Computational abstraction & Instruction Set Architecture

Isaac Pérez Andrade

ITESM Guadalajara
Department of Engineering & Architecture
Department of Computer Science

February - June 2021

References

The following material has been adopted and adapted from

Patterson, D. A., Hennessy, J. L., *Computer Organization and design: The hardware/software interface - ARM edition*, Morgan Kaufmann, 2017.

S. L. Harris and D. M. Harris, *Digital design and computer architecture - ARM edition*, Morgan Kaufman, 2016.

J. Yiu, *The definitive guide to ARM Cortex-M0 and Cortex-M0+ processors*, Second edition, Elsevier, 2015.

Computational abstraction

Computational abstraction

- How would you describe the operation of a computer?
- How do users communicate with Integrated Circuits (ICs) inside a computer?
  - Think about layers.
  - Between an application (Software (SW)) and the physics of a computer (Hardware (HW), ICs, transistors), there are multiple layers that communicate with each other.
  - Each layer deals with a different complexity level.
  - Can you name some of these layers?

Abstraction

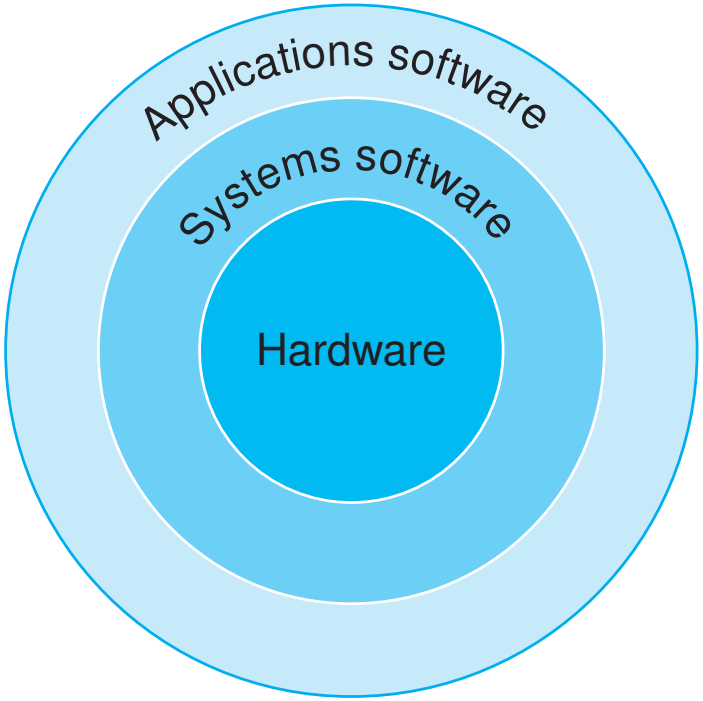


Figure 1: Simplified view of HW and SW as hierarchical layers.

- **Applications** provide direct user interaction.
- **System** layer consists of compilers and Operative Systems (OSs).
- **HW** layer relates to the actual physics of the computer, *i.e.*, signals, voltages, shapes, *etc.*

Abstraction

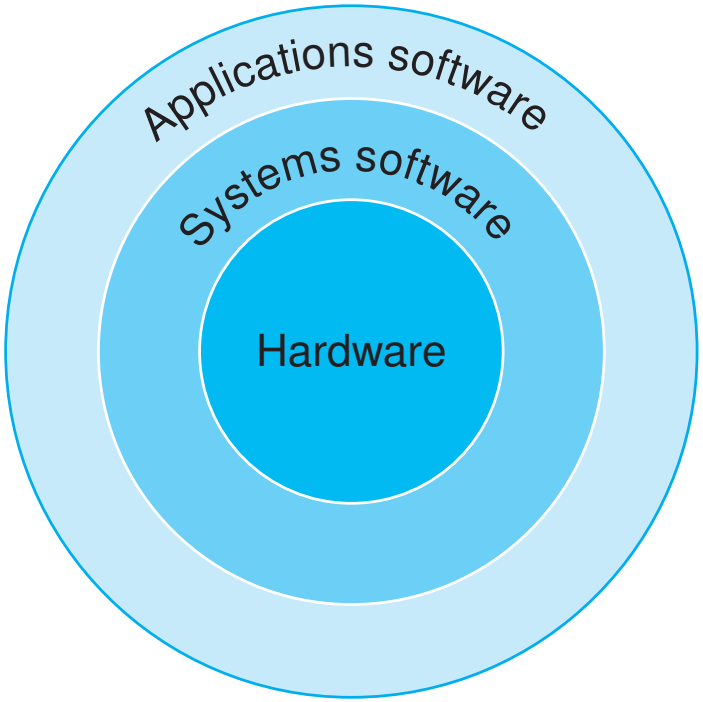


Figure 1: Simplified view of HW and SW as hierarchical layers.

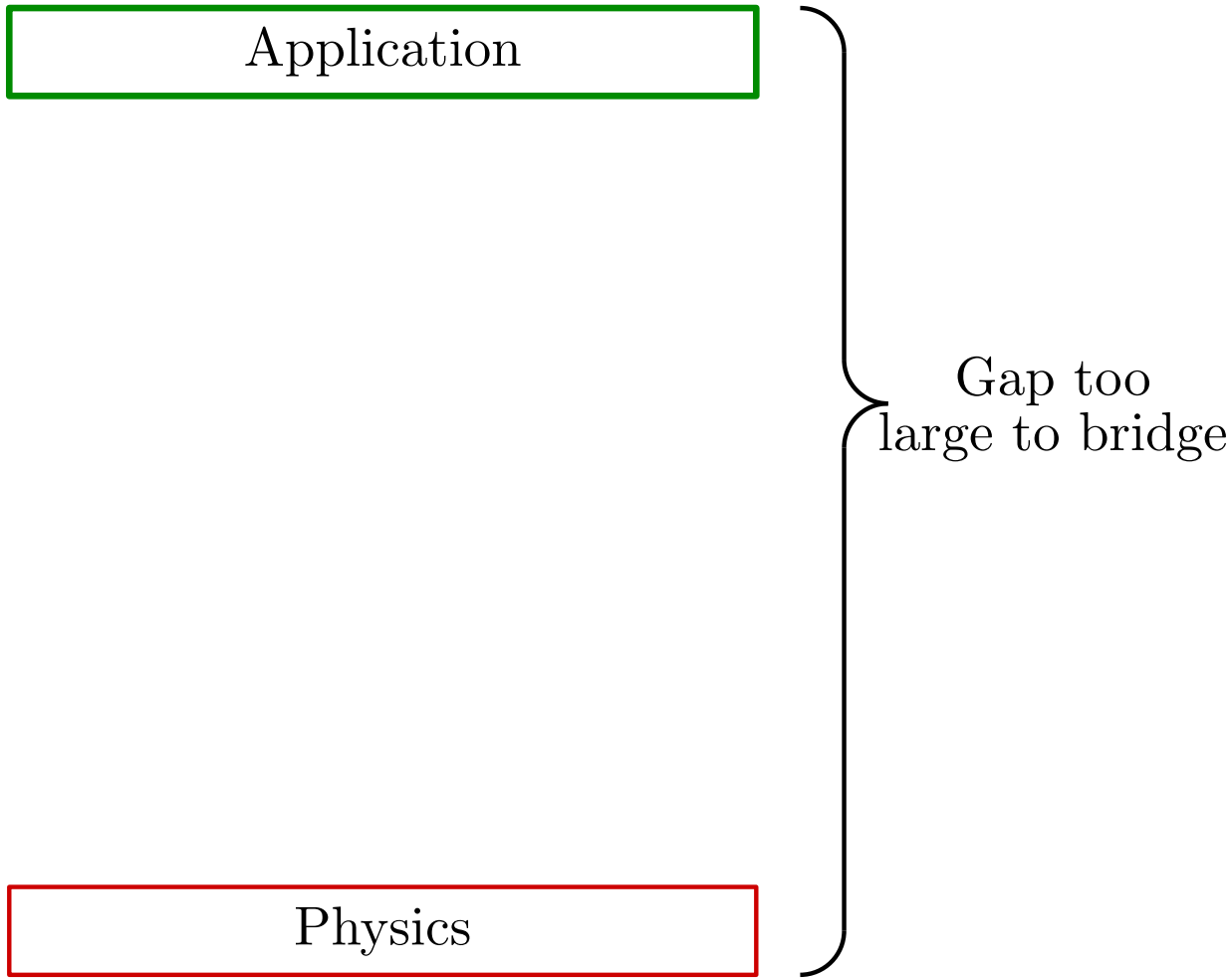
OS basic tasks.

- Provide interaction between user's programs and the HW.
- Handling basic Input/Output (IO) operations.
- Allocating storage and memory.
- Providing for protected sharing of the computer among multiple applications using it simultaneously.

Computational abstraction

- Applications such as word processors, internet browsers or media players consists of millions of lines of codes.
- Microprocessors (μPs) are only capable of executing extremely simple low-level instructions such as additions, comparisons, jumps, load/store from/to memory.
- Moreover, we must communicate with HW by simply using electrical signals.
- So how does a computing system perform such complex tasks using limited resources?

Computational abstraction



Computational abstraction

- **Q: How do we fill the gap between the application and the physics?**
- **A: The concept of abstraction.**
- Abstraction is no other thing than using different representations of a concept in order to deal with different levels of complexity.
- Hiding details when they are not important.
- This allows us to focus on a given complexity level and omit unnecessary details.

