Lecture 6
Addressing Modes

CPS310

Computer Organization II

WINTER 2022

© Dr. A. Sadeghian

ARC Pseudo-Ops

Pseudo-Op	Usage	Meaning
`.equ`	`X .equ #10`	Treat symbol X as $(10)_{16}$
`.begin`	`.begin`	Start assembling
`.end`	`.end`	Stop assembling
`.org`	`.org 2048`	Change location counter to 2048
`.dwb`	`.dwb 25`	Reserve a block of 25 words
`.global`	`.global Y`	Y is used in another module
`.extern`	`.extern Z`	Z is defined in another module
`.macro`	`.macro M a, b, ...`	Define macro M with formal parameters a, b, ...
`.endmacro`	`.endmacro`	End of macro definition
`.if`	`.if <cond>`	Assemble if <cond> is true
`.endif`	`.endif`	End of `.if` construct

Pseudo-ops are instructions to the assembler to perform some action at assembly time. Thet are not part of the ISA.

Pseudo-Ops vs Instructions

- **Instructions are specific to a given machine**
  (Instruction good for the architecture)

- **Pseudo-Ops are specific to a given assembler**
  (tell assembler to do some action)

Most commonly used Pseudo-Ops

.EQU: tell the assembler to equate a value or a string to a symbol this symbol then can be sued throughout the program

X .equ #10 ! X is $(10)_{16}$

.BEGIN / .END: tell the assemble where to start and stop the assembling process

Any instruction before **.BEGIN** & after **.END** are ignored

Most commonly used Pseudo-Ops

.org tell assembler where in the memory to put the next instruction

.org 2048 ! Next instruction goes to memory location 2048

.dwb tell assembler to set aside space in the memory

.dwb 25 ! Reserve a block of 25 words in the memory

A sample ‘C’ Program

```
/* to add 2 numbers */


main (){
	int x, y, z;
	x = 15;
	y = 9;
	z = x + y;
}
```

Assemble this ARC Program

An ARC assembly language program to add two integers:

```
! This programs adds two numbers
        .begin
        .org 2048
prog1:  ld      [x], %r1          ! Load x into %r1
        ld      [y], %r2          ! Load y into %r2
        addcc   %r1, %r2, %r3     ! %r3 ← %r1 + %r2
        st      %r3, [z]          ! Store %r3 into z
        jmpl    %r15 + 4, %r0     ! Return
x:      15
y:      9
z:      0
        .end
```

Your Program

Your Data

Side note – Registers and memory

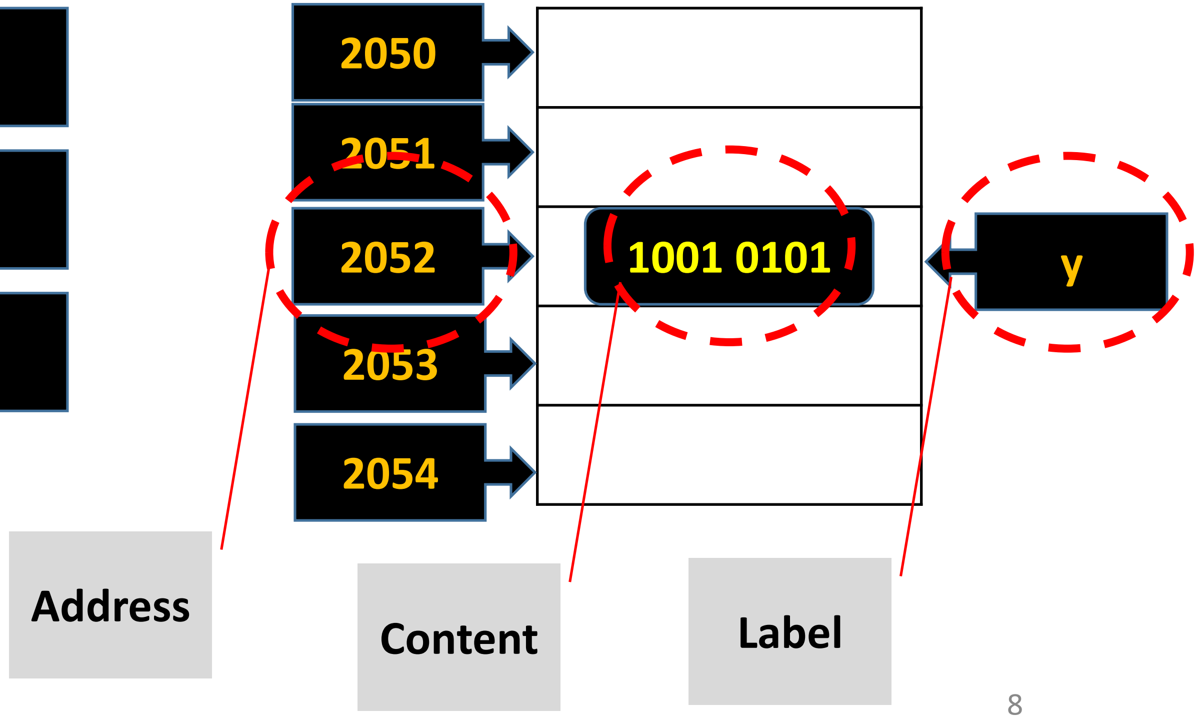
Registers are referred to by their names:
%r1, %r2, %r3, …

