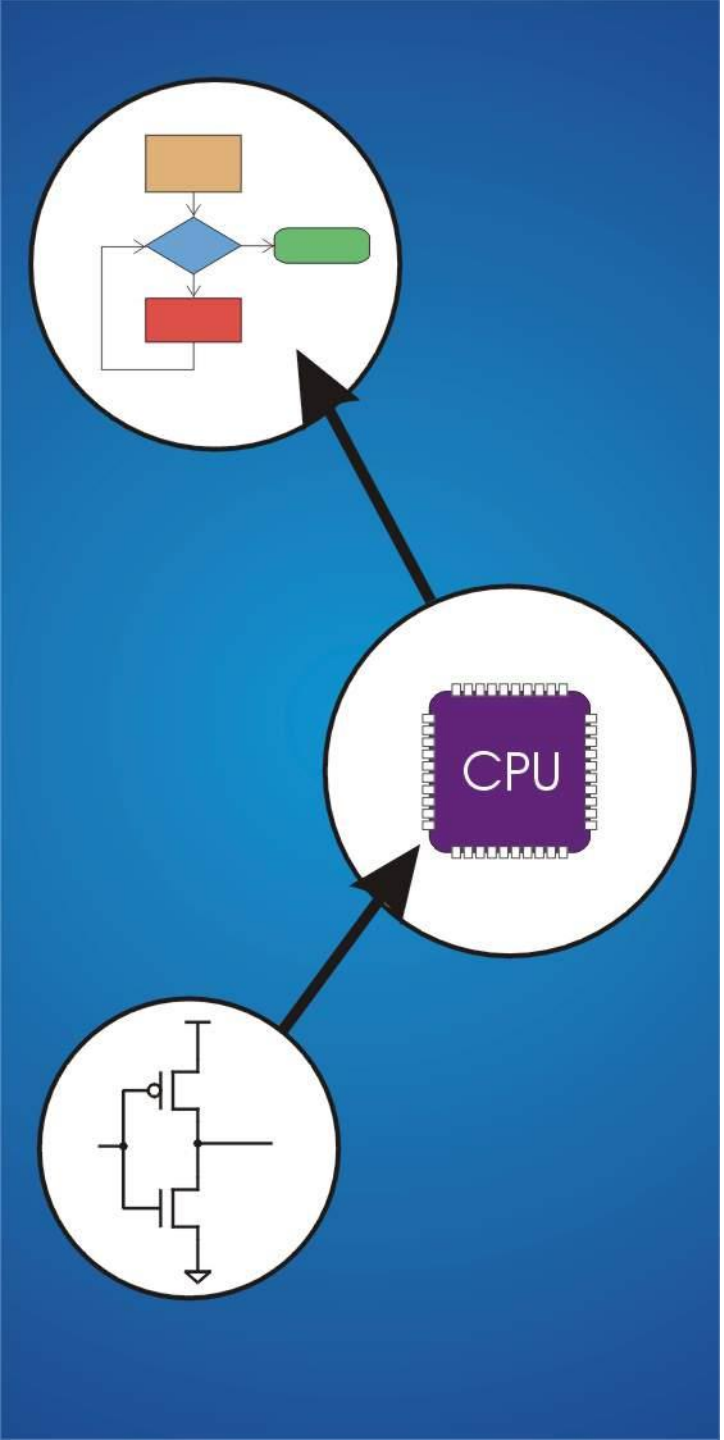
Introduction to Computer Engineering

CS/ECE 252, Spring 2017

Rahul Nayar

Computer Sciences Department

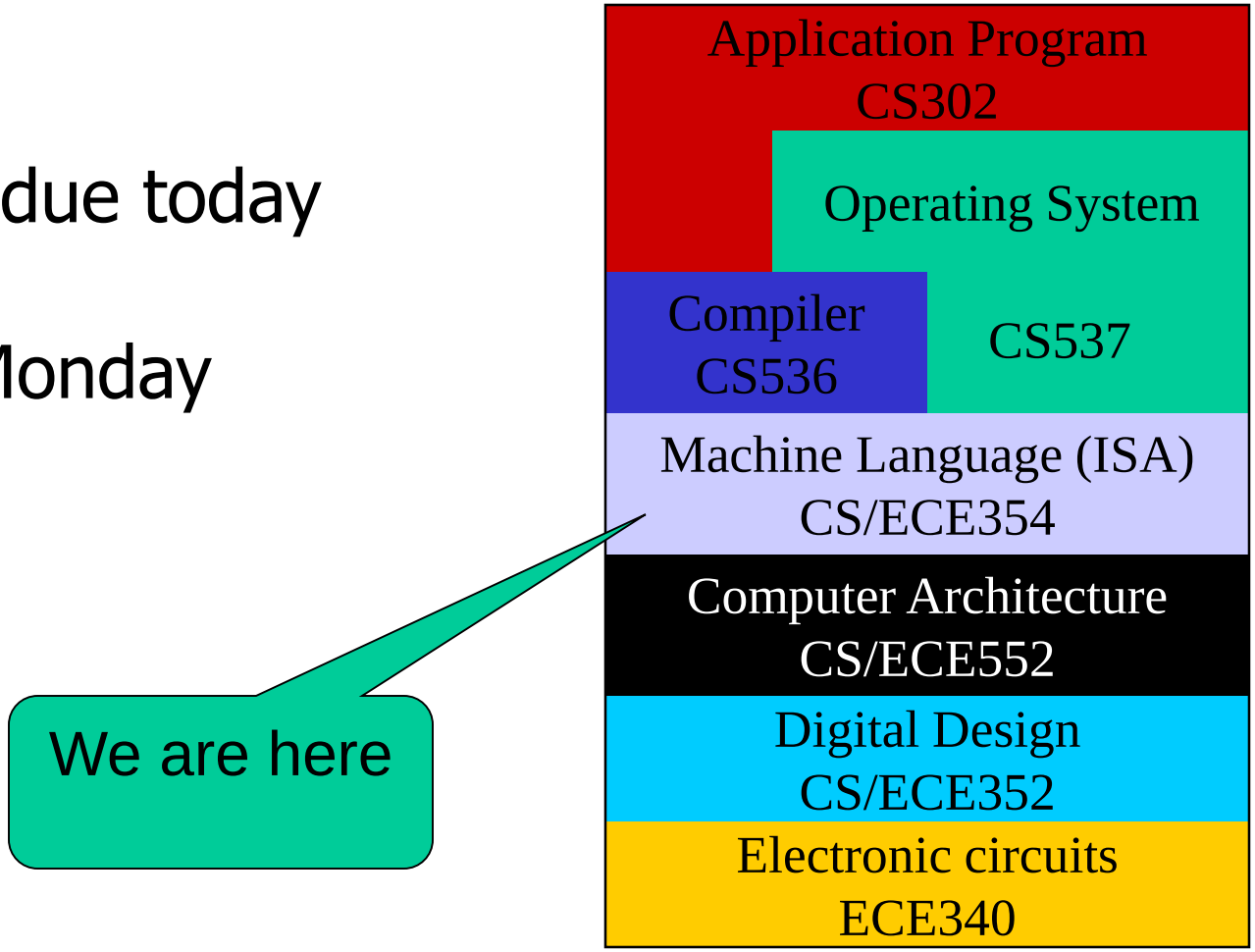
University of Wisconsin – Madison



Chapter 5
The LC-3

Announcements

- Homework 3 due today
- No class on Monday



Instruction Set Architecture

ISA = All of the *programmer-visible* components and operations of the computer

- **memory organization**
  - **address space -- how may locations can be addressed?**
  - **addressibility -- how many bits per location?**
- **register set**
  - **how many? what size? how are they used?**
- **instruction set**
  - **opcodes**
  - **data types**
  - **addressing modes**

ISA provides all information needed for someone that wants to write a program in machine language (or translate from a high-level language to machine language).

LC-3 Overview: Memory and Registers

Memory

- address space: 2^{16} locations (16-bit addresses)
- addressability: 16 bits

Registers

- temporary storage, accessed in a single machine cycle
  - accessing memory generally takes longer than a single cycle
- eight general-purpose registers: R0 - R7
  - each 16 bits wide
  - how many bits to uniquely identify a register?
- other registers
  - not directly addressable, but used by (and affected by) instructions
  - PC (program counter), condition codes

LC-3 Overview: Instruction Set

Opcodes

- **15 opcodes**
- *Operate* **instructions: ADD, AND, NOT**
- *Data movement* **instructions: LD, LDI, LDR, LEA, ST, STR, STI**
- *Control* **instructions: BR, JSR/JSRR, JMP, RTI, TRAP**
- **some opcodes set/clear *condition codes*, based on result:**
  - **N = negative, Z = zero, P = positive (> 0)**

Data Types

- **16-bit 2's complement integer**

Addressing Modes

- **How is the location of an operand specified?**
- **non-memory addresses: *immediate*, *register***
- **memory addresses: *PC-relative*, *indirect*, *base+offset***

Operate Instructions

Only three operations: ADD, AND, NOT

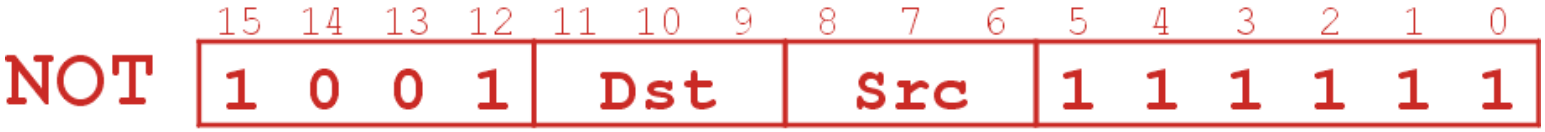
Source and destination operands are registers

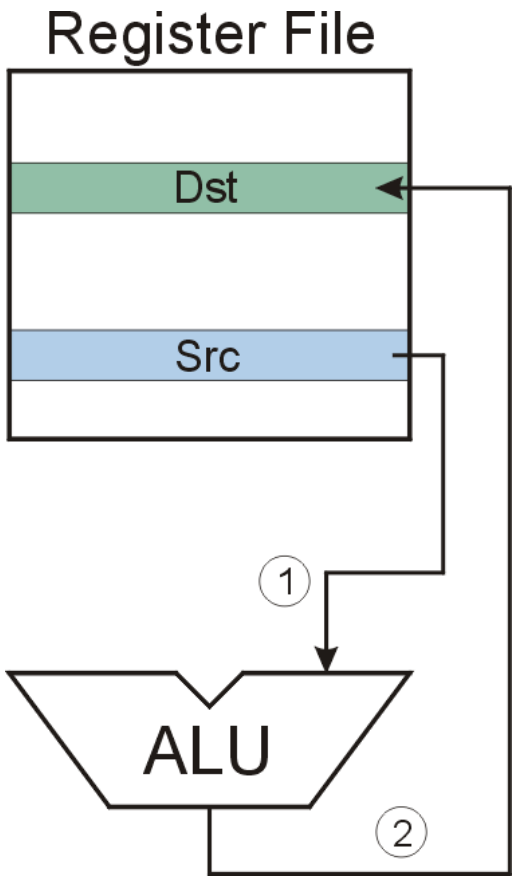
- **These instructions *do not* reference memory.**
- **ADD and AND can use "immediate" mode, where one operand is hard-wired into the instruction.**

Will show dataflow diagram with each instruction.

