Lecture 8 Instruction Sets: Characteristics and Functions

Objective:

Understand the factors involved in instruction set architecture design.

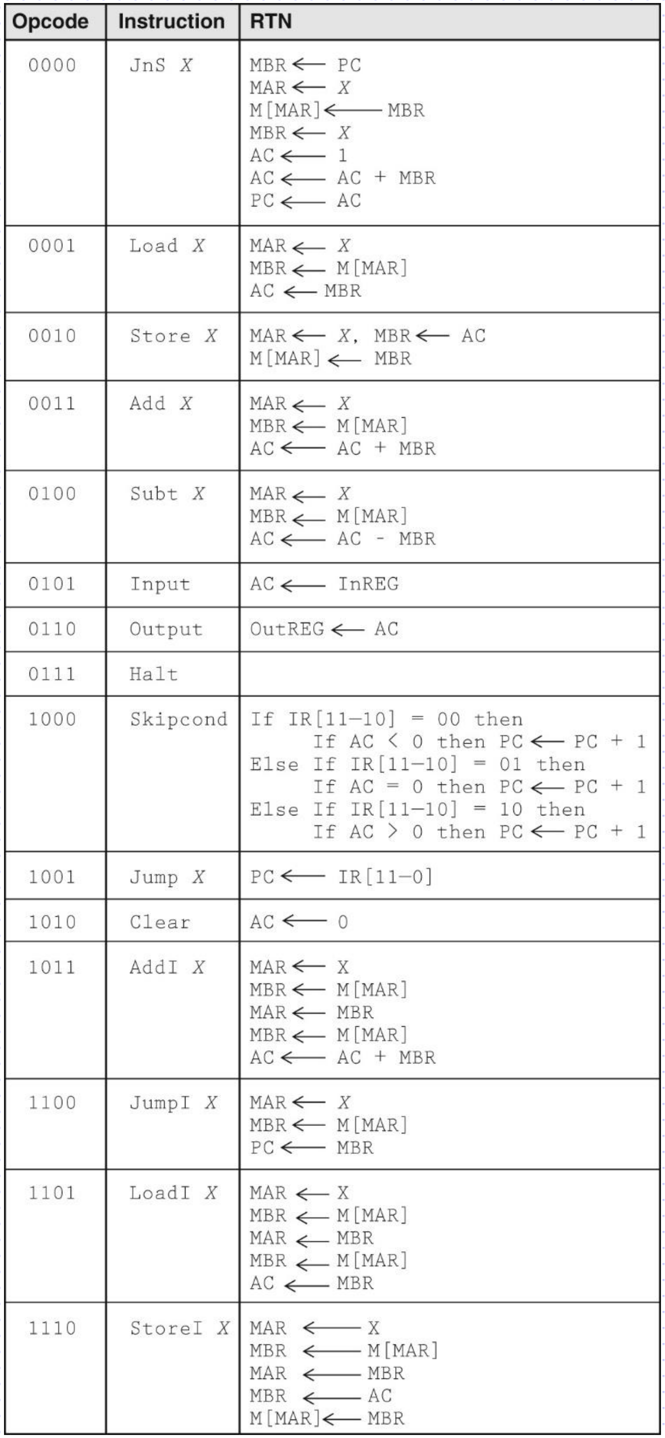
- ○ Instruction format
- ○ Instruction type
- ○ Architecture of data in CPU

Introduction

- This chapter builds upon the ideas up to Chapter 4.
- We present a detailed look at different instruction formats, operand types, and memory access methods.
- We will see the interrelation between machine organization and instruction formats.
- This leads to a deeper understanding of computer architecture in general.

What is an instruction set?

- The complete collection of instructions that are understood by a CPU
- Machine instruction or code (In Binary)
  - Usually represented by assembly codes (or mnemonics)



Opcode	Instruction	RTN

0000	JnS *X*	MBR ← PC
		MAR ← *X*
		M[MAR] ← MBR
		MBR ← *X*
		AC ← 1
		AC ← AC + MBR
		PC ← AC

0001	Load *X*	MAR ← *X*
		MBR ← M[MAR]
		AC ← MBR

0010	Store *X*	MAR ← *X*, MBR ← AC
		M[MAR] ← MBR

0011	Add *X*	MAR ← *X*
		MBR ← M[MAR]
		AC ← AC + MBR

0100	Subt *X*	MAR ← *X*
		MBR ← M[MAR]
		AC ← AC - MBR

0101	Input	AC ← InREG
0110	Output	OutREG ← AC
0111	Halt	
1000	Skipcond	If IR[11–10] = 00 then If AC < 0 then PC ← PC + 1 Else If IR[11–10] = 01 then If AC = 0 then PC ← PC + 1 Else If IR[11–10] = 10 then If AC > 0 then PC ← PC + 1
1001	Jump *X*	PC ← IR[11–0]
1010	Clear	AC ← 0