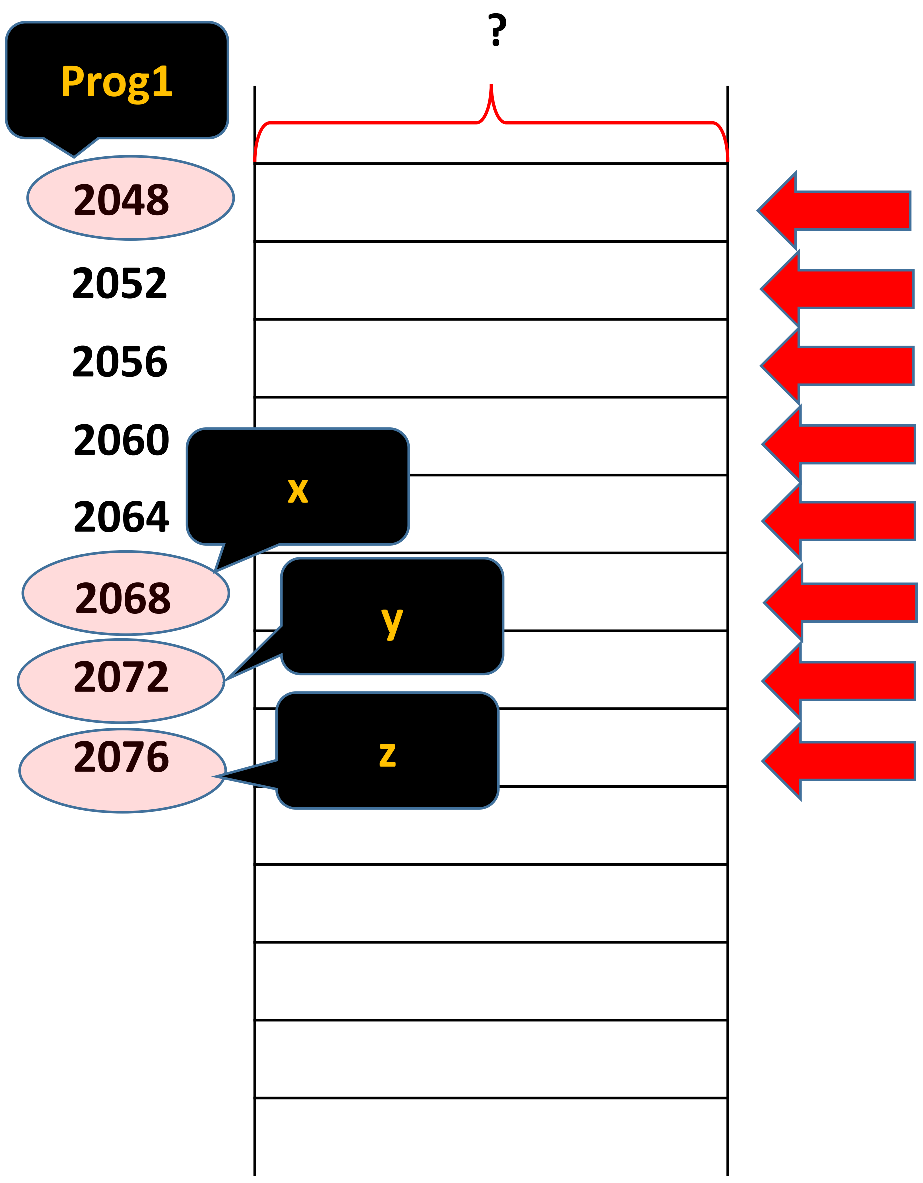
Memory is referred to by an address or a name (label):
2052
y

%r1	5	5
%r2	3	3
%r3	17	8

addcc %r1, %r2, %r3

%r1 + %r2 → %r3





[x] = 15

[y] = 9

[z] = 24

```
ld [x], %r1
ld [y], %r2
addcc %r1, %r2, %r3
st %r3, [z]
jmpl %r15 + 4, %r0
15
9
0
```

```
! This program adds 2 numbers
        . begin
        . org 2048
Prog1:  ld      [x], %r1
        ld      [y], %r2
        addcc   %r1,%r2,%r3
        st      %r3, [z]
        jmpl    %r15+4, %r0
x:      15
y:      9
z:      0
        . end
```

Memory Map – instructions addresses

Address		
2048	Prog1: ld	[x], %r1
2052	ld	[y], %r2
2056	addcc	%r1,%r2,%r3
2060	st	%r3, [z]
2064	jmpl	%r15+4, %r0
2068	x: 15	
2072	y: 9	
2076	z: 0	

In ARC, each instruction will take 32 bits

x is 2068
[x] is 15

Assembled Code vs instructions

ld [x], %r1
1100 0010 0000 0000 0010 1000 0001 0100
ld [y], %r2
1100 0100 0000 0000 0010 1000 0001 1000
addcc %r1,%r2,%r3
1000 0110 1000 0000 0100 0000 0000 0010
st %r3, [z]
1100 0110 0010 0000 0010 1000 0001 1100
jmpl %r15+4, %r0
1000 0001 1100 0011 1110 0000 0000 0100
15
0000 0000 0000 0000 0000 0000 0000 1111
9
0000 0000 0000 0000 0000 0000 0000 1001
0
0000 0000 0000 0000 0000 0000 0000 0000

