A6<br>AE<br>B6<br>BE | 87<br>8F<br>97<br>9F<br>A7<br>AF<br>B7<br>BF |
| [EAX*8]<br>[ECX*8]<br>[EDX*8]<br>[EBX*8]<br>none<br>[EBP*8]<br>[ESI*8]<br>[EDI*8] | 11 | 000<br>001<br>010<br>011<br>100<br>101<br>110<br>111 | C0<br>C8<br>D0<br>D8<br>E0<br>E8<br>F0<br>F8 | C1<br>C9<br>D1<br>D9<br>E1<br>E9<br>F1<br>F9 | C2<br>CA<br>D2<br>DA<br>E2<br>EA<br>F2<br>FA | C3<br>CB<br>D3<br>DB<br>E3<br>EB<br>F3<br>FB | C4<br>CC<br>D4<br>DC<br>E4<br>EC<br>F4<br>FC | C5<br>CD<br>D5<br>DD<br>E5<br>ED<br>F5<br>FD | C6<br>CE<br>D6<br>DE<br>E6<br>EE<br>F6<br>FE | C7<br>CF<br>D7<br>DF<br>E7<br>EF<br>F7<br>FF |

indexed  
(base +  
index)

scaled  
(base +  
index\*4)

## NOTES:

- The [\*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [\*] means disp8 or disp32 + [EBP]. This provides the following address modes:

MOD bits    Effective Address

- |    |                                 |
|----|---------------------------------|
| 00 | [scaled index] + disp32         |
| 01 | [scaled index] + disp8 + [EBP]  |
| 10 | [scaled index] + disp32 + [EBP] |

# X86 SIB-D Addressing Mode

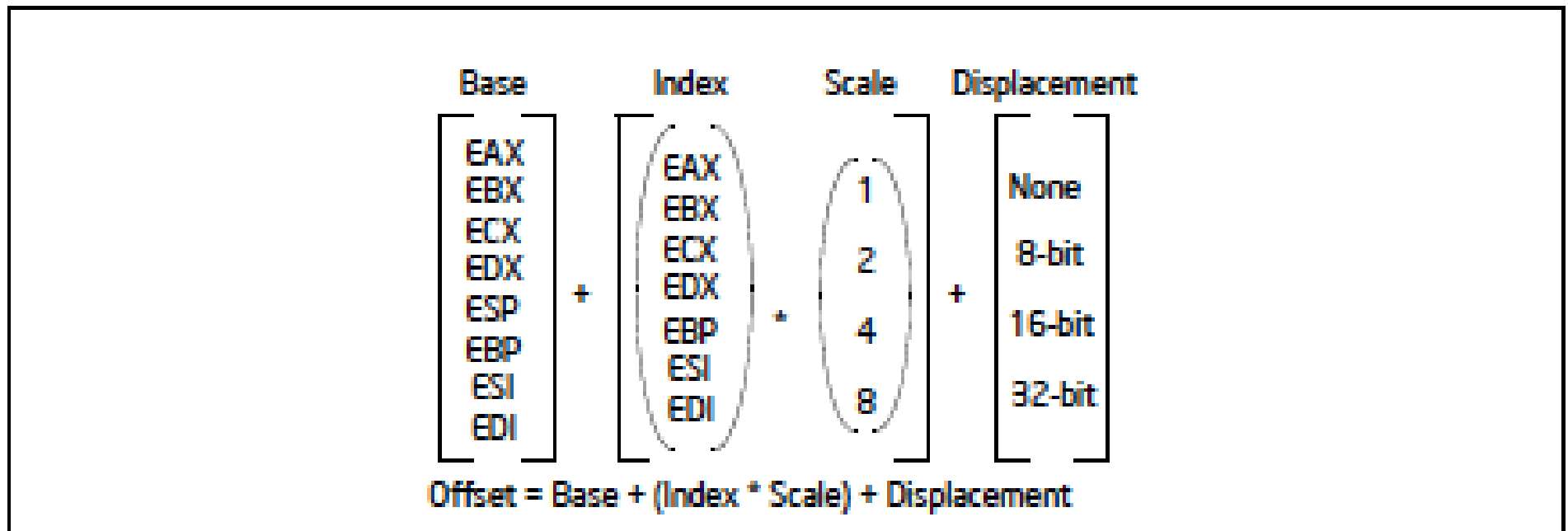


Figure 3-11. Offset (or Effective Address) Computation

x86 Manual Vol. 1, page 3-22 -- see course resources on website  
Also, see Section 3.7.3 and 3.7.5



# X86 Manual: Suggested Uses of Addressing Modes

The following addressing modes suggest uses for common combinations of address components.

- **Displacement** — A displacement alone represents a direct (uncomputed) offset to the operand. Because the displacement is encoded in the instruction, this form of an address is sometimes called an absolute or static address. It is commonly used to access a statically allocated scalar operand.
- **Base** — A base alone represents an indirect offset to the operand. Since the value in the base register can change, it can be used for dynamic storage of variables and data structures.
- **Base + Displacement** — A base register and a displacement can be used together for two distinct purposes:
  - As an index into an array when the element size is not 2, 4, or 8 bytes—The displacement component encodes the static offset to the beginning of the array. The base register holds the results of a calculation to determine the offset to a specific element within the array.
  - To access a field of a record: the base register holds the address of the beginning of the record, while the displacement is a static offset to the field.

An important special case of this combination is access to parameters in a procedure activation record. A procedure activation record is the stack frame created when a procedure is entered. Here, the EBP register is the best choice for the base register, because it automatically selects the stack segment. This is a compact encoding for this common function.

x86 Manual Vol. 1, page 3-22 -- see course resources on website

Also, see Section 3.7.3 and 3.7.5

# X86 Manual: Suggested Uses of Addressing Modes

- **(Index \* Scale) + Displacement** — This address mode offers an efficient way to index into a static array when the element size is 2, 4, or 8 bytes. The displacement locates the beginning of the array, the index register holds the subscript of the desired array element, and the processor automatically converts the subscript into an index by applying the scaling factor.
- **Base + Index + Displacement** — Using two registers together supports either a two-dimensional array (the displacement holds the address of the beginning of the array) or one of several instances of an array of records (the displacement is an offset to a field within the record).
- **Base + (Index \* Scale) + Displacement** — Using all the addressing components together allows efficient indexing of a two-dimensional array when the elements of the array are 2, 4, or 8 bytes in size.

x86 Manual Vol. 1, page 3-22 -- see course resources on website  
Also, see Section 3.7.3 and 3.7.5

# Other Example ISA-level Tradeoffs

- Condition codes vs. not
- VLIW vs. single instruction
- Precise vs. imprecise exceptions
- Virtual memory vs. not
- Unaligned access vs. not
- Hardware interlocks vs. software-guaranteed interlocking
- Software vs. hardware managed page fault handling
- Cache coherence (hardware vs. software)
- ...

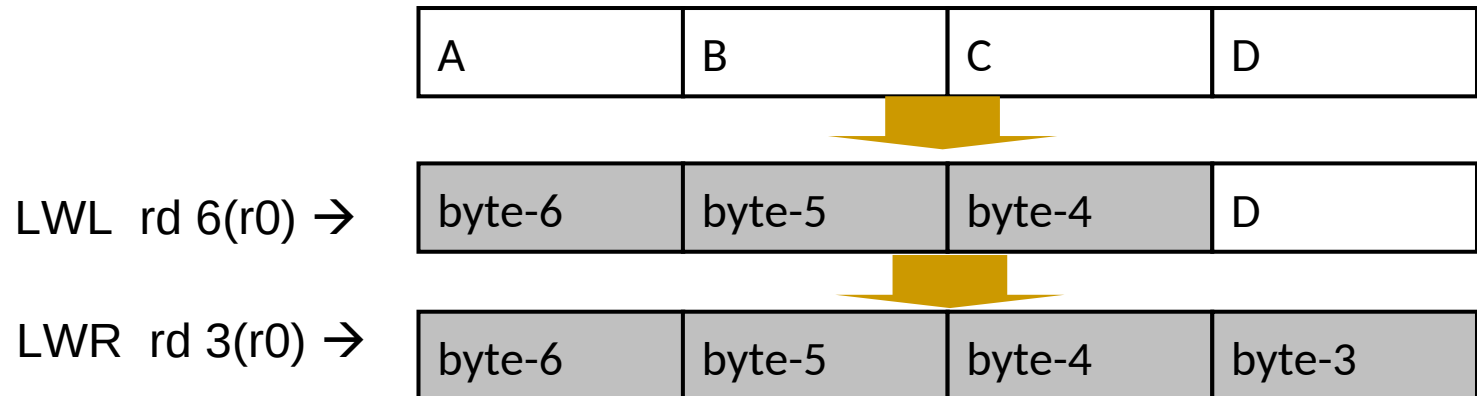
# Back to Programmer vs. (Micro)architect

- Many ISA features designed to aid programmers
- But, complicate the hardware designer's job
- Virtual memory
  - vs. overlay programming
  - Should the programmer be concerned about the size of code blocks fitting physical memory?
- Addressing modes
- Unaligned memory access
  - Compile/programmer needs to align data

# MIPS: Aligned Access

|     |        |        |        |        |     |
|-----|--------|--------|--------|--------|-----|
| MSB | byte-3 | byte-2 | byte-1 | byte-0 | LSB |
|     | byte-7 | byte-6 | byte-5 | byte-4 |     |

- LW/SW alignment restriction: 4-byte word-alignment
  - not designed to fetch memory bytes not within a word boundary
  - not designed to rotate unaligned bytes into registers
- Provide separate opcodes for the “infrequent” case



- LWL/LWR is slower
- Note LWL and LWR still fetch within word boundary

# X86: Unaligned Access

- LD/ST instructions automatically align data that spans a “word” boundary
- Programmer/compiler does not need to worry about where data is stored (whether or not in a word-aligned location)

## 4.1.1 Alignment of Words, Doublewords, Quadwords, and Double Quadwords

Words, doublewords, and quadwords do not need to be aligned in memory on natural boundaries. The natural boundaries for words, double words, and quadwords are even-numbered addresses, addresses evenly divisible by four, and addresses evenly divisible by eight, respectively. However, to improve the performance of programs, data structures (especially stacks) should be aligned on natural boundaries whenever possible. The reason for this is that the processor requires two memory accesses to make an unaligned memory access; aligned accesses require only one memory access. A word or doubleword operand that crosses a 4-byte boundary or a quadword operand that crosses an 8-byte boundary is considered unaligned and requires two separate memory bus cycles for access.