Computational abstraction

- Let's try to bridge the gap starting with the application layer.
- The application layer is the closest to the user.
- Application layer communicates with lower-level layers in order to instruct HW what to do.
- The following elements may be used for this purpose.
  - **High-level language.**
  - **Compiler.**
  - **Assembly language.**
  - **Assembler.**
  - **Machine language.**

Computational abstraction

- **High-level language.** Set of words and algebraic notation close to human language used to indicate a sequence of instructions.


```c
#include <stdio.h>

int main () {

      int a;

      /* for loop execution */
      for( a = 1; a <= 100; a++ )
      {
            printf("%d\n",a * a);
       }

      return 0;
}
```

- **Compiler.** Software that translates high-level language to assembly language.

Computational abstraction

- **Assembly language.** Set of words, also called mnemonics or dictionaries, that symbolically represent machine instructions.

```
sub_fcd7:
    clrc                    ;fcd7  81
    addcw a                 ;fcd8  23
    movw ix, a              ;fcd9  e2
    mov a, @ix+0x00         ;fcda  06 00
    mov mem_009e, a         ;fcdc  45 9e
    mov a, @ix+0x01         ;fcde  06 01
    mov mem_009f, a         ;fce0  45 9f
    mov a, @ix+0x02         ;fce2  06 02
    mov mem_00a0, a         ;fce4  45 a0
    call sub_fba3           ;fce6  31 fb a3
    ret                     ;fce9  20

sub_fcea:
    movw ix, #mem_02b6      ;fcea  e6 02 b6
    mov a, mem_01ef         ;fced  60 01 ef
    bne lab_fcf9            ;fcf0  fc 07
    mov a, #0x10            ;fcf2  04 10
    mov mem_01ef, a         ;fcf4  61 01 ef
    setb mem_0097:2         ;fcf7  aa 97

lab_fcf9:
    call sub_fd31           ;fcf9  31 fd 31
```

```
.text
main:

  # Polling keyboard input
poll_keyboard:
  lw      $t0, KEYBOARD_RX_CTRL
  beqz    $t0, poll_keyboard # 0xffff0000 = 0x00000001 ? -> Key pressed?

  # Load character from keyboard into $a0
  # $a0 contains character to be printed in syscall 11
  lbu     $a0, KEYBOARD_RX_DATA

  # Print character from keyboard and newline using syscall
  li      $v0, SYSCALL_PRINT_CHAR
  syscall
  # Polling display output
poll_display:
  lw      $t1, DISPLAY_TX_CTRL
  beqz    $t1, poll_display
  sw      $a0, DISPLAY_TX_DATA
  j poll_keyboard
```

- **Assembler.** Software that translates assembly language to machine language.
- **Machine language.** Sequence of bits that represent the most basic operations a machine may perform.

Computational abstraction

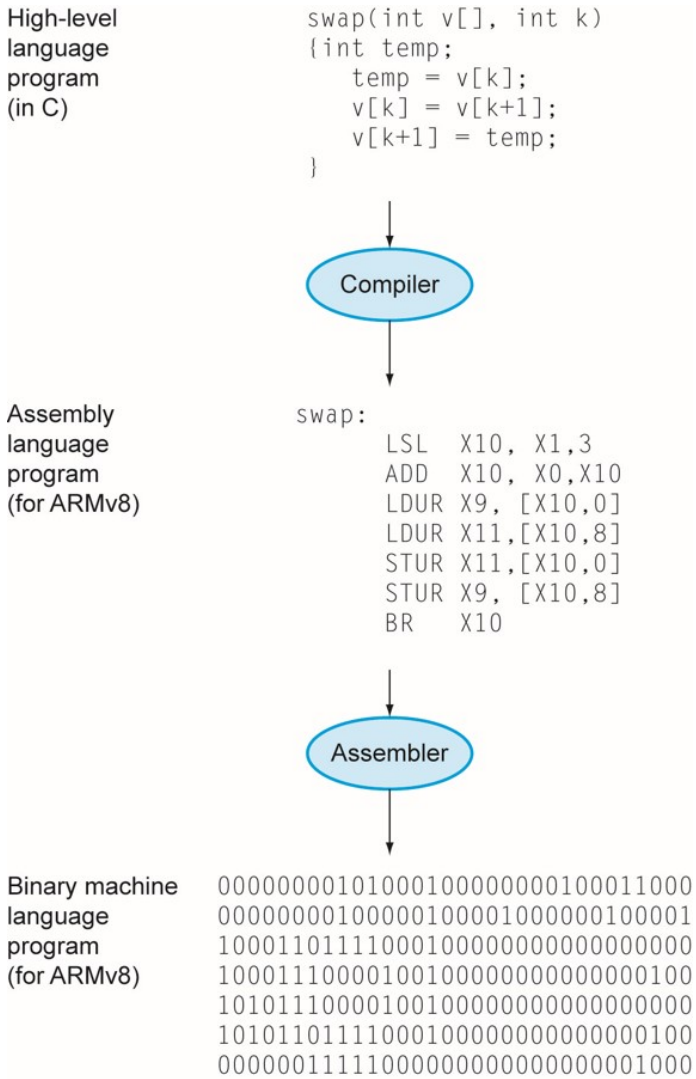
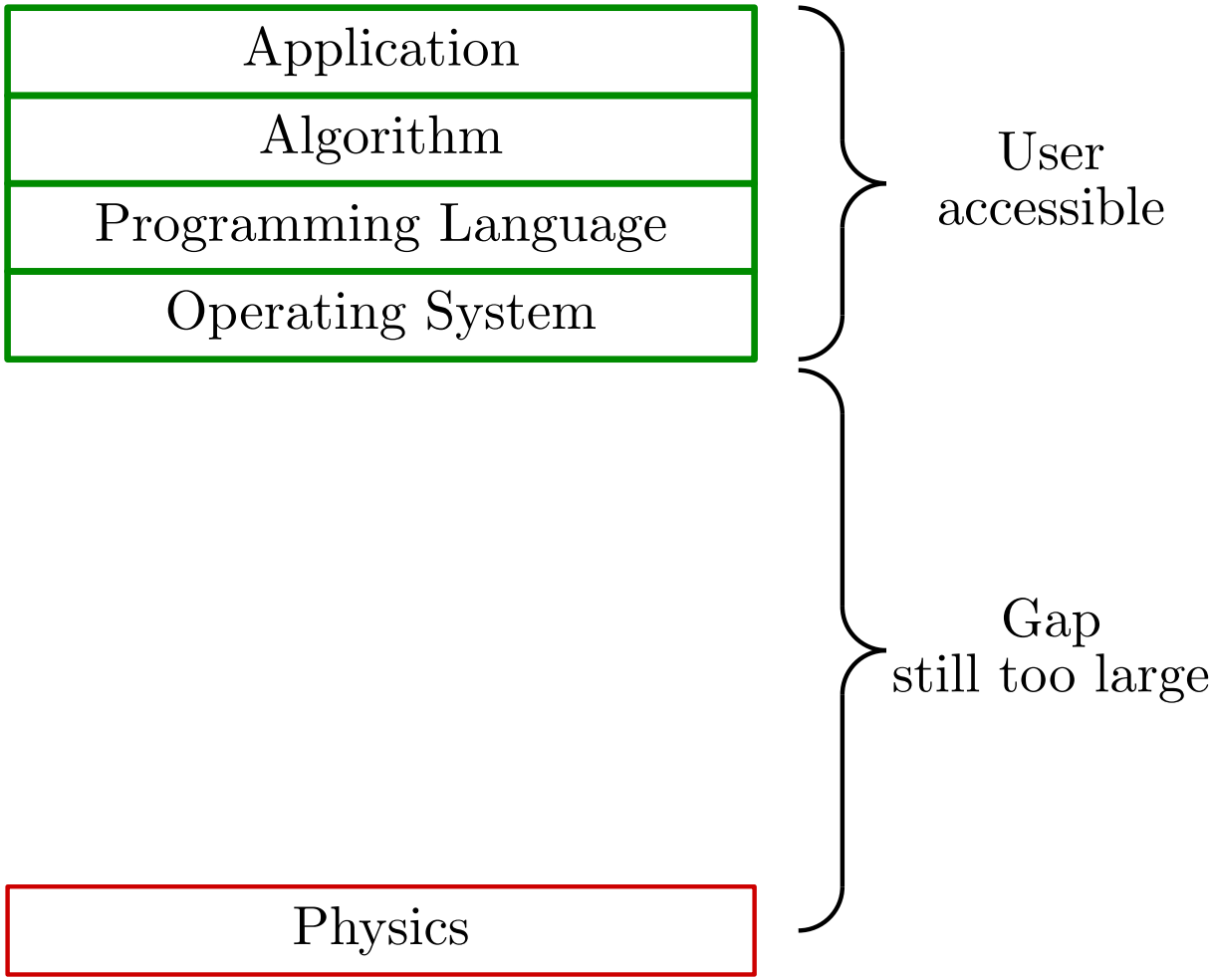


Figure 2: High-level language to machine language.

- The higher the level of the language, the more flexibility it provides.
- High-level languages allow programs to be independent of the computer on which they are developed and deployed.

Computational abstraction



Computational abstraction

- Let's now move to the physical layer.
- Do these two things represent the same functionality?

```
1 module NAND2(input logic A,
2                input logic B,
3                output logic X);
4   assign X = ~(A & B);
5 endmodule
```

Listing 1: SystemVerilog NAND2 module.

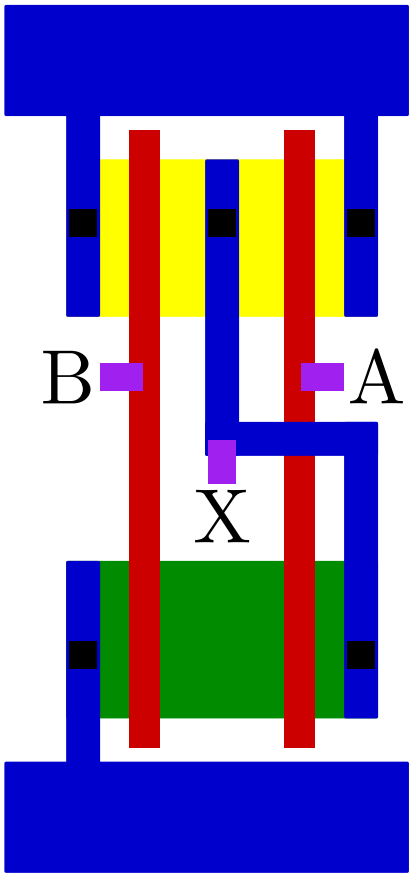


Figure 3: NAND2 layout.