- **illustrates *when* and *where* data moves to accomplish the desired operation**

NOT (Register)



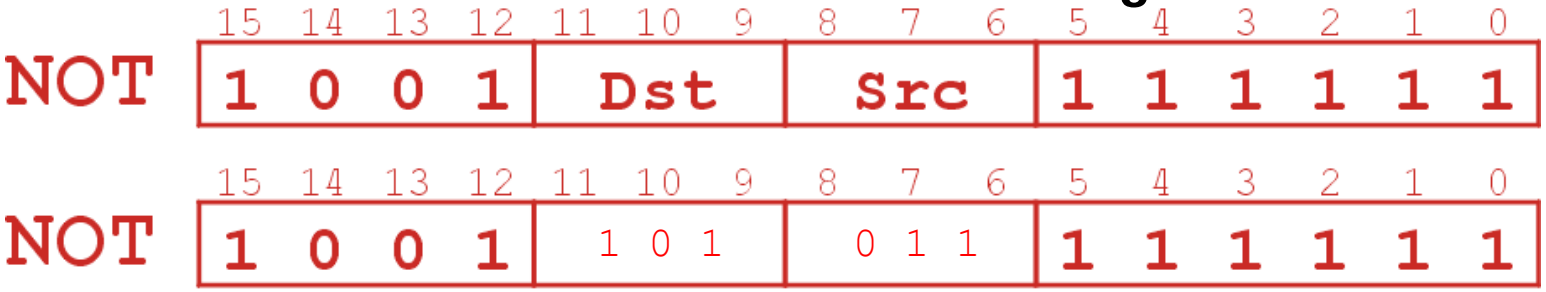


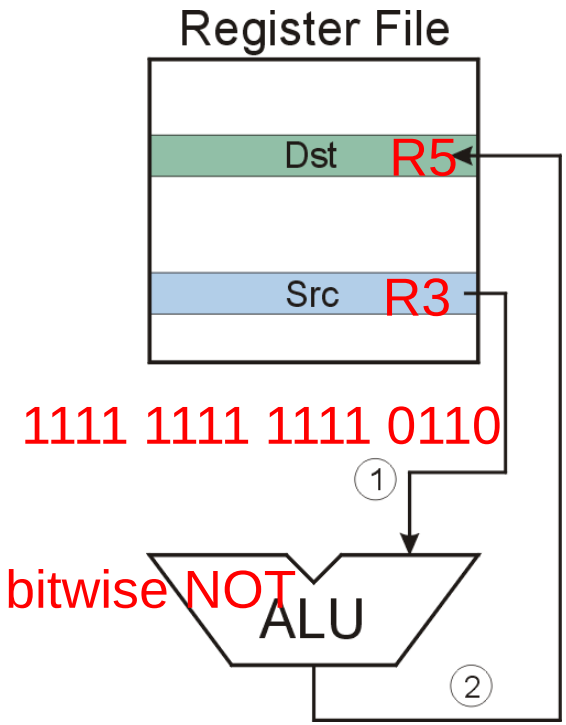
Note: Src and Dst could be the same register.

NOT (Register) Example

R3 is currently 1111 1111 1111 0110

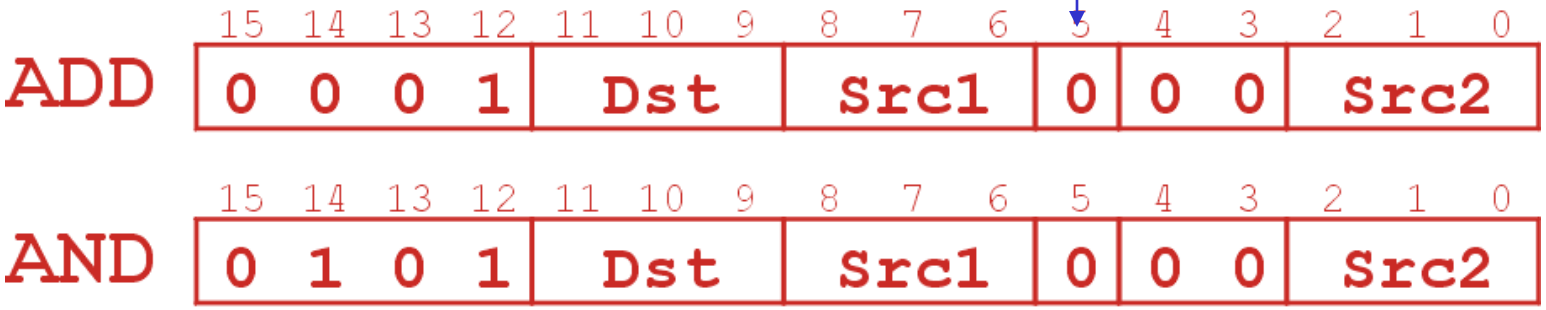
What is the contents of R5 after the following instruction is executed?

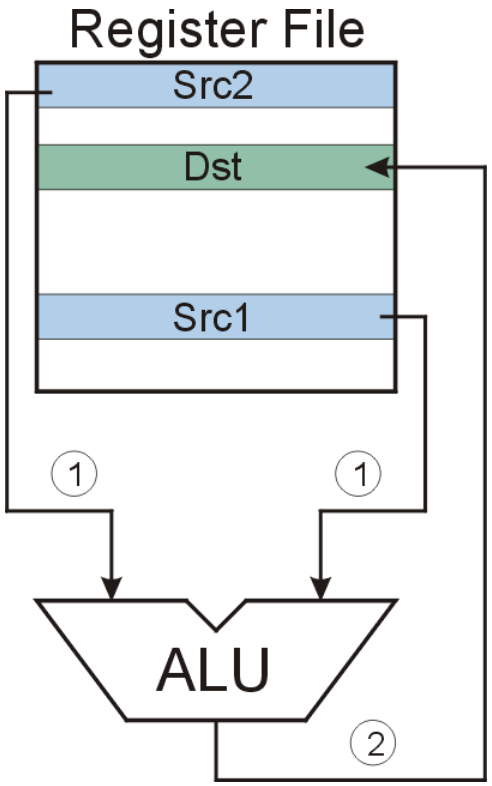




ADD/AND (Register)

this zero means "register mode"