# X86: Unaligned Access

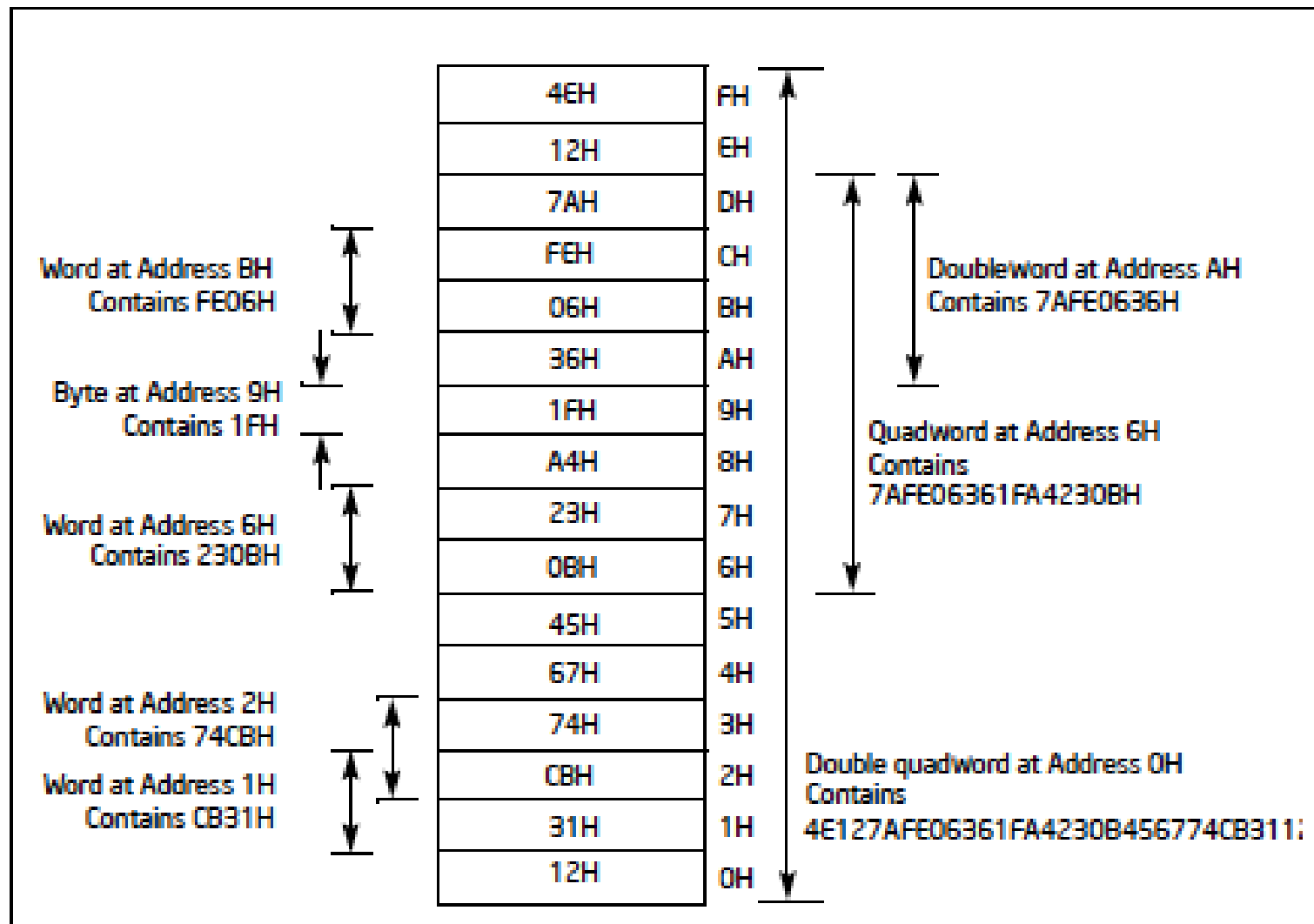
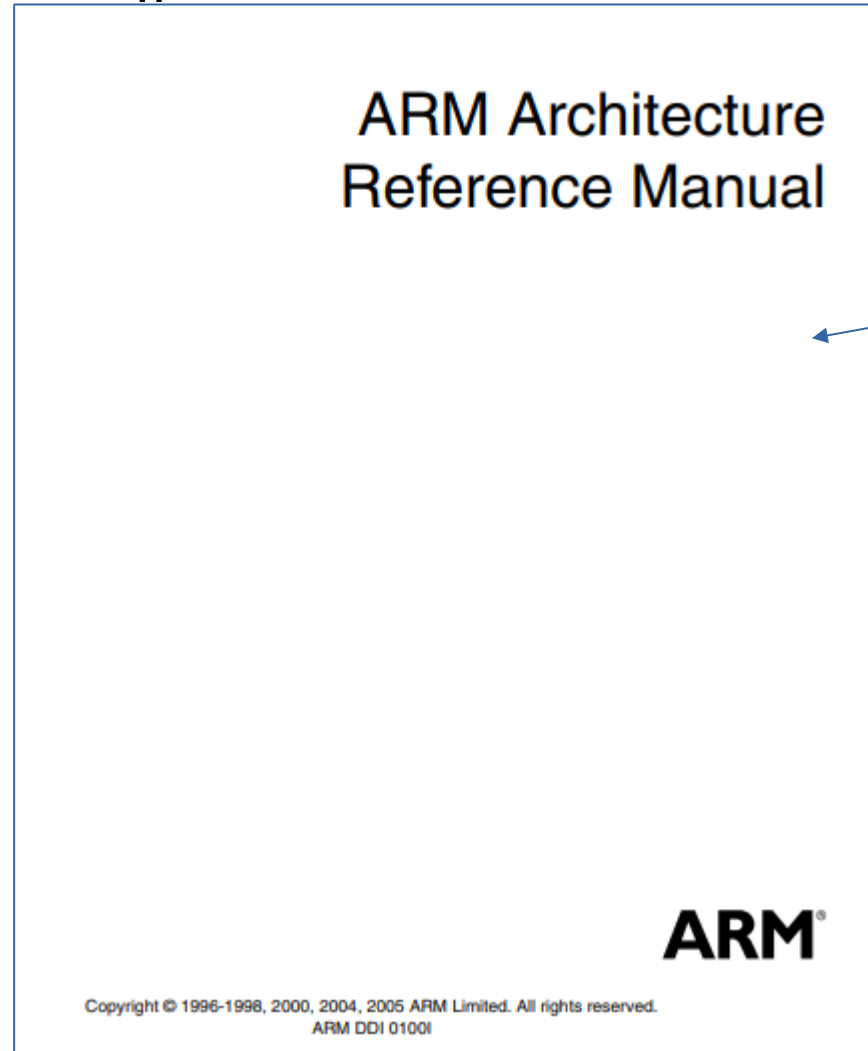


Figure 4-2. Bytes, Words, Doublewords, Quadwords, and Double Quadwords in Memory

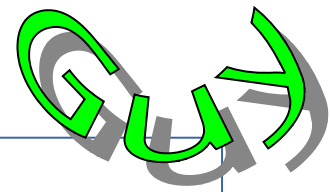
# What About ARM?

- [https://www.scss.tcd.ie/~waldroj/3d1/arm\\_arm.pdf](https://www.scss.tcd.ie/~waldroj/3d1/arm_arm.pdf)
  - Section A2.8 - - ראה גם שקף הבא



1138 עמודים!





## **A2.8 Unaligned access support**

The ARM architecture traditionally expects all memory accesses to be suitably aligned. In particular, the address used for a halfword access should normally be halfword-aligned, the address used for a word access should normally be word-aligned.

Prior to ARMv6, doubleword (LDRD/STRD) accesses to memory, where the address is not doubleword-aligned, are UNPREDICTABLE. Also, data accesses to non-aligned word and halfword data are treated as aligned from the memory interface perspective. That is:

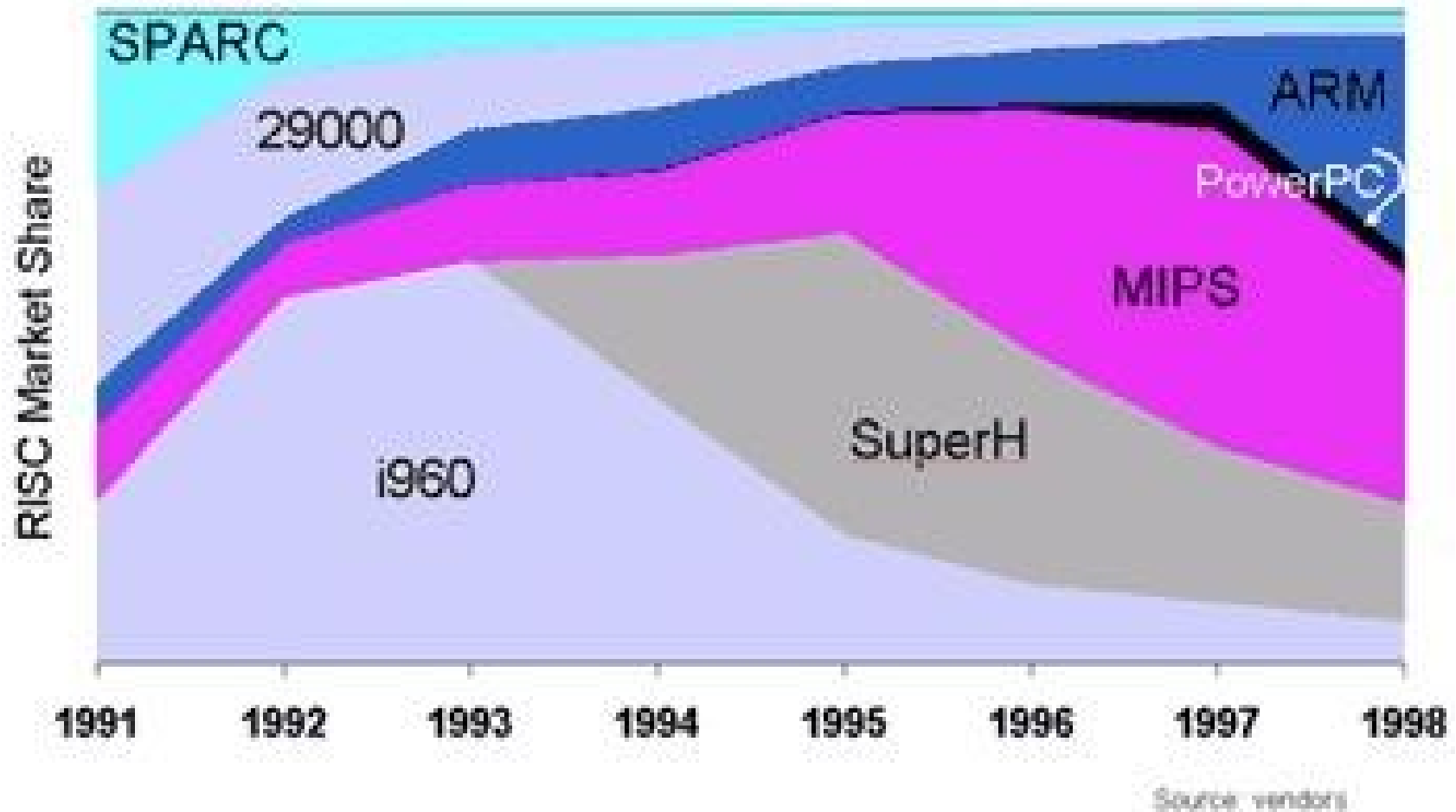
- the address is treated as truncated, with address bits[1:0] treated as zero for word accesses, and address bit[0] treated as zero for halfword accesses.
- load single word ARM instructions are architecturally defined to rotate right the word-aligned data transferred by a non word-aligned address one, two or three bytes depending on the value of the two least significant address bits.
- alignment checking is defined for implementations supporting a System Control coprocessor using the A bit in CP15 register 1. When this bit is set, a Data Abort indicating an alignment fault is reported for unaligned accesses.

ARMv6 introduces unaligned word and halfword load and store data access support. When this is enabled, the processor uses one or more memory accesses to generate the required transfer of adjacent bytes transparently to the programmer, apart from a potential access time penalty where the transaction crosses an IMPLEMENTATION DEFINED cache-line, bus-width or page boundary condition. Doubleword accesses must be word-aligned in this configuration.

# Aligned vs. Unaligned Access

- Pros of having no restrictions on alignment
- Cons of having no restrictions on alignment
- Filling in the above: an exercise for you...

# Market share of embedded RISC processors. From ExtremeTech.



**BGU 361-1-4201**

**Computer Architecture**

**Lecture 2: ISA Tradeoffs (Continued) and  
RISC-V ISA**

**Lecturer: Dr. Guy Tel-Zur**

Based on lectures and slides  
by Prof. Onur Mutlu  
Kevin Chang  
Carnegie Mellon University  
Spring 2015

גיא: שינויים, הרחבות, והתאמות לארכי' RISC-V, אפריל 2026

# ISA-level Tradeoffs: Instruction Length

**Fixed length:** Length of all instructions the same

- + Easier to decode single instruction in hardware
- + Easier to decode multiple instructions concurrently (superscalar processors – next slide)
- -- Wasted bits in instructions (Why is this bad?)
- -- Harder-to-extend ISA (how to add new instructions?)