Computational abstraction

- What about these representations?
- Truth table.

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

- Schematic symbol.

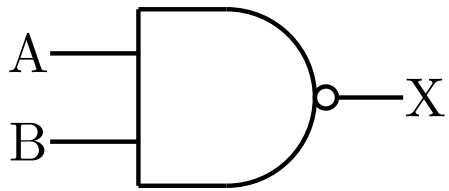


Figure 4: NAND2 gate.

Computational abstraction

- What about these representations?

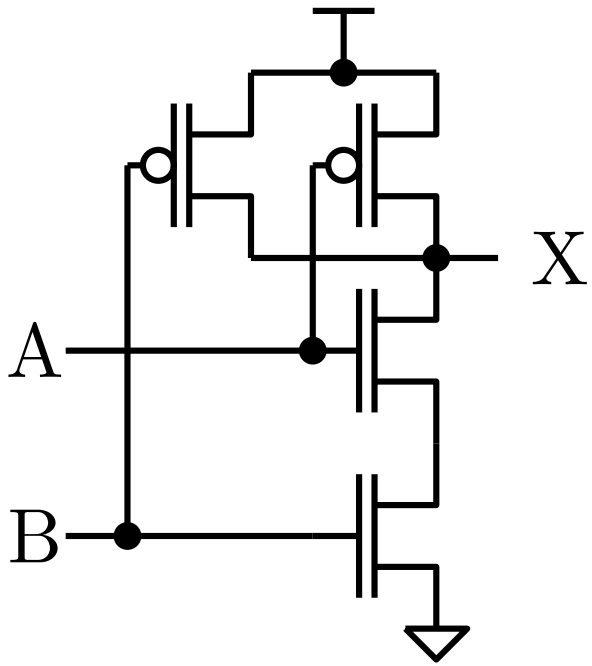


Figure 5: NAND2 transistor level.

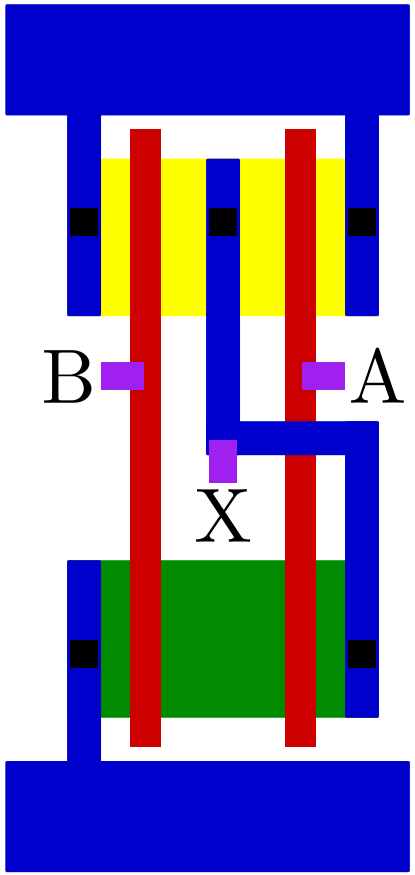
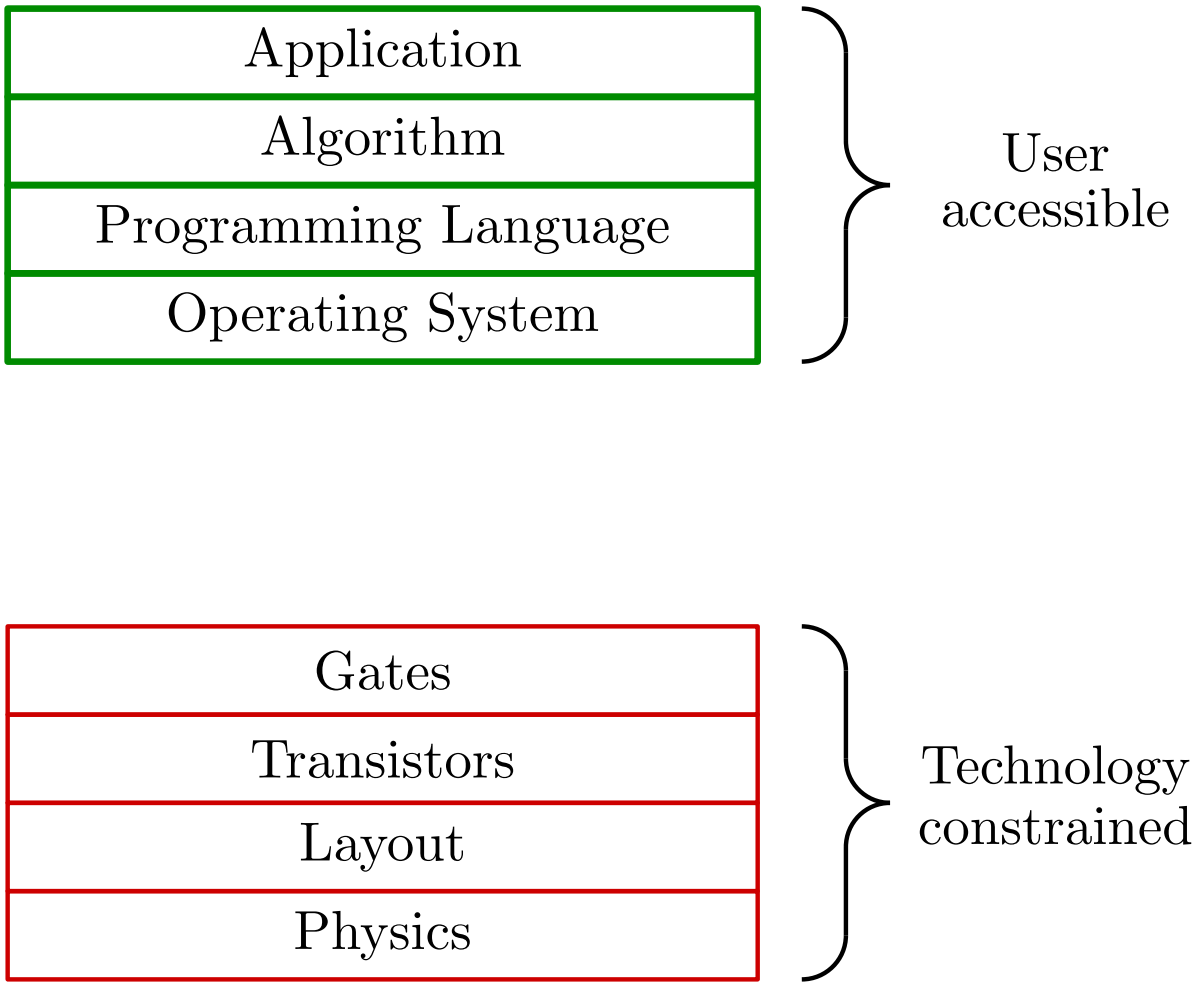


Figure 6: NAND2 layout.

Computational abstraction



Computational abstraction

- Even with the **user** and **technology** layers, there is a gap to bridge.
- **Q: How do we fill this gap?**
- **A: Computer Architecture.**