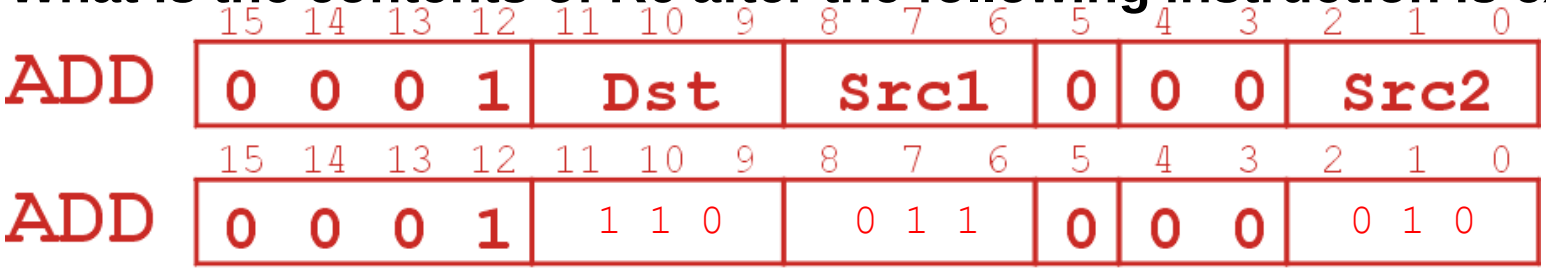


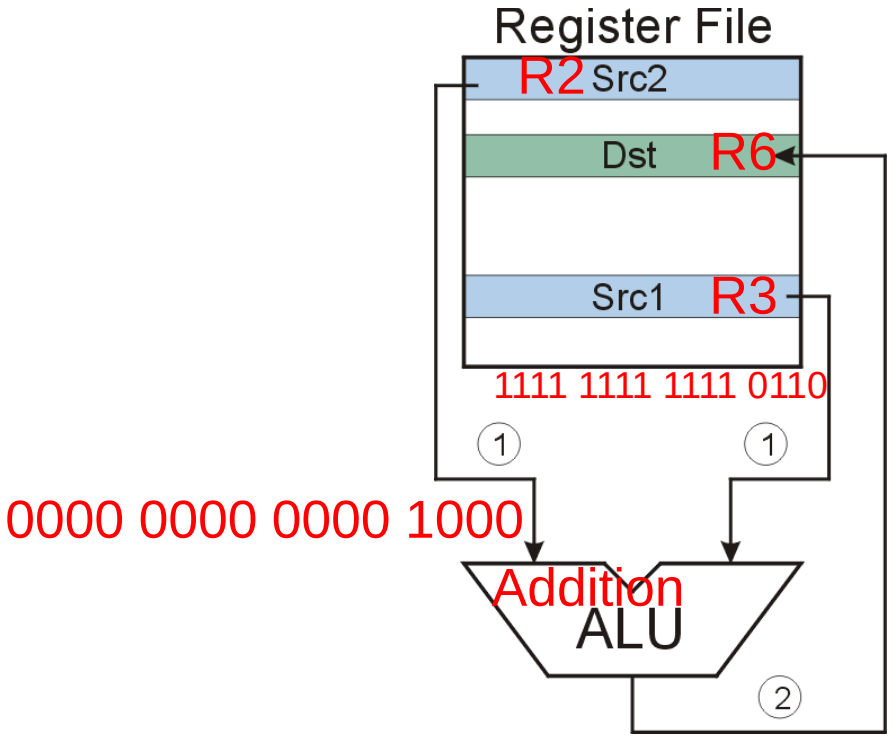
ADD (Register) Example

R3 is currently 1111 1111 1111 0110

R2 is currently 0000 0000 0000 1000

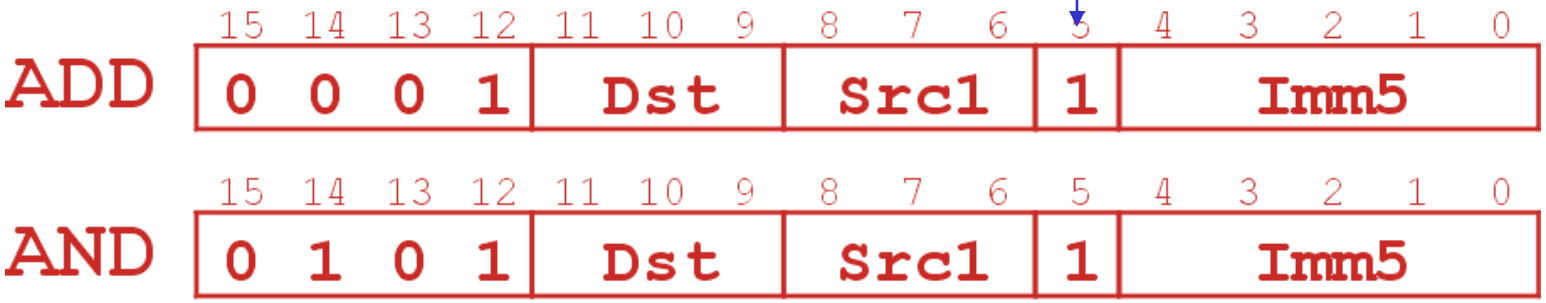
What is the contents of R6 after the following instruction is executed?



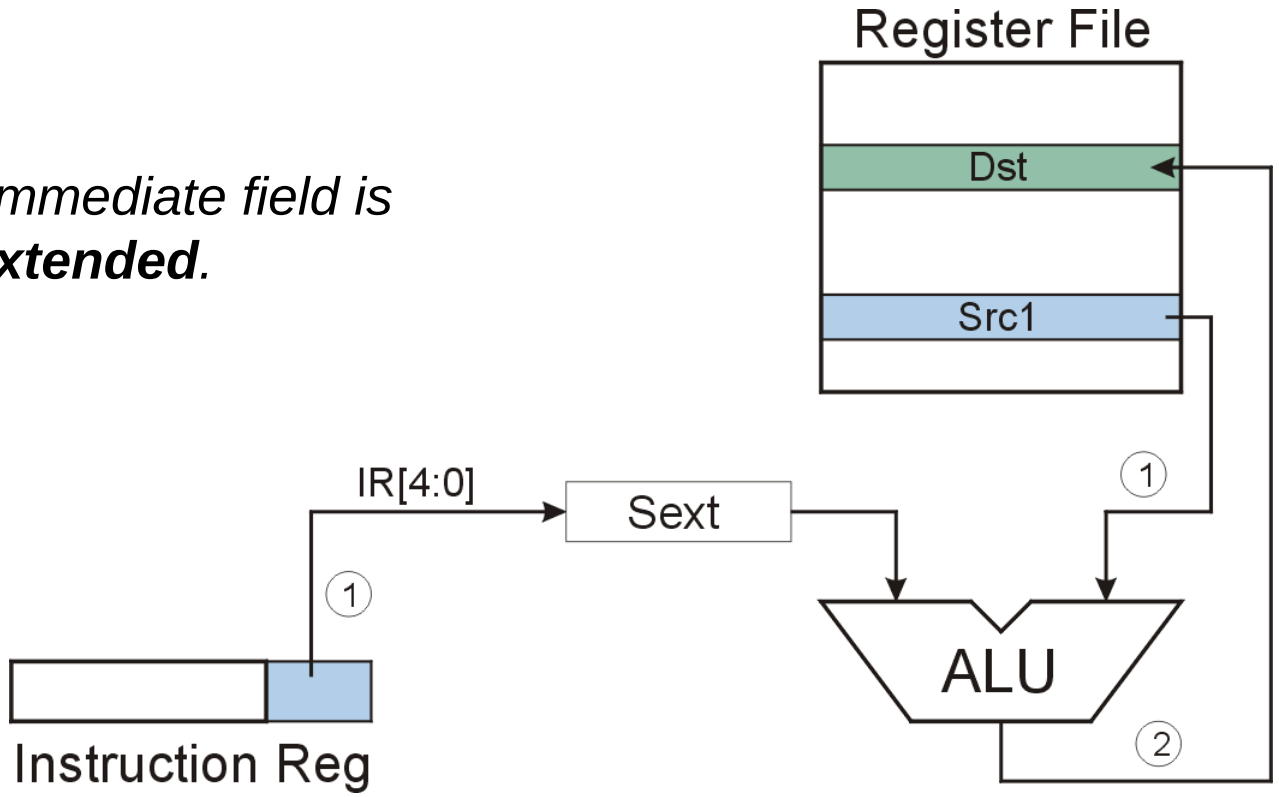


ADD/AND (Immediate)

this one means "immediate mode"



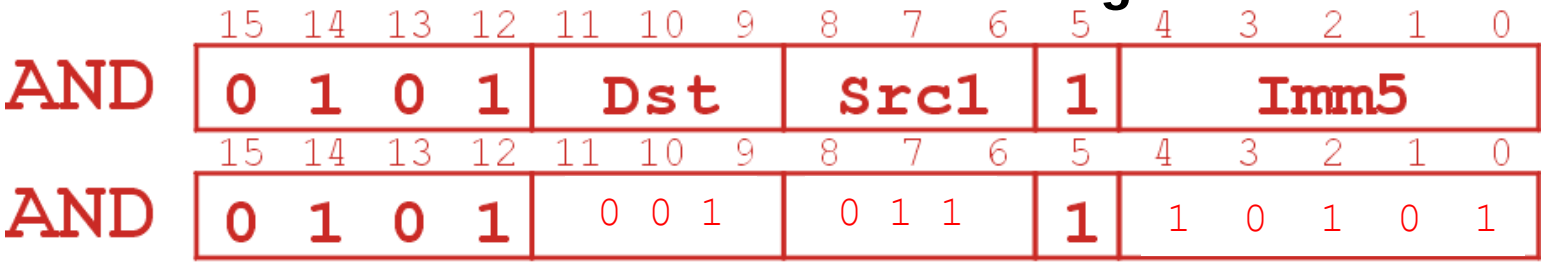
Note: Immediate field is ***sign-extended****.*



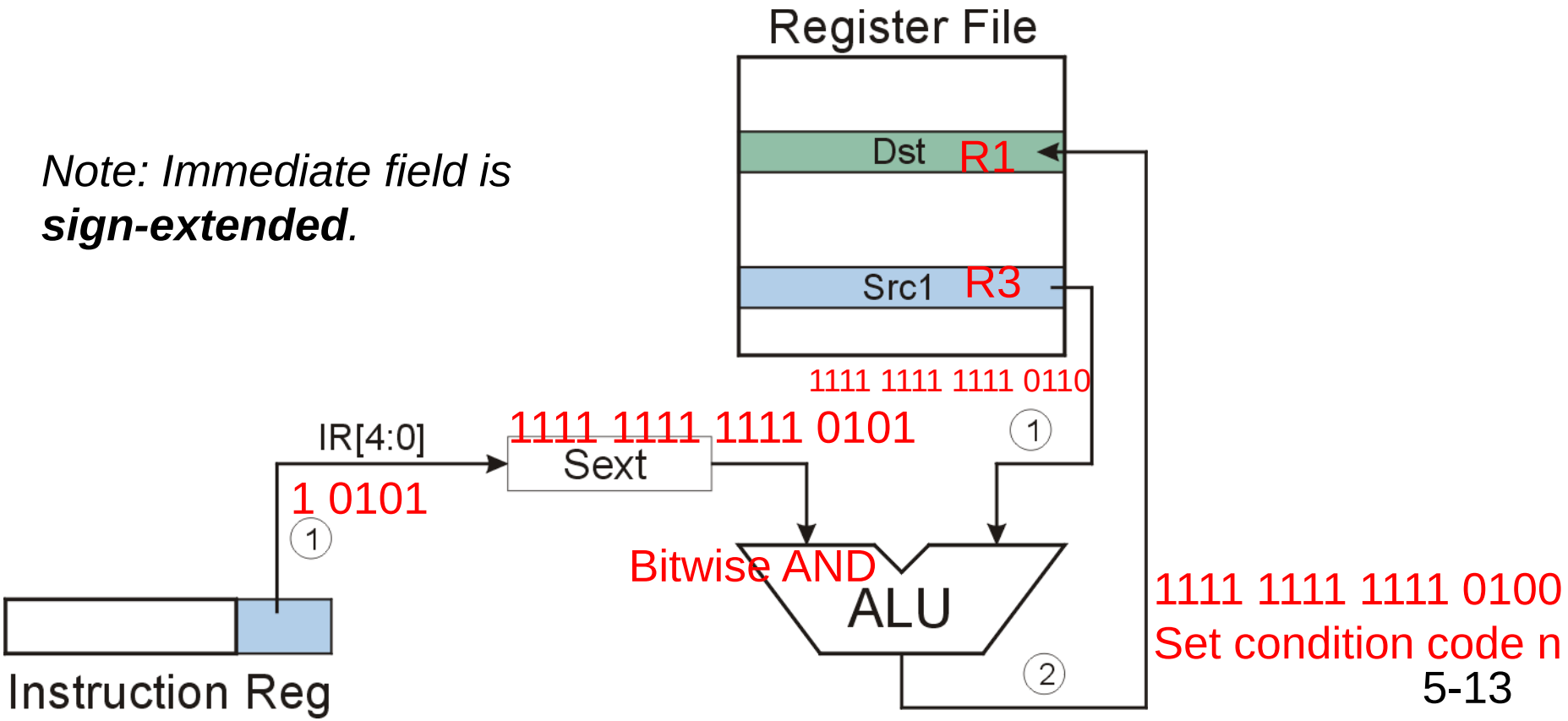
ADD/AND (Immediate) Example

R3 is currently 1111 1111 1111 0110

What is the contents of r1 after the following instruction is executed?