1011	AddI *X*	MAR ← X
		MBR ← M[MAR]
		MAR ← MBR
		MBR ← M[MAR]
		AC ← AC + MBR

1100	JumpI *X*	MAR ← *X*
		MBR ← M[MAR]
		PC ← MBR

1101	LoadI *X*	MAR ← X
		MBR ← M[MAR]
		MAR ← MBR
		MBR ← M[MAR]
		AC ← MBR

1110	StoreI *X*	MAR ← X
		MBR ← M[MAR]
		MAR ← MBR
		MBR ← AC
		M[MAR] ← MBR

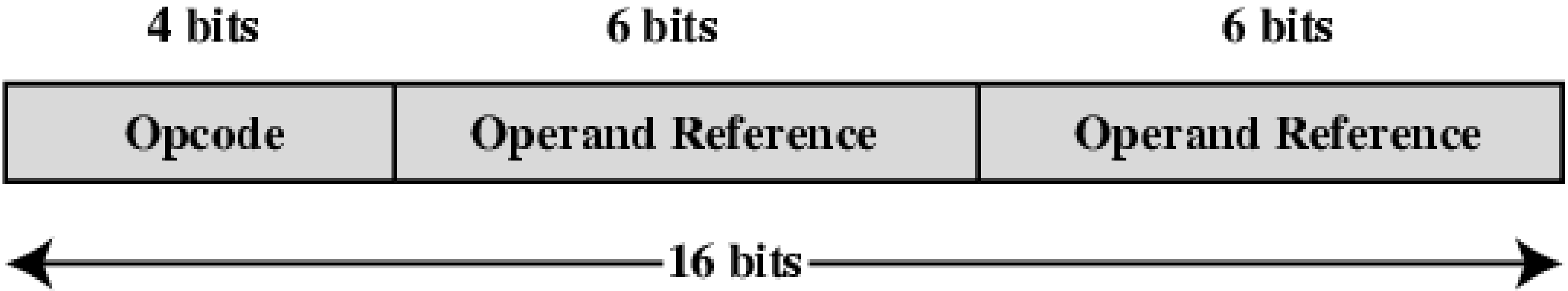
Instruction set architecture of MARIE

Instruction Formats

In designing an instruction set, consideration is given to:

- Instruction length.
  - Whether short, long, or variable.
- Number of operands.
- Number of addressable registers.
- Memory organization.
  - Whether byte- or word addressable.
- Addressing modes.
  - Choose any or all: direct, indirect or indexed.

Sample Instruction Format



4 bits	6 bits	6 bits
Opcode	Operand Reference	Operand Reference

16 bits

Instruction Representation

- In machine code each instruction has a unique bit pattern
  - Divided into fields
- For human, a symbolic representation is used (Called *mnemonics*)
  - e.g. ADD, SUB, LOAD,STORE
- Operands can also be represented symbolically
  - ADD *A,B*
    - *Note that operation is performed on the contents, not on its address!*
- Machine language is rare use, but useful for describing machine instructions

Types of Operand (data referencing)

Machine operations operate on data with the general categories:

- Addresses
- Numbers
  - Integer (fixed point)/packed decimal
  - Limited floating point representation (magnitude vs precision)
- Characters
  - ASCII, EBCDIC etc.
- Logical Data
  - Bits or flags

Instruction types

1. Data transfer or movement (to and from register, memory, I/O)
2. Data processing (arithmetic and logic instructions)
   - Arithmetic: computational capabilities for processing numeric data
   - Logic (Boolean): operate on the bits of a word
     - Perform on data in CPU registers
3. Data storage (main memory)
4. Program flow control (Test and branch)

1.Data Transfer

Specify

- Source
- Destination
- Amount of data
- Mode of addressing

May be different instructions for different movements: L, LH, LR in IBM 370

- (L for Load from memory to register, LH for load halfword, LR for load from register to register)

Or one instruction (MOV) and different addresses

- e.g. VAX → mov op1, op2

VAX’s approach is simpler but less compact

2. Data processing: Arithmetic

- Add, Subtract, Multiply, Divide
- Signed Integer
- Floating point
- Packed decimal
- Maybe single-operand
    - Increment (a++) → add 1 to operand
    - Decrement (a--)
    - Negate (-a)

2. Data processing: Logical

- Bitwise operations
- Boolean operation → AND, OR, NOT
- Binary test → EQUAL
  - (R1)=10100101
(R2)=00001111
  - (R1) AND (R2)=00000101
- Logical shift and rotate