Computational abstraction

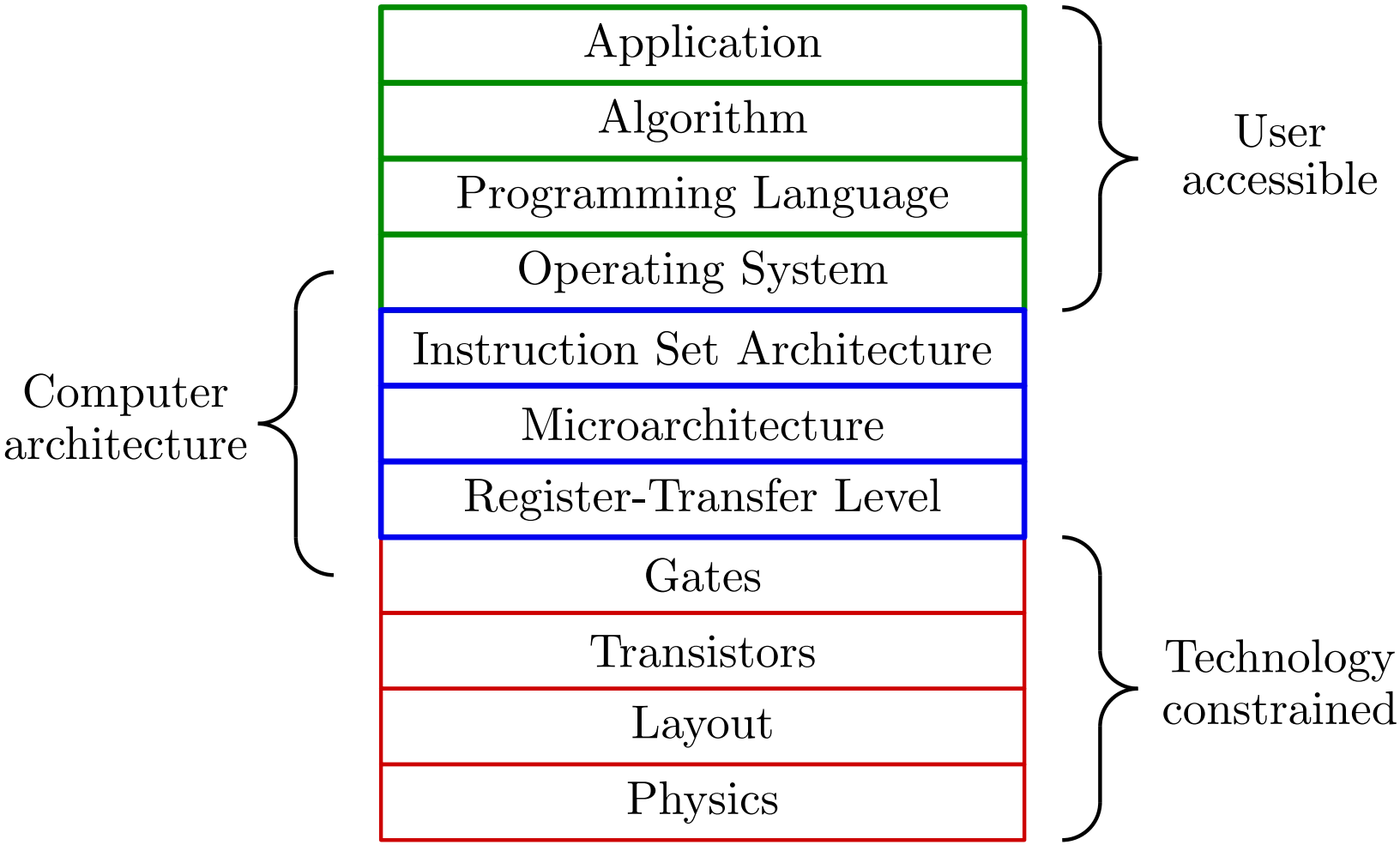


Figure 7: Simplified abstraction levels in a computing system.

Computational abstraction

- Computer architecture is the bridge between the user and the physics.
- Computer architecture may be seen as the interaction of the basic HW building blocks of a computer, as well as the semantics and rules required for such interaction.
- The three main components of computer architecture are
  - Instruction Set Architecture (ISA).
  - Microarchitecture (μA).
  - Register-Transfer Level (RTL).

Summary

- Abstraction allows us to concentrate on what really matters.
- Engineers deal with an abstraction level according to their specialization. For example:
  - SW engineers do not directly interact with HW. However, they must consider some hardware aspects such as memory allocation and usage.
  - Logic designers may deal with RTL and netlist.
  - Physical design engineers may deal with netlists and layouts.
  - Layout designers work in routing all the different semiconductor layers in an IC.
  - Analogue designers may work at the transistor and schematic levels.
- The higher the abstraction level, the more it may enclose.
- Moreover, the concept of abstraction allowed to see where this course is focused on.

Instruction Set Architecture

Instruction Set Architecture (ISA)

- It is an **abstract** concept which defines the portion of a computer that is visible to both the programmer and the compiler.
- It is part of the link between an application (something a human does such as video recording, playing music, editing a spreadsheet, etc) and the physical layer of the computer, *i.e.* HW.

- ISA **theoretically** describes how a computer executes its programs.
- It describes:
  - The fundamental operations, which are simply referred to as **instructions**, that the computer can execute.
  - How these instructions are executed.
  - The semantics and rules required for the interaction of the different building blocks of a computer.

- Overall, ISA provides valuable information to the programmer.
  - Is a computer stack-, accumulator- or register-based?
  - Does the computer have memory? Does it have registers?
  - How many steps, *i.e.* clock cycles, does it take to execute instructions?
  - Where are operands fetched from?
  - Where is the result stored?
  - How big are data types?

μA

- μA is more closely related to the **physical** implementation of a design, *i.e.*, μA determines how the ISA is implemented in HW.
  - For example, it describes which building are necessary in order to model a μP and how these building blocks are connected with each other.

- The same ISA may be physically implemented in a variety of μAs.
  - For example, one ISA could be implemented by different HW approaches and vendors such as Intel, ARM or AMD, and all three could have different performances.
- A naive adder example:
  - ISA specifies data width as 64-bits.
  - μA defines the adder as ripple-carry, carry-lookahead, carry-save, carry-select, etc.

- The goal of a processor designer is to evaluate the different trade-offs between ISA and μA in order to find a Pareto optimal system.
  - Power consumption.
  - Latency - How long does it take to complete a task.
  - Throughput - How many tasks can be completed in a given time.
  - Chip area.

Instruction Set Architecture

Same ISA, different μA- 45 nm technology

- x86 ISA.
- Quad Core.
- 2.6 GHz.
- 125 W.

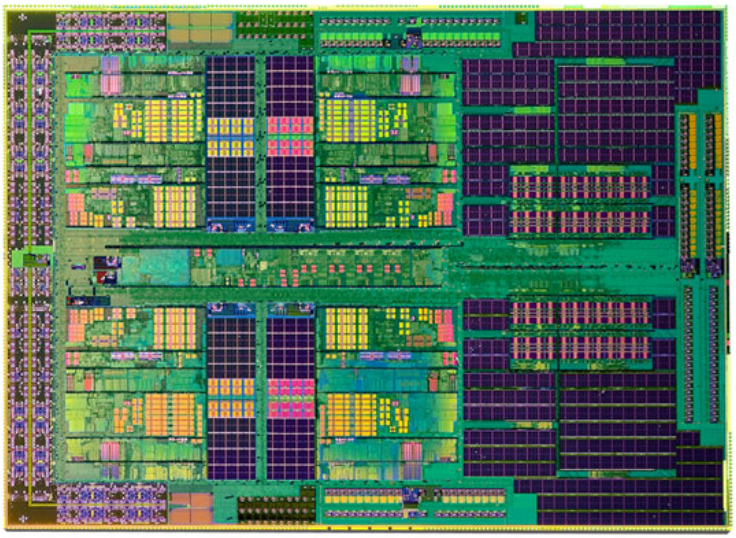


Figure 8: AMD Phenom II X4

- x86 ISA.
- Dual Core.
- 1.6 GHz.
- 4 W.

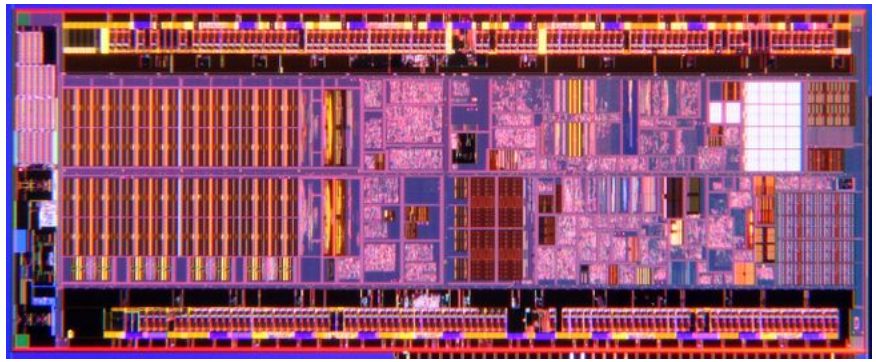


Figure 9: Intel Atom 230

- x86 ISA.
- Quad Core.
- 2.6 GHz.
- 125 W.

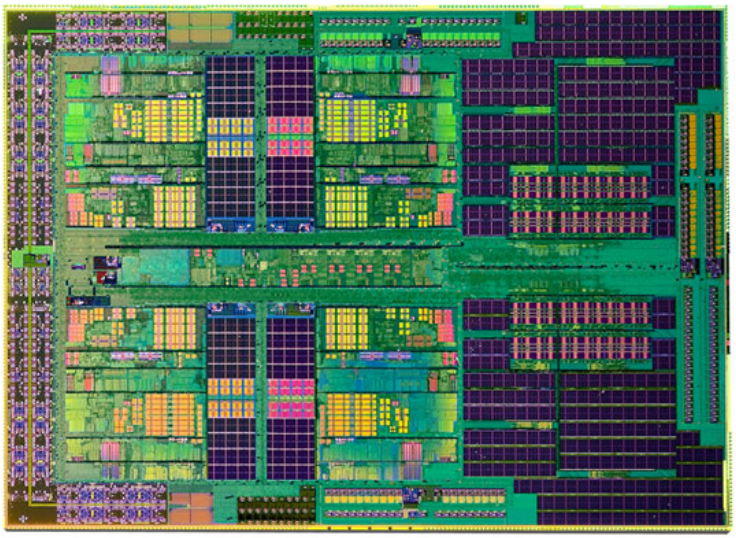


Figure 10: AMD Phenom II X4

- Power ISA.
- Octa Core.
- 4.25 GHz.
- 200 W.

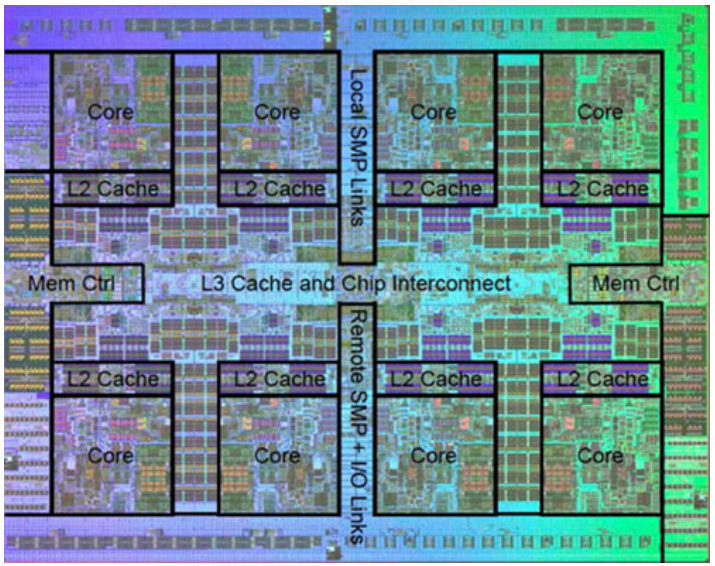


Figure 11: IBM Power7

ISA characteristics

ISA characteristics

- Type and size of instructions.
- Type and size of operands.
- Instruction encoding.
- Addressing modes.
- Registers

Registers

A register is a small memory element that stores data for quick access. This is list of registers commonly found in μPs and μP-based systems. Note that generally, systems do not implement every single register in this list.

