Module 6

Contents

- Addressing Modes
- Types and sizes of operands
- Types of Instruction

Addressing Modes [1]

- A machine instruction can find its **operand** in one of three places:

I. As a part of the instruction (Immediate addressing mode)
II. In a general-purpose register
III. In memory

- Addressing modes refers to the way in which the operand of an instruction is specified.

- Consequently, addressing modes can be broadly classified into:

(a) *direct* – the address field specifies the operand address and
(b) *indirect*– the address field specifies a location that contains the operand address.

I. Immediate Addressing Mode

- Operand is a constant encoded in the instruction itself

```
Add R4, #3        /* R4 <- R4 + 3
```

- No memory reference to fetch data

- Fast, but Limited range

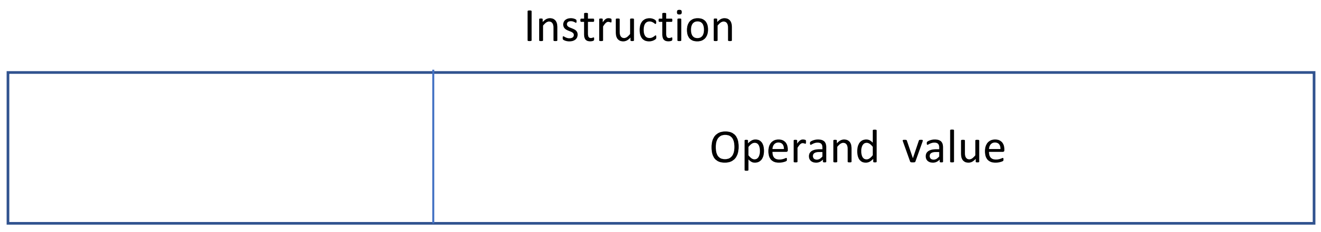
- Used to define and use constants or set initial values of variables

- Disadvantage:
- Size of the operand is limited to the size of address field of instruction. In most instruction set address field of operand is small compared to instruction size.

Immediate Addressing Mode

- Operand is a constant encoded in the instruction itself

```
Add R4, #3
```
/* R4 <- R4 + 3



Immediate mode

Direct Addressing Mode

–**Direct (Absolute):** Address is a constant ,encoded in the instruction

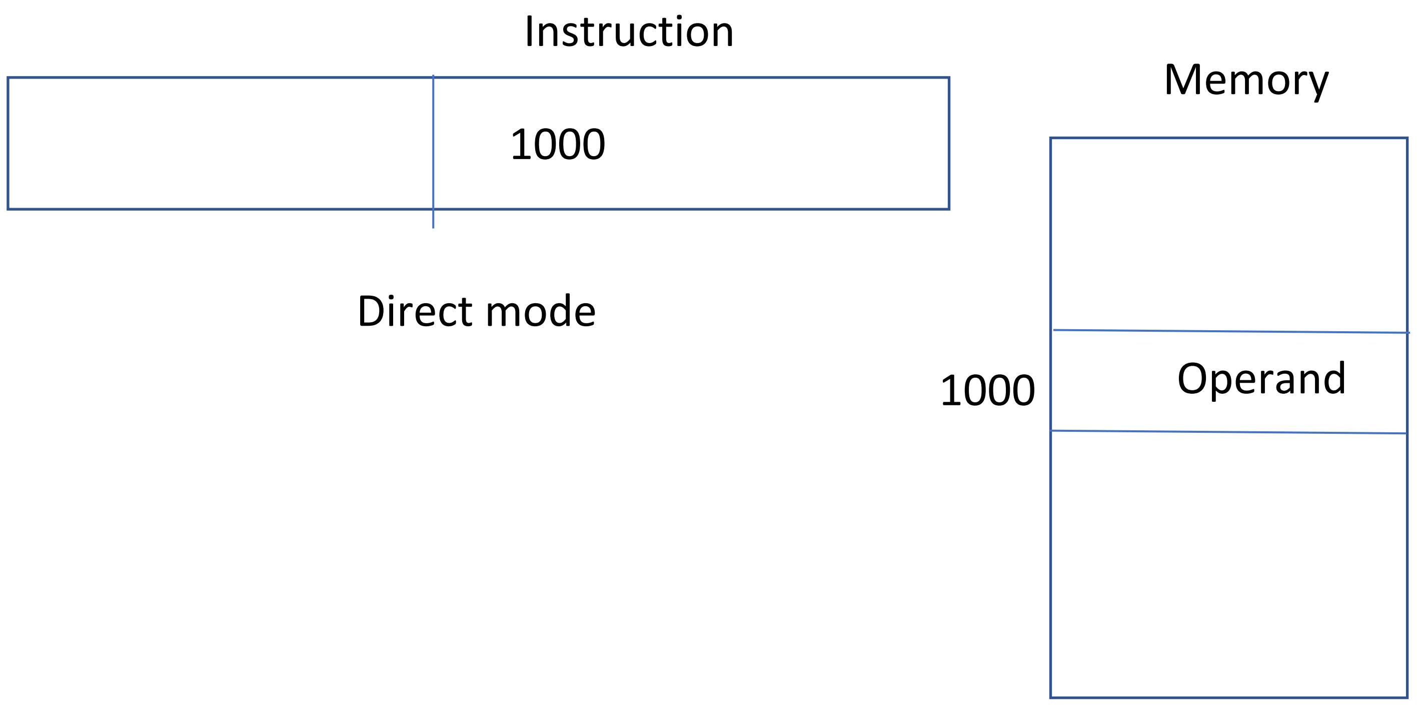
- `Add R1,(1001)` /* R1 =R1 + M[1001]
- + Single memory reference to access data
- + No additional calculations to work out effective address
- - Limited address space

- **Use:** accessing static data

Direct Addressing Mode

–**Direct (Absolute):** Address is a constant ,encoded in the instruction

– `Add R1,(1001)` /* R1 =R1 + M[1001]



Indirect Addressing Mode

–**Why Indirect addressing?**

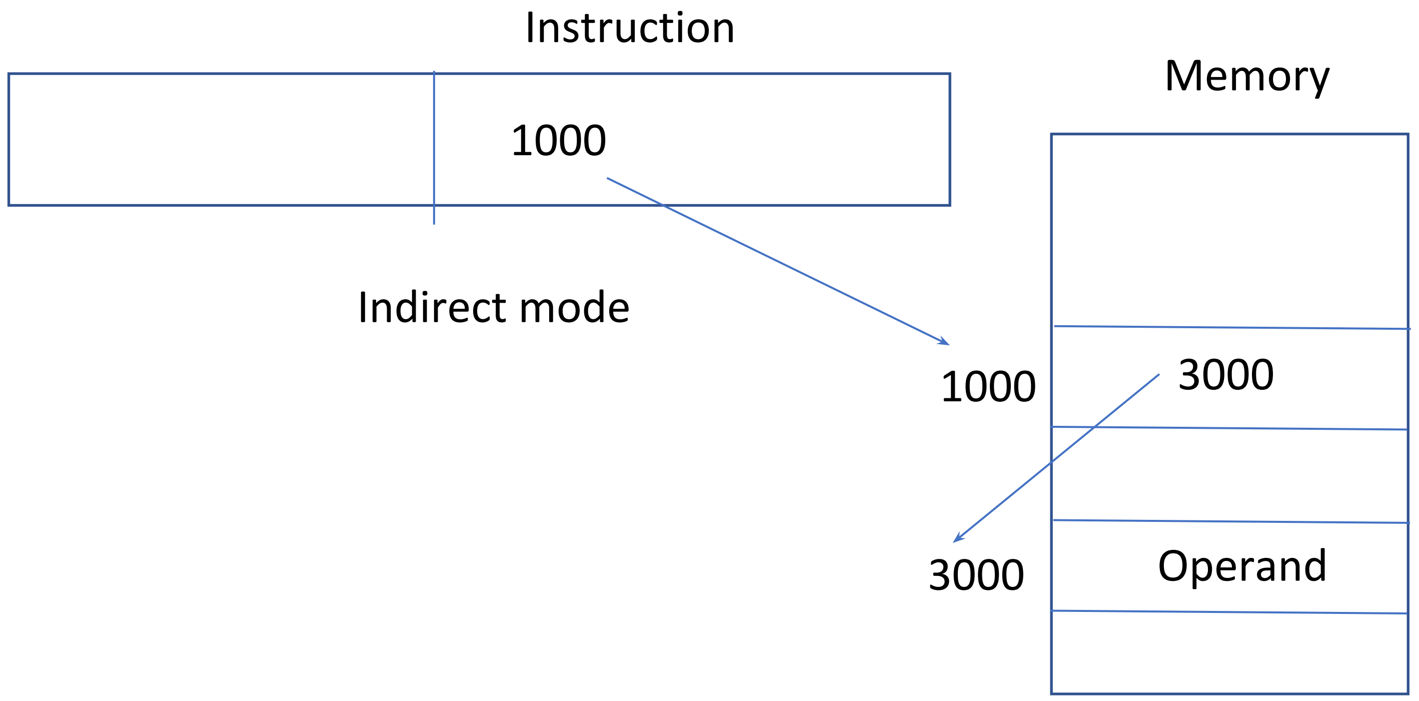
- **In direct addressing length of address field of operand is less than the word size, so it provides limited address space.**
- **Using indirect addressing mode, full-length address of operand can be used.**

- **Indirect :** effective address is in the memory, and corresponding memory address will be maintained in the address field of an instruction.
  - `Add R1,@1001` /* R1 =R1 + M[[1001]]

–+

–+

Indirect Addressing Mode



Two memory reference to access data

— + For word length N, available address space is 2^N

— + The number of different effective addresses that may be referenced at any one time is limited to 2^*K, where K is the length of the address field*