Complete Memory Map

Address		Memory Content
2048	ld [x], %r1	1100 0010 0000 0000 0010 1000 0001 0100
2052	ld [y], %r2	1100 0100 0000 0000 0010 1000 0001 1000
2056	addcc %r1,%r2,%r3	1000 0110 1000 0000 0100 0000 0000 0010
2060	st %r3, [z]	1100 0110 0010 0000 0010 1000 0001 1100
2064	jmpl %r15+4, %r0	1000 0001 1100 0011 1110 0000 0000 0100
2068	15	0000 0000 0000 0000 0000 0000 0000 1111
2072	9	0000 0000 0000 0000 0000 0000 0000 1001
2076	0	0000 0000 0000 0000 0000 0000 0000 0000

Another Look at the Memory Map

Address		Memory Content
…		
2056	addcc %r1,%r2,%r3	1000 0110
2057		1000 0000
2058		0100 0000
2059		0000 0010
2060	st %r3, [z]	1100 0110
2061		0010 0000
2062		0010 1000
2063		0001 1100
…		

In ARC, each instruction will take 32 bits

A Sample C Program using Arrays

25 + (-10) + 33 + (-5) + 7 = 50

```
/* add elements of an array */

main(){
        int Len = 5;
        int A[5] = {25, -10, 33, -5, 7};
        int i, sum = 0;

        for ( i = Len – 1; i >=0; i--)
                sum += a[i];
}
```

```
! This program sums LENGTH numbers
! Register usage:        %r1 – Length of array a
!                        %r2 – Starting address of array a
!                        %r3 – The partial sum
!                        %r4 – Pointer into array a
!                        %r5 – Holds an element of a
            .begin                  ! Start assembling
            .org   2048             ! Start program at 2048
a_start     .equ   3000             ! Address of array a
            ld     [length], %r1
            ld     [address],%r2    ! %r2 ← address of a
            andcc  %r3, %r0, %r3    ! %r3 ← 0
loop:       andcc  %r1, %r1, %r0    ! Test # remaining elements
            be     done             ! Finished when length=0
            addcc  %r1, -4, %r1     ! Decrement array length
            addcc  %r1, %r2, %r4    ! Address of next element
            ld     %r4, %r5         ! %r5 ← Memory[%r4]
            addcc  %r3, %r5, %r3    ! Sum new element into r3
            ba     loop             ! Repeat loop.

done:       jmpl   %r15 + 4, %r0    ! Return to calling routine
length:             20
address:            a_start
            .org   a_start          ! Start of array a
a:                  25              ! length/4 values follow
                   -10
                    33
                    -5
                     7
            .end                    ! Stop assembling
```

Your Program

Your Data

Address ?

Address ?

andcc %r3, %r0, %r3 – bit to bit ANDing

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
%r3	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
AND																																
%r0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
%r3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

bit to bit AND

%r3 AND %r0 (bit to bit ANDing)

result

andcc %r3, %r1, %r3 – bit to bit ANDing

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
%r3	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
AND																																
%r1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
%r3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	?	?	?	?

bit to bit AND

%r3 AND %r0 (bit to bit ANDing)

result

	2048	ld [length], %r1	11000010 00000000 00101000 00101100
	2052	ld [address], %r2	11000100 00000000 00101000 00110000
	2056	andcc %r3, %r0, %r3	10000110 10001000 11000000 00000000
loop	2060	andcc %r1, %r1, %r0	10000000 10001000 01000000 00000001
	2064	be done	00000010 10000000 00000000 00000110
	2068	addcc %r1, -4, %r1	10000010 10000000 01111111 11111100
	2072	addcc %r1, %r2, %r4	10001000 10000000 01000000 00000010
	2076	ld %r4, %r5	11001010 00000001 00000000 00000000
	2080	addcc %r3, %r5, %r3	10000110 10111111 11000000 00000101
	2084	ba loop	00010000 10111111 11111111 11111010
done	2088	jmpl %r15 + 4, %r0	10000001 11000011 11100000 00000100
length	2092	20	00000000 00000000 00000000 00010100
	2096	3000	00000000 00000000 00001011 10111000
a	3000	25	00000000 00000000 00000000 00011001
	3004	-10	11111111 11111111 11111111 11110110
	3008	33	00000000 00000000 00000000 00100001
	3012	-5	11111111 11111111 11111111 11111011
	3016	7	00000000 00000000 00000000 00000111

Load from memory *length* to r1

r2 ← 3000

r3 → 0

if zero flag is set branch to done

address of the next element

r5 + r3 → r3

return back to the calling routine

	ld [length], %r1
	ld [address], %r2
	andcc %r3, %r0, %r3
loop:	andcc %r1, %r1, %r0
	be done
	addcc %r1, -4, %r1
	addcc %r1, %r2, %r4
	ld %r4, %r5
	addcc %r3, %r5, %r3
	ba loop
done:	jmpl %r15 + 4, %r0
length:	20
address:	3000
a:	25
	-10
	33
	-5
	7

r1 ← 20

Load from memory *address* to r2

is r1 == 0 ?