- **Program Counter (PC)**. Also called **Instruction Pointer (IP)**, points to the memory address of the next instruction to be executed.
- **Register File (RF)**. Set of registers used to store data.
- **Stack Pointer (SP)**. Points to the next location in the stack. Used in `PUSH` and `POP` operations.

Basic registers of a computer

- **Instruction Register (IR)**. Holds the instruction currently being executed.
- **Memory Address Register (MAR)**. Also called **Address Register (AR)**, points to the memory address to/from which data is stored/fetched to/from.
- **Instruction Memory (IM)**. Memory that stores the instructions that the μP will execute.

Basic registers of a computer

- **Accumulator (ACC)**. Holds the result of arithmetic and logic operations.
- **Data Memory (DM)**. Also called **Data Register (DR)**, holds operand(s) to be used in arithmetic and logic operations.
- **General Purpose Register (GPR)**. Registers to temporary store data or addresses.
- **Status Register (SR)**. Also called **Flag Register (FR)**, holds the special conditions of the result of arithmetic and logic operations, as well as branch and jump status. For example, indicates if a comparison resulted in an equality, if the result of an operation is zero, negative, overflow, *etc.*

Basic registers of a computer

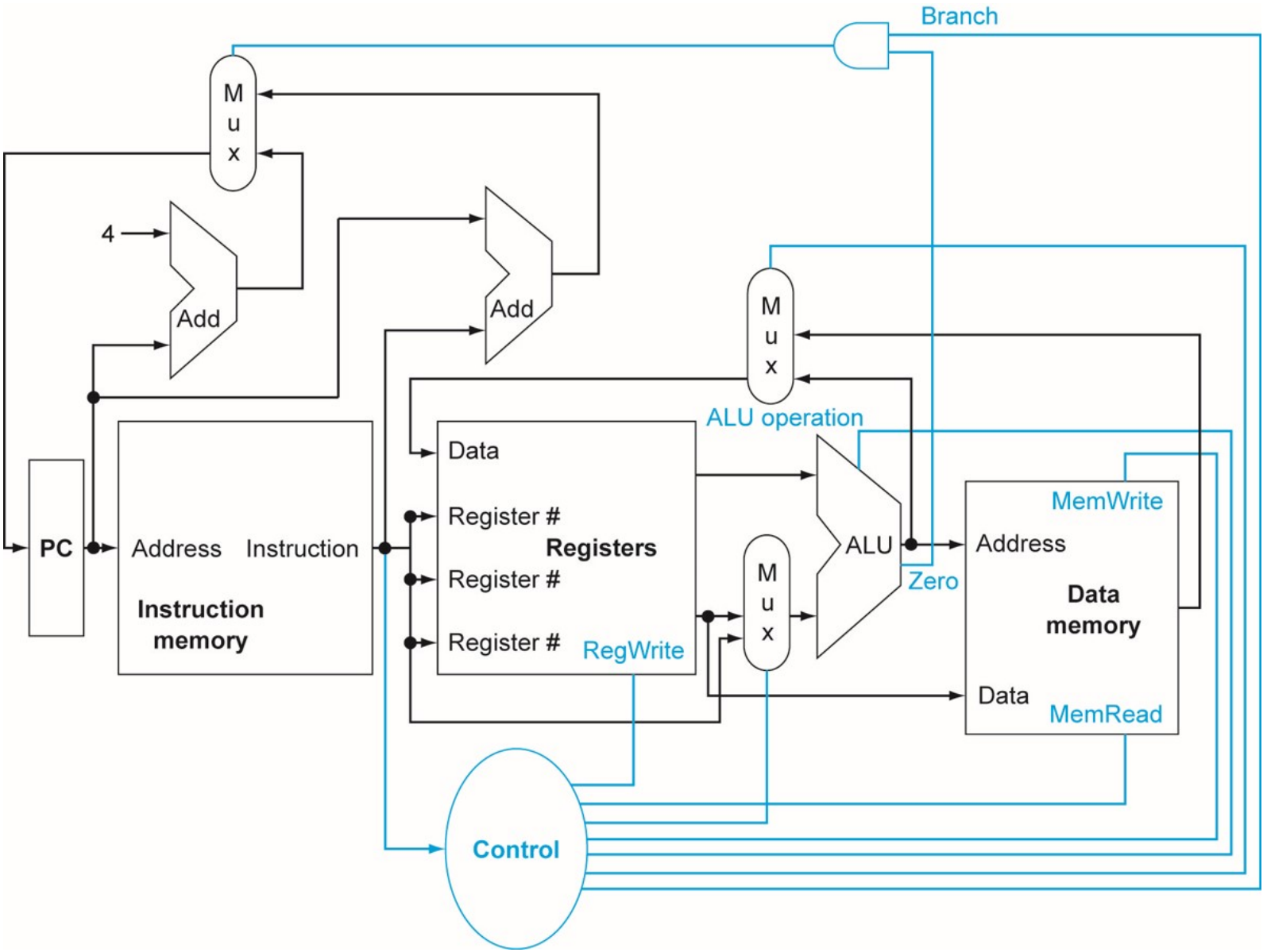


Figure 12: Basic structure of an ARM μP-based system.

Instructions

ISA characteristics

Instructions

- Instructions are the basic operations that a μP can understand and perform.
- Programmers must use basic operations in order to complete and implement complex and high-level algorithms.
- The complete set of instructions that a μP may perform is called **instruction set**, **which is not the same as Instruction Set Architecture!**

Instruction type	**Example**[1]
Arithmetic & logical	`ADD`, `SUB`, `AND`, `OR`
Data transfer	`LOAD`, `SW`, `MOV`, `PUSH`, `POP`
Conditional branch	`BNE`, `BEQ`
Unconditional jumps	`JMP`, `JAL`, `CALL`, `RET`
System	`RD_INT`, `PRNT_CHR`, `SYSCALL`
Floating Point	`FADD`, `FMULT`
String	`MOVSB`, `STR_MV`, `STR_CMP`
Signal processing[2]	`ADD_ARRAY`, `MULT_ARRAY`, `FFT`

[1] These examples are not specific to a particular ISA.
[2] Typically found in Single Instruction Multiple Data (SIMD) ISAs.

Table 1: Intel's x86 top ten instructions based on five SPECint92 programs[3].

Rank	Type	Distribution
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	**96%**

[3] J. L. Hennessy and D. A. Patterson, *Computer architecture: A quantitative approach*, 6th ed., p A-4, Morgan Kaufmann, 2019.

Operands and operations

ISA characteristics

Operands and operations

- Where do operands come from?
- Where are results stored?
- What is the size of the operands?
- How many steps does an instruction take?