Register Addressing Mode

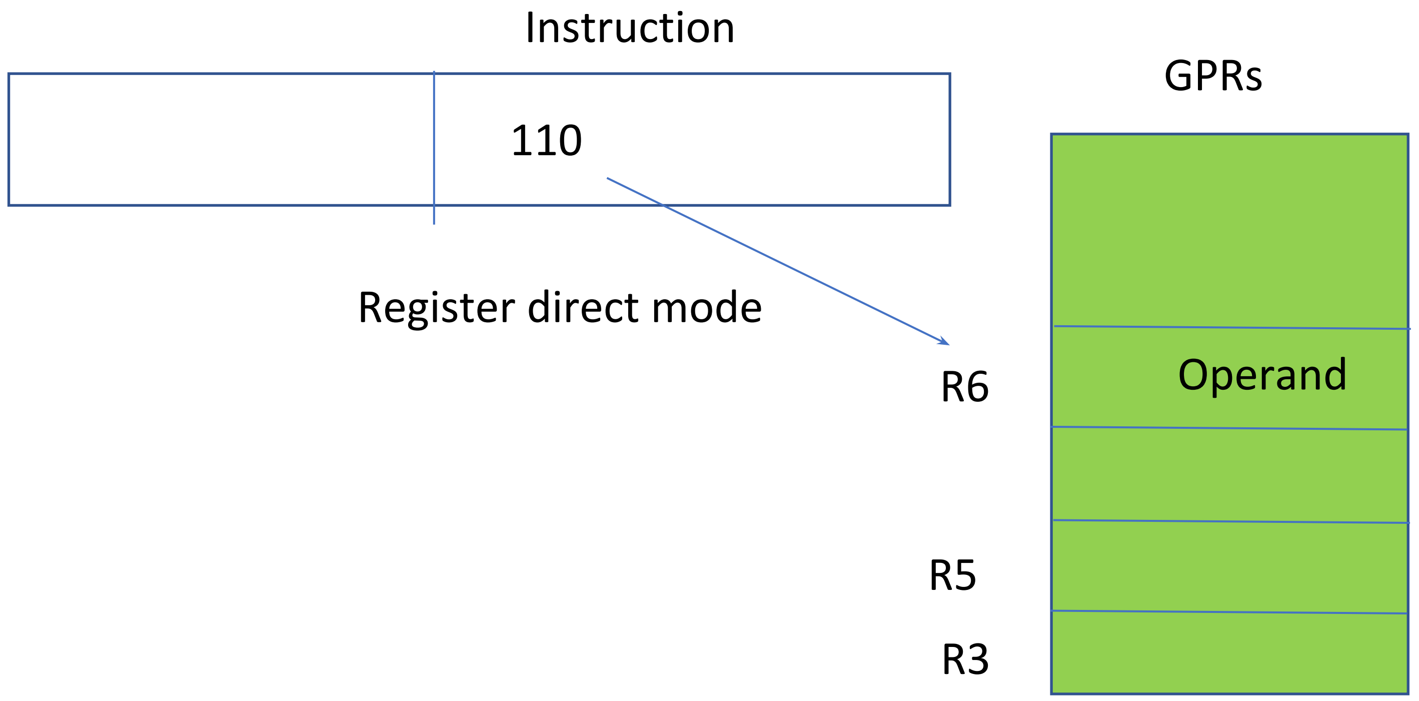
Register Mode: Operand is held in register named in the address filed of the instruction

`Add R4,R3` /* R4=R4+R3

If content of address field of instruction is 6 then register R6 is the intended address, and the operand value is contained in R5

Generally, 3 to 5 bits is used in address field, so only 8 to 32 register can be referenced.

Register Addressing Mode



Small address field is needed in instruction

No memory reference

Limited address space

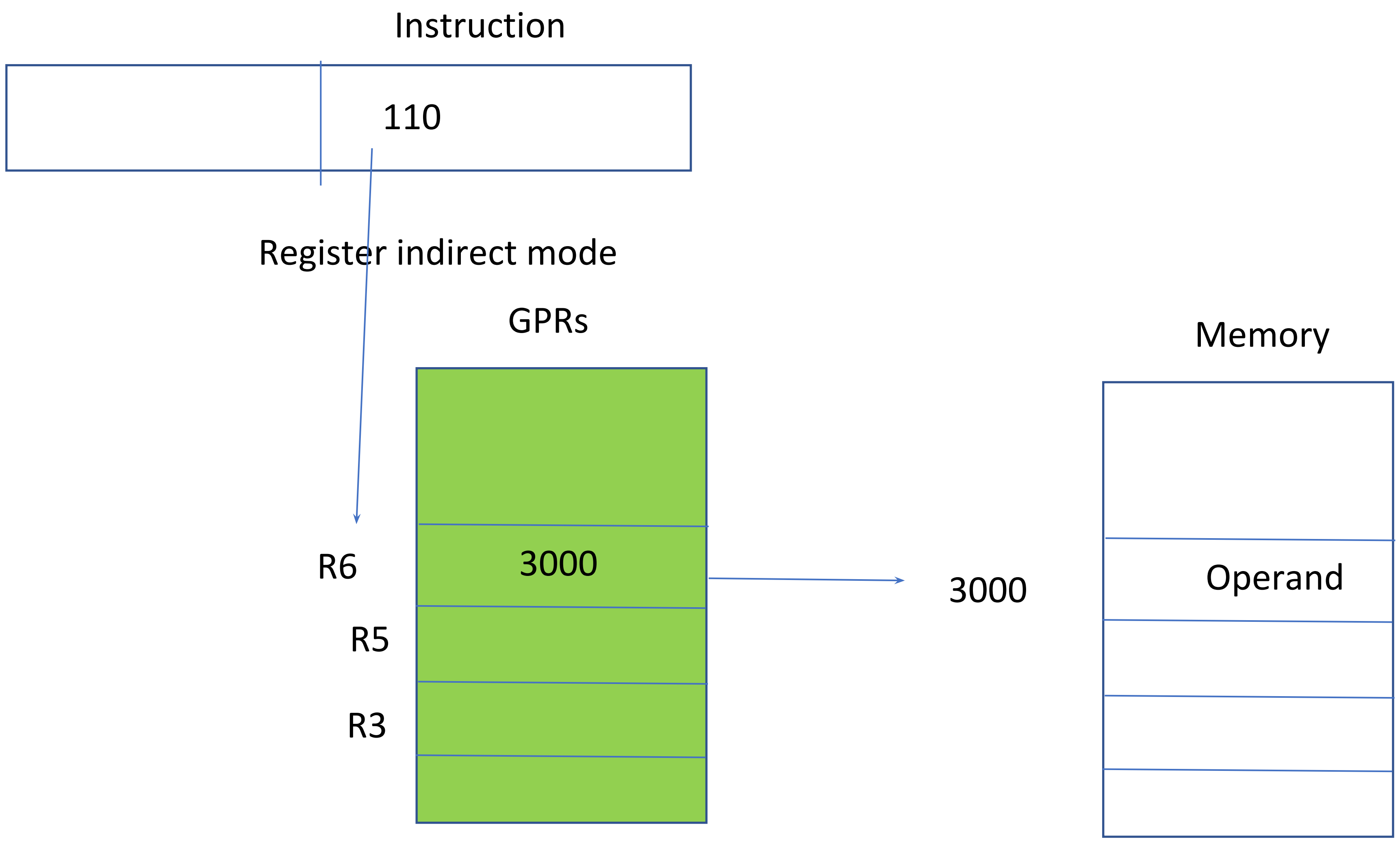
Register Indirect Addressing Mode

Register Indirect Mode: As register addressing is similar to direct addressing, **register indirect addressing** is similar to indirect addressing

```
Add R4,(R1)
```
/* R4=R4+M[R1]

+ Large address space
- Multiple memory accesses to find operand, Hence slower
- **Use:** accessing a pointer or computed address

Register Indirect Addressing Mode



Register Indirect Addressing Mode

e advantages and limitations of register indirect addressing are basically the same as
indirect addressing.

both cases, the address space limitation (limited range of addresses) of the address
ld is overcome by having that field refer to a word- length location containing an
dress. In addition, register indirect addressing uses one less memory reference than
lirect addressing[4].