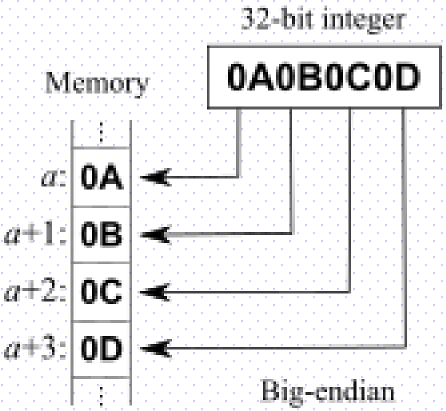
3. Data storage: Byte Order Issues

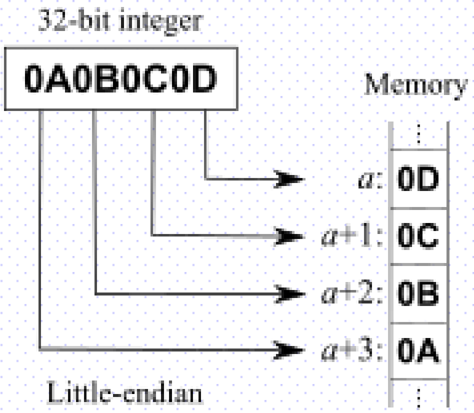
- When keeping data in the storage, Byte ordering, or *endianness*, is another architectural consideration.
- What order do we read numbers that occupy more than one byte (multibyte value)
- e.g. (numbers in hex to make it easy to read)
  - 123456AB can be stored in 4 x 8bit locations as follows

To keep 123456AB at address 184, the byte order can be:

Address	Big-endian	Little-endian
184	12	AB
185	34	56
186	56	34
187	AB	12

- i.e. read top down (big-endian) or bottom up (little-endian)?



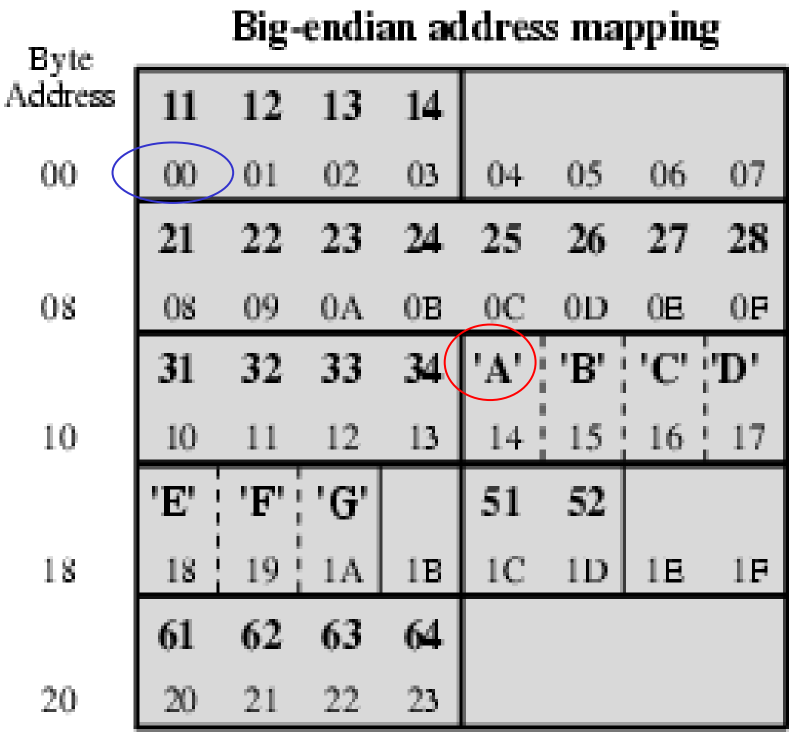


Byte Order: Big- vs Little-Endian

- The system with the least significant byte in the lowest address is called *little-endian*
  - *Most processors follow this, e.g. Intels, ARM, AMD*
  - *More and more of systems are little-endian*
- The system with the most significant byte at the lowest address is called *big-endian*
  - *Networking, e.g. TCP/IP follows this.*
  - This is a concern if you have to work on the lower network layer!
- The system can handle both, called *bi-endian*
  - *ARM v.3.0 upwards*

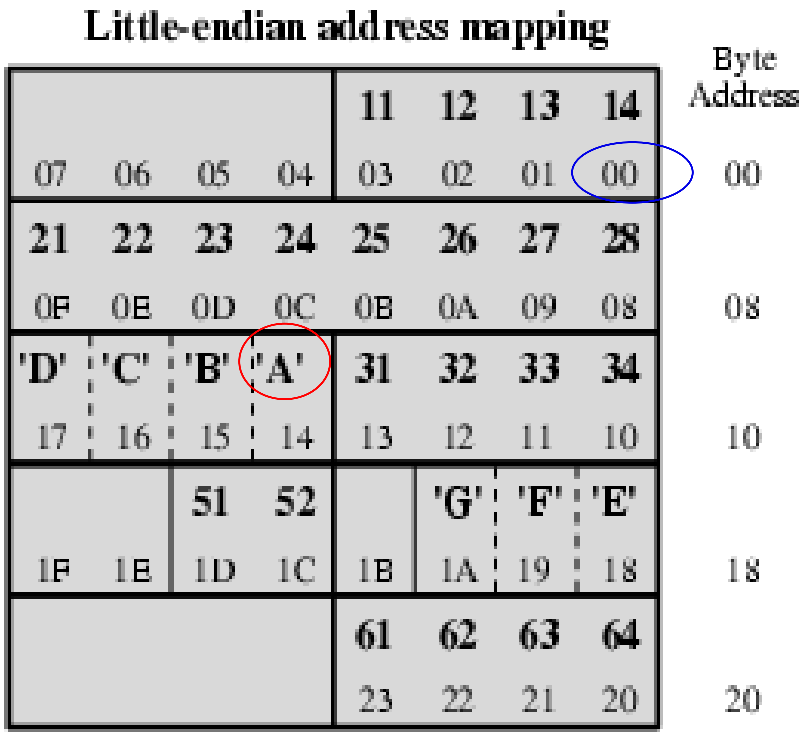
Example of C Data Structure (a 32-bit word)

```
struct{
   int     a;     //0x1112_1314                   word
   int     pad;   //
   double  b;     //0x2122_2324_2526_2728         doubleword
   char*   c;     //0x3132_3334                   word
   char    d[7];  //'A'.'B','C','D','E','F','G'   byte array
   short   e;     //0x5152                        halfword
   int     f;     //0x6161_6364                   word
} s;
```