Note: Immediate field is ***sign-extended****.*



Using Operate Instructions

With only ADD, AND, NOT…

- **How do we subtract?**

  Subtract: R3 = R1 - R2
  Take 2's complement of R2, then add to R1.
  (1) R2 = NOT(R2)
  (2) R2 = R2 + 1
  (3) R3 = R1 + R2

- **How do we OR?**

  Use DeMorgan's Law -- invert R1 and R2, AND, then invert result.
  A OR B is same as NOT(NOT(A) AND NOT(B))

  OR: R3 = R1 OR R2
  (1) R1 = NOT(R1)
  (2) R2 = NOT(R2)
  (3) R3 = R1 AND R2
  (4) R3 = NOT(R3)

- **How do we copy from one register to another?**

  Register-to-register copy: R3 = R2
  R3 = R2 + 0 (Add-immediate) OR
  R3 = R2 AND FFFF (AND-immediate)

- **How do we initialize a register to zero?**

  Initialize to zero: R1 = 0
  R1 = R1 AND 0 (And-immediate)

Data Movement Instructions

Load -- read data from memory to register

- **LD:** PC-relative mode
- **LDR:** base+offset mode
- **LDI:** indirect mode

Store -- write data from register to memory

- **ST:** PC-relative mode
- **STR:** base+offset mode
- **STI:** indirect mode

Load effective address -- compute address, save in register

- **LEA:** immediate mode
- ***does not access memory***

PC-Relative Addressing Mode

Want to specify address directly in the instruction

- **But an address is 16 bits, and so is an instruction!**
- **After subtracting 4 bits for opcode and 3 bits for register, we have 9 bits available for address.**

Solution:

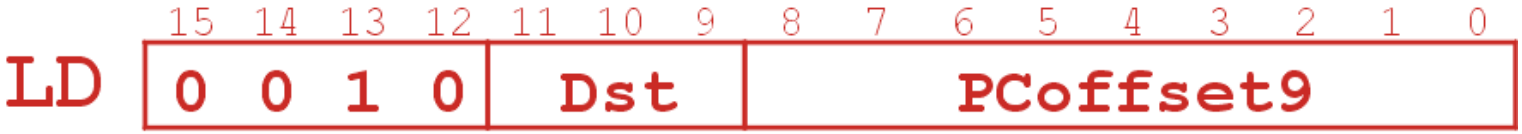
- **Use the 9 bits as a *signed offset* from the current PC.**

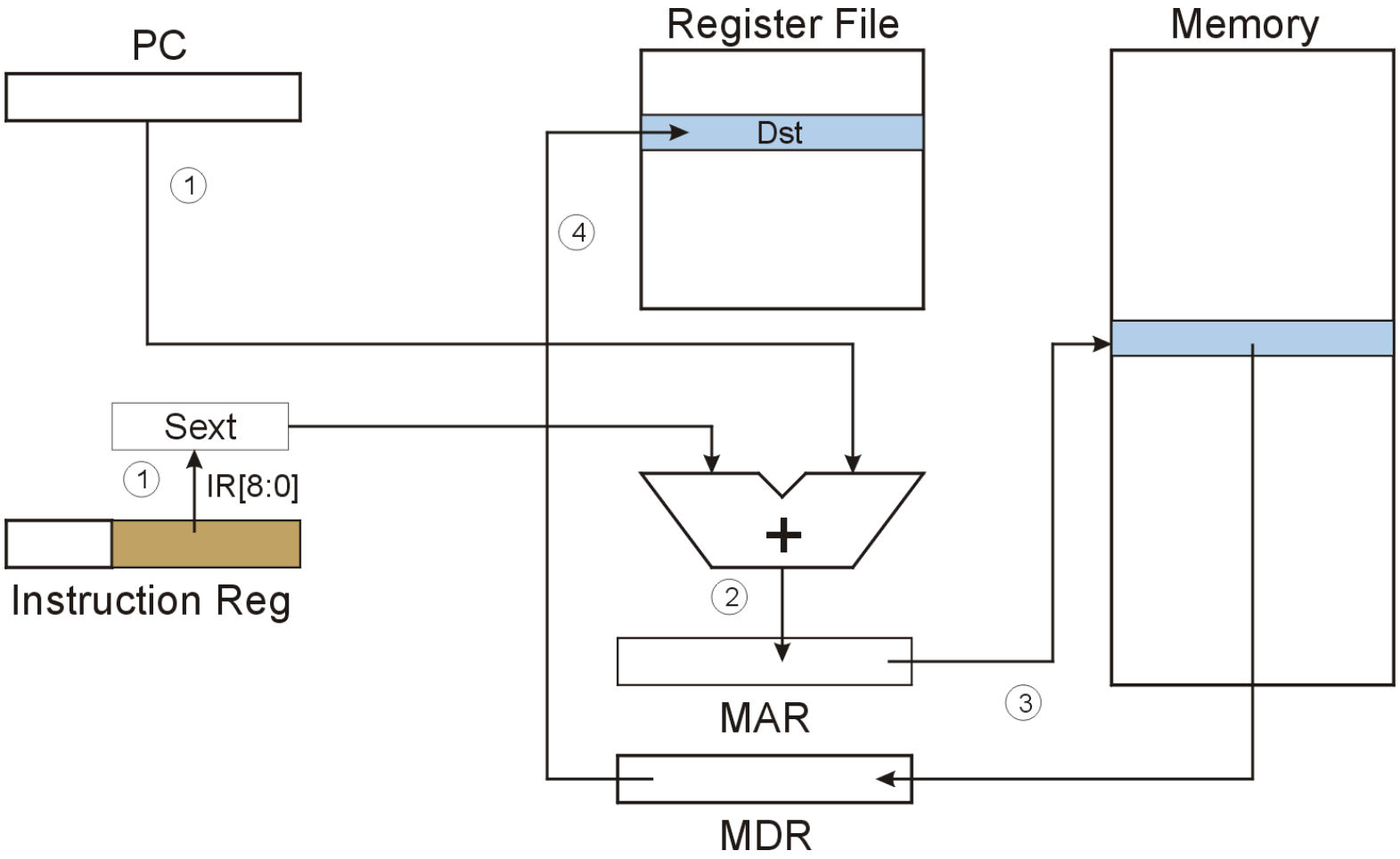
9 bits: $-256 \leq \text{offset} \leq +255$

Can form any address X, such that: $PC - 256 \leq X \leq PC + 255$

Remember that PC is incremented as part of the FETCH phase;
This is done before the EVALUATE ADDRESS stage.

LD (PC-Relative)