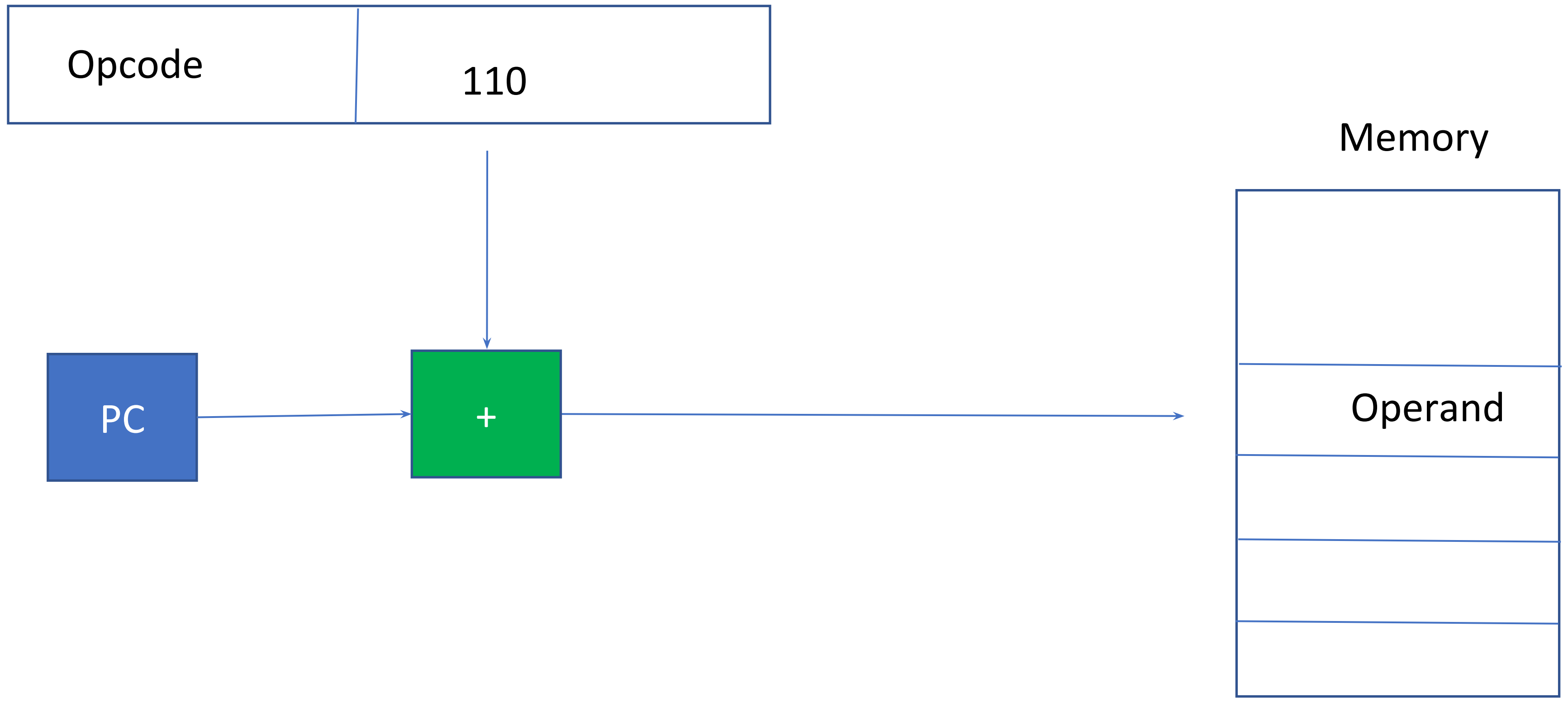
$$\begin{tikzcd}
 & \text{Relative addressing} \\
\textbf{Displacement Addressing} \arrow[ur] \arrow[r] \arrow[dr] & \text{Base-register addressing} \\
 & \text{Indexing}
\end{tikzcd}$$

Combines the capabilities of direct addressing and register indirect addressing

Relative Addressing

Content of the program counter is added to the address part of the instruction in order to obtain the effective address.

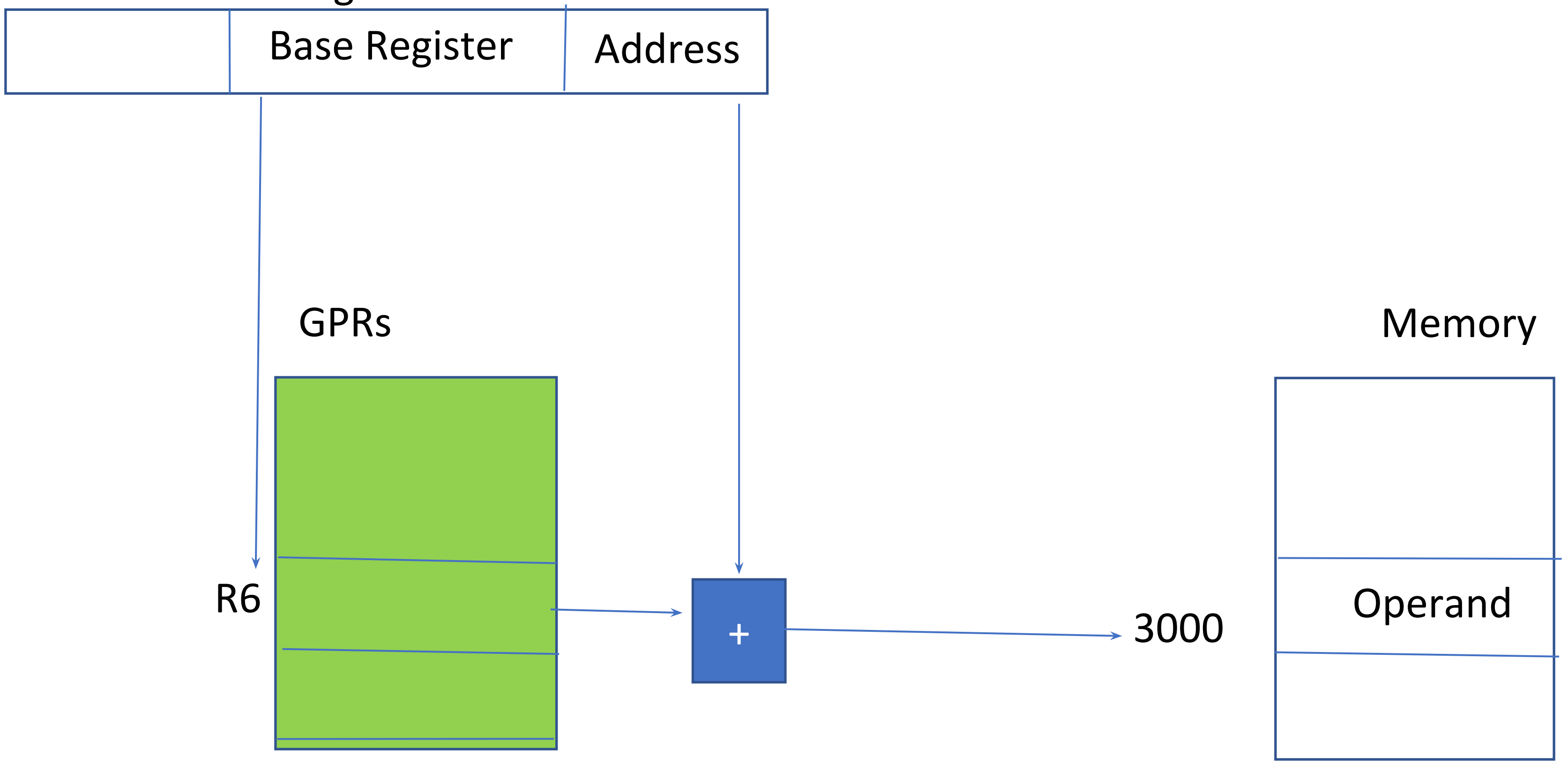
Effective address= PC + Address field value



Base-Register Addressing

Content of a base register is added to the address part of the instruction to obtain the effective address.

EA= Base register + Address field value. Load R1 20(R2) then EA=20+[R2] , R2 is base register



Base-Register Addressing

It is use to implement segmentation.

The base register holds the base address of segment and address field of instruction holds displacement.

Address field size= K

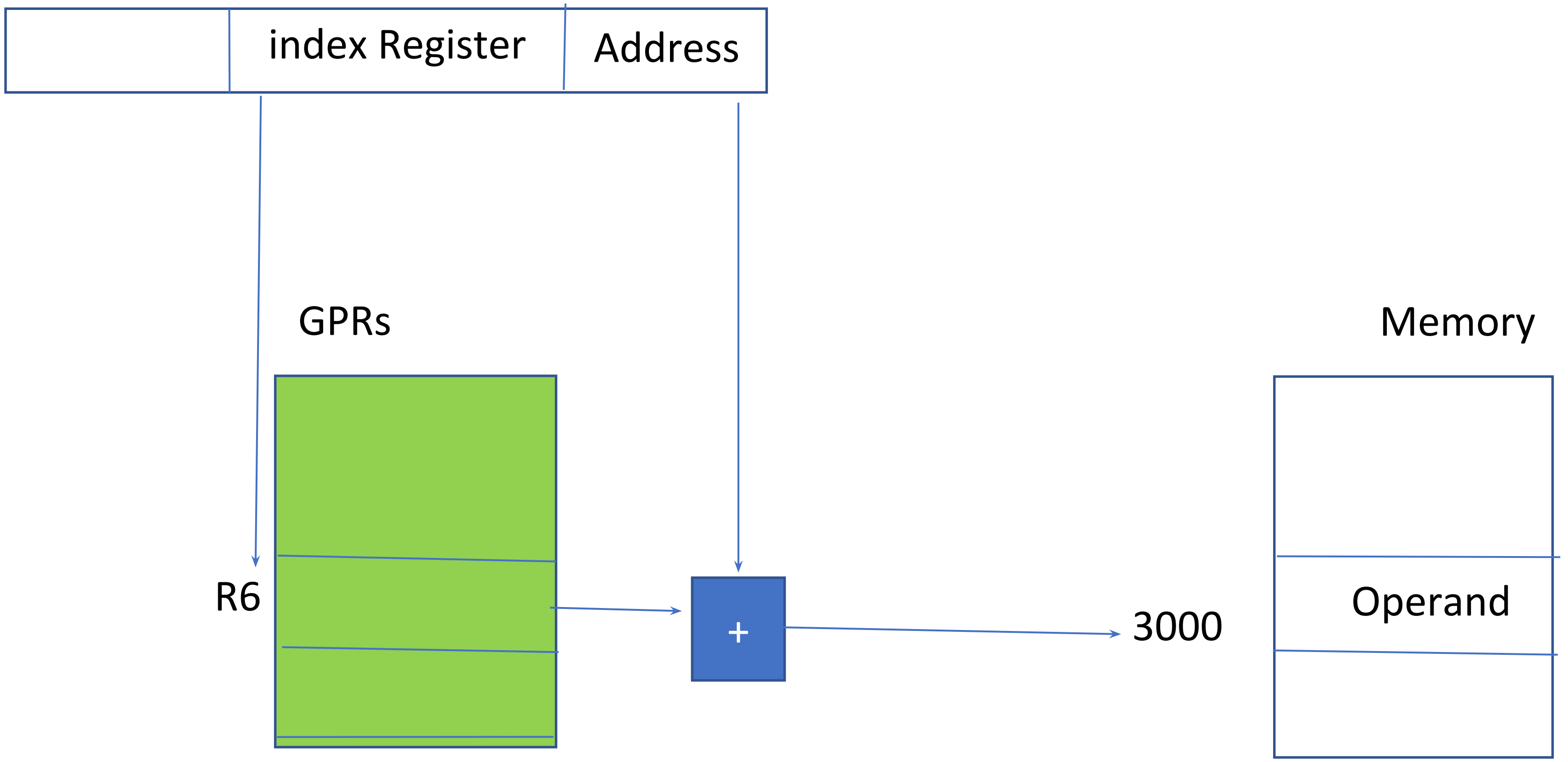
No of possible register = N

Then one instruction can reference any one of *N areas of* 2^K *words.*

Index-Register Addressing

Content of a index register is added to the address part of the instruction to obtain the effective address.