Big-endian address mapping

Byte Address								
00	11 00	12 01	13 02	14 03	04	05	06	07
08	21 08	22 09	23 0A	24 0B	25 0C	26 0D	27 0E	28 0F
10	31 10	32 11	33 12	34 13	'A' 14	'B' 15	'C' 16	'D' 17
18	'E' 18	'F' 19	'G' 1A	1B	51 1C	52 1D	1E	1F
20	61 20	62 21	63 22	64 23				



Little-endian address mapping

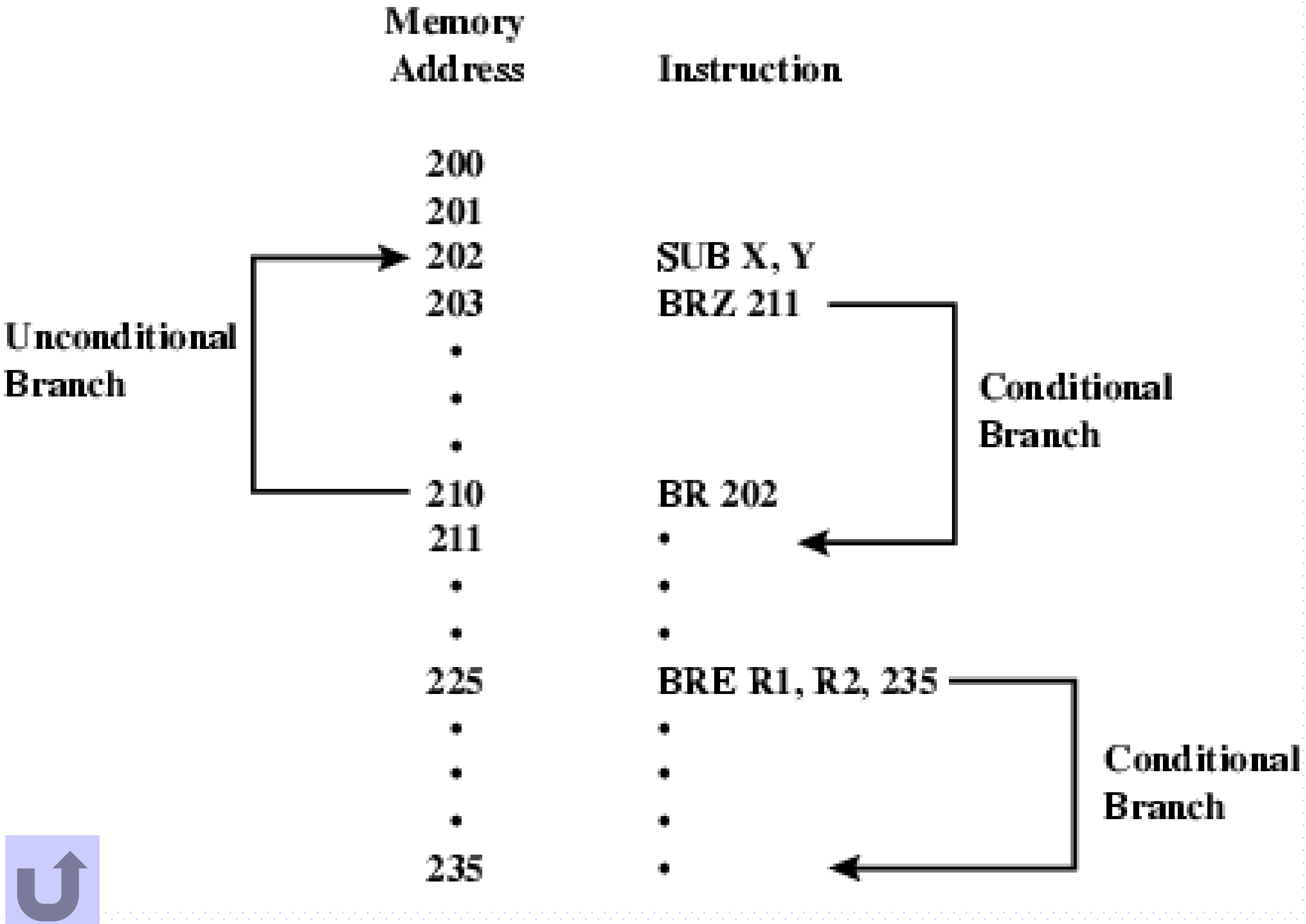
								Byte Address
07	06	05	04	11 03	12 02	13 01	14 00	00
21 0F	22 0E	23 0D	24 0C	25 0B	26 0A	27 09	28 08	08
'D' 17	'C' 16	'B' 15	'A' 14	31 13	32 12	33 11	34 10	10
1F	1E	51 1D	52 1C	1B	'G' 1A	'F' 19	'E' 18	18
				61 23	62 22	63 21	64 20	20