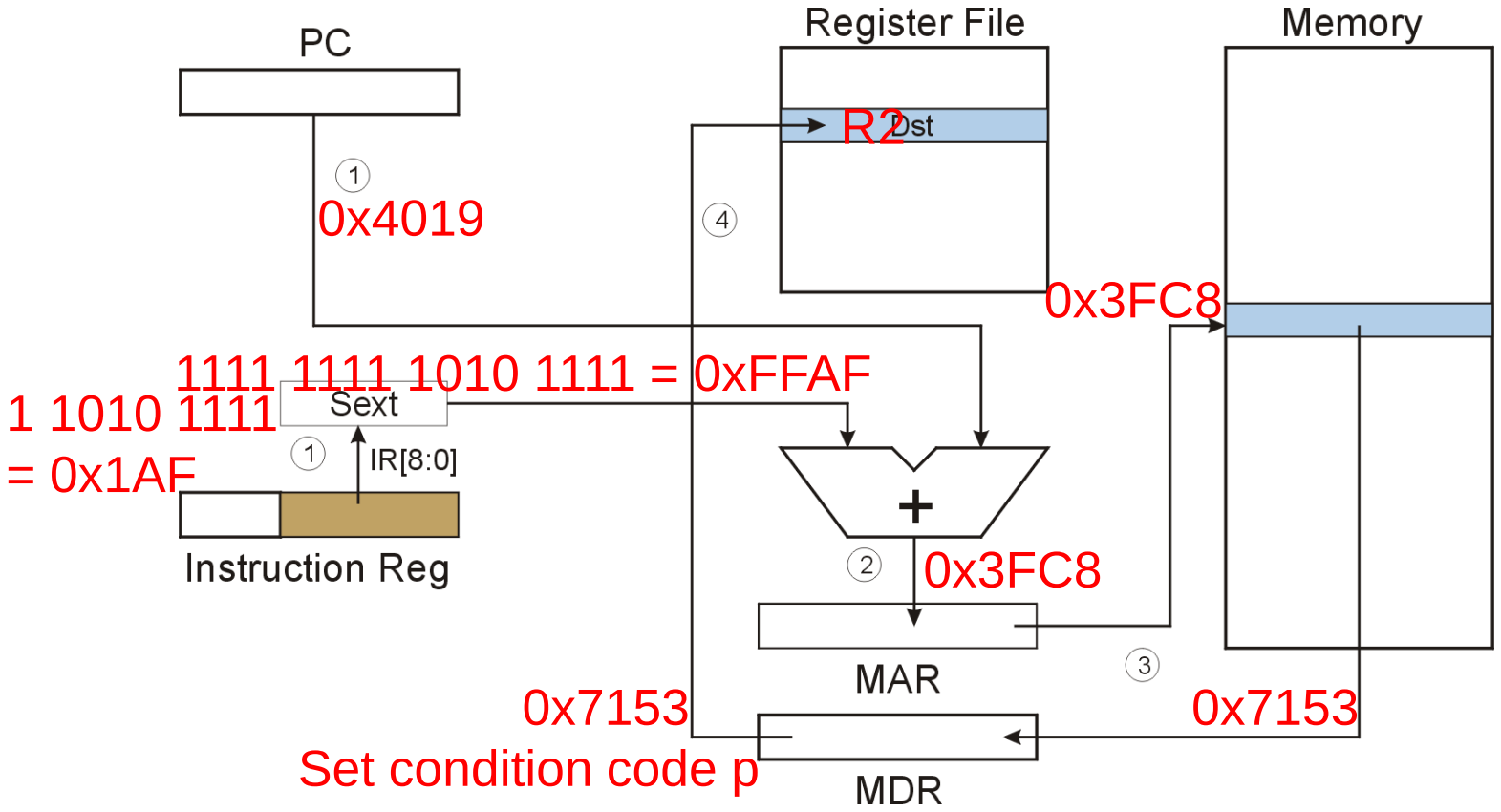


LD (PC-Relative) Example

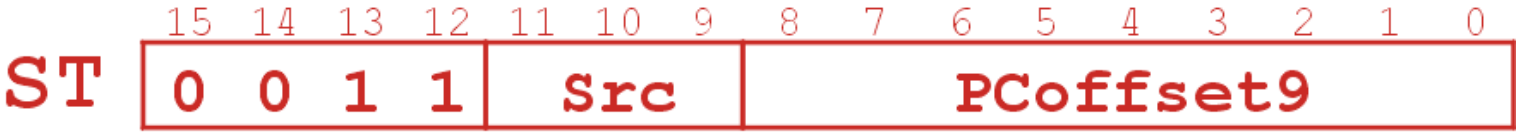
PC is currently 0x4018

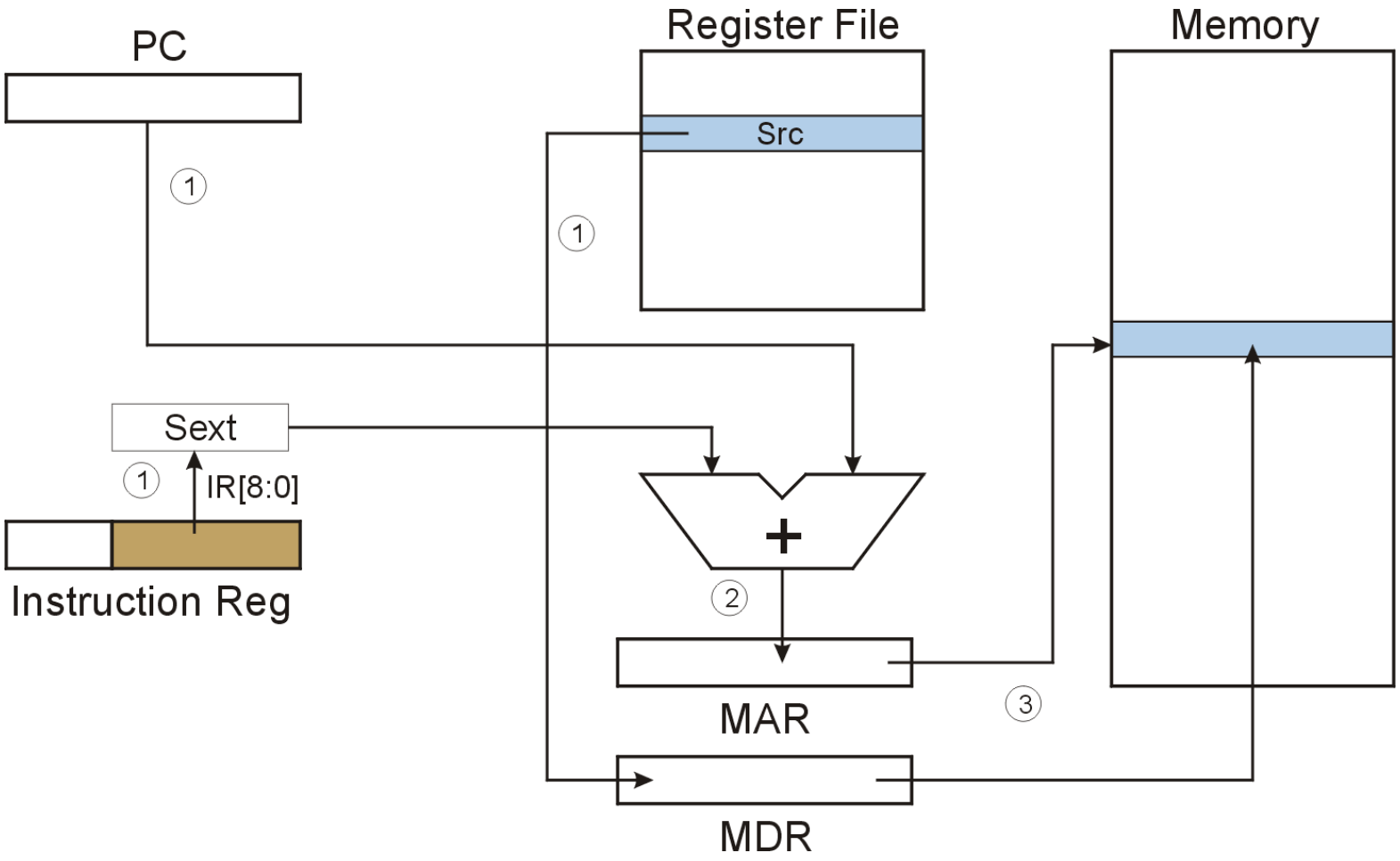
M[0x3FC8] is currently 0x7153

What is the contents of R2 after the following instruction is executed?



ST (PC-Relative)



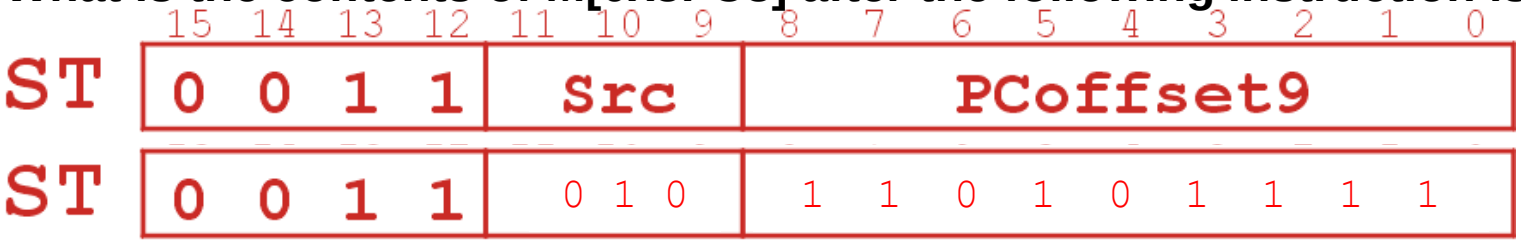


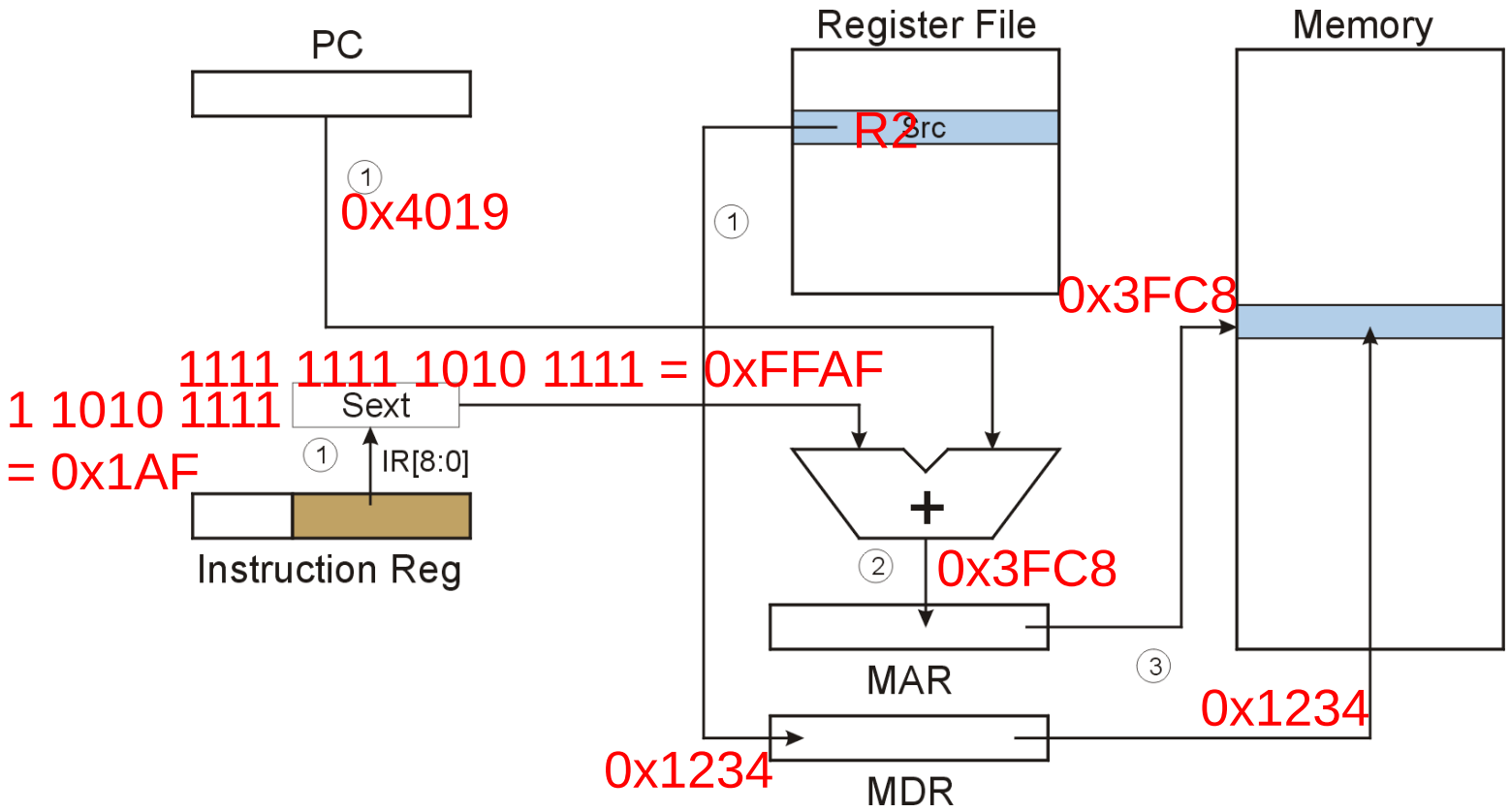
ST (PC-Relative) Example

PC is currently 0x4018

R2 is currently 0x1234

What is the contents of M[0x3FC8] after the following instruction is executed?