EA= index register + Address field value.   Load   R1 20(R2) then EA= 20+[R2], R2 is index register



Index-Register Addressing

The address field references a main memory address, and the referenced register contains a positive displacement from that address.

Consider, for example, a list of numbers stored starting at location A.

Suppose that we would like to add 1 to each element on the list. We need to fetch each value, add 1 to it, and store it back

The sequence of effective addresses that we need is A, A + 1, A + 2, . . . , up to the last location on the list.

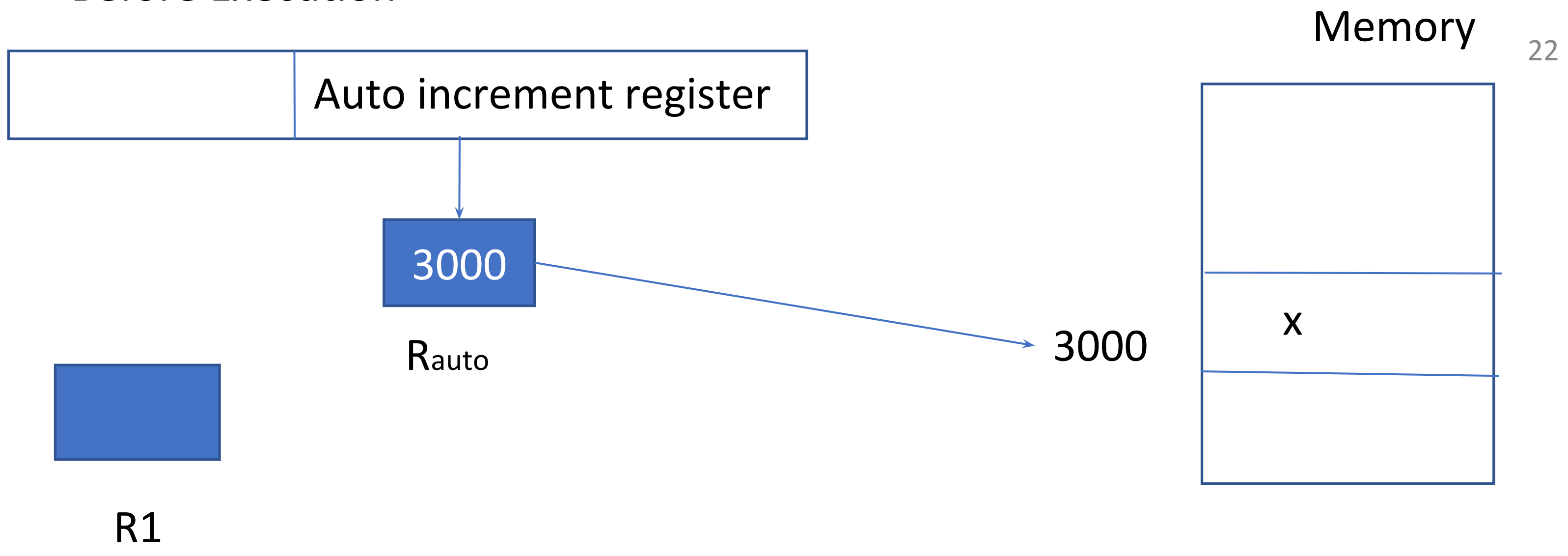
With indexing, this is easily done. The value A is stored in the instruction's address field, and the chosen register, called an *index register, is initialized to 0. After each* operation, the index register is incremented by 1.

Auto Indexed (increment mode):

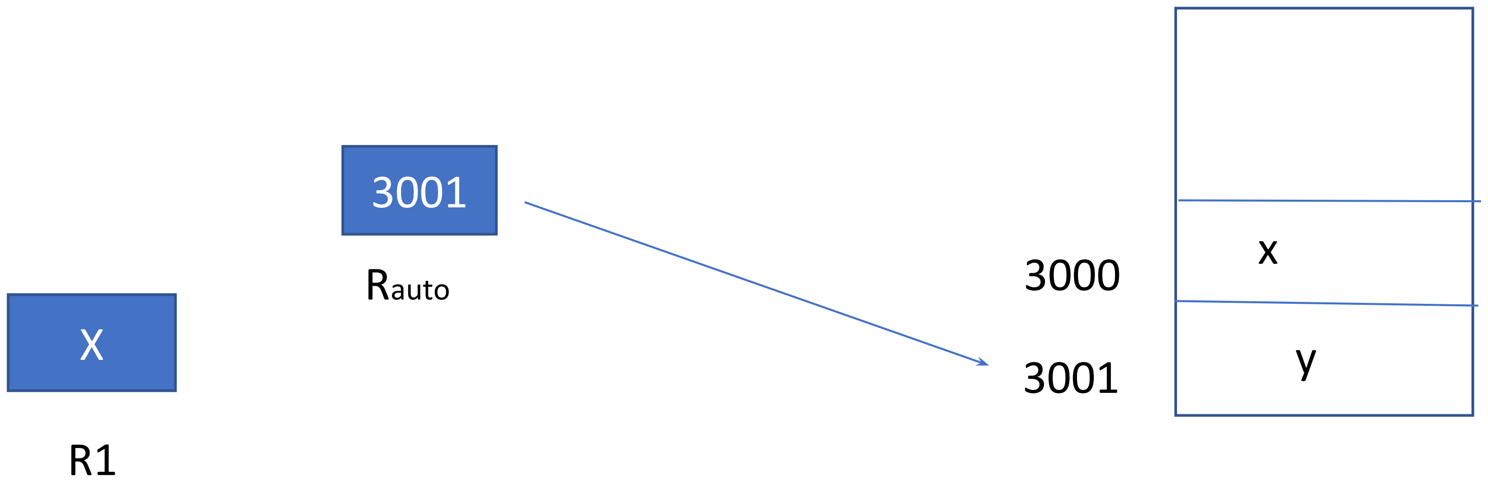
Effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next consecutive memory location.**(R1)+**.
Here one register reference, one memory reference and one ALU operation is required to access the data.

Example: `Add R1, (R2)+  // OR`
```
R1 = R1 +M[R2]
R2 = R2 + d
```
Useful for stepping through arrays in a loop. R2 – start of array d – size of an element

Before Execution



After Execution



Auto Indexed (decrement mode):

Effective address of the operand is the contents of a register specified in the instruction. Before accessing the operand, the contents of this register are automatically decremented to point to the previous consecutive memory location. –**(R1)**

Here one register reference, one memory reference and one ALU operation is required to access the data.

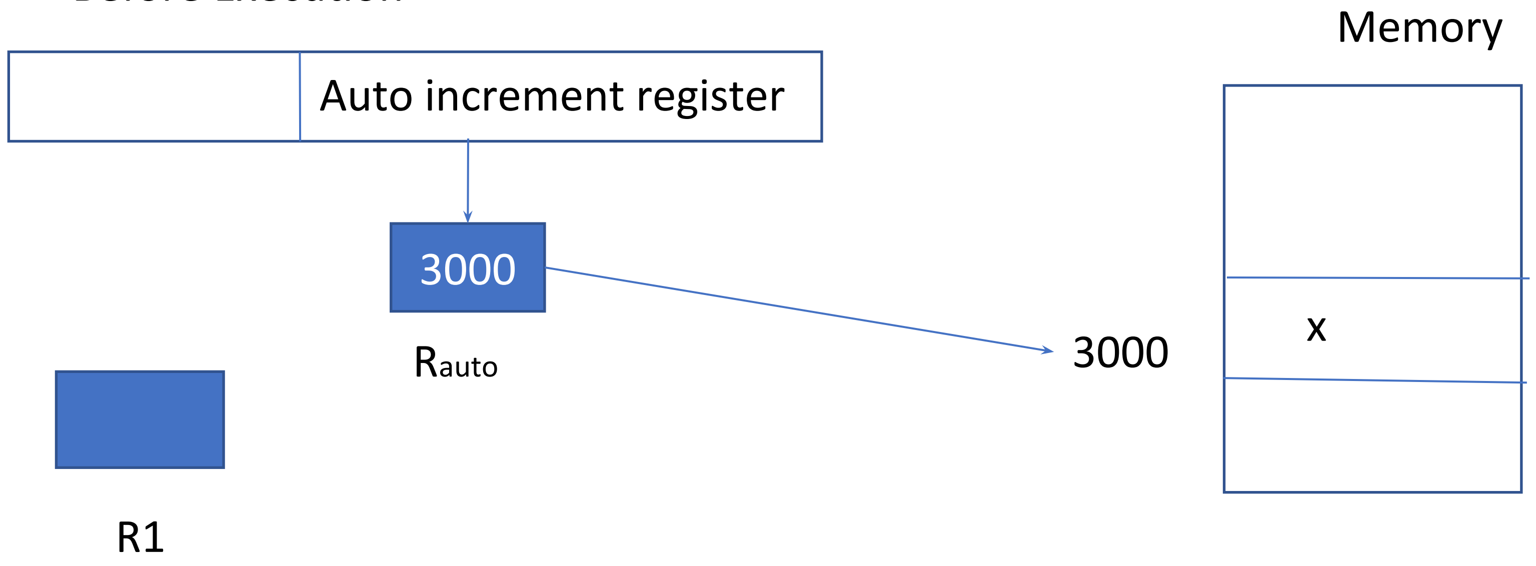
Example:

Add R1,-(R2) //OR

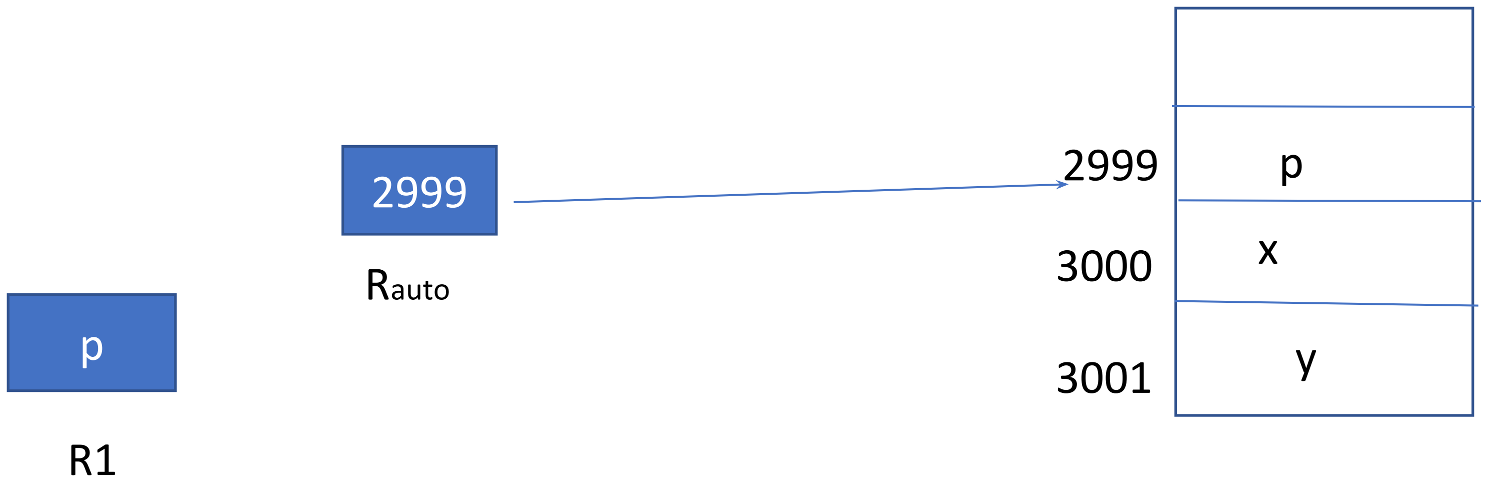
R2 = R2-*d*

R1 = R1 + M[R2]

Before Execution



After Execution



Stack Addressing

The stack is a reserved block of locations. Items are appended to the top of the stack so that, at any given time, the block is partially filled.

Associated with the stack is a pointer whose value is the address of the top of the stack.

Alternatively, the top two elements of the stack may be in processor registers, in which case the stack pointer references the third element of the stack.

The stack pointer is maintained in a register. Thus, references to stack locations in memory are in fact register indirect addresses.

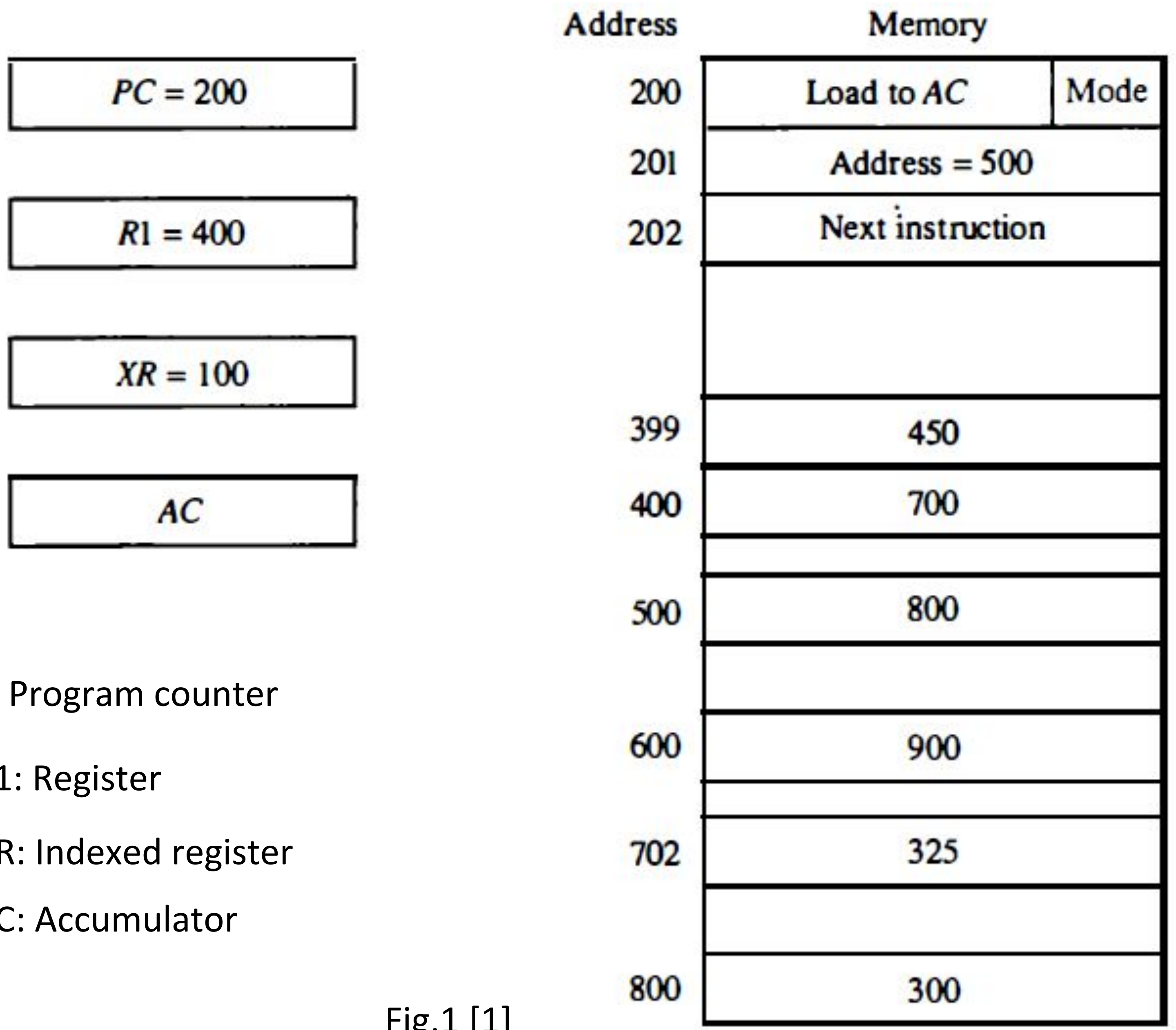
Summarizing All Addressing Modes

Addressing Mode	Example Instruction	Meaning	When Used
Register	Add R4, R3	R4 □ R4 + R3	When a value is in a register
Immediate	Add R4, #3	R4 □ R4 + 3	For constants
Displacement	Add R4, 100(R1)	R4 □ R4 + Mem[100+R1]	Accessing local variables
Register deferred *or* Indirect	Add R4, (R1)	R4 □ R4 + Mem[R1]	Accessing using pointer or computed address
Indexed	Add R3, (R1+R2)	R3 □ R3 + Mem[R1+R2]	Array addressing; R1 = base of array, R2 = index amount
Direct or Absolute	Add R1, (1001)	R1 □ R1 + Mem[1001]	Accessing static data; addr. constant may need to be big

Summarizing All Addressing Modes

Addressing Mode	Example Instruction	Meaning	When Used
Memory indirect *or* Memory deferred	Add R1, @(R3)	R1 □ R1 + Mem[Mem[R3]]	If R3 is the address of a pointer *p*, then mode yields **p*
Autoincrement	Add R1, (R2)+	R1 □ R1+Mem[R2]; R2 □ R2 + *d*	Useful for stepping through arrays within a loop; R2 points to start of array; each ref. increments R2 by *d*
Autodecrement	Add R1, -(R2)	R1 □ R1-Mem[R2]; R2 □ R2 + *d*	Same as autoincrement; can be used for push/pop on stack
Scaled	Add R1, 100(R2), [R3]	R1 □ R1 + Mem[100+R2+R3**d*]	Used to index arrays