Good and bad points for Big and Little Endian

- Big endian:
  - Is more natural.
  - The sign of the number can be determined by looking at the byte at address offset 0.
  - Strings and integers are stored in the same order
- Little endian:
  - Makes it easier to place and deal with values on non-word boundaries, e.g. multiple-precision math.
  - Conversion from a 16-bit integer address to a 32-bit integer address does not require any arithmetic.
  - Easy to check odd or even number

Ref: http://people.cs.umass.edu/~verts/cs32/endian.html

Transfer of Control: Branch Instruction



Memory Address	Instruction
200	
201	
202	SUB X, Y
203	BRZ 211
•	
•	
•	
210	BR 202
211	•
•	•
•	•
225	BRE R1, R2, 235
•	•
•	•
•	•
235	•

Transfer of Control

See Figure
on the next page

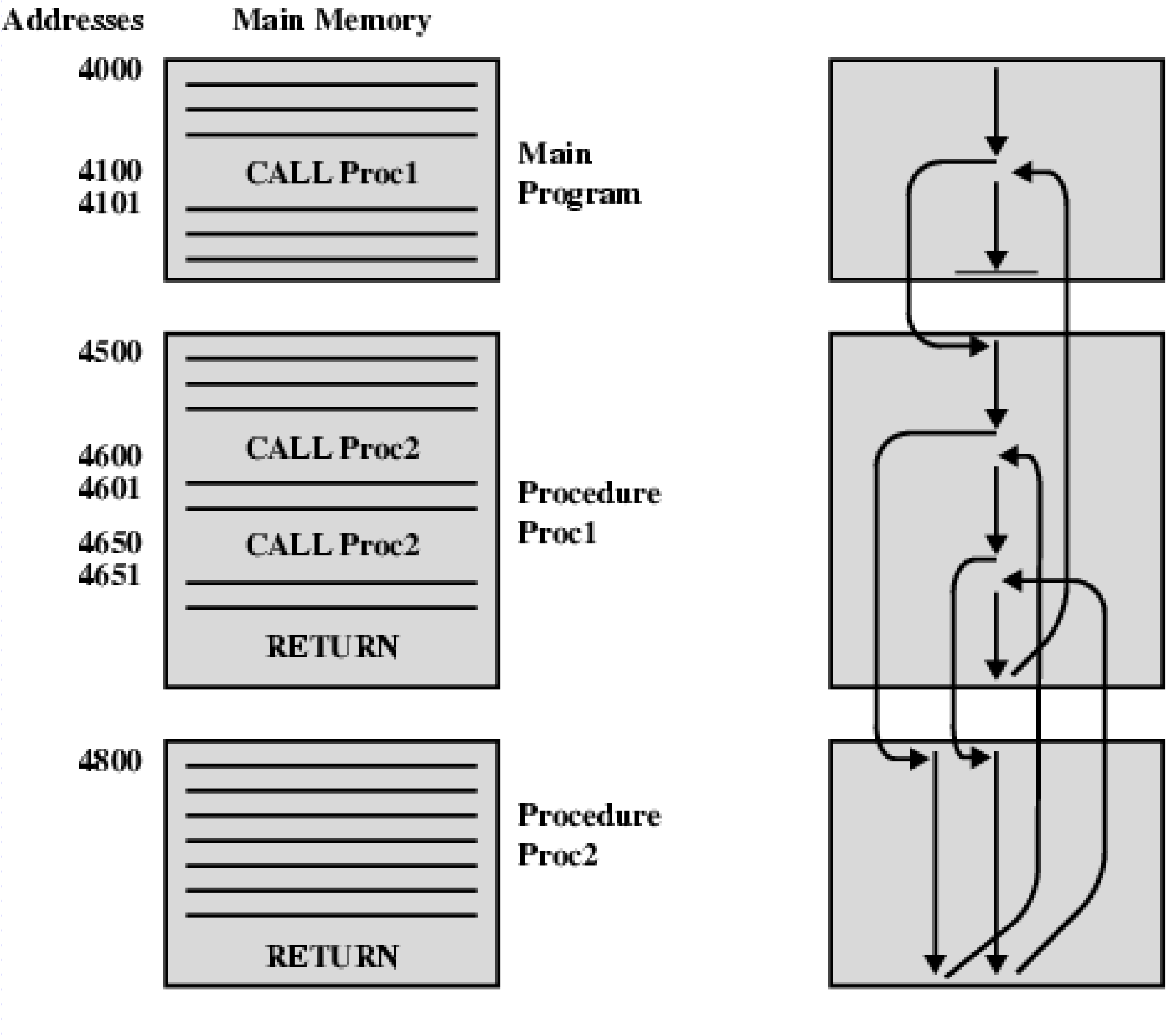
- Branch (or jump instruction)
  - e.g. branch to x if result is zero [*BRZ*]
  - Branch to x if contents of R1 = contents of R2 [*BRE*]
- Skip
  - e.g. increment and skip if zero
    - ISZ Register1
- Procedure call
  - A self-contained program incorporated into a larger program
  - For the economy and modularity
  - Involve 2 basic instructions: call and return (both are forms of branching)