**Variable length:** Length of instructions different (determined by opcode and sub-opcode)

- + Compact encoding (Why is this good?)
- Intel 432: Huffman encoding (sort of). 6 to 321 bit instructions. How?
- - More logic to decode a single instruction מעבדים סופרסקלריים יותר מורכבים
- - Harder to decode multiple instructions concurrently

## Tradeoffs

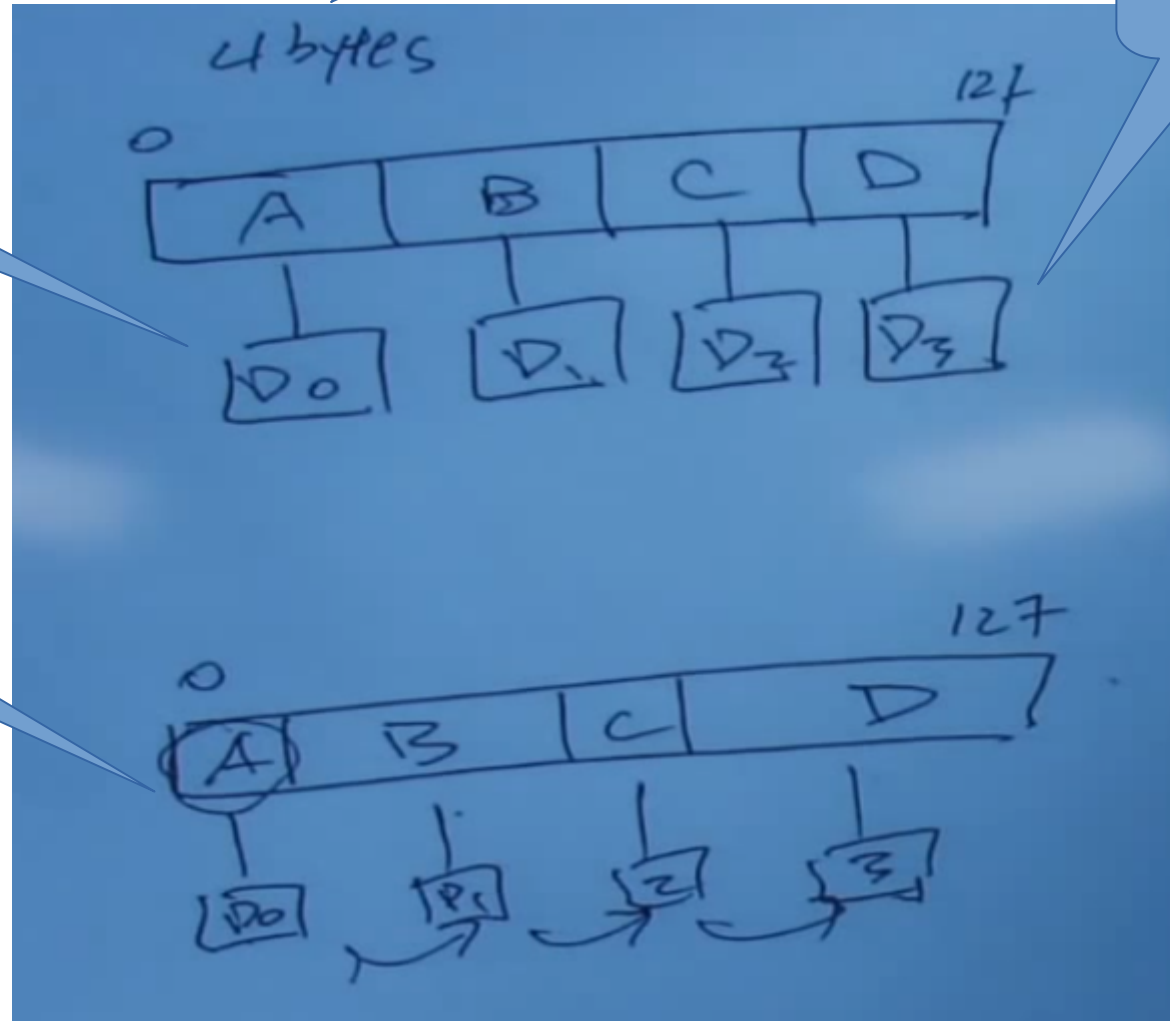
- Code size (memory space, bandwidth, latency) vs. hardware complexity
- ISA extensibility and expressiveness vs. hardware complexity
- Performance? Energy? Smaller code vs. ease of decode
- ביצועים – אם נסתכל על יחס החישוב לתקשורת: אורך משתנה של פקודות מיטיב עם הזכרון ולכן יתכן שיש יתרון לאורך משתנה על פני אורך קבוע.
- מבחינת אנרגיה אין שחור ולבן...

4 הוראות באורך 4 בייטים

4 מפענחים  
העובדים באופן  
בלתי תלוי

עם הוראות בגודל קבוע  
ניתן למקבל את עבודת ה-  
instruction decoders

עם הוראות בגודל משתנה  
יש תלות סריאלית בעבודת ה-  
instruction decoders



סופר-סקאלר

# מתוך התיעוד של iAPX432



GDP = General Data Processor

iAPX 432 GDP

Program Organization

## INSTRUCTION OBJECTS

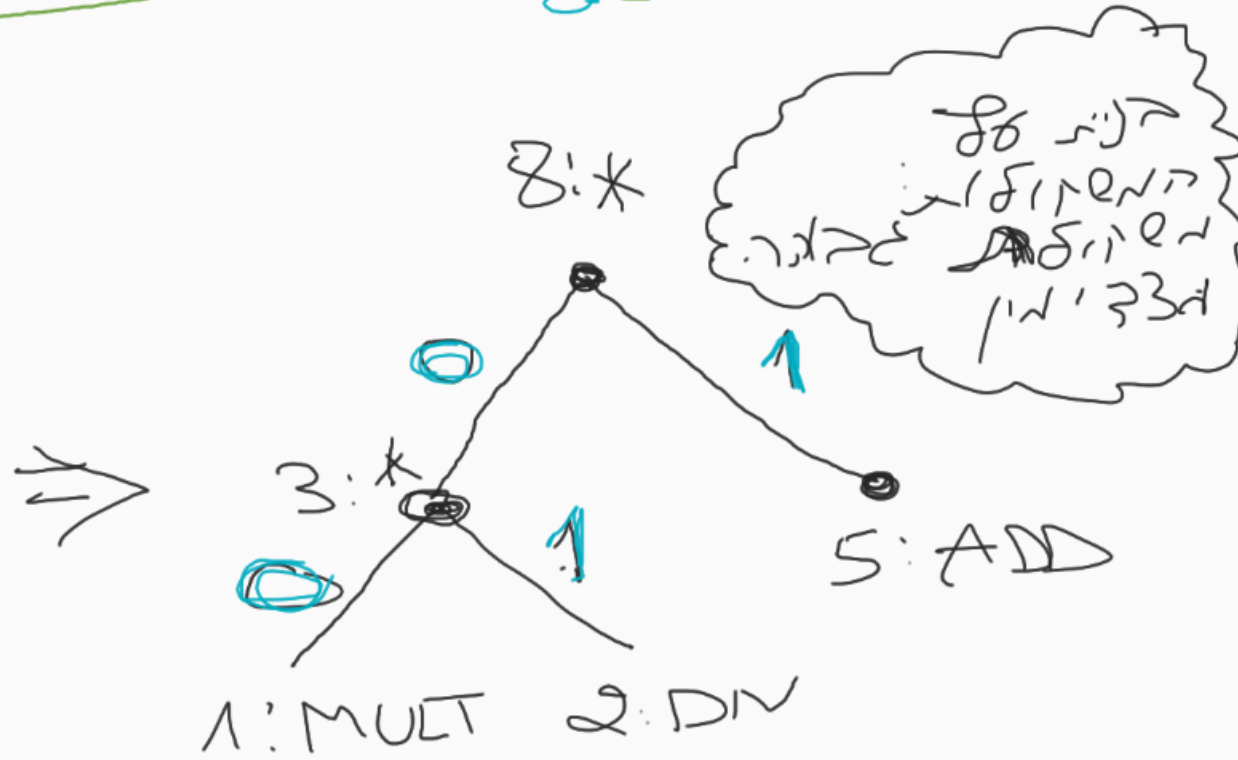
The GDP represents a procedure as one or more instruction objects (see Figure 2-1). When a procedure (an instruction object) is called, the GDP requires certain information for the context object that represents the call. This information is contained in an Instruction Object Header in the first eight bytes of the instruction object's data part. The remainder of the data part can contain instructions.

GDP instructions are not aligned on byte or word boundaries and are varying-length sequences of bits. Instruction fields are frequency encoded (Huffman encoded) so that more frequent operation codes, formats, classes, and addressing modes are encoded in fewer bits.

נעשה פרופיילינג של כל הפקודות ונבדוק את תדירות ההופעה שלהן

# Huffman Encoding

| פעולה | תדירות |
|-------|--------|
| MULT  | 1      |
| DIV   | 2      |
| ADD   | 5      |



| פעולה | קוד בינארי | סך הכל |
|-------|------------|--------|
| ADD   | 1          | 0110   |
| DIV   | 01         | 0101   |
| MULT  | 00         | 0011   |

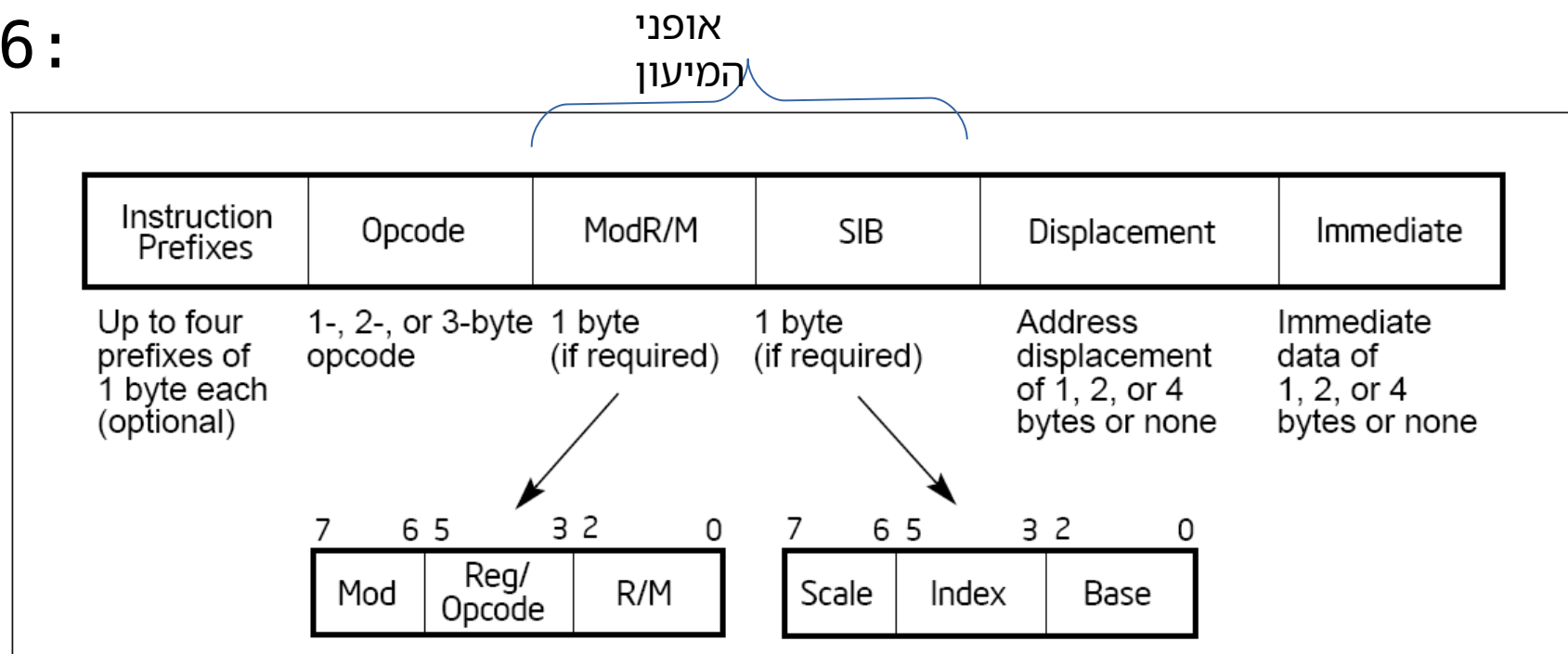


# ISA-level Tradeoffs: Uniform Decode

- **Uniform decode:** Same bits in each instruction correspond to the same meaning
  - Opcode is always in the same location (as in RISC-V)
  - Ditto operand specifiers, immediate values, ...
  - Many “RISC” ISAs: Alpha, MIPS, SPARC
  - + Easier decode, simpler hardware
  - + Enables parallelism: generate target address before knowing the instruction is a branch
  - Restricts instruction format (fewer instructions?) or wastes space
  
- **Non-uniform decode**
  - E.g., opcode can be the 1st-7th byte in x86 – קיים טווח של גדלים וזה מאפשר מבנה יותר קומפקטי
  - + More compact and powerful instruction format
  - More complex decode logic

# x86 vs. Alpha Instruction Formats

## ■ x86:



## ■ Alpha

בזכות המבנה הקבוע מתאפשר מיקבול: בזמן שנעשה decode של הפקודה ניתן בצורה ספקולטיבית להעריך את התוכן של 21 הביטים בחלק השני של הפקודה

|        |    |        |      |          |    |    |   |    |                |                |
|--------|----|--------|------|----------|----|----|---|----|----------------|----------------|
| 31     | 26 | 25     | 21   | 20       | 16 | 15 | 5 | 4  | 0              |                |
| Opcode |    | Number |      |          |    |    |   |    |                | PALcode Format |
| Opcode |    | RA     | Disp |          |    |    |   |    |                | Branch Format  |
| Opcode |    | RA     | RB   | Disp     |    |    |   |    |                | Memory Format  |
| Opcode |    | RA     | RB   | Function |    |    |   | RC | Operate Format |                |

The SIB byte is structured as follows:

[Scale] [Index] [Base]

\* Scale (2 bits): This field specifies a scaling factor that multiplies the value in the index register.