Indirect Addressing Mode

With PC-relative mode, can only address data within 256 words of the instruction.

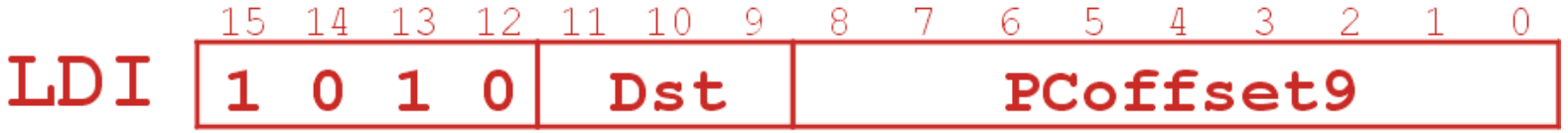
- **What about the rest of memory?**

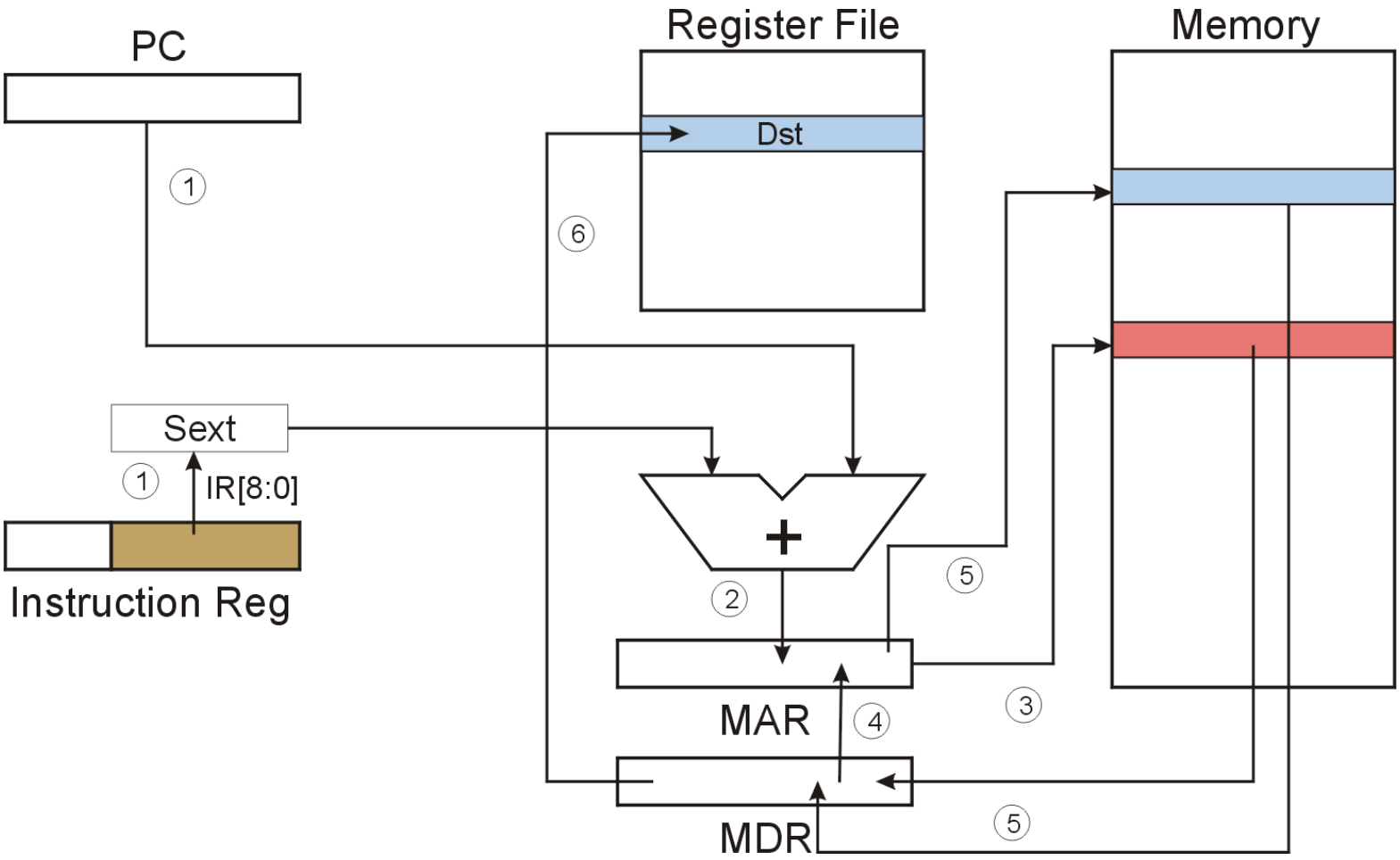
Solution #1:

- **Read address from memory location, then load/store to that address.**

First address is generated from PC and IR (just like PC-relative addressing), then content of that address is used as target for load/store.

LDI (Indirect)