Nested Procedure Calls

Note:
1. Can be called from > 1 location

2. Can be nested

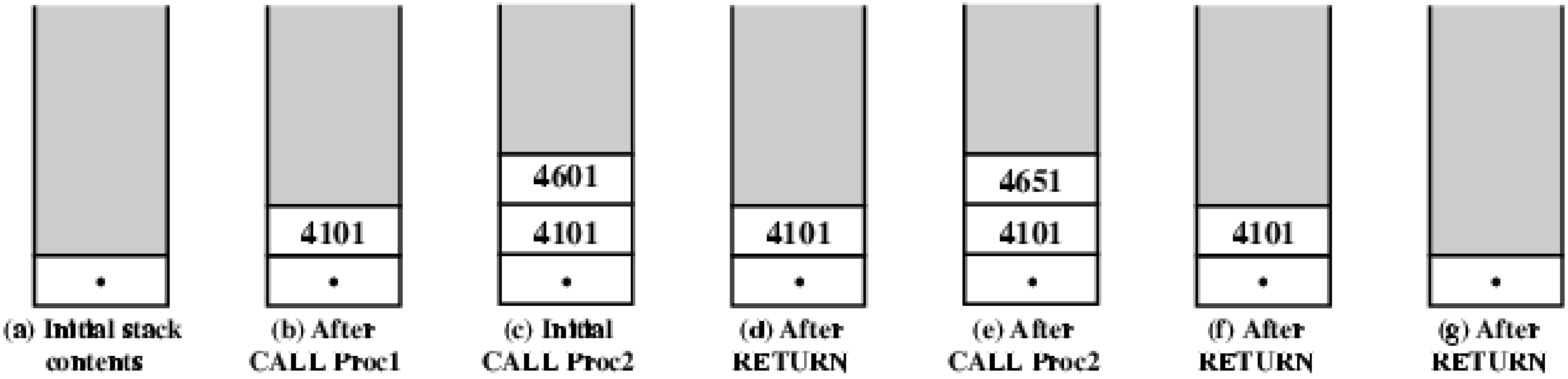
3. Each call is matched by a return
4. May use stack to control the calls and return



(a) Calls and returns

(b) Execution sequence

Use of Stack



(a) Initial stack contents

(b) After CALL Proc1

(c) Initial CALL Proc2

(d) After RETURN

(e) After CALL Proc2

(f) After RETURN

(g) After RETURN

(1) To manage (recursive) procedure call

(2) Alternate form of addressing

- Basic operation : Push, Pop
- Typical implemented to grow from higher addresses to lower

Architecture for Data in CPU

- The next consideration for architecture design concerns how the CPU will store data.

  1. A stack architecture
  2. An accumulator architecture (as seen in simple CPU like MARIE)
  3. A general purpose register architecture.

- In choosing one over the other, the tradeoffs are simplicity (and cost) of hardware design with execution speed and ease of use.

Instruction Formats: GPR

- Most systems today are GPR systems.
- There are three types:
  - Memory-memory where two or three operands may be in memory.
  - Register-memory where at least one operand must be in a register.
  - Load-store where no operands may be in memory.
- The number of operands and the number of available registers has a direct affect on instruction length.

Instruction Formats: Stacking

- Stack machines use one - and zero-operand instructions.
- **`LOAD`** and **`STORE`** instructions require a single memory address operand.
- Other instructions use operands from the stack implicitly.
- **`PUSH`** and **`POP`** operations involve only the stack’s top element.
- Binary instructions (e.g., **`ADD`**, **`MULT`**) use the top two items on the stack.

Instruction Formats:
Number of addresses in ISA

- Let's see how to evaluate an infix expression using different instruction formats.
- With a three-address ISA, (e.g.,mainframes), the infix expression,

  Z = X × Y + W × U

  might look like this:

```
MULT R1,X,Y
MULT R2,W,U
ADD  Z,R1,R2
```

- 3 addresses
  - Result, Operand 1, Operand 2

Instruction Formats

- In a two-address ISA, (e.g.,Intel, Motorola), the infix expression,

  Z = X × Y + W × U

  might look like this:

```
LOAD  R1,X
MULT  R1,Y
LOAD  R2,W
MULT  R2,U
ADD   R1,R2
STORE Z,R1
```

Note: Two-address ISAs usually require one operand to be a register.