- The possible values are: 00: Scale = 1, 01: Scale = 2, 10: Scale = 4, 11: Scale = 8
- The scaled index register is used to access elements of arrays with different element sizes.

\* Index (3 bits):

- This field specifies an index register (e.g., EAX, ECX, EDX, EBX, ESI, EDI).
- The value in the index register is multiplied by the scale factor.

\* Base (3 bits):

- This field specifies a base register (e.g., EAX, ECX, EDX, EBX, ESI, EDI, ESP, EBP).
- The value in the base register is added to the scaled index value and any displacement to calculate the final memory address.

\* Address Calculation. The final memory address is calculated as follows:

$$\text{Address} = \text{Base} + (\text{Index} * \text{Scale}) + \text{Displacement}$$

- `Base`: The value in the base register (or 0 if no base register).
- `Index`: The value in the index register (or 0 if no index register).
- `Scale`: The scaling factor (1, 2, 4, or 8).
- `Displacement`: An optional signed displacement value (8, 32 bit).

\* Example:

Consider the following assembly instruction:

**mov eax, [ebx + ecx \* 4]**

In this case:

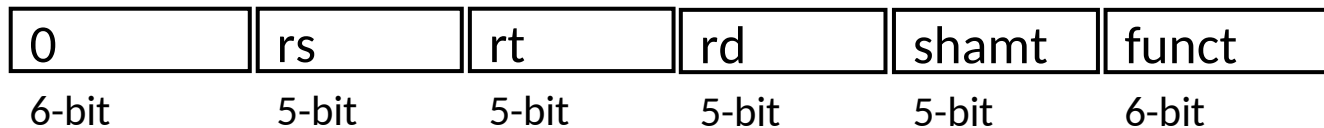
- `ebx` is the base register.
- `ecx` is the index register.
- The scale factor is 4.

The corresponding SIB byte would be:

- Scale: 10 (which is 4)
- Index: The encoding for `ecx`
- Base: The encoding for `ebx`

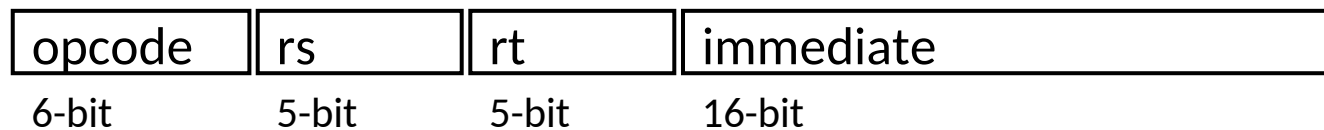
# MIPS Instruction Format

- R-type, 3 register operands



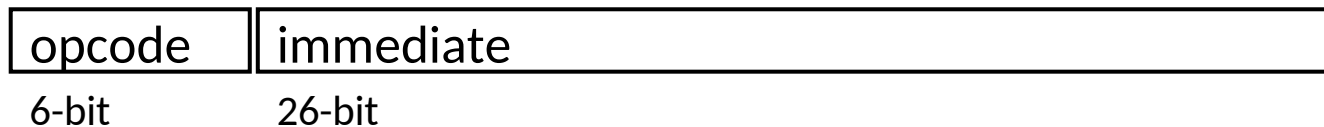
R-type

- I-type, 2 register operands and 16-bit immediate operand



I-type

- J-type, 26-bit immediate operand



J-type

- Simple Decoding

- 4 bytes per instruction, regardless of format
- must be 4-byte aligned (2 lsb of PC must be 2b'00)
- format and fields easy to extract in hardware

# RISC-V Instruction Format

| 31                    | 27 | 26 | 25 | 24  | 20 | 19  | 15 | 14     | 12 | 11          | 7 | 6      | 0 |        |
|-----------------------|----|----|----|-----|----|-----|----|--------|----|-------------|---|--------|---|--------|
| funct7                |    |    |    | rs2 |    | rs1 |    | funct3 |    | rd          |   | opcode |   | R-type |
| imm[11:0]             |    |    |    |     |    | rs1 |    | funct3 |    | rd          |   | opcode |   | I-type |
| imm[11:5]             |    |    |    | rs2 |    | rs1 |    | funct3 |    | imm[4:0]    |   | opcode |   | S-type |
| imm[12 10:5]          |    |    |    | rs2 |    | rs1 |    | funct3 |    | imm[4:1 11] |   | opcode |   | B-type |
| imm[31:12]            |    |    |    |     |    |     |    |        |    | rd          |   | opcode |   | U-type |
| imm[20 10:1 11 19:12] |    |    |    |     |    |     |    |        |    | rd          |   | opcode |   | J-type |

R - Register-register operations

I - Short immediates and loads

S - Stores

B - Conditional branches

U - Long immediates

J - Unconditional jumps

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0  
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

|      |   |   |   |        |                      |   |   |   |      |     |   |               |    |   |           |     |   |        |    |   |     |        |    |    |                                   |                                                 |                                                |                            |
|------|---|---|---|--------|----------------------|---|---|---|------|-----|---|---------------|----|---|-----------|-----|---|--------|----|---|-----|--------|----|----|-----------------------------------|-------------------------------------------------|------------------------------------------------|----------------------------|
| Cond | 0 | 0 | 1 | Opcode |                      |   |   | S | Rn   |     |   | Rd            |    |   | Operand 2 |     |   |        |    |   |     |        |    |    |                                   |                                                 | <i>Data Processing / PSR Transfer</i>          |                            |
| Cond | 0 | 0 | 0 | 0      | 0                    | 0 | A | S | Rd   |     |   | Rn            |    |   | Rs        |     |   | 1      | 0  | 0 | 1   | Rm     |    |    |                                   | <i>Multiply</i>                                 |                                                |                            |
| Cond | 0 | 0 | 0 | 0      | 1                    | U | A | S | RdHi |     |   | RdLo          |    |   | Rn        |     |   | 1      | 0  | 0 | 1   | Rm     |    |    |                                   | <i>Multiply Long</i>                            |                                                |                            |
| Cond | 0 | 0 | 0 | 1      | 0                    | B | 0 | 0 | Rn   |     |   | Rd            |    |   | 0         | 0   | 0 | 0      | 1  | 0 | 0   | 1      | Rm |    |                                   |                                                 | <i>Single Data Swap</i>                        |                            |
| Cond | 0 | 0 | 0 | 1      | 0                    | 0 | 1 | 0 | 1    | 1   | 1 | 1             | 1  | 1 | 1         | 1   | 1 | 1      | 1  | 0 | 0   | 0      | 1  | Rn |                                   |                                                 |                                                | <i>Branch and Exchange</i> |
| Cond | 0 | 0 | 0 | P      | U                    | 0 | W | L | Rn   |     |   | Rd            |    |   | 0         | 0   | 0 | 0      | 1  | S | H   | 1      | Rm |    |                                   |                                                 | <i>Halfword Data Transfer: register offset</i> |                            |
| Cond | 0 | 0 | 0 | P      | U                    | 1 | W | L | Rn   |     |   | Rd            |    |   | Offset    |     |   | 1      | S  | H | 1   | Offset |    |    |                                   | <i>Halfword Data Transfer: immediate offset</i> |                                                |                            |
| Cond | 0 | 1 | 1 | P      | U                    | B | W | L | Rn   |     |   | Rd            |    |   | Offset    |     |   |        |    |   |     |        |    |    |                                   |                                                 | <i>Single Data Transfer</i>                    |                            |
| Cond | 0 | 1 | 1 |        |                      |   |   |   |      |     |   |               |    |   |           |     |   |        |    |   | 1   |        |    |    |                                   | <i>Undefined</i>                                |                                                |                            |
| Cond | 1 | 0 | 0 | P      | U                    | S | W | L | Rn   |     |   | Register List |    |   |           |     |   |        |    |   |     |        |    |    |                                   |                                                 | <i>Block Data Transfer</i>                     |                            |
| Cond | 1 | 0 | 1 | L      | Offset               |   |   |   |      |     |   |               |    |   |           |     |   |        |    |   |     |        |    |    |                                   |                                                 | <i>Branch</i>                                  |                            |
| Cond | 1 | 1 | 0 | P      | U                    | N | W | L | Rn   |     |   | CRd           |    |   | CP#       |     |   | Offset |    |   |     |        |    |    | <i>Coprocessor Data Transfer</i>  |                                                 |                                                |                            |
| Cond | 1 | 1 | 1 | 0      | CP Opc               |   |   |   | CRn  |     |   | CRd           |    |   | CP#       |     |   | CP     |    | 0 | CRm |        |    |    | <i>Coprocessor Data Operation</i> |                                                 |                                                |                            |
| Cond | 1 | 1 | 1 | 0      | CP Opc               |   |   |   | L    | CRn |   |               | Rd |   |           | CP# |   |        | CP |   | 1   | CRm    |    |    |                                   | <i>Coprocessor Register Transfer</i>            |                                                |                            |
| Cond | 1 | 1 | 1 | 1      | Ignored by processor |   |   |   |      |     |   |               |    |   |           |     |   |        |    |   |     |        |    |    |                                   |                                                 | <i>Software Interrupt</i>                      |                            |

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0  
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

Figure 4-1: ARM instruction set formats

# A Note on RISC vs. CISC

- Usually, ...
- RISC
  - Simple instructions
  - Fixed length
  - Uniform decode
  - Few addressing modes
- CISC
  - Complex instructions
  - Variable length
  - Non-uniform decode
  - Many addressing modes



# ISA-level Tradeoffs: Number of Registers

## ■ Affects:

- Number of bits used for encoding register address
- Number of values kept in fast storage (register file)
- (uarch) Size, access time, power consumption of register file

המידע הזה תופס מקום

## ■ Large number of registers:

- + Enables better **register allocation** (and optimizations) by compiler
- + Fewer saves/restores
- Larger instruction size
- Larger register file size (יותר הספק)

השקף הבא מדגים את האתגר ב- register allocation

צוהר - לה' יעזר, בתקוהים ?

st = store  
ld = load

רבי עזריאל מרומניץ

o'd'c'n f'e'n'd, p'u'n'e'n q' y'd e' E'k

. 3^A K \rightarrow J S C W S A p \sim J S \rightarrow O N . i n t

set  $z$  ("spilling")  $\Leftrightarrow$  push  $z$

ld 2 (filling back) pop 2

[illegible]

2022/12/27

Spilling = 110's & 20R"

# ISA-level Tradeoffs: Addressing Modes

- Addressing mode specifies how to obtain an operand of an instruction
  - Register
  - Immediate
  - Memory (displacement, register indirect, indexed, absolute, memory indirect, auto-increment, auto-decrement, ...)
- More addressing modes: ( → smaller semantic gap)
  - + help better support programming constructs (arrays, pointer-based accesses)
  - make it harder for the architect to design
  - too many choices for the compiler?
    - Many ways to do the same thing complicates compiler design
    - Wulf, “*Compilers and Computer Architecture*,” *IEEE Computer* 1981

# הסבר נוסף

## **Immediate** (or literal)

ex: `ADD r1, r2, 1`

means:  $r1 \leftarrow r2 + 1$

comment: used when a constant is needed.

## **Direct**

ex: `ADD r1, r2, (100)`

means:  $r1 \leftarrow r2 + M[100]$

comment: used to access static data; the address of the operand is include in the instruction; space must be provided to accommodate a whole address.

## **Register indirect** (or register deferred)

ex: `ADD r1, r2, (r3)`

means:  $r1 \leftarrow r2 + M[r3]$

comment: the register (r3 in this example) contains the address of a memory location.

## **Memory indirect** (or memory deferred)

ex: `ADD r1, r2, @r3`

means:  $r1 \leftarrow r2 + M[M[r3]]$

comment: used in pointer addressing; if r3 contains the address of a pointer p, then  $M[M[r3]]$  yields \*p.

# נושאים מתקדמים – Other Examples of ISA-level Tradeoffs

- Condition codes vs. not – פעולה על סמך קיומו של תנאי – see "[Status Register](#)"
- VLIW (=Very Long Instruction Word) vs. single instruction
- Precise vs. imprecise exceptions
- Virtual memory vs. not
- Unaligned access vs. not
- Hardware interlocks vs. software-guaranteed interlocking
- Software vs. hardware managed page fault handling
- Cache coherence (hardware vs. software)
- ...

בדיקת תלות ע"י החומרה או התוכנה

# Back to *Programmer vs. (Micro)architect*

- Many ISA features designed to aid programmers
- But, complicate the hardware designer's job

דוגמאות לכך:

- Virtual memory
  - vs. overlay programming
  - Should the programmer be concerned about the size of code blocks fitting physical memory?