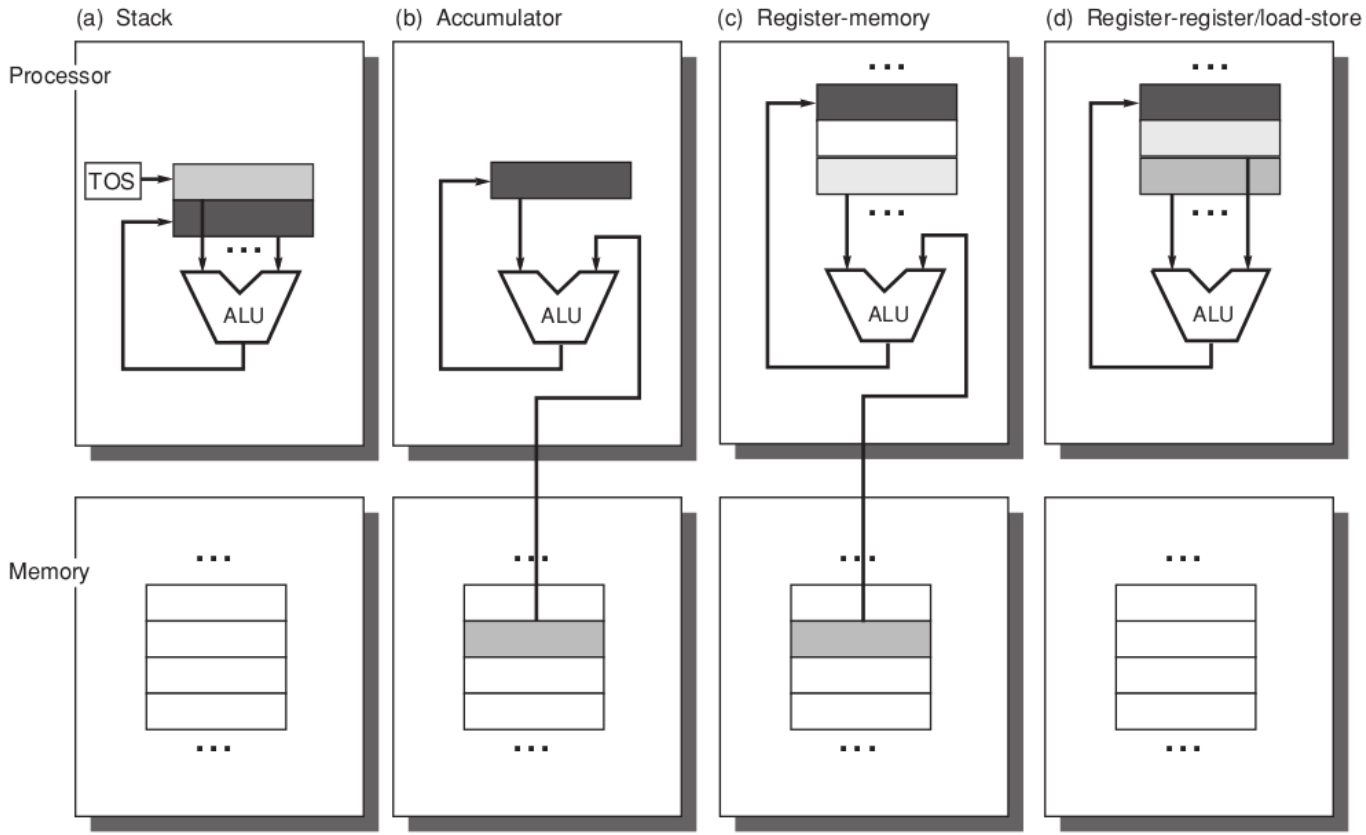
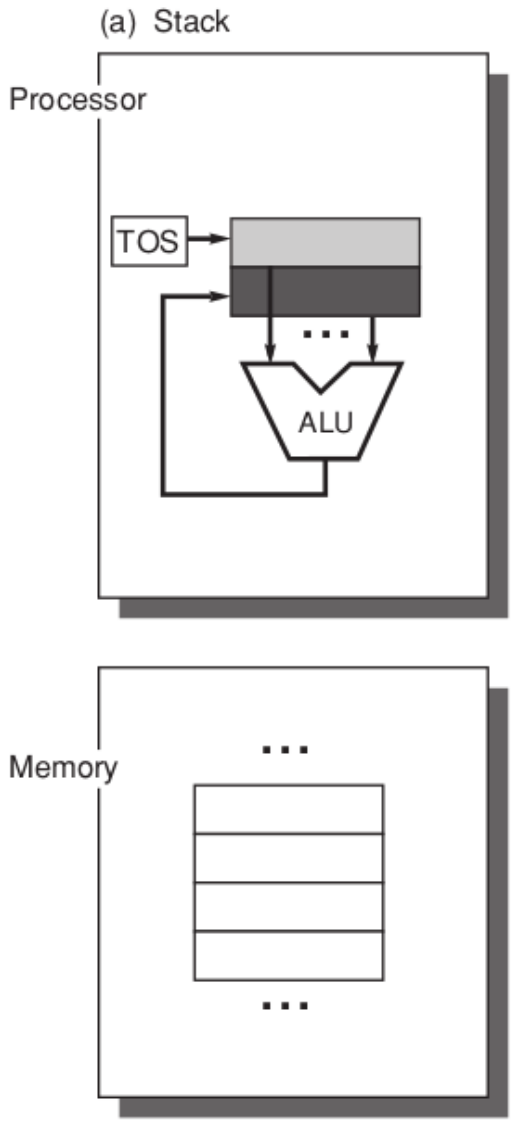


Figure 13: Operand locations for different ISAs [Figure A.1] [4].

[4] J. L. Hennessy and D. A. Patterson, *Computer architecture: A quantitative approach*, 6th ed., p A-4, Morgan Kaufmann, 2019.



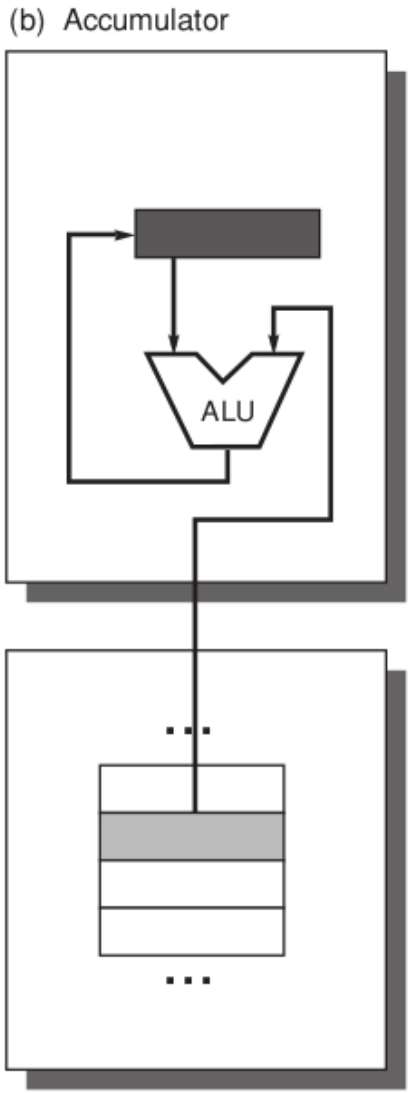
Stack-based ISA

```
C = A + B
```

```
Push   A
Push   B
Add
Pop    C
```

ISA characteristics

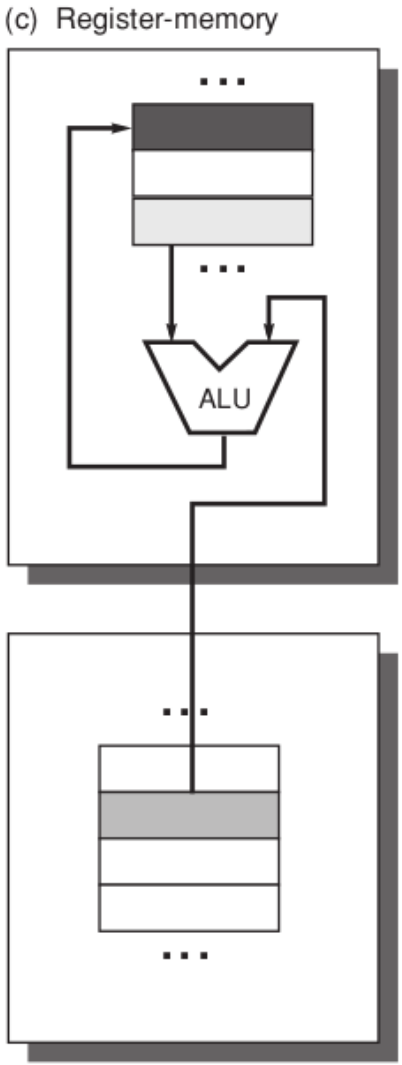
Operands and operations



Accumulator-based ISA

```
C = A + B
```

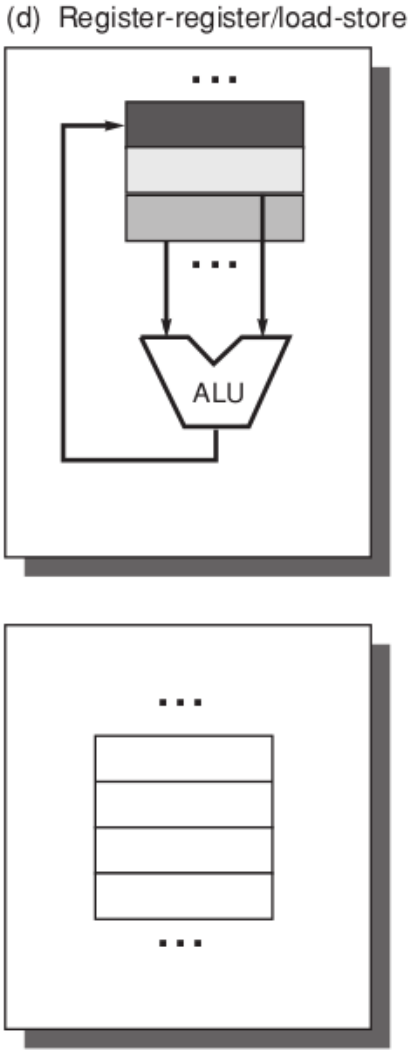
```
Load    A
Add     B
Store   C
```



Memory-Register-based ISA

```
C = A + B
```

```
Load    R1   A
Add     R3   R1   B
Store   R3   C
```



Register-Register-based ISA

```
C = A + B
```

```
Load    R1   A
Load    R2   B
Add     R3   R1   R2
Store   R3   C
```

Addressing Modes

- How can we read/write data from/into memory?
- What types of memory exist?

Table 2: Examples of addressing modes

Mode	**Example**	**Meaning**
Immediate	`ADD R4 , 3`	`R4` ← `R4` + `3`
Register	`ADD R4 , R3`	`R4` ← `R4` + `R3`
Absolute (Direct)	`ADD R2 , (100)`	`R2` ← `R2` + `Mem[100]`
Register indirect	`ADD R4 , (R1)`	`R4` ← `R4` + `Mem[R1]`
Indexed	`ADD R3 , (R1 + R2)`	`R3` ← `R3` + `Mem[R1 + R2]`
Displacement	`ADD R4 , 100(R1)`	`R4` ← `R4` + `Mem[100+R1]`
Memory indirect	`ADD R1 , @(R3)`	`R1` ← `R1` + `Mem[Mem[R3]]`

Table 3: Examples of addressing modes

Mode	Example	Meaning
Autoincrement	`ADD R1 , (R2)+`	`R1` $\leftarrow$ `R1` + `Mem[R2]` `R2` $\leftarrow$ `R2` + d
Autodecrement	`ADD R1 , -(R2)`	`R2` $\leftarrow$ `R2 -` d `R1` $\leftarrow$ `R1` + `Mem[R2]`
Scaled	`ADD R1 , 100(R2)[R3 ]`	`R1` $\leftarrow$ `R1` + `Mem[100+R2` + `R3 *d]`

Note that Autoincrement and Autodecrement modes, the order of + and - signs influence the order of the operations. For example, `-(R2)` indicates decrementing `R2` before accessing to memory.