Example



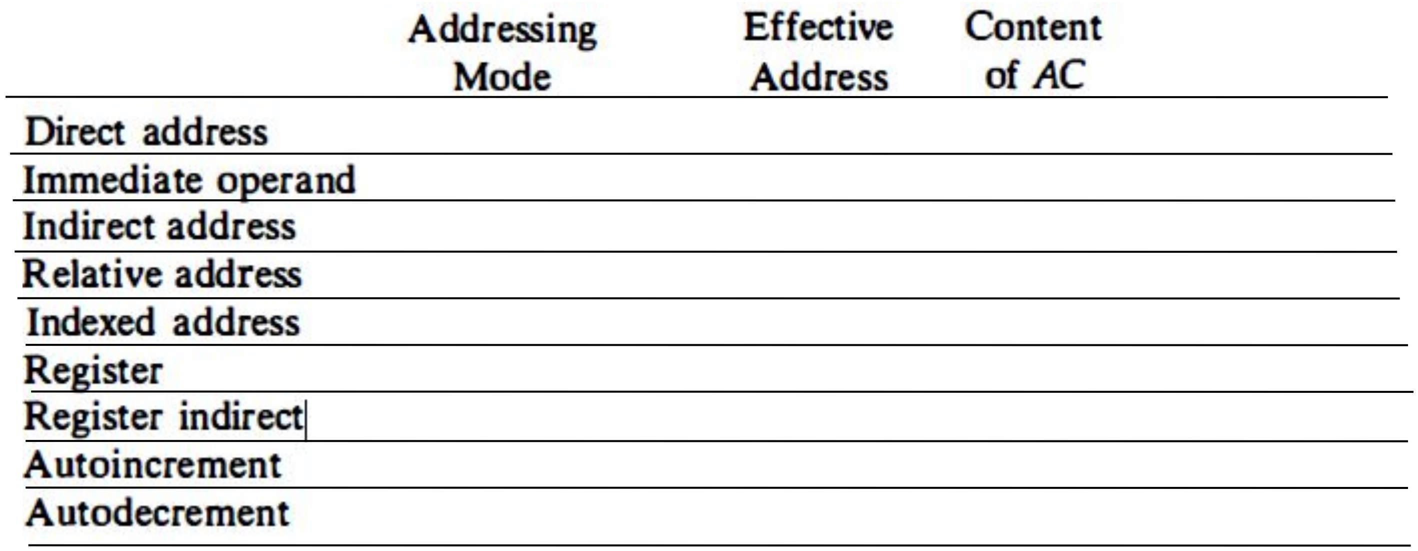
PC: Program counter

R1: Register

XR: Indexed register

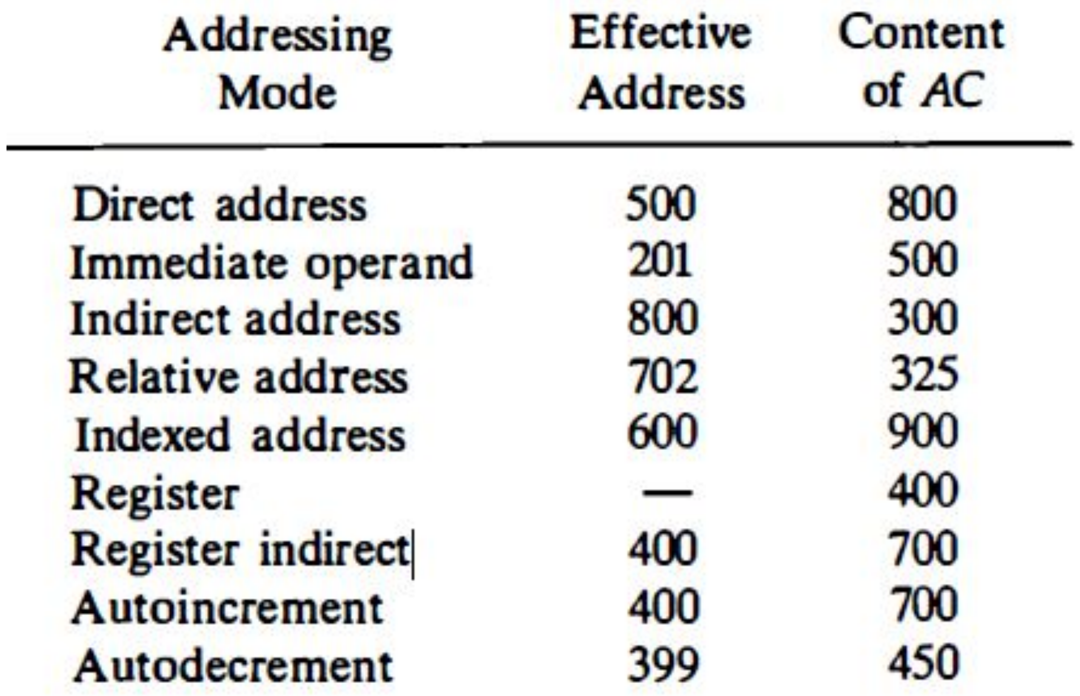
AC: Accumulator

Fig.1 [1]



Addressing Mode	Effective Address	Content of AC
Direct address		
Immediate operand		
Indirect address		
Relative address		
Indexed address		
Register		
Register indirect		
Autoincrement		
Autodecrement		

Fig.2 [1]



Addressing Mode	Effective Address	Content of *AC*
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

Fig.3 [1]

Types and sizes of operands

- How is the type of an operand designated?
  - Encoded in the opcode
  - Annotated by tags
- Common operand types:

– Character -	8bits
– Half word -	16 bits, 2 bytes
– Word -	32 bits, 4 bytes
– Single precision floating point -	32 bits, 4 bytes
– Double precision floating point -	64 bits, 8 bytes

- Most instructions have three operands ($z = x + y$)
- Well-known ISAs specify 0-3 (explicit) operands per instruction

- Operands can be specified implicitly or explicitly (*see previous slides of classifying ISA*)

Types and sizes of operands

- Encoding of characters:
  - ASCII
  - UNICODE
- Encoding of integers(signed):
  - Two’s complement binary numbers

    http://en.wikipedia.org/wiki/Two's_complement
- Encoding of floating point numbers:
  - IEEE standard 754

Expanding Opcodes [2]

- We have seen that how the number of operands in an instruction is dependent on the instruction length; we must have enough bits for the opcode and for the operand addresses

- However, not all instructions require the same number of operands

- **Expanding opcodes** represent a compromise between the need for a rich set of opcodes and the desire to have short opcodes, and thus short instructions.

- The idea is to make some opcodes short, but have a means to provide longer ones when needed.

  When the opcode is short, a lot of bits are left to hold operands (which mean we could have two or three operands per instruction).

Expanding Opcodes

- When we don’t need any space for operands (for an instruction such as **Halt** or because the machine uses a stack), **all the bits can be used for the opcode**, which allows for many unique instructions

  In between, there are longer opcodes with fewer operands as well as shorter opcodes with more operands

Consider a machine with 16-bit instructions and 16 registers. Because we now have a register set instead of one simple accumulator, we need to use 4 bits to specify a unique register. We could encode 16 instructions, each with 3 register operands (which implies any data to be operated on must first be loaded into a register).

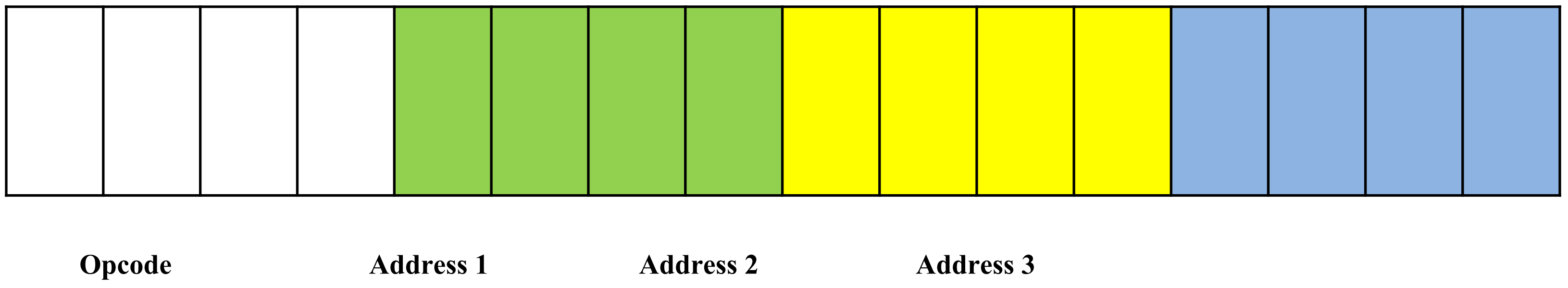
Or,

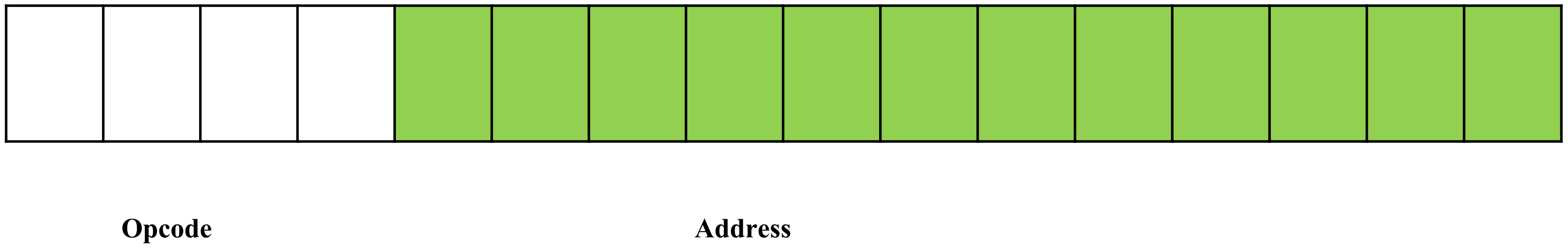
Use 4 bits for the opcode and 12 bits for a memory address (assuming a memory of size 4K). However, if all data in memory is first loaded into a register in this register set, the instruction can select that particular data element using only 4 bits (assuming 16 registers).