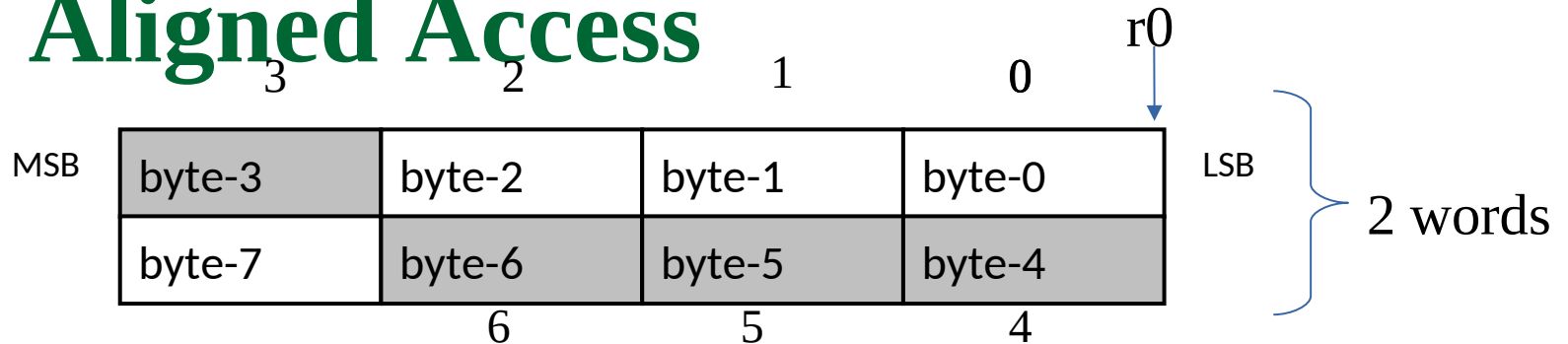
- Addressing modes

- Unaligned memory access

- Compiler/programmer needs to align data

במערכות Embedded, כאשר תכנית גדולה לא נכנסת לזיכרון המוגבל ואז היא מחולקת למודולים שנטענים לזיכרון עפ"י הצורך. כמו-כן נעשה Swap עם מקטעים של תכניות אחרות

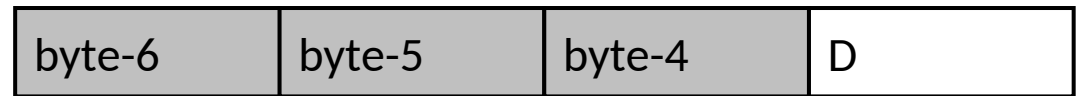
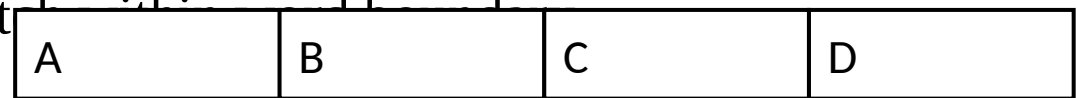
# MIPS: Aligned Access



- LW/SW alignment restriction: 4-byte word-alignment
  - not designed to fetch memory bytes not within a word boundary
  - not designed to rotate unaligned bytes into registers
- Provide separate opcodes for the “infrequent” case
- LWL/LWR (=Load Word Left/Right) is slower
  - Note LWL and LWR still fetch 4 bytes

מלא לשמאל את מה  
שנמצא ממקום מס' 6 ימינה  
(הספירה היא בבייטים)

LWL rd 6(r0) →



הצמד ימינה את מה  
שנמצא במקום 3 ושמאלה  
(הספירה היא בבייטים)

LWR rd 3(r0) →



איך בודקים אם כתובת A היא מיושרת (aligned) לגודל N-bytes?

150 פשוט בודקים את המודול  $A \% N == 0$  ? עבור MIPS: N=4

# Demo

In the next slide there is a screen shot of the MARS MIPS simulator

For the instructor:

working directory:

`/home/telzur/science/Teaching/CPU/lectures/02/code`

Code: `lwl_lwr_demo.asm`

Open mars in the terminal.



$$123456_{10} = 0001E240_{16}$$

| 3 | 2 | 1 | 0 |
|---|---|---|---|
|   |   |   |   |

|      |    |            |
|------|----|------------|
| \$t3 | 11 | 0x40000000 |
| \$t4 | 12 | 0x0001e240 |
| \$t5 | 13 | 0xe2400000 |
| \$t6 | 14 | 0x000001e2 |
| \$t7 | 15 | 0x01e24000 |
| \$s0 | 16 | 0x00000001 |
| \$s1 | 17 | 0x0001e240 |
| \$s2 | 18 | 0x00000000 |

טבלת ערכי  
הרגיסטרים  
בהגדלה

```

1 .data
2 number: .word 256
3 .text
4 addi $t0,$zero,123456
5 la $t1,number
6 sw $t0,0($t1)
7 lw $t2,0($t1)
8 lwl $t3, 0($t1)
9 lwr $t4, 0($t1)
10 lwl $t5, 1($t1)
11 lwr $t6, 1($t1)
12 lwl $t7, 2($t1)
13 lwr $s0, 2($t1)
14 lwl $s1, 3($t1)
15 lwr $s2, 3($t1)

```

המקום  
בבייטים

| Text Segment |            |            |                          |                                                                                                |
|--------------|------------|------------|--------------------------|------------------------------------------------------------------------------------------------|
| Bkpt         | Address    | Code       | Basic                    | Source                                                                                         |
|              | 0x00400000 | 0x3c010001 | lui \$1,0x00000001       | 4: addi \$t0,\$zero,123456 # initialize an arbitrary number = 0x0001E240, 123456(10)=1E240(16) |
|              | 0x00400004 | 0x3421e240 | ori \$1,\$1,0x0000e240   |                                                                                                |
|              | 0x00400008 | 0x00014020 | add \$8,\$0,\$1          |                                                                                                |
|              | 0x0040000c | 0x3c011001 | lui \$1,0x00001001       | 5: la \$t1,number # set a memory location                                                      |
|              | 0x00400010 | 0x34290000 | ori \$9,\$1,0x00000000   |                                                                                                |
|              | 0x00400014 | 0xad280000 | sw \$8,0x00000000(\$9)   | 6: sw \$t0,0(\$t1) # store the number in that location                                         |
|              | 0x00400018 | 0x8d2a0000 | lw \$10,0x00000000(\$9)  | 7: lw \$t2,0(\$t1) # load it again to verify                                                   |
|              | 0x0040001c | 0x892b0000 | lwl \$11,0x00000000(\$9) | 8: lwl \$t3, 0(\$t1) #                                                                         |
|              | 0x00400020 | 0x992c0000 | lwr \$12,0x00000000(\$9) | 9: lwr \$t4, 0(\$t1) #                                                                         |
|              | 0x00400024 | 0x892b0001 | lwl \$11,0x00000001(\$9) | 10: lwl \$t3, 1(\$t1) #                                                                        |
|              | 0x00400028 | 0x992c0001 | lwr \$12,0x00000001(\$9) | 11: lwr \$t4, 1(\$t1) #                                                                        |
|              | 0x0040002c | 0x892d0002 | lwl \$13,0x00000002(\$9) | 12: lwl \$t5, 2(\$t1) #                                                                        |
|              | 0x00400030 | 0x992e0002 | lwr \$14,0x00000002(\$9) | 13: lwr \$t6, 2(\$t1) #                                                                        |
|              | 0x00400034 | 0x892f0003 | lwl \$15,0x00000003(\$9) | 14: lwl \$t7, 3(\$t1) #                                                                        |
|              | 0x00400038 | 0x99380003 | lwr \$24,0x00000003(\$9) | 15: lwr \$t8, 3(\$t1) #                                                                        |

| Registers | Coproc 1 | Coproc 0   |
|-----------|----------|------------|
| Name      | Number   | Value      |
| \$zero    | 0        | 0x00000000 |
| \$at      | 1        | 0x10010000 |
| \$v0      | 2        | 0x00000000 |
| \$v1      | 3        | 0x00000000 |
| \$a0      | 4        | 0x00000000 |
| \$a1      | 5        | 0x00000000 |
| \$a2      | 6        | 0x00000000 |
| \$a3      | 7        | 0x00000000 |
| \$t0      | 8        | 0x0001e240 |
| \$t1      | 9        | 0x10010000 |
| \$t2      | 10       | 0x0001e240 |
| \$t3      | 11       | 0xe2400000 |
| \$t4      | 12       | 0x000001e2 |
| \$t5      | 13       | 0x01e24000 |
| \$t6      | 14       | 0x00000001 |
| \$t7      | 15       | 0x0001e240 |
| \$s0      | 16       | 0x00000000 |
| \$s1      | 17       | 0x00000000 |
| \$s2      | 18       | 0x00000000 |
| \$s3      | 19       | 0x00000000 |
| \$s4      | 20       | 0x00000000 |
| \$s5      | 21       | 0x00000000 |
| \$s6      | 22       | 0x00000000 |
| \$s7      | 23       | 0x00000000 |
| \$t8      | 24       | 0x00000000 |

# The RISC-V way

The RISC-V specification allows the standard lw (load word) instruction to handle misaligned addresses directly.

On high-performance cores: The hardware includes logic to handle the misaligned access automatically (often in a single cycle or with a small penalty). You simply use lw.

On simpler cores: If the hardware doesn't support misaligned access, the CPU will trap to a software handler (M-mode) which emulates the load using byte-sized loads and shifts, then returns control to your program.

To replace the 2-instruction MIPS sequence, a RISC-V software equivalent for an unaligned load at address might look like this:

**# Loading a 4-byte word from a misaligned address into t0**

**lbu t1, 0(a0) # t1 = [000000 | Byte0] (Overwrites everything)**

**lbu t2, 1(a0) # t2 = [000000 | Byte1]**

**lbu t3, 2(a0) # t3 = [000000 | Byte2]**

**lbu t4, 3(a0) # t4 = [000000 | Byte3]**

**slli t2, t2, 8 # Shift Byte1 to the second position**

**slli t3, t3, 16 # Shift Byte2 to the third position**

**slli t4, t4, 24 # Shift Byte3 to the fourth position**

**or t0, t1, t2 # Combine them**

**or t0, t0, t3**

**or t0, t0, t4**

# MIPS vs. RISC-V

| Feature              | MIPS                                   | RISC-V                                      |
|----------------------|----------------------------------------|---------------------------------------------|
| Instruction          | lw                                     | lw                                          |
| Address Constraint   | Must be a multiple of 4.               | Can be any byte address.                    |
| Failure Result       | Address Error Exception (Crash).       | Guaranteed success (Hardware or Emulation). |
| Special Instructions | LWL / LWR required for unaligned data. | None needed.                                |

# The RISC-V compiler will follow 4 bytes alignment (left) unless it is forced otherwise (right)

```
struct MyData {  
    char a;    // 1 byte  
    // 3 bytes of invisible padding  
    // inserted here!  
    int b;     // 4 bytes (must start  
    // at an address divisible by 4)  
};
```

```
struct PackedData {  
    char a;  
    int b;  
} __attribute__((packed));  
// Total size: 5 bytes. 'b' will be  
// misaligned at address +1.
```