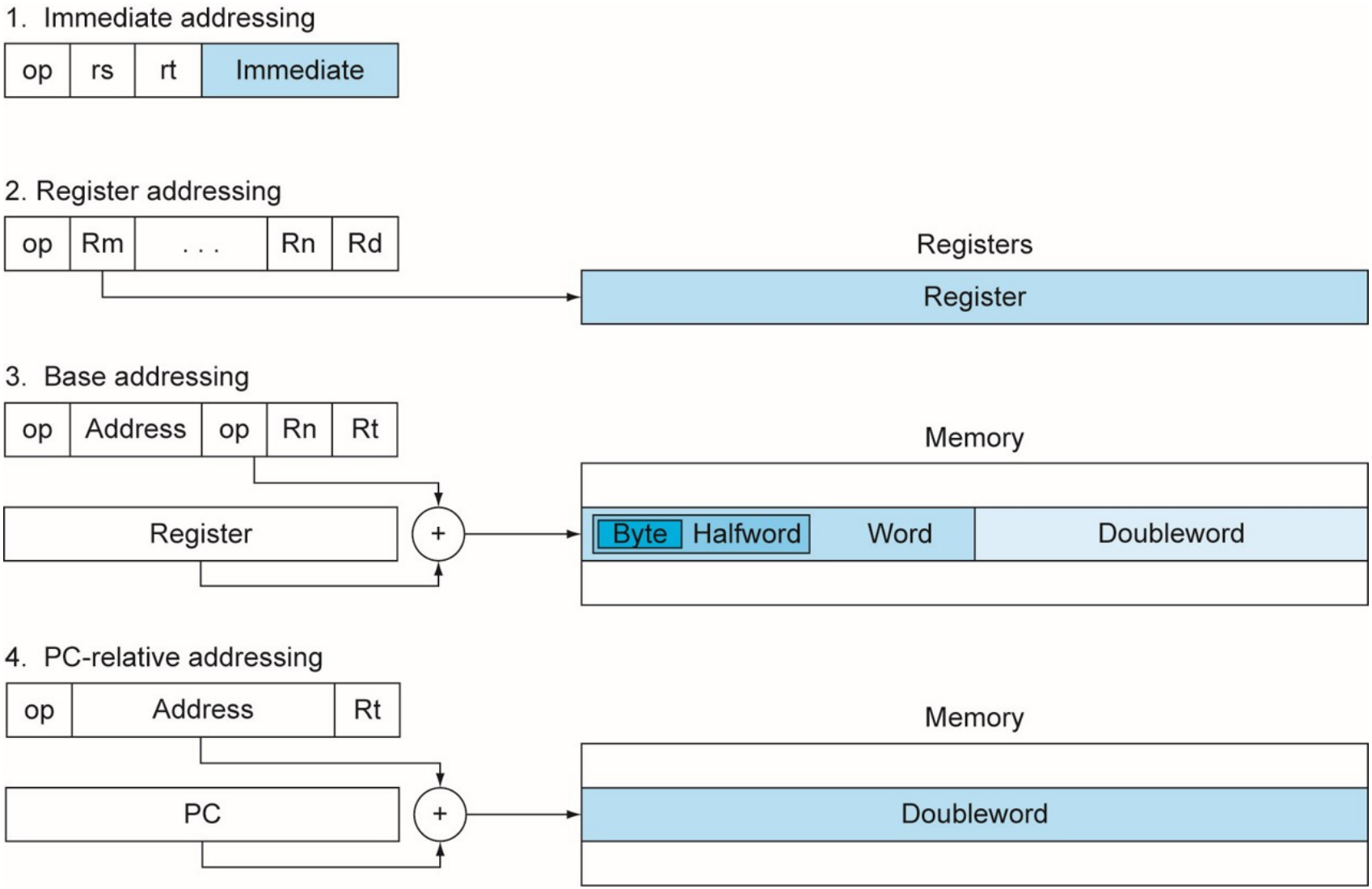


Figure 14: ARMv6 basic addressing modes.

- Let's assume that registers `Ri`, $i \in [1, 4]$, and selected memory locations of a computer store the following values.

Reg	Val
R1	23
R2	11
R3	7
R4	19

Mem	Val
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

Mode	Example	Meaning
Immediate	`ADD R4 , 3`	`R4` $\leftarrow$ `R4` + `3`

Reg	Val
R1	23
R2	11
R3	7
R4	19

Mem	Val
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

`R4` = ?

Mode	Example	Meaning
Register	`ADD R4 , R3`	`R4` $\leftarrow$ `R4` + `R3`

Reg	Val
R1	23
R2	11
R3	7
R4	19

Mem	Val
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

`R4` $= ?$

Mode	Example	Meaning
Absolute (Direct)	`ADD R2 , (100)`	`R2` $\leftarrow$ `R2` + `Mem[100]`

Reg	Val
R1	23
R2	11
R3	7
R4	19

Mem	Val
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

`R2` = ?

Mode	Example	Meaning
Register indirect	`ADD R4 , (R1)`	`R4` $\leftarrow$ `R4` + `Mem[R1]`

Reg	Val
R1	23
R2	11
R3	7
R4	19

Mem	Val
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

`R4` = ?

Mode	Example	Meaning
Indexed	`ADD R3 , (R1 + R2)`	`R3` $\leftarrow$ `R3` + `Mem[R1 + R2]`

Reg	Val
R1	23
R2	11
R3	7
R4	19

Mem	Val
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

`R3` = ?

Mode	Example	Meaning
Displacement	`ADD R4 , 100(R1)`	`R4` $\leftarrow$ `R4` + `Mem[100+R1]`

Reg	Val
R1	23
R2	11
R3	7
R4	19

Mem	Val
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

`R4` $= ?$

Mode	Example	Meaning
Memory indirect	`ADD R1 , @(R3)`	`R1` $\leftarrow$ `R1` + `Mem[Mem[R3]]`

Reg	Val
R1	23
R2	11
R3	7
R4	19

Mem	Val
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

`R1` = ?

Mode	Example	Meaning
Autoincrement	`ADD R1 , (R2)+`	`R1` $\leftarrow$ `R1` + `Mem[R2]` `R2` $\leftarrow$ `R2` + d

Reg	Val
R1	23
R2	11
R3	7
R4	19

Mem	Val
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

$d = 3$

`R1` $= ?$

Mode	Example	Meaning
Autodecrement	`ADD R1 , -(R2)`	`R2` $\leftarrow$ `R2 -` d `R1` $\leftarrow$ `R1` $+$ `Mem[R2]`

Reg	Val
R1	23
R2	11
R3	7
R4	19

Mem	Val
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

$d = 4$

`R1` $= ?$

Mode	Example	Meaning
Scaled	`ADD R1 , 100(R2)[R3 ]`	`R1` $\leftarrow$ `R1` + `Mem[100+R2` + `R3 *d]`

Reg	Val
R1	23
R2	11
R3	7
R4	19

Mem	Val
7	23
11	13
13	31
23	17
34	37
100	13
123	29
132	41

$d = 3$

`R1` $= ?$

Instruction encoding

- In stored-program computers, instructions and data are stored in memory.
- So, how does a processor know
  - how to differentiate between operations and operands?
  - which instruction to perform?
  - which `Reg` or `Mem` locations are the operands located?
  - which `Reg` or `Mem` locations should the result be stored to?
  - how to differentiate between

    `R1` $\leftarrow$ `R1 + R1`

    `R1` $\leftarrow$ `R1 + Mem[R1]`

    `R1` $\leftarrow$ `R1 + Mem[Mem[R1]]`

    `R1` $\leftarrow$ `Mem[R1] + Mem[R1]`
- Instruction encoding is a **convention used to differentiate the various operations in a μP, as well as operations from operands**.

ISA characteristics

Instructions encoding

- Instructions are encoded using binary representation.
- Suppose we want to design a processor that can implement the following instructions.

```
R1 ← R1 + R1
R1 ← R1 + Mem[R1]
R1 ← R1 + Mem[Mem[R1]]
R1 ← Mem[R1] + Mem[R1]
```

- We could assign a binary code to each of these 4 operations.

Instruction	**Binary code**
`R1 ← R1 + R1`	00
`R1 ← R1 + Mem[R1]`	01
`R1 ← R1 + Mem[Mem[R1]]`	10
`R1 ← Mem[R1] + Mem[R1]`	11

- Let's try to expand this implementation.

- Let `Rd` be the destination register and `Rsi` the source register, where `d` $\in [0, 3]$ and `i` $\in [0, 3]$.
- We could assign a binary code for each combination of `Rd` and `Rsi` in the instruction `Rd` $\leftarrow$ `Rs1` + `Rs2`.

Instruction	Binary code	Hex code
`R0 ← R0 + R0`	00 0000	`00h`
`R0 ← R0 + R1`	00 0001	`01h`
⋮	⋮	⋮
`R1 ← R2 + R3`	01 1011	`1Bh`
⋮	⋮	⋮
`R2 ← R3 + R0`	10 1100	`2Ch`
⋮	⋮	⋮
`R3 ← R3 + R3`	11 1111	`3Fh`

- What about the μA of this encoding?