decrement r1 by 4

Load from memory whose address is shown by r4 to r5

branch back to loop

3000

3004

3008

3012

3016

r1	r2	r3	r4	r5
20	3000	0		
16		7	3016	7
12		2	3012	-5
8		35	3008	33
4		25	3004	-10
0		50	3000	25

1, 2, 3 - address instructions

- In machines with many registers, most operations can be done using registers
- In machines with fewer registers, memory itself is being used more often
- We can have **3**-address, **2**-address, and **1**-address instructions
- For example, consider this expression: **A = B x C + D**

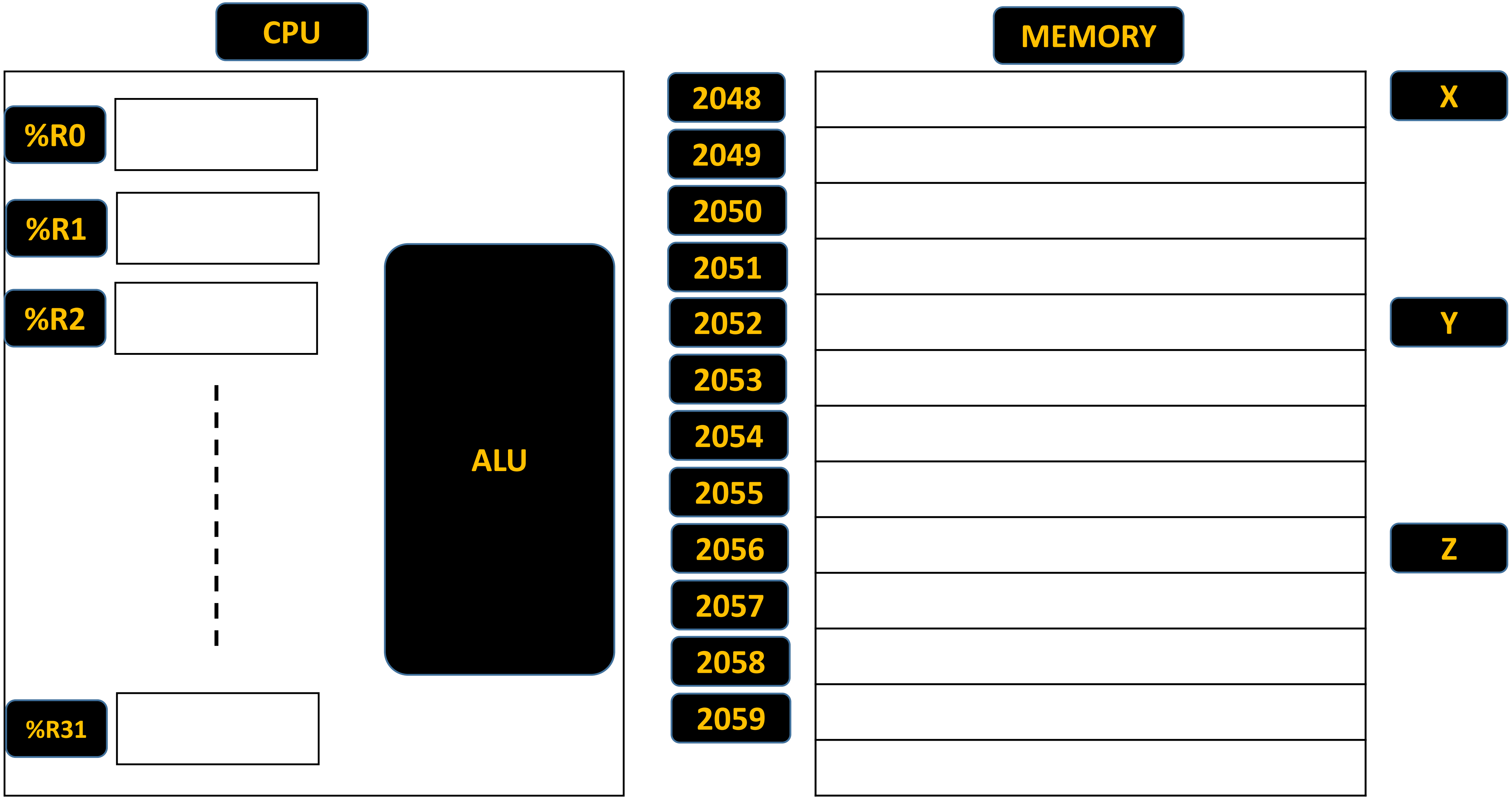
One, Two, Three-Address Machines

- For comparison purposes we evaluate same expression using each of the one, two, and three-address instruction types

Assumptions:

- Addresses are **2 bytes**
- Data are **2 bytes**
- Opcodes are **1 byte**
- Operands are moved to and from memory 1 word (**2 bytes**) at a time

Side note – Register vs Memory again



Three-Address Instructions - A = B x C + D

The expression A = B x C + D might be coded as:

```
mult    B, C, A    ! B x C -> A
add     D, A, A    ! D + A -> A
```