# X86: Unaligned Access

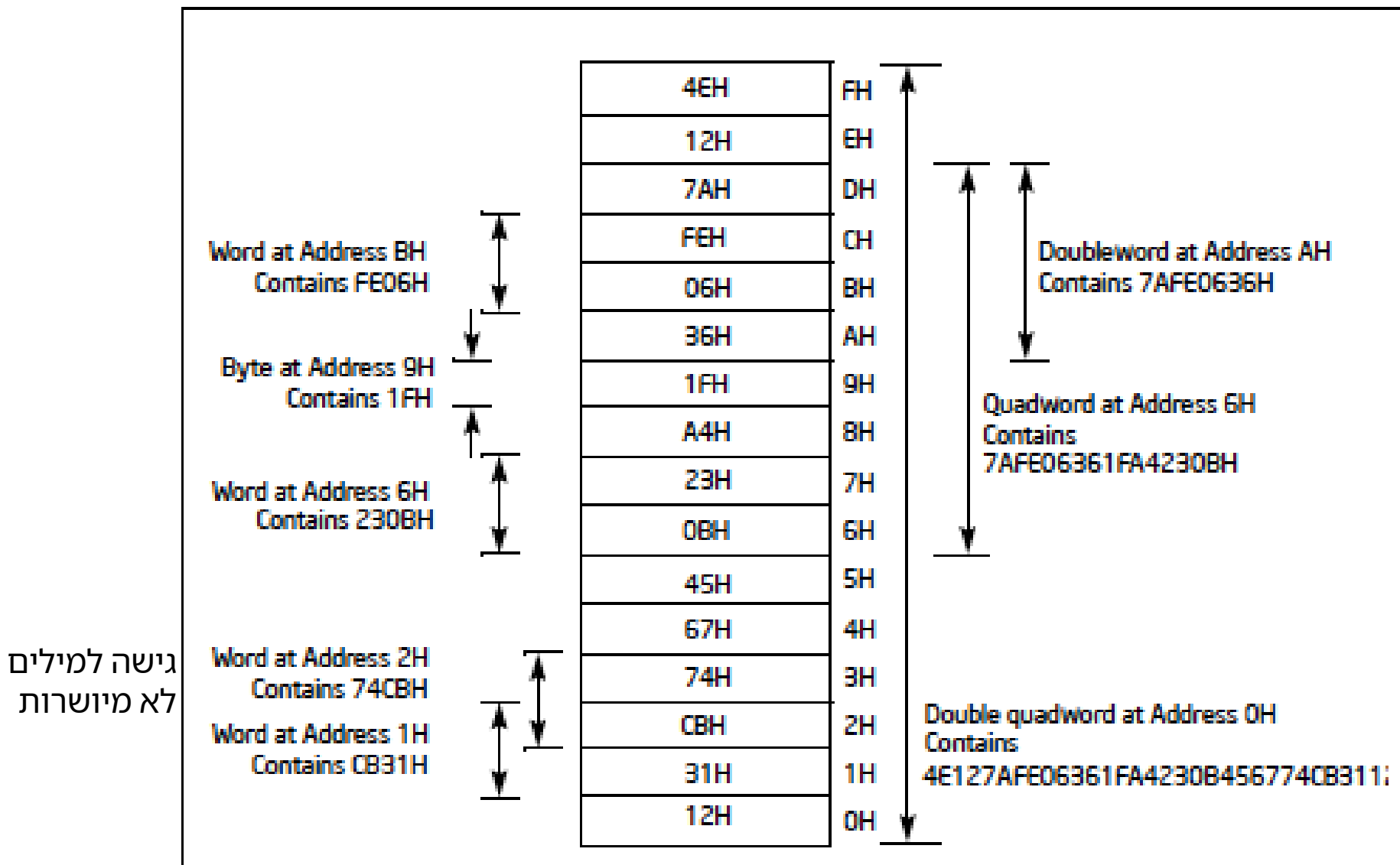


Figure 4-2. Bytes, Words, Doublewords, Quadwords, and Double Quadwords in Memory

2 bytes = 1 word

# X86: Unaligned Access

- LD/ST instructions automatically align data that spans a “word” boundary
- Programmer/compiler does not need to worry about where data is stored (whether or not in a word-aligned location)

## 4.1.1 Alignment of Words, Doublewords, Quadwords, and Double Quadwords

Words, doublewords, and quadwords do not need to be aligned in memory on natural boundaries. The natural boundaries for words, double words, and quadwords are even-numbered addresses, addresses evenly divisible by four, and addresses evenly divisible by eight, respectively. However, to improve the performance of programs, data structures (especially stacks) should be aligned on natural boundaries whenever possible. The reason for this is that the processor requires two memory accesses to make an unaligned memory access; aligned accesses require only one memory access. A word or doubleword operand that crosses a 4-byte boundary or a quadword operand that crosses an 8-byte boundary is considered unaligned and requires two separate memory bus cycles for access.

# What About ARM?

[https://www.scss.tcd.ie/~waldroj/3d1/arm\\_arm.pdf](https://www.scss.tcd.ie/~waldroj/3d1/arm_arm.pdf)

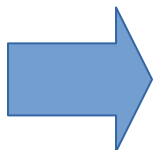
- ARM Architecture Reference Manual, Section A2.8

## A2.8 Unaligned access support

The ARM architecture traditionally expects all memory accesses to be suitably aligned. In particular, the address used for a halfword access should normally be halfword-aligned, the address used for a word access should normally be word-aligned.

Prior to ARMv6, doubleword (LDRD/STRD) accesses to memory, where the address is not doubleword-aligned, are UNPREDICTABLE. Also, data accesses to non-aligned word and halfword data are treated as aligned from the memory interface perspective. That is:

- the address is treated as truncated, with address bits[1:0] treated as zero for word accesses, and address bit[0] treated as zero for halfword accesses.
- load single word ARM instructions are architecturally defined to rotate right the word-aligned data transferred by a non word-aligned address one, two or three bytes depending on the value of the two least significant address bits.
- alignment checking is defined for implementations supporting a System Control coprocessor using the A bit in CP15 register 1. When this bit is set, a Data Abort indicating an alignment fault is reported for unaligned accesses.



ARMv6 introduces unaligned word and halfword load and store data access support. When this is enabled, the processor uses one or more memory accesses to generate the required transfer of adjacent bytes transparently to the programmer, apart from a potential access time penalty where the transaction crosses an IMPLEMENTATION DEFINED cache-line, bus-width or page boundary condition. Doubleword accesses must be word-aligned in this configuration.

# Aligned vs. Unaligned Access

תרגיל עבורכם...

- Pros of having no restrictions on alignment
- Cons of having no restrictions on alignment
- Filling in the above: an exercise for you...



# RISC vs. CISC

| RISC                                                                   | CISC                                                                    |
|------------------------------------------------------------------------|-------------------------------------------------------------------------|
| Multiple register sets, often consisting of more than 256 registers    | Single register set, typically 6 to 16 registers total                  |
| Three register operands allowed per instruction (e.g., add R1, R2, R3) | One or two register operands allowed per instruction (e.g., add R1, R2) |
| Parameter passing through efficient on-chip register windows           | Parameter passing through inefficient off-chip memory                   |
| Single-cycle instructions (except for load and store)                  | Multiple-cycle instructions                                             |
| Hardwired control                                                      | Microprogrammed control                                                 |
| Highly pipelined                                                       | Less pipelined                                                          |
| Simple instructions that are few in number                             | Many complex instructions                                               |
| Fixed-length instructions                                              | Variable-length instructions                                            |
| Complexity in compiler                                                 | Complexity in microcode                                                 |
| Only load and store instructions can access memory                     | Many instructions can access memory                                     |
| Few addressing modes                                                   | Many addressing modes                                                   |

Credit:

“The Essentials of Computer Organization and Architecture”,  
5<sup>th</sup> edition by Linda Null

עד כאן הצגת נושא ה-ISA באופן כללי.

בשקפים הבאים נכיר את ISA של מעבד MIPS (רשות, לא יילמד בכיתה).

ה-ISA של RISC-V: פרק 6 מתוך H&H  
במהדורה החדשה – במצגת נפרדת.

# MIPS ISA

במצגת הבאה נלמד בפירוט  
על ה-ISA של מעבד MIPS.

בהמשך מצגת זו שקפים אודות ה-ISA  
של MIPS במבט-על

חומר הלימוד על אסמבלי של MIPS (המצגת  
הבאה) נלקח מתוך פרק 6 בספר של H&H. המצגות  
נמצאות במודל.

# MIPS R2000 Program *Visible State*

# Program Counter

32-bit memory address  
of the current instruction

|        |
|--------|
| M[0]   |
| M[1]   |
| M[2]   |
| M[3]   |
| M[4]   |
|        |
| M[N-1] |

## Memory

$2^{32}$  by 8-bit locations (4 Giga Bytes)

## 32-bit address

|               |
|---------------|
| **Note** r0=0 |
| r1            |
| r2            |

General Purpose  
Register File  
32 32-bit words  
named r0...r31

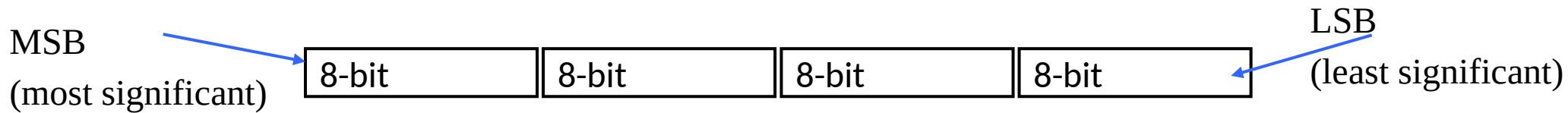
# Data Format

- ◆ Most things are 32 bits
  - instruction and data addresses
  - signed and unsigned integers
  - just bits
- ◆ Also 16-bit half word and 8-bit byte
- ◆ Floating-point numbers
  - IEEE standard 754
  - float: 8-bit exponent, 23-bit significant
  - double: 11-bit exponent, 52-bit significant

# Big Endian vs. Little Endian

(Part I, Chapter 4, Gulliver's Travels)

- ◆ 32-bit signed or unsigned integer comprises 4 bytes



- ◆ On a byte-addressable machine . . . . .

| MSB     | Big Endian |         |         | LSB |
|---------|------------|---------|---------|-----|
| byte 0  | byte 1     | byte 2  | byte 3  |     |
| byte 4  | byte 5     | byte 6  | byte 7  |     |
| byte 8  | byte 9     | byte 10 | byte 11 |     |
| byte 12 | byte 13    | byte 14 | byte 15 |     |
| byte 16 | byte 17    | byte 18 | byte 19 |     |

pointer points to the big end

| MSB     | Little Endian |         |         | LSB |
|---------|---------------|---------|---------|-----|
| byte 3  | byte 2        | byte 1  | byte 0  |     |
| byte 7  | byte 6        | byte 5  | byte 4  |     |
| byte 11 | byte 10       | byte 9  | byte 8  |     |
| byte 15 | byte 14       | byte 13 | byte 12 |     |
| byte 19 | byte 18       | byte 17 | byte 16 |     |

pointer points to the little end

- ◆ What difference does it make?

htonl = Host TO Network Long

It's used in network programming to convert a 32-bit integer from host byte order to network byte order (big-endian). This is important because different systems may use different byte orders, and network protocols typically use big-endian order.

```
telzur@TUF:~/science/Teaching/CPU/lectures/02/code$ cat ./test_htonl2.c
#include <stdio.h>
#include <arpa/inet.h>

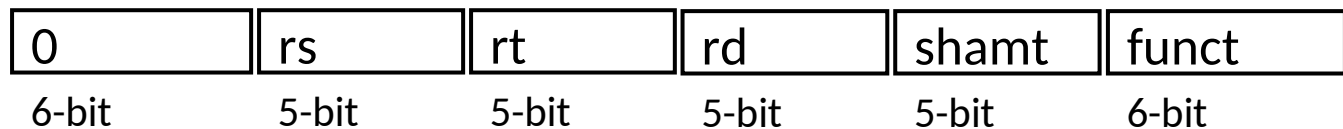
int main() {
    unsigned long hostlong = 0x12345678;
    unsigned long netlong = htonl(hostlong);
    printf("Host order: 0x%lx\n", hostlong);
    printf("Network order: 0x%lx\n", netlong);
    return 0;
}
telzur@TUF:~/science/Teaching/CPU/lectures/02/code$ ./test_htonl2
Host order: 0x12345678
Network order: 0x78563412
telzur@TUF:~/science/Teaching/CPU/lectures/02/code$
```

My computer byte order is Little Endian  
and this is why the order got reversed

# Instruction Formats

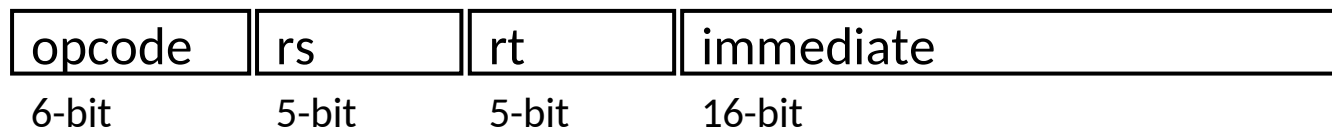
- ◆ 3 simple formats

R-type, 3 register operands

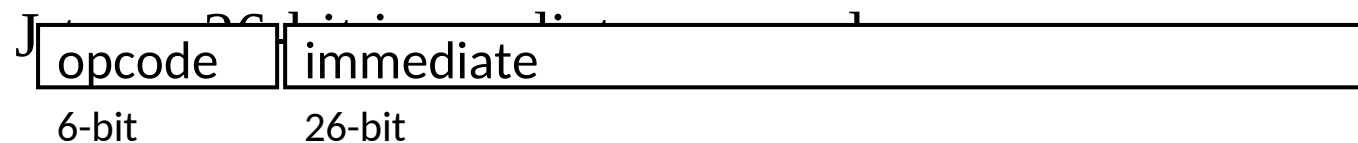


R-type

I-type, 2 register operands and 16-bit immediate operand



I-type



J-type

- ◆ Simple Decoding

4 bytes per instruction, regardless of format

must be 4-byte aligned (2 lsb of PC must be 2b'00)

format and fields readily extractable

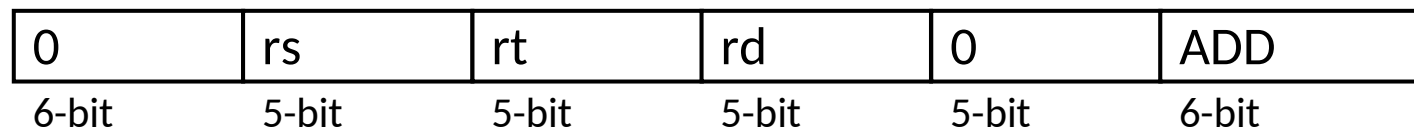


# ALU Instructions

- ◆ Assembly (e.g., register-register signed addition)