These two choices are illustrated in the following figure:

Example

Fig. Two Possibilities for a 16-Bit Instruction Format



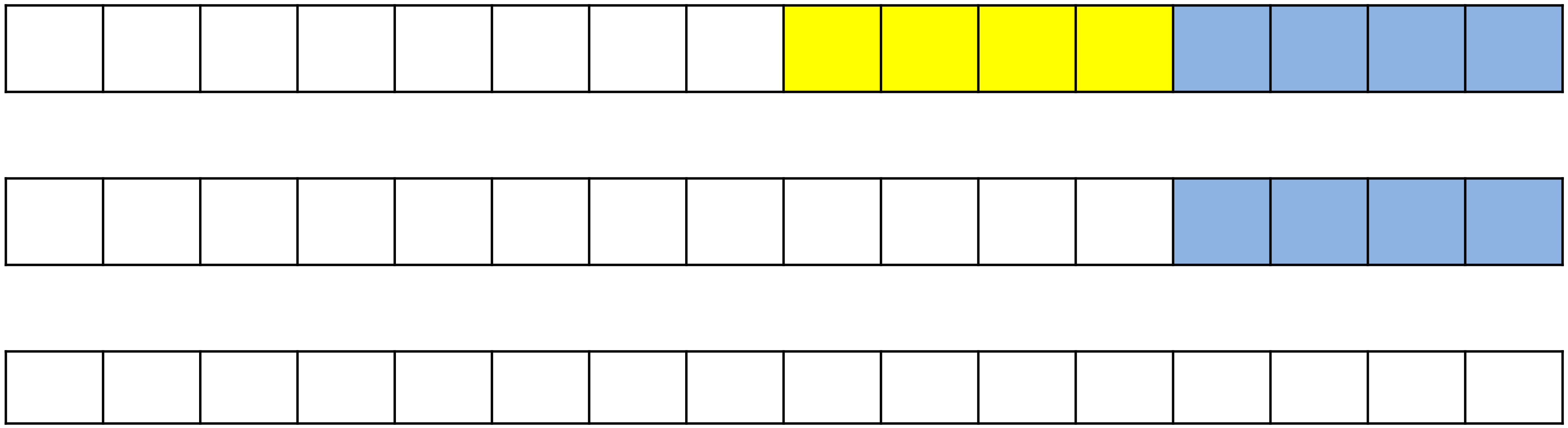


But why limit the opcode to only 4 bits?

- If we allow the length of the opcode to vary, that changes the number of remaining bits that can be used for operand addresses.

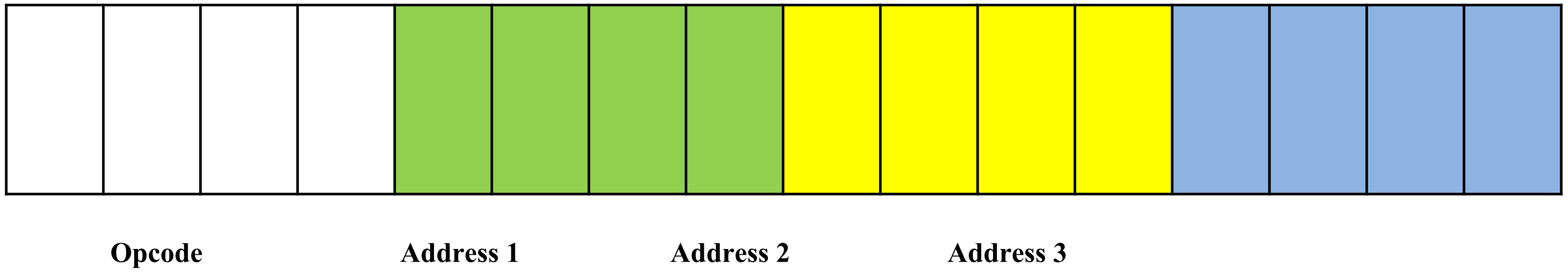
- Using expanding opcodes, we could allow for opcodes of 8 bits that require two register operands; or we could allow opcodes of 12 bits that operate on one register; or we could allow for 16-bit opcodes that require no operands.

Three More Possibilities for a 16-Bit Instruction Format



Consider a machine in which instructions are 16 bits long and addresses are 4 bits long.

- This might be reasonable on a machine that has 16 registers on which all arithmetic operations take place.
- One design would be a 4-bit opcode and three addresses in each instruction, giving 16 three address instructions.[5]



However, if the designers need 15 three-address instructions, 14 two-address instructions, 31 one-address instructions, and 16 instructions with no address at all, they can use opcodes 0 to 14 as three-address instructions but interpret opcode 15 differently.

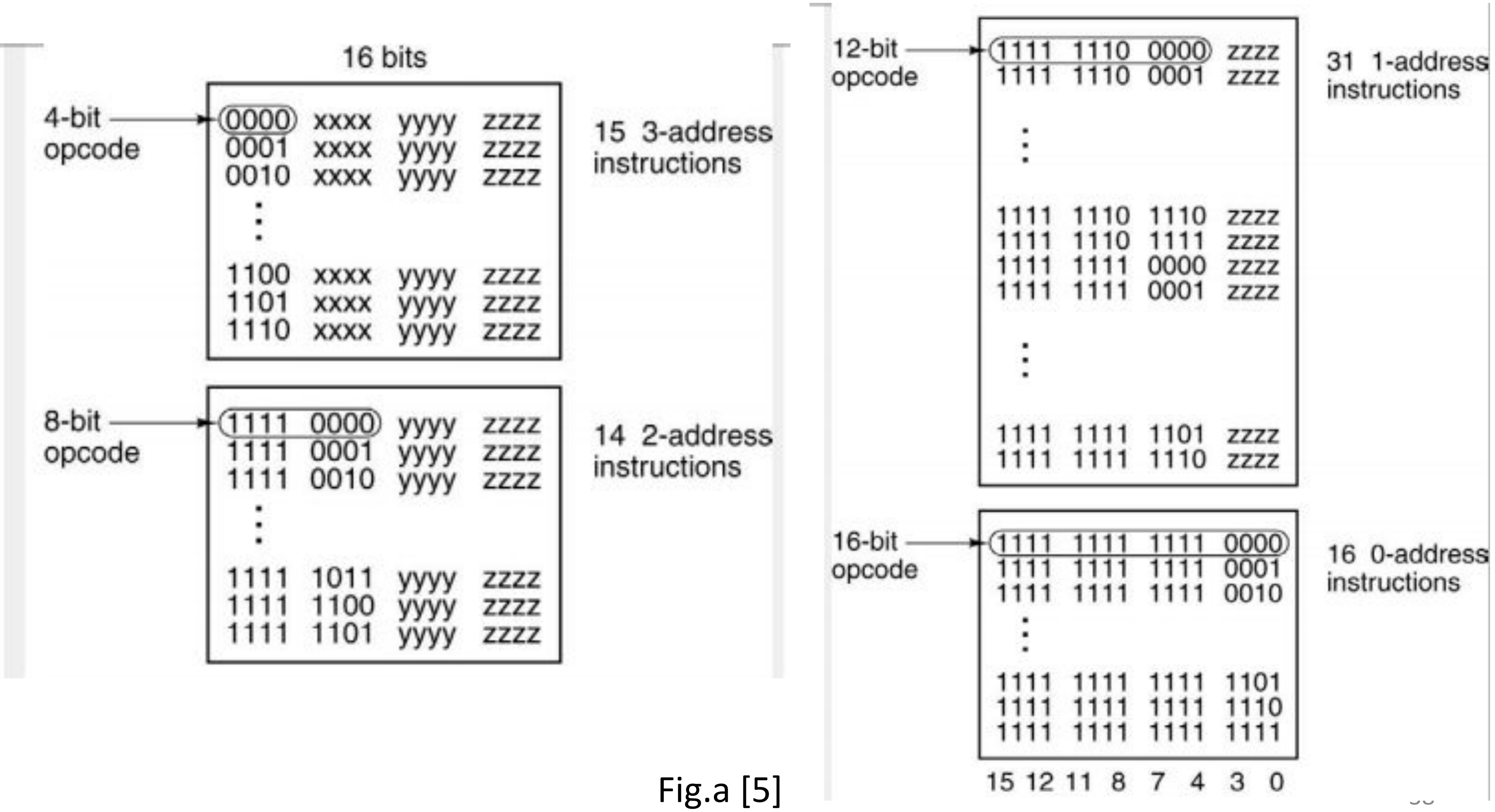


Fig.a [5]

Escape Opcode

- The only issue is that we need a method to determine when the instructions should be interpreted as having a 4-bit, 8-bit, 12-bit, or 16-bit opcode
- The trick is to use an "**escape opcode**" to indicate which format should be used
- While allowing for a wider variety of instructions, this expanding opcode scheme also makes the decoding more complex. Instead of simply looking at a bit pattern and deciding which instruction it is, we need to decode the instruction something like this:

```
if (leftmost four bits != 1111 ) {
    Execute appropriate three-address instruction}
else if (leftmost seven bits != 1111 111 ) {
    Execute appropriate two-address instruction}
else if (leftmost twelve bits != 1111 1111 1111 ) {
    Execute appropriate one-address instruction }
else {
    Execute appropriate zero-address instruction
}
```

Escape Opcode

- At each stage, one spare code—**the escape code**—is used to indicate that we should now look at more bits

- This is another example of the types of trade-offs hardware designers continually face:

  Here, we trade opcode space for operand space.

Example: Given 8-bit instructions, it is possible to use expanding opcodes to allow the following to be encoded? If so, show the encoding.

- 3 instructions with two 3-bit operands
- 2 instructions with one 4-bit operand
- 4 instructions with one 3-bit operand

Example

First, we must determine if the encoding is possible:

$3 * 2^3 * 2^3 = 3 * 2^6 = 192$

$2 * 2^4 = 32$

$4 * 2^3 = 32$

If we sum the required number of bit patterns, we get 192 + 32 + 32 = 256. 8 bits in the instruction means a total of $2^8 = 256$ bit patterns, so we have an exact match (which means the encoding is possible, but every bit pattern will be used in creating it).