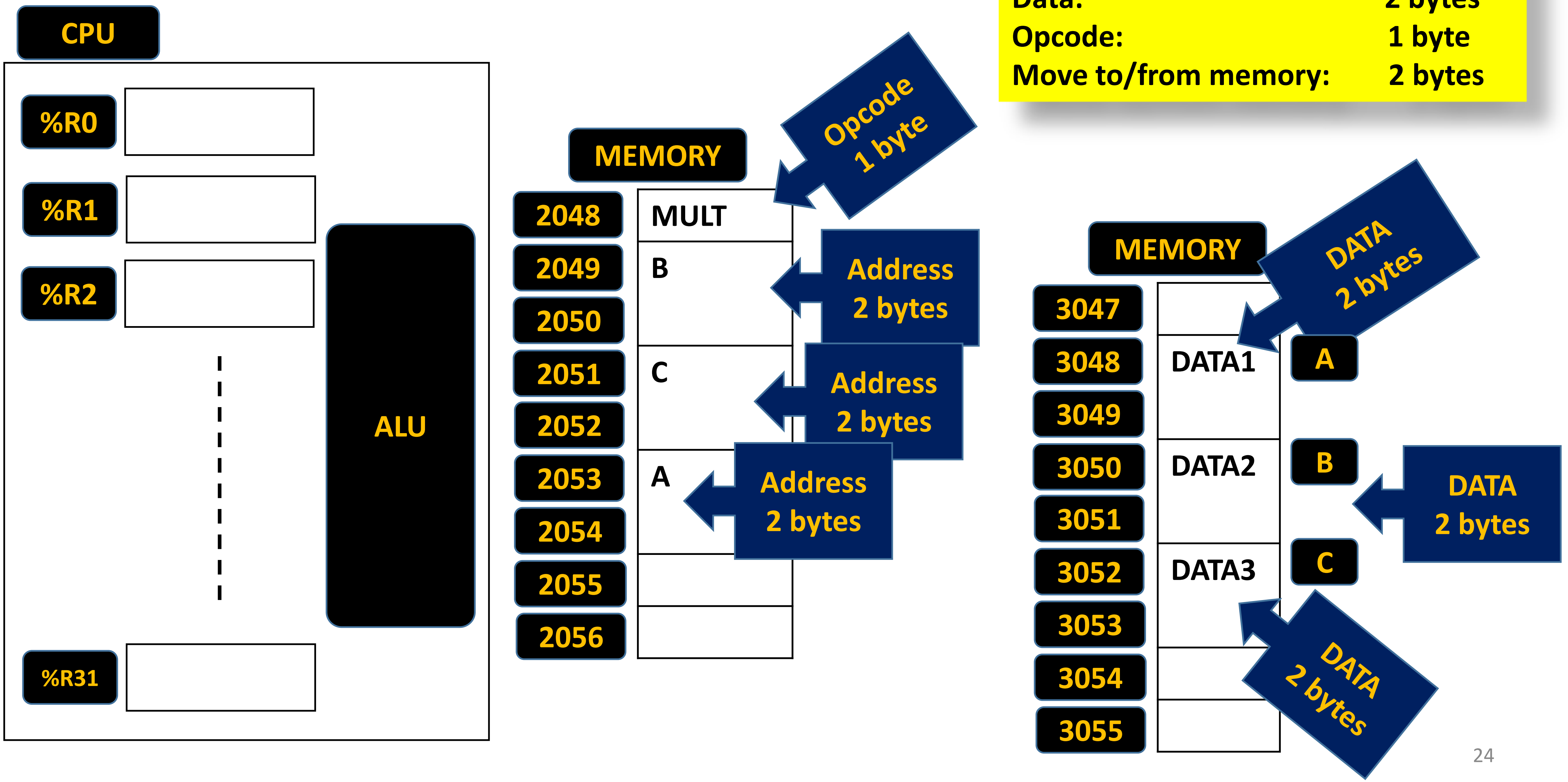
- B, C, A are addresses
- multiply data in B by data in C and store the result at address A
- add the data in D to data in A and store the result at address A

Note: The mult and add operations are generic;

not necessarily ARC instructions

Three-Address Instructions – MULT B, C, A

Address:	**2 bytes**
Data:	**2 bytes**
Opcode:	**1 byte**
Move to/from memory:	**2 bytes**



3-Address Instructions – Memory Traffic Calculation

Address:	2 bytes
Data:	2 bytes
Opcode:	1 byte
Move to/from memory:	2 bytes

CPU

1 byte

2, 2, 2 bytes

memory

mult

B, C, A

At this point the CPU knows that it should perform MULT, it also knows that the 1st operand is in address B, the 2nd operand is in address C and the result of multiplication should go to address A

Get the 1st operand from address B

Get the 2nd operand from address C

Perform MULT and put the result in address A

address	memory	comment
	mult B, C, A	! B x C -> A
	add D, A, A	! D + A -> A
A		
B		
C		
D		

2 bytes

2 bytes

2 bytes

Total Traffic: 7 + 6 byes (13 bytes)