ADD  $rd_{reg}$   $rs_{reg}$   $rt_{reg}$

- ◆ Machine encoding



R-type

- ◆ Semantics

$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

$PC \leftarrow PC + 4$

- ◆ Exception on “overflow”

- ◆ Variations

Arithmetic: {signed, unsigned} x {ADD, SUB}

Logical: {AND, OR, XOR, NOR}

Shift: {Left, Right-Logical, Right-Arithmetic}

# Reg-Reg Instruction Encoding

3 ביט → 8 מצבים

|       |                   | SPECIAL function |                   |                   |                     |                     |                     |                     |
|-------|-------------------|------------------|-------------------|-------------------|---------------------|---------------------|---------------------|---------------------|
| 2...0 |                   |                  |                   |                   |                     |                     |                     |                     |
| 5...3 | 0                 | 1                | 2                 | 3                 | 4                   | 5                   | 6                   | 7                   |
| 0     | SLL               | *                | SRL               | SRA               | SLLV                | *                   | SRLV                | SRAV                |
| 1     | JR                | JALR             | *                 | *                 | SYSCALL             | BREAK               | *                   | SYNC                |
| 2     | MFHI              | MTHI             | MFLO              | MTLO              | DSLLV <sub>ε</sub>  | *                   | DSRLV <sub>ε</sub>  | DSRAV <sub>ε</sub>  |
| 3     | MULT              | MULTU            | DIV               | DIVU              | DMULT <sub>ε</sub>  | DMULTU <sub>ε</sub> | DDIV <sub>ε</sub>   | DDIVU <sub>ε</sub>  |
| 4     | ADD               | ADDU             | SUB               | SUBU              | AND                 | OR                  | XOR                 | NOR                 |
| 5     | *                 | *                | SLT               | SLTU              | DADD <sub>ε</sub>   | DADDU <sub>ε</sub>  | DSUB <sub>ε</sub>   | DSUBU <sub>ε</sub>  |
| 6     | TGE               | TGEU             | TLT               | TLTU              | TEQ                 | *                   | TNE                 | *                   |
| 7     | DSLL <sub>ε</sub> | *                | DSRL <sub>ε</sub> | DSRA <sub>ε</sub> | DSLL32 <sub>ε</sub> | *                   | DSRL32 <sub>ε</sub> | DSRA32 <sub>ε</sub> |

[MIPS R4000 Microprocessor User's Manual]



ההוראות  
היסודיות

(סה"כ 6 ביט)

הוראות מיוחדות -  
הרחבת ה-ISA - ראו  
השקף הבא.

What patterns do you see? Why are they there?

# הרחבה של ה-ISA ב-MIPS R4000

Table 1-16 Extensions to the ISA: Exception Instructions

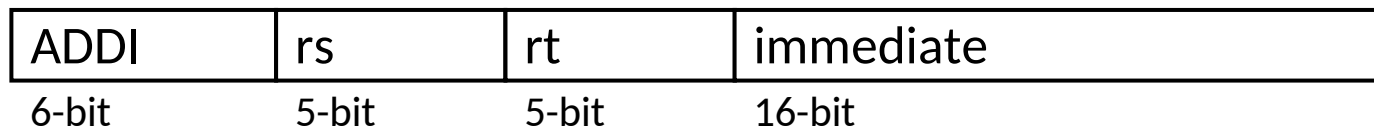
| OpCode | Description                                      |
|--------|--------------------------------------------------|
| TGE    | Trap if Greater Than or Equal                    |
| TGEU   | Trap if Greater Than or Equal Unsigned           |
| TLT    | Trap if Less Than                                |
| TLTU   | Trap if Less Than Unsigned                       |
| TEQ    | Trap if Equal                                    |
| TNE    | Trap if Not Equal                                |
| TGEI   | Trap if Greater Than or Equal Immediate          |
| TGEIU  | Trap if Greater Than or Equal Immediate Unsigned |
| TLTI   | Trap if Less Than Immediate                      |
| TLTIU  | Trap if Less Than Immediate Unsigned             |
| TEQI   | Trap if Equal Immediate                          |
| TNEI   | Trap if Not Equal Immediate                      |

# ALU Instructions

- ◆ Assembly (e.g., regi-immediate signed additions)

ADDI  $rt_{reg}$   $rs_{reg}$   $immediate_{16}$

- ◆ Machine encoding



I-type

- ◆ Semantics

$GPR[rt] \leftarrow GPR[rs] + \text{sign-extend}(\text{immediate})$

$PC \leftarrow PC + 4$

- ◆ Exception on “overflow”

- ◆ Variations

Arithmetic: {signed, unsigned} x {ADD, SUB}

Logical: {AND, OR, XOR, LUI}

# Assembly Programming 101

- ◆ Break down high-level program constructs into a sequence of elemental operations

- ◆ E.g. High-level Code

```
f = ( g + h ) - ( i + j )
```

- ◆ Assembly Code

suppose **f**, **g**, **h**, **i**, **j** are in  $r_f$ ,  $r_g$ ,  $r_h$ ,  $r_i$ ,  $r_j$

suppose  $r_{temp}$  is a free register

```
add  $r_{temp}$   $r_g$   $r_h$  #  $r_{temp} = g+h$ 
```

```
add  $r_f$   $r_i$   $r_j$  #  $r_f = i+j$ 
```

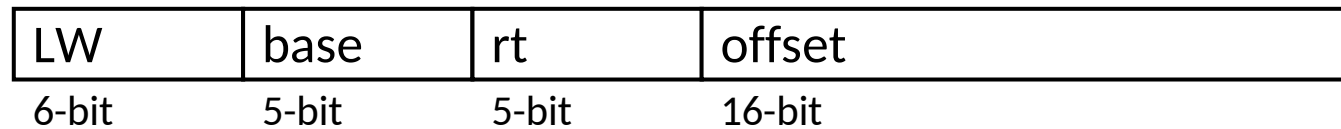
```
sub  $r_f$   $r_{temp}$   $r_f$  #  $f = r_{temp} - r_f$ 
```

# Load Instructions

- ◆ Assembly (e.g., load 4-byte word)

LW  $rt_{reg}$   $offset_{16}$  ( $base_{reg}$ )

- ◆ Machine encoding



I-type

- ◆ Semantics

$effective\_address = sign\_extend(offset) + GPR[base]$

$GPR[rt] \leftarrow MEM[translate(effective\_address)]$

$PC \leftarrow PC + 4$

- ◆ Exceptions

address must be “word-aligned”

What if you want to load an unaligned word?

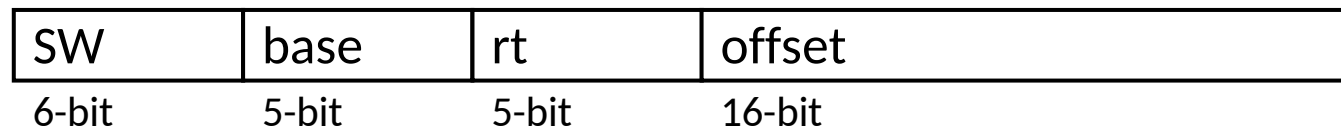
MMU (=Memory Management Unit) exceptions

# Store Instructions

- ◆ Assembly (e.g., store 4-byte word)

$\text{SW } \text{rt}_{\text{reg}} \text{ offset}_{16} (\text{base}_{\text{reg}})$

- ◆ Machine encoding



I-type

- ◆ Semantics

$\text{effective\_address} = \text{sign-extend}(\text{offset}) + \text{GPR}[\text{base}]$

$\text{MEM}[\text{translate}(\text{effective\_address})] \leftarrow \text{GPR}[\text{rt}]$

$\text{PC} = \text{PC} + 4$

- ◆ Exceptions

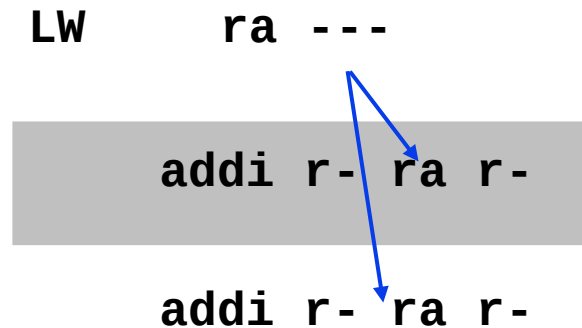
address must be “word-aligned”

MMU exceptions





# Load Delay Slots



- ◆ R2000 load has an architectural latency of 1 inst\*.
  - the instruction immediately following a load (in the “delay slot”)  
still sees the old register value
  - the load instruction no longer has an atomic semantics

Why would you do it this way?
- ◆ Is this a good idea? (hint: R4000 redefined LW to complete atomically)

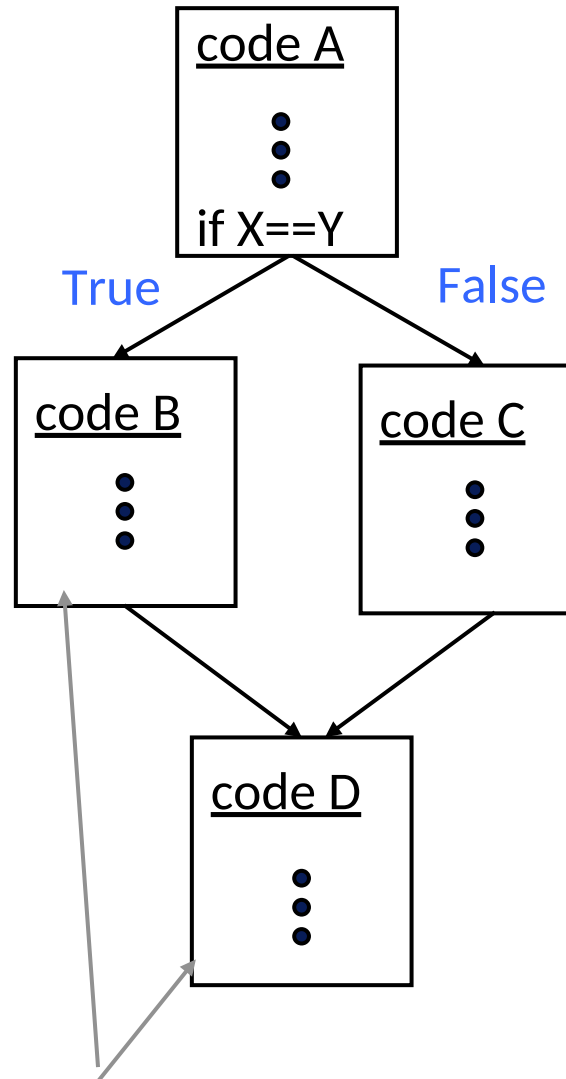
\*BTW, notice that latency is defined in “instructions” not cyc. or sec.