The encoding we can use is as follows:

```
00 xxx xxx
01 xxx xxx      3 instructions with two 3-bit operands
10 xxx xxx
11 – Escape opcode
1100 xxxx
1101 xxxx             2 instructions with one 4-bit operand
1110 – Escape opcode
1111 – Escape opcode
11100 xxx
11101 xxx             4 instructions with one 3-bit operand
11110 xxx
11111 xxx
```

Types of Instruction [3]

Depending on operation they perform, all instructions are divided in several groups:

- Arithmetic Instructions
- Data Transfer Instructions
- Logic Instructions
- Bit-oriented Instructions
- Branch Instructions

MNEMONICS and OPERAND

- The **first part** of each instruction, called MNEMONIC refers to the operation an instruction performs (copy, addition, logic operation etc.).
- Mnemonics are abbreviations of the name of operation being executed.
- For example:
  - INC R1: Increment register R1 (increment register R1);
  - JNZ LOOP: Jump if Not Zero LOOP (if the value in the accumulator is not 0, jump to the address marked as LOOP);

- The other part of instruction, called **OPERAND** (separated from mnemonic by at least one whitespace) defines data being processed by instructions.
- Some of the instructions have no operand, while some of them have one, two or three. If there is more than one operand in an instruction, they are separated by a comma. For example:

MNEMONICS and OPERAND

- Some of the instructions have no operand, while some of them have one, two or three.
- If there is more than one operand in an instruction, they are separated by a comma. For example:
- RET - return from a subroutine;
- JZ TEMP - if the number in the accumulator is not 0, jump to the address marked as TEMP;
- ADD A,R3 - add R3 and accumulator;
- CJNE A,#20,LOOP - compare accumulator with 20. If they are not equal, jump to the address marked as LOOP;

Arithmetic Instructions

- Arithmetic instructions perform several basic operations such as addition, subtraction, division, multiplication etc.
- After execution, the result is stored in the first operand.
- For example:

ADD A,R1 - The result of addition (A+R1) will be stored in the accumulator.

Arithmetic Instructions

Mnemonic	Description	Byte	Cycle
ADD A,Rn	Adds the register to the accumulator	1	1
ADD A,direct	Adds the direct byte to the accumulator	2	2
DD A,@Ri	Adds the indirect RAM to the accumulator	1	2
ADD A,#data	Adds the immediate data to the accumulator	2	2
ADDC A,Rn	Adds the register to the accumulator with a carry flag	1	1
ADDC A,direct	Adds the direct byte to the accumulator with a carry flag	2	2
ADDC A,@Ri	Adds the indirect RAM to the accumulator with a carry flag	1	2
ADDC A,#data	Adds the immediate data to the accumulator with a carry flag	2	2
SUBB A,Rn	Subtracts the register from the accumulator with a borrow	1	1

Data Transfer Instructions

- Data transfer instructions move the content of one register to another.
- The register the content of which is moved remains unchanged.
- If they have the suffix “X” (MOVX), the data is exchanged with external memory.

Mnemonic	Description	Byte	Cycle
MOV A,Rn	Moves the register to the accumulator	1	1
MOV A,direct	Moves the direct byte to the accumulator	2	2
MOV A,@Ri	Moves the indirect RAM to the accumulator	1	2
MOV A,#data	Moves the immediate data to the accumulator	2	2
MOV Rn,A	Moves the accumulator to the register	1	2

Data Transfer Instructions

Mnemonic	Description	Byte	Cycle
MOV A,Rn	Moves the register to the accumulator	1	1
MOV A,direct	Moves the direct byte to the accumulator	2	2
MOV A,@Ri	Moves the indirect RAM to the accumulator	1	2
MOV A,#data	Moves the immediate data to the accumulator	2	2
MOV Rn,A	Moves the accumulator to the register	1	2
MOV Rn,direct	Moves the direct byte to the register	2	4
MOV Rn,#data	Moves the immediate data to the register	2	2
MOV direct,A	Moves the accumulator to the direct byte	2	3
MOV direct,Rn	Moves the register to the direct byte	2	3
MOV direct,direct	Moves the direct byte to the direct byte	3	4
MOV direct,@Ri	Moves the indirect RAM to the direct byte	2	4

Data Transfer Instructions

Mnemonic	Description	Byte	Cycle
MOV direct,#data	Moves the immediate data to the direct byte	3	3
MOV @Ri,A	Moves the accumulator to the indirect RAM	1	3
MOV @Ri,direct	Moves the direct byte to the indirect RAM	2	5
MOV @Ri,#data	Moves the immediate data to the indirect RAM	2	3
MOVC A,@A+PC	Moves the code byte relative to the PC to the accumulator (address=A+PC)	1	3
MOVX A,@Ri	Moves the external RAM (8-bit address) to the accumulator	1	3-10
MOVX @Ri,A	Moves the accumulator to the external RAM (8-bit address)	1	4-11
PUSH direct	Pushes the direct byte onto the stack	2	4
POP direct	Pops the direct byte from the stack/td>	2	3

Data Transfer Instructions

Mnemonic	Description	Byte	Cycle
XCH A,Rn	Exchanges the register with the accumulator	1	2
XCH A,direct	Exchanges the direct byte with the accumulator	2	3
XCH A,@Ri	Exchanges the indirect RAM with the accumulator	1	3
XCHD A,@Ri	Exchanges the low-order nibble indirect RAM with the accumulator	1	3

Logic Instructions

- Logic instructions perform logic operations upon corresponding bits of two registers.
- After execution, the result is stored in the first operand.

Mnemonic	Description	Byte	Cycle
ANL A,Rn	AND register to accumulator	1	1
ANL A,direct	AND direct byte to accumulator	2	2
ANL A,@Ri	AND indirect RAM to accumulator	1	2
ANL A,#data	AND immediate data to accumulator	2	2
ANL direct,A	AND accumulator to direct byte	2	3
ANL direct,#data	AND immediae data to direct register	3	4

Logic Instructions

Mnemonic	Description	Byte	Cycle
ORL A,Rn	OR register to accumulator	1	1
ORL A,direct	OR direct byte to accumulator	2	2
ORL A,@Ri	OR indirect RAM to accumulator	1	2
ORL direct,A	OR accumulator to direct byte	2	3
ORL direct,#data	OR immediate data to direct byte	3	4
XRL A,Rn	Exclusive OR register to accumulator	1	1
XRL A,direct	Exclusive OR direct byte to accumulator	2	2
XRL A,@Ri	Exclusive OR indirect RAM to accumulator	1	2
XRL A,#data	Exclusive OR immediate data to accumulator	2	2
XRL direct,A	Exclusive OR accumulator to direct byte	2	3
XORL direct,#data	Exclusive OR immediate data to direct byte	3	4

Logic Instructions

Mnemonic	Description	Byte	Cycle
CLR A	Clears the accumulator	1	1
CPL A	Complements the accumulator (1=0, 0=1)	1	1
SWAP A	Swaps nibbles within the accumulator	1	1
RL A	Rotates bits in the accumulator left	1	1
RLC A	Rotates bits in the accumulator left through carry	1	1
RR A	Rotates bits in the accumulator right	1	1
RRC A	Rotates bits in the accumulator right through carry	1	1

Bit-oriented Instructions

- Similar to logic instructions, bit-oriented instructions perform logic operations.
- The difference is that these are performed upon single bits.

Mnemonic	Description	Byte	Cycle
CLR C	Clears the carry flag	1	1
CLR bit	Clears the direct bit	2	3
SETB C	Sets the carry flag	1	1
SETB bit	Sets the direct bit	2	3
CPL C	Complements the carry flag	1	1
CPL bit	Complements the direct bit	2	3
ANL C,bit	AND direct bit to the carry flag	2	2

Bit-oriented Instructions

Mnemonic	Description	Byte	Cycle
ANL C,/bit	AND complements of direct bit to the carry flag	2	2
ORL C,bit	OR direct bit to the carry flag	2	2
ORL C,/bit	OR complements of direct bit to the carry flag	2	2
MOV C,bit	Moves the direct bit to the carry flag	2	2
MOV bit,C	Moves the carry flag to the direct bit	2	3

Branch Instructions

There are two kinds of branch instructions:

1. Unconditional jump instructions: upon their execution a jump to a new location from where the program continues execution is executed.
2. Conditional jump instructions: a jump to a new program location is executed only if a specified condition is met. Otherwise, the program normally proceeds with the next instruction.

Mnemonic	Description	Byte	Cycle
ACALL addr11	Absolute subroutine call	2	6
LCALL addr16	Long subroutine call	3	6
RET	Returns from subroutine	1	4
RETI	Returns from interrupt subroutine	1	4
AJMP addr11	Absolute jump	2	3

Branch Instructions

Mnemonic	Description	Byte	Cycle
LJMP addr16	Long jump	3	4
SJMP rel	Short jump (from –128 to +127 locations relative to the following instruction)	2	3
JC rel	Jump if carry flag is set. Short jump.	2	3
JNC rel	Jump if carry flag is not set. Short jump.	2	3
JB bit,rel	Jump if direct bit is set. Short jump.	3	4
JBC bit,rel	Jump if direct bit is set and clears bit. Short jump.	3	4
JMP @A+DPTR	Jump indirect relative to the DPTR	1	2
JZ rel	Jump if the accumulator is zero. Short jump.	2	3
JNZ rel	Jump if the accumulator is not zero. Short jump.	2	3
NOP	No operation	1	1

References

1. M. Morris Mano, Computer System Architecture, Prentice Hall of India Pvt Ltd, 3rd Edition (updated) , 30 June 2017
2. the Essentials of Computer Organization and Architecture (4th Edition), Linda Null and Julia Lobur, Jones and Barlett Publishers, ISBN: 978-1-284-07448-2.
3. Barry B. Brey, The Intel Microprocessors: 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, Pentium Pro Processor, Pentium II, Pentium III, Pentium 4, and Core2 with 64-bit Extensions : Architecture, Programming, and Interfacing. Pearson Education India, Eigth Edition, 2019
4. Computer Organization and Architecture Designing for Performance Tenth Edition William Stallings
5. http://www.personal.kent.edu/~aguercio/CS35101Slides/Tanenbaum/CA_Ch05_PartII.pdf