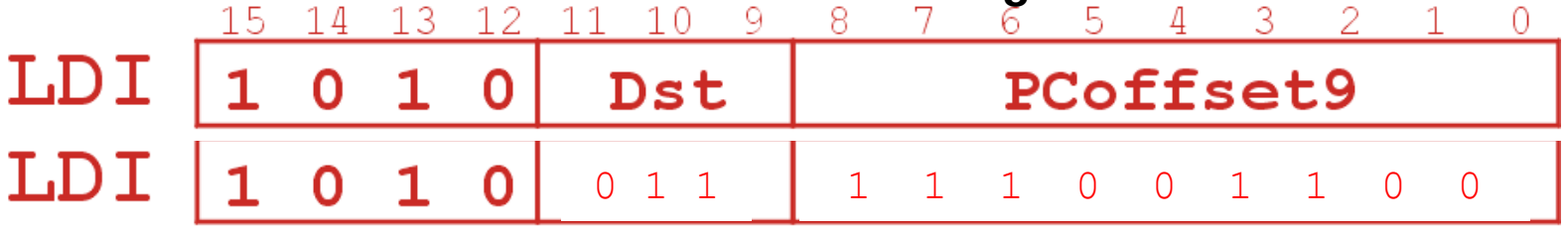
LDI (Indirect) Example

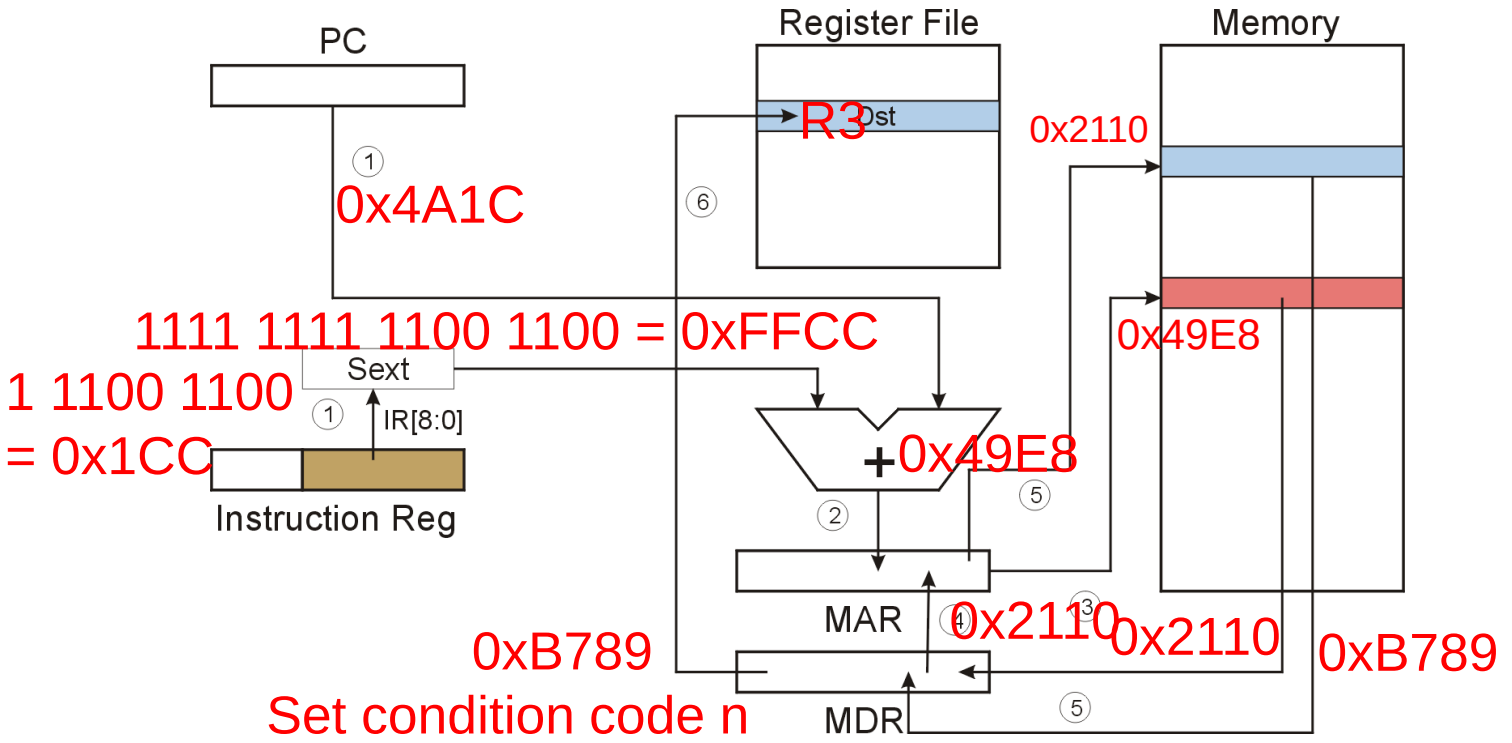
PC is currently 0x4A1B

M[0x49E8] is currently 0x2110

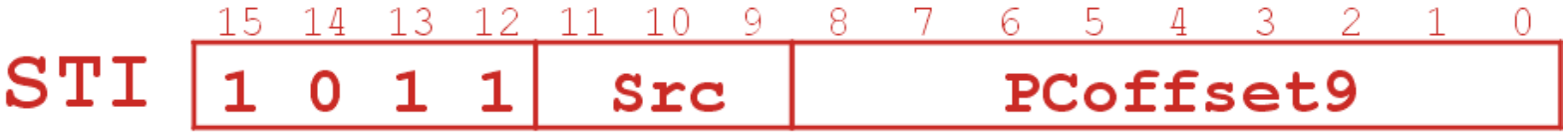
M[0x2110] is currently 0xB789

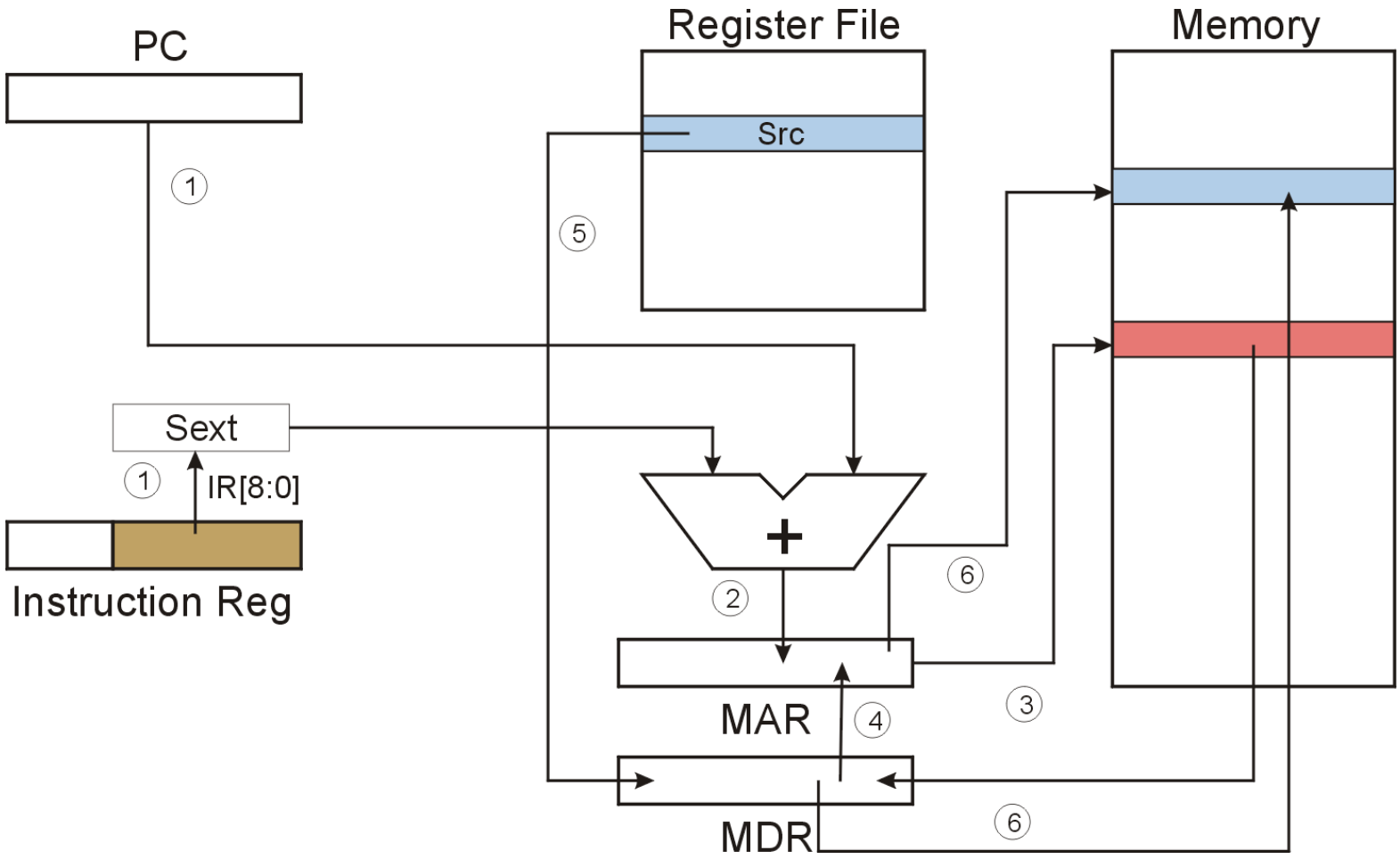
What is the contents of R3 after the following instruction is executed?





STI (Indirect)





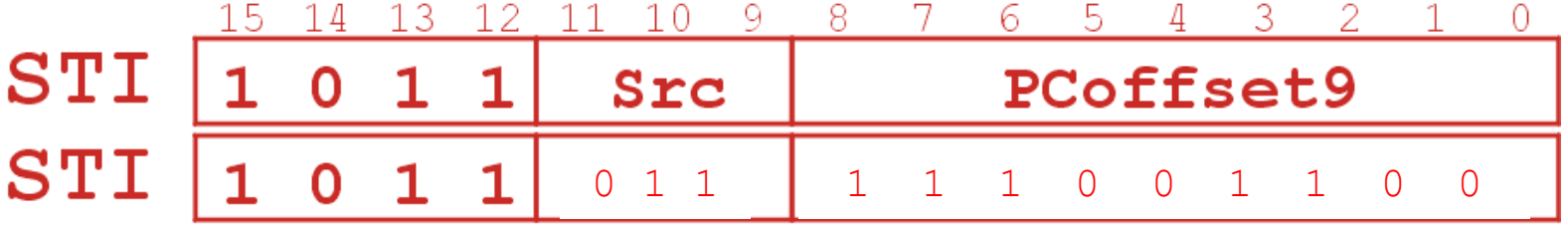
STI (Indirect) Example

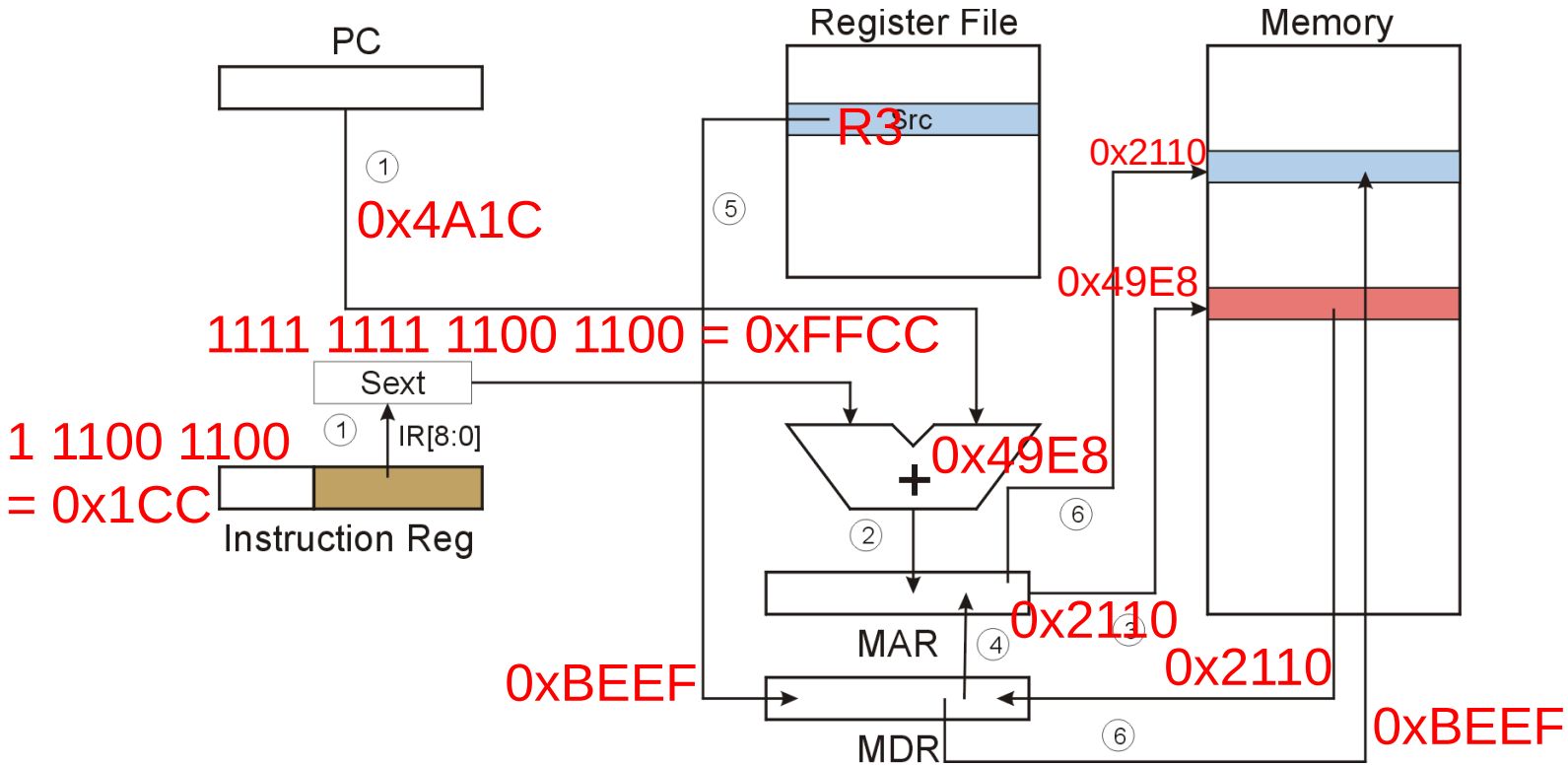
PC is currently 0x4A1B

M[0x49E8] is currently 0x2110

R3 is currently 0xBEEF

What is the contents of M[0x2110] after the following instruction is executed?