Instruction Formats

- In a one-address ISA, like MARIE, the infix expression,

  Z = X × Y + W × U

  looks like this:

```
LOAD X
MULT Y
STORE TEMP
LOAD W
MULT U
ADD TEMP
STORE Z
```

- **This can be implemented based on Accumulator architecture.**

Instruction Formats

- In a stack ISA (zero-addressing), the postfix expression,

  Z = X Y × W U × +

  might look like this:

```
PUSH X
PUSH Y
MULT
PUSH W
PUSH U
MULT
ADD
POP Z
```

Instruction Formats

- Stack architectures require us to think about arithmetic expressions a little differently.
- We are accustomed to writing expressions using *infix* notation, such as: Z = X + Y.
- Stack arithmetic requires that we use *postfix* notation: Z = XY+.
  - This is also called *reverse Polish notation*, (somewhat) in honor of its Polish inventor, Jan Lukasiewicz (1878 - 1956).

Instruction Formats

- The principal advantage of postfix notation is that parentheses are not used.
- For example, the infix expression,

  Z = (X × Y) + (W × U),

  becomes:

  Z = X Y × W U × +

  in postfix notation.

Instruction Formats

- Example: Convert the infix expression (2+3) - 6/3 to postfix:

2 3+ - 6/3	The sum 2 + 3 in parentheses takes precedence; we replace the term with 2 3 +.

Instruction Formats

- Example: Convert the infix expression (2+3) - 6/3 to postfix:

2 3+ - 6 3/

The division operator takes next precedence; we replace 6/3 with 6 3 /.

Instruction Formats

- Example: Convert the infix expression (2+3) - 6/3 to postfix:

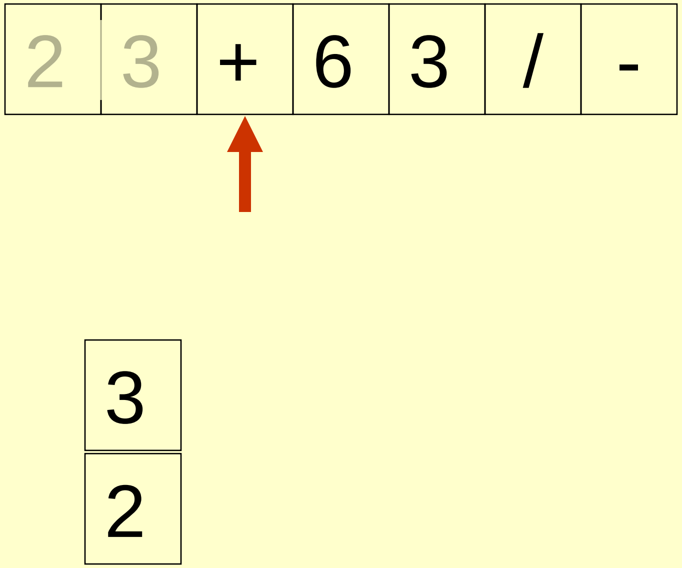
2 3+ 6 3/ -

The quotient 6/3 is subtracted from the sum of 2 + 3, so we move the - operator to the end.

Instruction Formats

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

Scanning the expression from left to right, push operands onto the stack, until an operator is found



2	3	+	6	3	/	-

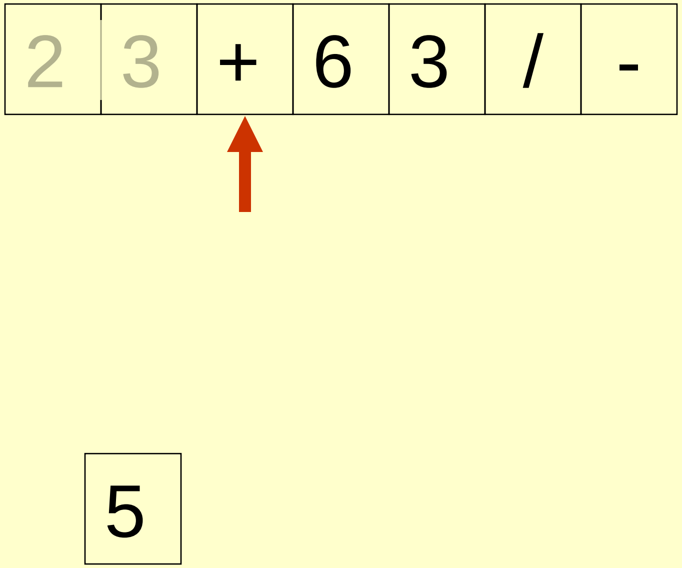
3

2

Instruction Formats

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

Pop the two operands and carry out the operation indicated by the operator. Push the result back on the stack.