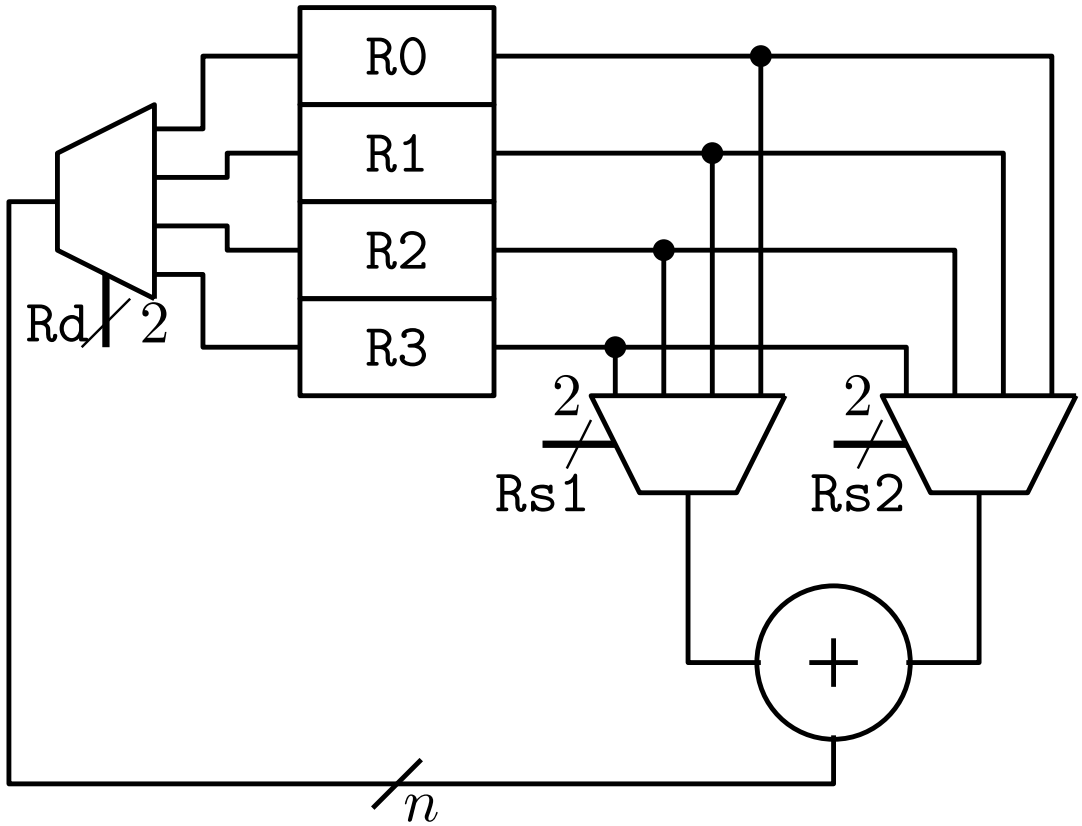


Figure 15: A naive μA for adding two registers.

- Is our previous scheme feasible?
- What's wrong with it?
- Could it be generalised?
- What about other addressing modes?
- How could we include other operations such as subtractions or jumps?
- Are all instructions represented using the same number of bits?

- We can continue to expand this scheme in order to add other operations, *e.g.*, subtraction and logical operations.
- Moreover, we can continue to include bits that represent different addressing modes.
- The ultimate goal of this, is to design an encoding feasible for all operations and addressing modes in our ISA.

ISA characteristics

Instructions encoding

- We could have a bit for selecting addition/subtraction in our previous design.

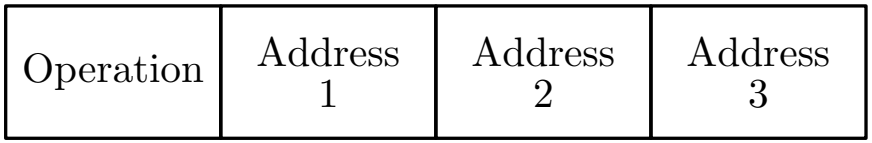
Table 4: Naive encoding for adding and subtracting two numbers.

Instruction	Binary code	Hex code
R0 ← R0 + R0	**0**00 0000	00h
⋮	⋮	⋮
R1 ← R2 + R3	**0**01 1011	1Bh
⋮	⋮	⋮
R3 ← R3 + R3	**0**11 1111	3Fh
R0 ← R0 - R0	**1**00 0000	40h
⋮	⋮	⋮
R1 ← R2 - R3	**1**01 1011	5Bh
⋮	⋮	⋮
R3 ← R3 - R3	**1**11 1111	7Fh

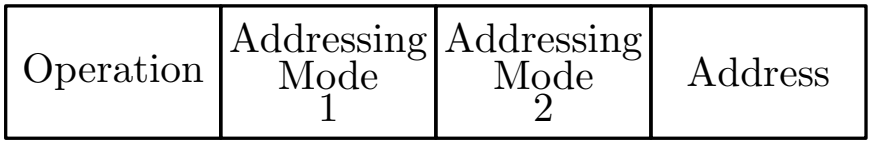
- As seen in the previous example, the source, destination and operation type may be represented with a **single** binary code.
- This concept may be further expanded for other operations and addressing modes.
- For this purpose, we may use longer binary codes, which may be divided into several fields.
- For example, we may use a field called **opcode** in order to represent the type of operation to be performed.
- Another field may represent the **source**, *i.e.*, the memory location where data to operate with is.
- Another field may represent the **destination**, *i.e.*, the memory location where the result of the operation should be stored to.

Variable encoding

Fixed encoding



Hybrid encoding



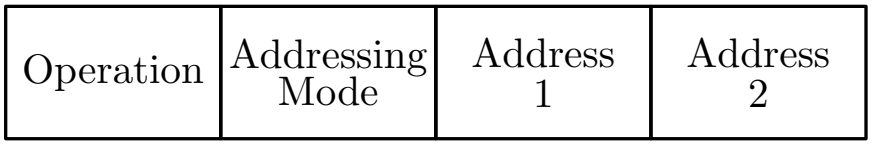


Figure 16: Generalised instruction encoding.

ISA characteristics

Instructions encoding

- **Variable encoding.**
  - Supports any number of operands, with each operand having a specific addressing mode.
  - The number of encoded bits varies between instructions.
  - **Compact machine programs.**
  - **Harder to decode.**
- **Fixed encoding.**
  - Every instruction has the same number of operands, with addressing modes specified in the opcode.
  - The number of encoded bits is always the same regardless of the instruction or addressing modes of the operands.
  - **Easier to decode.**
  - **Wasted bits in some instructions.**
- **Hybrid encoding.**
  - Instructions use two different encoding lengths (16- and 32-bits, for example).

- ISAs may be classified into two main categories according to the complexity of their instruction encoding.
- **Reduced Instruction Set Computer (RISC).**
- **Complex Instruction Set Computer (CISC).**

CISC

- ISAs that perform complex operations and the instruction formats are not uniform.
- Large number of instructions available.
- Microcode approach.
  - A single instruction may be divided into several smaller instructions.
  - For example, a single instruction may perform a load from memory, an arithmetic operation and a store to memory.
- Reduced size of the compiled code due to **variable-length** encoding.
  - Shortest encodings represent the most commonly used instructions.

RISC

- ISAs that have a small number of simple, **fixed-length** instructions.
- Single-cycle instructions.
- Load-store approach.
  - Only load and store instructions are used for transferring data between registers and memory.

```
 1:  mul16:
 2:          pushl   %ebp                ; 01010101
 3:          movl    %esp, %ebp          ; 1000100111100101
 4:          movl    8(%ebp), %ecx       ; 100010000100110100001000
 5:          pushl   %ebx                ; 01010011
 6:          movl    12(%ebp), %edx      ; 100010110101010100001100
 7:          xorl    %ebx, %ebx          ; 0011000111011011
 8:          movl    $15, %eax           ; 1011100000001111
 9:          .p2align 2,,3               ; 00000000000000000000000
                                         ; 10001101011101100000000
10:  .L6:
11:          testb   $1, %dl             ; 11110110110000100000001
12:          je      .L5                 ; 0111010000000010
13:          addl    %ecx, %ebx          ; 0000000111001011
14:  .L5:
15:          sall    %ecx                ; 1101000111100001
16:          shrl    %edx                ; 1101000111101010
17:          decl    %eax                ; 01001000
18:          jns     .L6                 ; 0111100111110010
19:          movl    %ebx, %eax          ; 1000100111011000
20:          popl    %ebx                ; 01011011
21:          leave                       ; 11001001
22:          ret                         ; 11000011
```

Figure 17: CISC code example.

```
 1:  mul16:
 2:          move    $6, $0              # 00000000000000000011000000100001
 3:          li      $3, 15              # 00100100000000110000000000001111
 4:  $L6:
 5:          andi    $2, $5, 0x1         # 00110000101000100000000000000001
 6:          addiu   $3, $3, -1          # 00100100011000111111111111111111
 7:          beq     $2, $0, $L5         # 00010000010000000000000000000010
 8:          srl     $5, $5, 1           # 00000000000001010010100001000010
 9:          addu    $6, $6, $4          # 00000000110001000011000000100001
10:  $L5:
11:          bgez    $3, $L6            # 00000100011000011111111111111010
12:          sll     $4, $4, 1          # 00000000000001000010000001000000
13:          j       $31                # 00000011111100000000000000001000
14:          move    $2, $6             # 00000000110000000001000000100001
```

Figure 18: RISC code example.

QUIZ

- Which of the following are affected by the instruction encoding?
  - A) The execution time of each instruction.
  - B) The μA of the processor.
  - C) Global warming.
  - D) The size of the compiled program.
  - E) All of the above.
  - F) None of the above.

QUIZ

- Which of the following are affected by the instruction encoding?
  - A) **The execution time of each instruction.**[5]
  - B) **The μA of the processor.**
  - C) Global warming.
  - D) **The size of the compiled program.**
  - E) All of the above.
  - F) None of the above.

[5] Think about CISC and RISC differences.

Summary

- ISA is the link between applications and HW.
- μA refers to the physical implementation of the ISA.
- The same ISA can be implemented in different μAs.
- ISA encloses
  - Type and size of instructions and operands.
  - Addressing modes.
  - Instruction encoding.
- There are RISC and CISC ISAs.
- There are several trade-offs associated between ISAs and μAs, and our goal is to find a Pareto-optimal design.

Further Reading

- Read about the difference between Von-Neumann and Harvard architectures.