3-Address Instructions – Memory Traffic Calculation

Address:	2 bytes
Data:	2 bytes
Opcode:	1 byte
Move to/from memory:	2 bytes

CPU

1 byte

2, 2, 2 bytes

memory

add | D, A, A

At this point the CPU knows that it should perform ADD, it also knows that the 1st operand is in address D, the 2nd operand is in address A and the result of multiplication should go to address A

Get the 1st operand from address D

2 bytes

Get the 2nd operand from address A

2 bytes

Perform ADD and put the result in address A

2 bytes

address	memory		comment
	mult	B, C, A	! B x C -> A
	add	D, A, A	! D + A -> A
A			
B			
C			
D			

Total Traffic: 7 + 6 byes (13 bytes)

3-Address Instructions - A = B x C + D

Address:	2 bytes
Data:	2 bytes
Opcode:	1 byte
Move to/from memory:	2 bytes

Each instruction is 1 + 2 + 2 + 2 = 7 bytes

The program size is then 7 x 2 = 14 bytes

Traffic for each instruction:

Instruction 1: fetch 7 bytes, data traffic 6 bytes – (mult B, C, A ! B x C -> A)

Instruction 2: fetch 7 bytes, data traffic 6 bytes – (add D, A, A ! D + A -> A)

Memory traffic is (7 + 6) x 2 = 26 bytes

```
mult    B, C, A    ! B x C -> A
add     D, A, A    ! D + A -> A
```

2-Address Instructions - A = B x C + D

In a two-address instruction, one of the operands is overwritten by the final result

Example:	A = B x C + D		
	load	B, A	! B → A
	mult	C, A	! A x C → A
	add	D, A	! A + D → A D

2-Address Instructions – Memory Traffic Calculation

Address: 2 bytes
Data: 2 bytes
Opcode: 1 byte
Move to/from memory: 2 bytes

CPU

1 byte

2, 2 bytes

memory

load B, A

At this point the CPU knows that it should perform LOAD, it also knows that the 1st operand is in address B, it should get the operand from address B and to write to address A

Get the operand from address B

Copy the operand to address A

2 bytes

2 bytes

address			comment
	load	B, A	! B → A
	mult	C, A	! C x A → A
	add	D, A	! D + A → A
A			
B			
C			
D			

Total Traffic: 5 + 4 byes (9 bytes)

2-Address Instructions – Memory Traffic Calculation

Address:	2 bytes
Data:	2 bytes
Opcode:	1 byte
Move to/from memory:	2 bytes

CPU

1 byte

2, 2 bytes

memory

mult C, A

At this point the CPU knows that it should perform MULT, it also knows that the 1st operand is in address C, the 2nd operand is in address A, and the result should go to address A

Get the operand from address C

Get the 2nd operand from address A

Perform MULT and put the result in address A

2 bytes

2 bytes

2 bytes

address			comment
	load	B, A	! B → A
	mult	C, A	! C x A → A
	add	D, A	! D + A → A
A			
B			
C			
D			

Total Traffic: 5 + 6 byes (11 bytes)

2-Address Instructions – Memory Traffic Calculation

Address:	2 bytes
Data:	2 bytes
Opcode:	1 byte
Move to/from memory:	2 bytes

CPU

1 byte

2, 2 bytes

memory

add D, A

At this point the CPU knows that it should perform ADD, it also knows that the 1st operand is in address D, the 2nd operand is in address A, and the result should go to address A

Get the operand from address D

Get the 2nd operand from address A

Perform ADD and put the result in address A

2 bytes

2 bytes

2 bytes

address			comment
	load	B, A	! B → A
	mult	C, A	! C x A → A
	add	D, A	! D + A → A
A			
B			
C			
D			

Total Traffic: 5 + 6 byes (11 bytes)

2-address instructions - A = B x C + D

Instruction 1: fetch 5 bytes, data traffic 4 bytes – (load B, A ! A ← B)

Instruction 2: fetch 5 bytes, data traffic 6 bytes – (mult C, A ! A <- A x C)

Instruction 3: fetch 5 bytes, data traffic 6 bytes – (add D, A ! A <- A + D)

15 bytes of fetch

16 bytes of data traffic

Memory traffic is 31 bytes

Address:	2 bytes
Data:	2 bytes
Opcode:	1 byte
Move to/from memory:	2 bytes

DT: 2, 2 bytes

F: 1 byte

F: 2, 2 bytes

```
load    B, A    ! A ← B
mult    C, A    ! A <- A x C
add     D, A    ! A <- A + D
```

DT: 2, 2, 2 bytes

F: 1 byte

F: 2, 2 bytes

F: 1 byte

F: 2, 2 bytes

DT: 2, 2, 2 bytes

1-Address (Accumulator) Instructions

- A one-address instruction employs a single register in the CPU, known as the accumulator. In this case, the code for the expression
- A = B x C + D is:

```
load    B    ! ACC   <-   B
mult    C    ! ACC   <-   ACC  x  C
add     D    ! ACC   <-   ACC  +  D
store   A    ! A     <-   ACC
```

1-Address (Accumulator) Instructions

Each instruction fetch: 1 + 2 = 3 bytes
Each instruction data traffic: 2 bytes

Total of 12 bytes of fetch
8 bytes of data traffic
memory traffic is 20 bytes

Address:	2 bytes
Data:	2 bytes
Opcode:	1 byte
Move to/from memory:	2 bytes

DT: 2 bytes

F: 1 byte

F: 2, bytes

```
load    B    ! ACC   <- B
mult    C    ! ACC   <- ACC x C
add     D    ! ACC   <- ACC + D
store   A    ! A     <- ACC
```