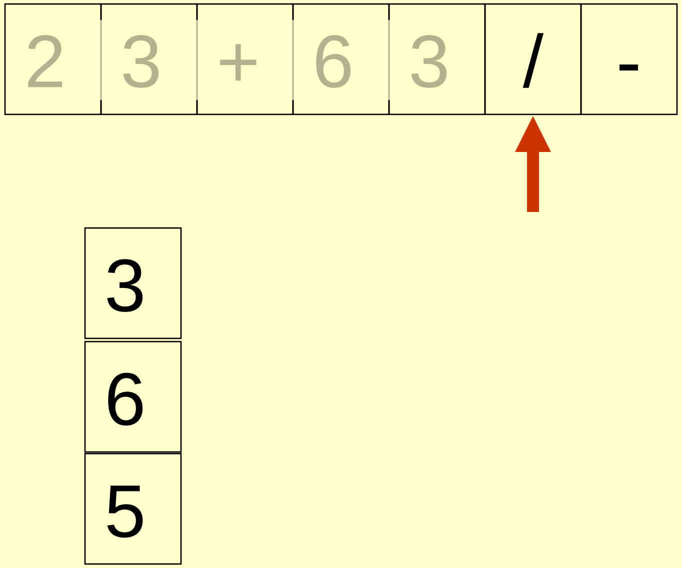
2	3	+	6	3	/	-

5

Instruction Formats

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

Push operands until another operator is found.



2	3	+	6	3	/	-

3

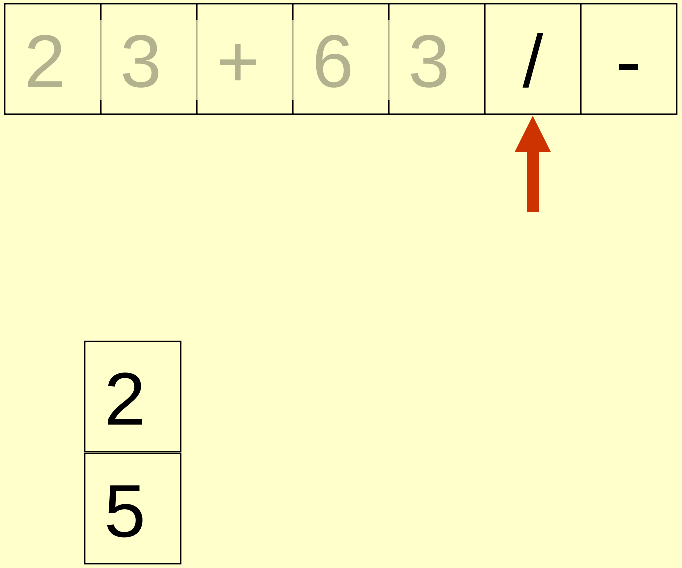
6

5

Instruction Formats

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

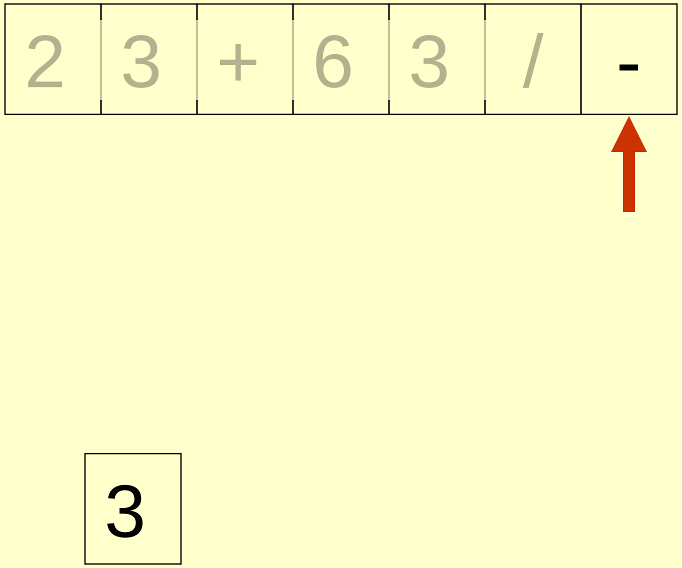
Carry out the operation and push the result.



Instruction Formats

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

Finding another operator, carry out the operation and push the result.
The answer is at the top of the stack.



2	3	+	6	3	/	-

3

Pros and cons of different architecture for data

- In a stack architecture, instructions and operands are implicitly taken from the stack.
    - But! A stack cannot be accessed randomly.
- In an accumulator architecture, one operand of a binary operation is implicitly in the accumulator.
    - One operand is in memory, creating lots of bus traffic.
- In a general purpose register (GPR) architecture, registers can be used instead of memory.
    - Faster than accumulator architecture.
    - Efficient implementation for compilers.
    - Results in longer instructions
- Most systems today are GPR
    - https://datacadamia.com/computer/cpu/register/general#cpu_register_-_general_purpose_register_gpr

Conclusion

- A set of factor impacting instruction set designed were reviewed.
- Instruction format can be fixed or variable to suit the need
- 4 Instruction type can be:
  - Data movement, processing, storing, and flow control
- Architecture of data in CPU
  - Accumulator-based, GPR, and stack-based.