# Control Flow Instructions

## ◆ C-Code

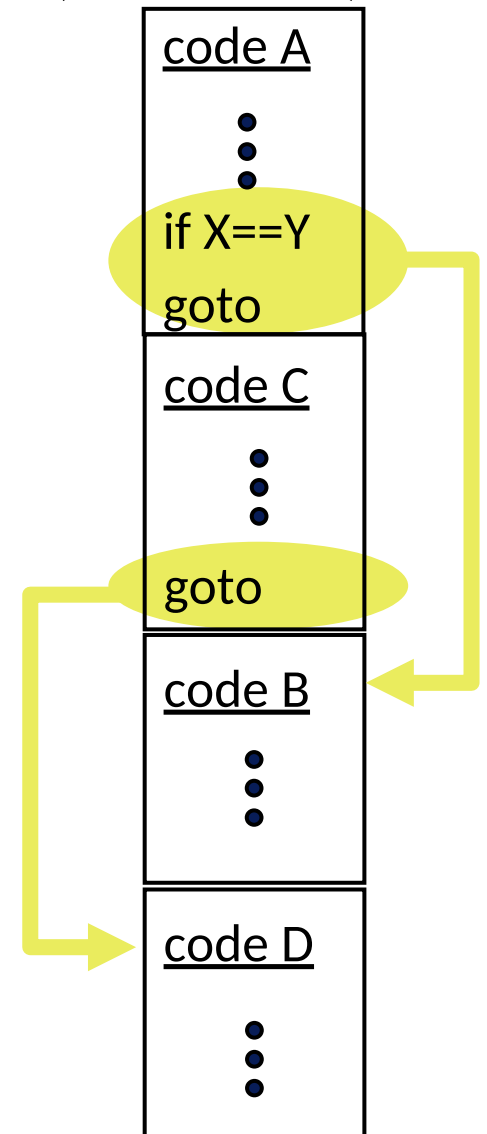
```
{ code A }  
if X==Y then  
    { code B }  
else  
    { code C }  
{ code D }
```

## Control Flow Graph



these things are called basic blocks

## Assembly Code (linearized)

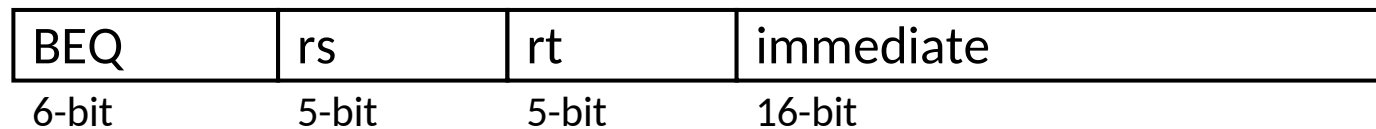


# (Conditional) Branch Instructions

- ◆ Assembly (e.g., branch if equal)

BEQ  $rs_{reg}$   $rt_{reg}$   $immediate_{16}$

- ◆ Machine encoding



I-type

- ◆ Semantics

$target = PC + \text{sign-extend}(immediate) \times 4$

if  $GPR[rs] == GPR[rt]$  then  $PC = target$

else  $PC = PC + 4$

- ◆ How far can you jump?

- ◆ Variations

BEQ, BNE, BLEZ, BGTZ

**PC + 4 w/  
branch delay slot**

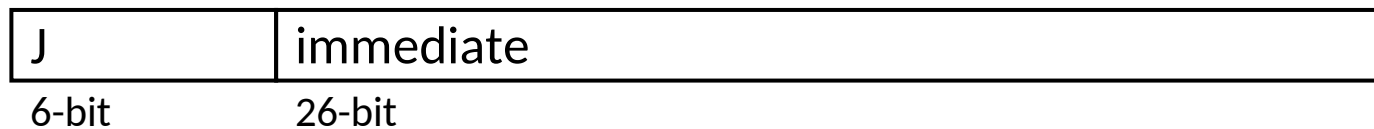
Why isn't there a BLE or BGT instruction?

# Jump Instructions

- ◆ Assembly

J immediate<sub>26</sub>

- ◆ Machine encoding



J-type

- ◆ Semantics

target = PC[31:28]x2<sup>28</sup> |<sub>bitwise-or</sub> zero-extend(immediate)x4

PC ← target

- ◆ How far can you jump?

- ◆ Variations

- Jump and Link
- Jump Registers

**PC + 4 w/  
branch delay slot**

# Assembly Programming 301

## ◆ E.g. High-level Code

```
if (i == j) then
    e = g
else
    e = h
f = e
```

## ◆ Assembly Code

- suppose  $e, f, g, h, i, j$  are in  $r_e, r_f, r_g, r_h, r_i, r_j$

```
bne  $r_i$   $r_j$  L1 # L1 and L2 are addr labels
```

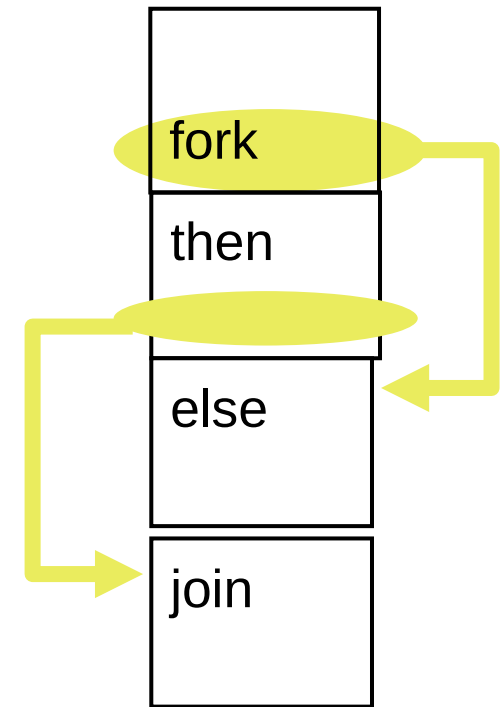
```
# assembler computes offset
```

```
add  $r_e$   $r_g$   $r0$  #  $e = g$ 
```

```
j L2
```

```
L1: add  $r_e$   $r_h$   $r0$  #  $e = h$ 
```

```
L2: add  $r_f$   $r_e$   $r0$  #  $f = e$ 
```



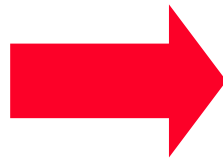
# Branch Delay Slots

- ◆ R2000 branch instructions also have an architectural latency of 1 instructions

the instruction immediately after a branch is always executed (in fact PC-offset is computed from the delay slot instruction)

branch target takes effect on the 2<sup>nd</sup> instruction

```
bne ri rj L1  
  
add re rg r0  
j L2  
  
L1: add re rh r0  
  
L2: add rf re r0  
  
. . . .
```



```
bne ri rj L1  
nop  
  
j L2  
add re rg r0  
  
L1: add re rh r0  
  
L2: add rf re r0  
  
. . . .
```

# Function Call and Return

## ◆ Jump and Link: JAL offset<sub>26</sub>

return address = PC + 8

target = PC[31:28]x2<sup>28</sup> |<sub>bitwise-or</sub> zero-extend(immediate)x4

PC ← target

GPR[r31] ← return address

On a function call, the callee needs to know where to go back to afterwards

## ◆ Jump Indirect: JR rs<sub>reg</sub>

target = GPR [rs]

PC ← target

PC-offset jumps and branches always jump to the same target every time the same instruction is executed

Jump Indirect allows the same instruction to jump to any location specified by rs (usually r31)

# Assembly Programming 401

## Caller

```
... code A ...  
JAL  _myfxn  
... code C ...  
JAL  _myfxn  
... code D ...
```

## Callee

```
_myfxn: ... code B ...  
        JR  r31
```

- ◆ ..... **A**  **B**  **C**  **B**  **D** .....
- ◆ How do you pass argument between caller and callee?
- ◆ If **A** set **r10** to 1, what is the value of **r10** when **B** returns to **C**?
- ◆ What registers can **B** use?
- ◆ What happens to **r31** if **B** calls another function



# Caller and Callee Saved Registers

## ◆ Callee-Saved Registers

Caller says to callee, “The values of these registers should not change when you return to me.”

Callee says, “If I need to use these registers, I promise to save the old values to memory first and restore them before I return to you.”

## ◆ Caller-Saved Registers

Caller says to callee, “If there is anything I care about in these registers, I already saved it myself.”

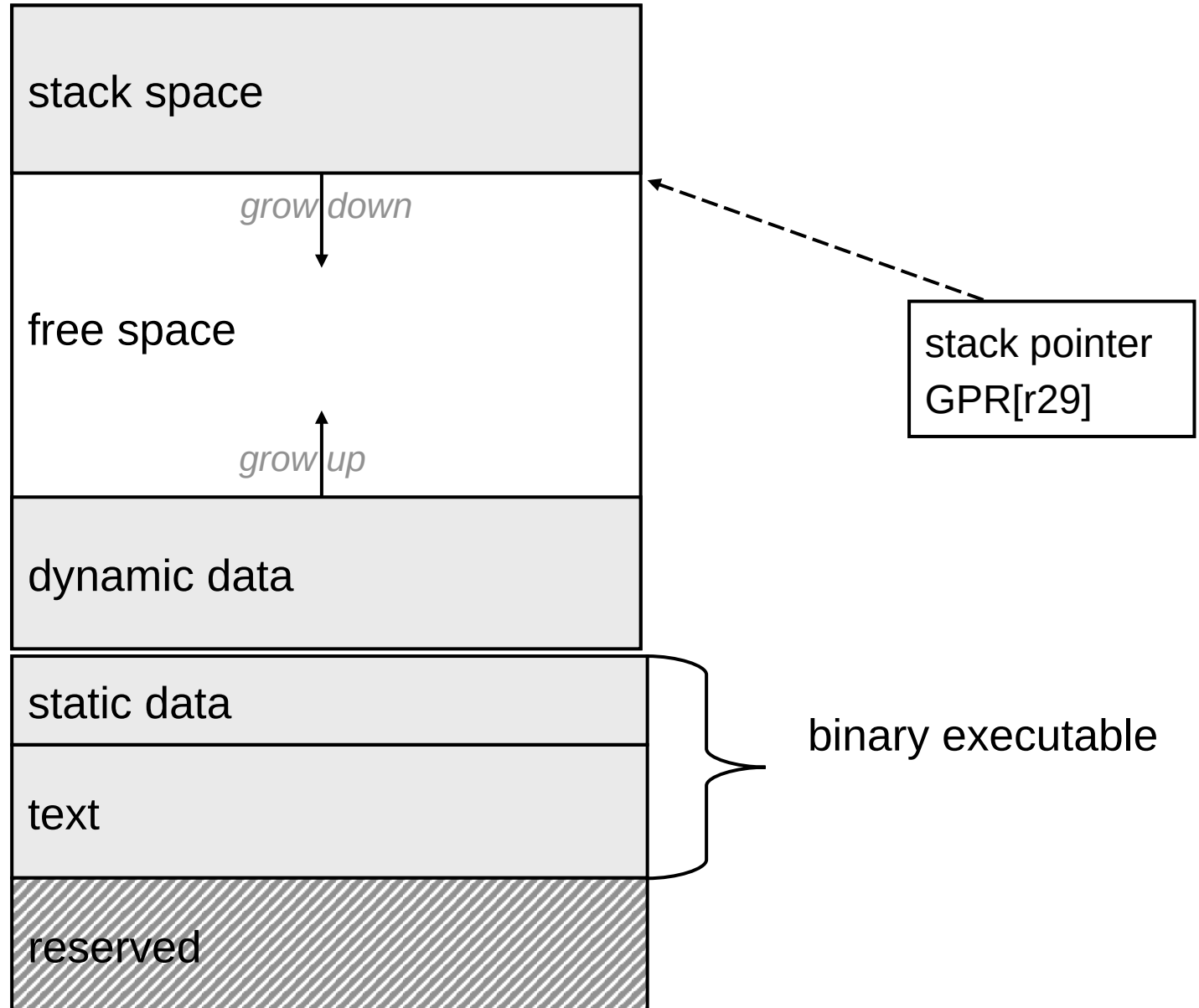
Callee says to caller, “Don’t count on them staying the same values after I am done.”

# R2000 Register Usage Convention

- ◆ r0: always 0
- ◆ r1: reserved for the assembler
- ◆ r2, r3: function return values
- ◆ r4~r7: function call arguments
- ◆ r8~r15: “caller-saved” temporaries
- ◆ r16~r23 “callee-saved” temporaries
- ◆ r24~r25 “caller-saved” temporaries
- ◆ r26, r27: reserved for the operating system
- ◆ r28: global pointer
- ◆ r29: stack pointer
- ◆ r30: callee-saved temporaries
- ◆ r31: return address

# R2000 Memory Usage Convention

high address



low address

# Calling Convention

.....

1. caller saves caller-saved registers
2. caller loads arguments into r4~r7
3. caller jumps to callee using JAL
4. callee allocates space on the stack (dec. stack pointer)
5. callee saves callee-saved registers to stack (also r4~r7, old r29, r31)

..... body of callee (can “nest” additional calls) .....

1. callee loads results to r2, r3
2. callee restores saved register values
3. JR r31
4. caller continues with return values in r2, r3

prologue

epilogue

.....

# To Summarize: RISC-V / MIPS / RISC

- ◆ Simple operations
  - 2-input, 1-output arithmetic and logical operations
  - few alternatives for accomplishing the same thing
- ◆ Simple data movements
  - ALU ops are register-to-register (need a large register file)
  - “Load-store” architecture
- ◆ Simple branches
  - limited varieties of branch conditions and targets
- ◆ Simple instruction encoding
  - all instructions encoded in the same number of bits
  - only a few formats

Loosely speaking, an ISA intended for compilers rather than assembly programmers