Addressing Modes in ARC

Addressing mode explains how an operand refers to the **data** we are interested in for a particular instruction

In the Fetch part of the instruction cycle, there are generally three ways to handle addressing in the instruction

1. **Immediate Addressing**
2. **Direct Addressing**
3. **Indirect Addressing**

1. Immediate Addressing Mode

The operand directly contains the value we are interested in working with, for example,

ADD #5, … !add **number 5** to something

This uses **immediate addressing** for the value 5

The 2’s comp rep for the number 5 is directly stored in the ADD instruction

We must know value at assembly time

2. Direct Addressing Mode

The operand contains an address with the data

ADD 100h, … !add the **content of Memory Location 100h** to something

Downside: Need to fit entire address in the instruction, may limit address space

For example, 32 bit word size and 32 bit addresses.
Do we have a problem here?

2. Direct Addressing Mode

The address could also be a register

ADD %r5, … !add the **content of Register 5** to something

Upside: don't have the previous problem

3. Indirect Addressing Mode

The operand contains an address

This address contains the address of the data

Add [100h], …

The data at memory location **100h** is an address:

1. Go to that address, and
2. Get the data, and
3. Add it to the Accumulator

Downside: Requires additional memory access

3. Indirect Memory Address

Can also do Indirect Addressing with registers

Add [%r3], …

The data in register 3 is an address:

1. Go to that address,
2. Get the data, and
3. Add it to the Accumulator

More Addressing Modes – ***Beyond ARC***

Refer to how an operand of an instruction is specified

Different variations of what we discussed before

1. Immediate	Reference to a constant
2. Direct	Access data using its address or register
3. Indirect	Access data using a pointer
4. Register Indirect	Access data using a pointer
5. Register Indexed	
6. Register Based	
7. Register Based Indexed	

Addressing Modes – *Beyond ARC*

Addressing Mode	Syntax	Meaning
Immediate	#K	K
Direct	K	M[K]
Indirect	(K)	M[M[K]]
Register	(Rn)	M[Rn]
Register Indexed	(Rm + Rn)	M[Rm + Rn]
Register Based	(Rm + X)	M[Rm + X]
Register Based Indexed	(Rm + Rn + X)	M[Rm + Rn + X]

Direct Mode – *Beyond ARC*

Gives either the operand or its address explicitly

Register mode: The operand is in a register
ADD R1, R2

Absolute mode: The operand in somewhere in the memory
The address of the memory location is given explicitly
ADD $2400, R1

Immediate mode: The operand is given as an immediate (explicitly)
ADD #$100, R1

Indirect Mode – *Beyond ARC*

The **effective address** of the operand is the content of a register (it is given in an register) or memory location whose address appeared in the instruction

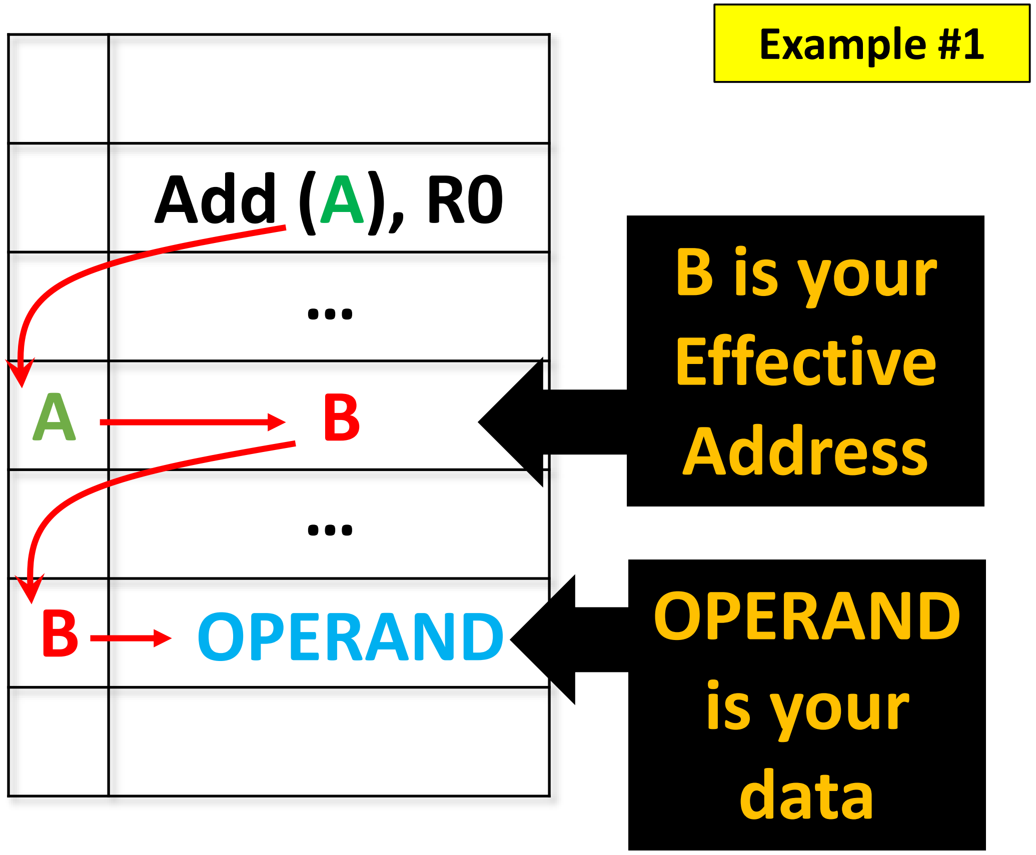
The instruction does not give the operand or its address explicitly

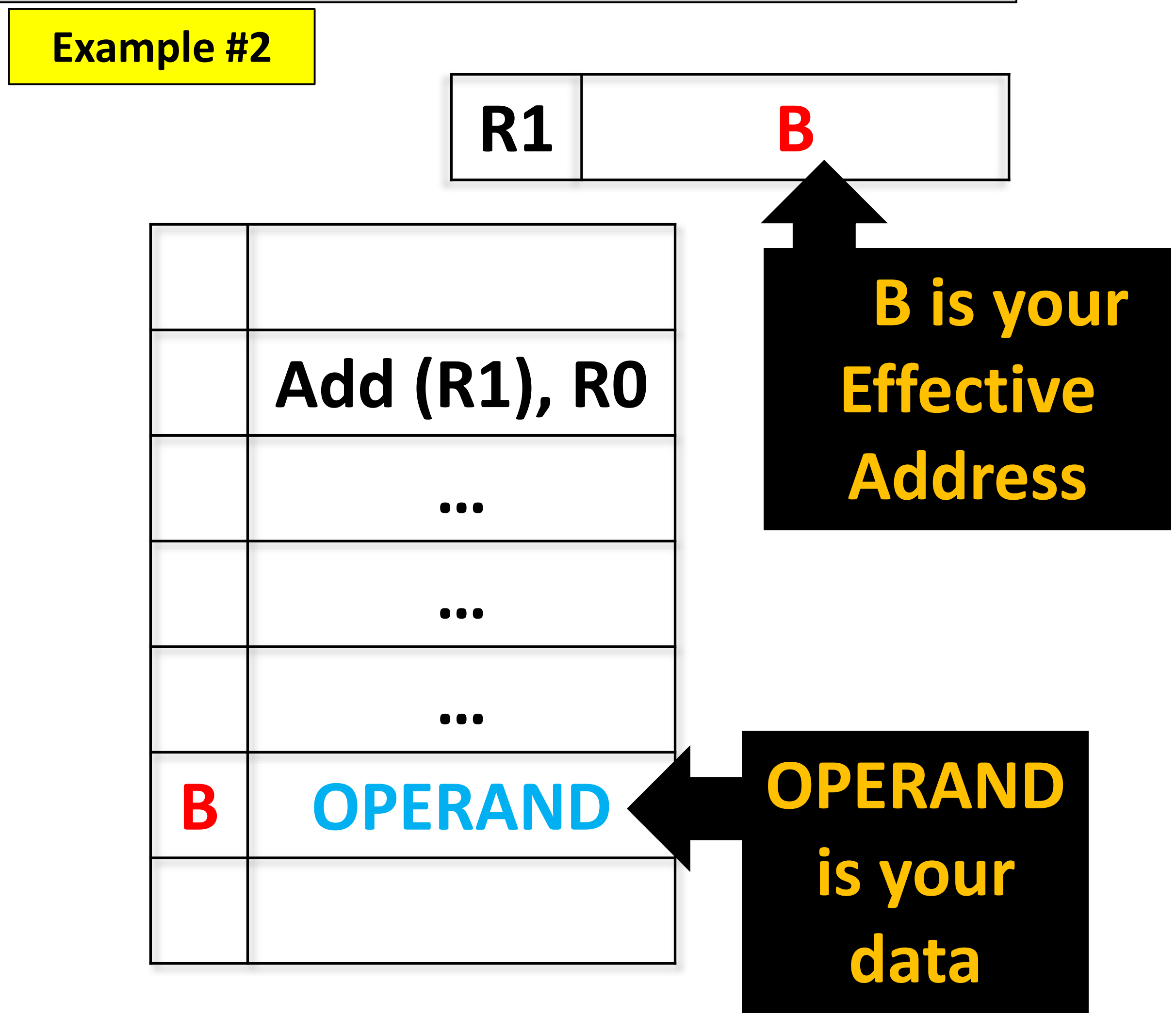
The effective address of the operand is the content of a

1. A register add (A), R0
2. Memory location add (R1), R0

Typically indirection is shown by placing the register or the memory location within parentheses.

Indirect Mode – *Beyond ARC*





Index Mode – *Beyond ARC*

The effective address of the operand is given by adding a constant value to the register content

	Add 20(R1), R2
1070	**5**

R1	**1050**

R2	**3**

20 is known as offset

X(Ri) → effective address is sum of X and Ri

(Ri, Rj) → effective address is sum of Ri and Rj

X(Ri, Rj) → effective address is sum of X and Ri and Rj