Points Covered So far…

- Understanding abstraction layer
- Importance of ISA
  - Using the same program in different machine organizations
  - ISA provides all information needed for someone that wants to write a program in machine language
- Instructions
  - Fixed or Variable length
- Operate Instructions
  - AND ,ADD and NOT
  - Using these to perform SUB, OR … etc
- Addressing modes
  - Non-memory addresses: register, immediate
  - Memory addresses: PC offset, indirect, base+Offset

Base + Offset Addressing Mode

With PC-relative mode, can only address data within 256 words of the instruction.

- **What about the rest of memory?**

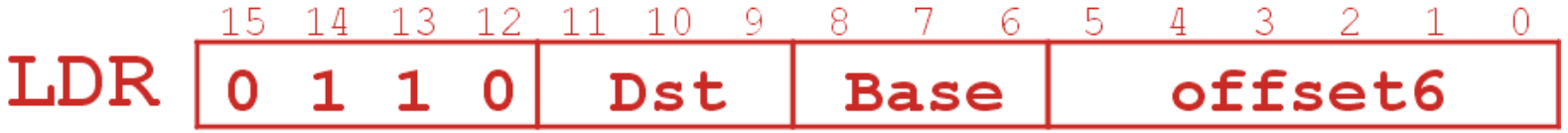
Solution #2:

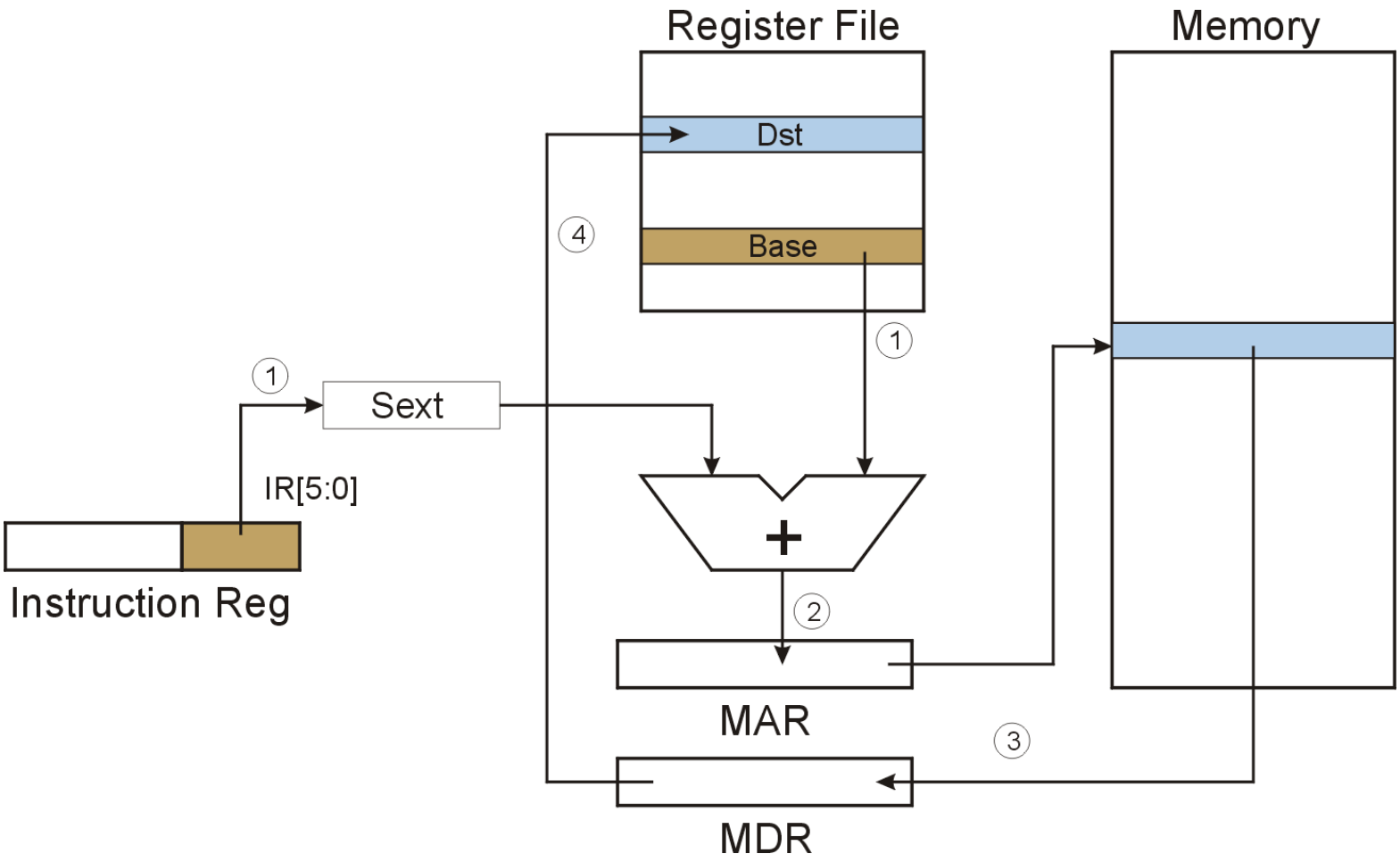
- **Use a register to generate a full 16-bit address.**

4 bits for opcode, 3 for src/dest register,
3 bits for *base* register -- remaining 6 bits are used as a *signed offset*.

- Offset is *sign-extended* before adding to base register.

LDR (Base+Offset)





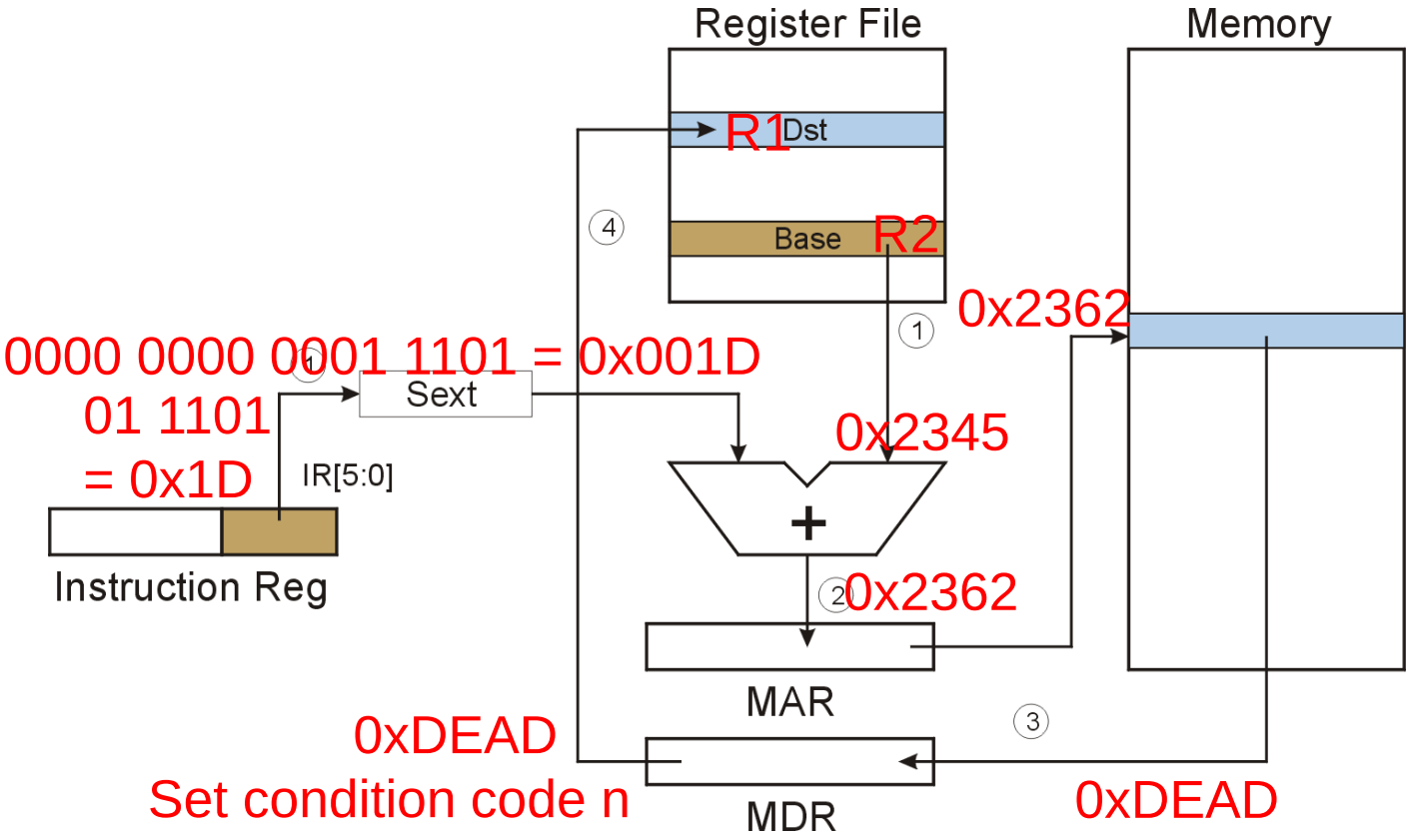
LDR (Base+Offset) Example

R2 is currently 0x2345

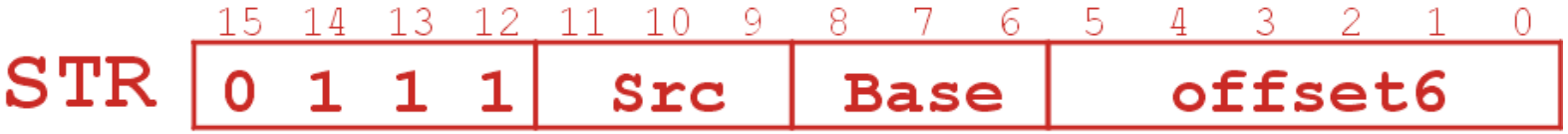
M[0x2362] is currently 0xDEAD

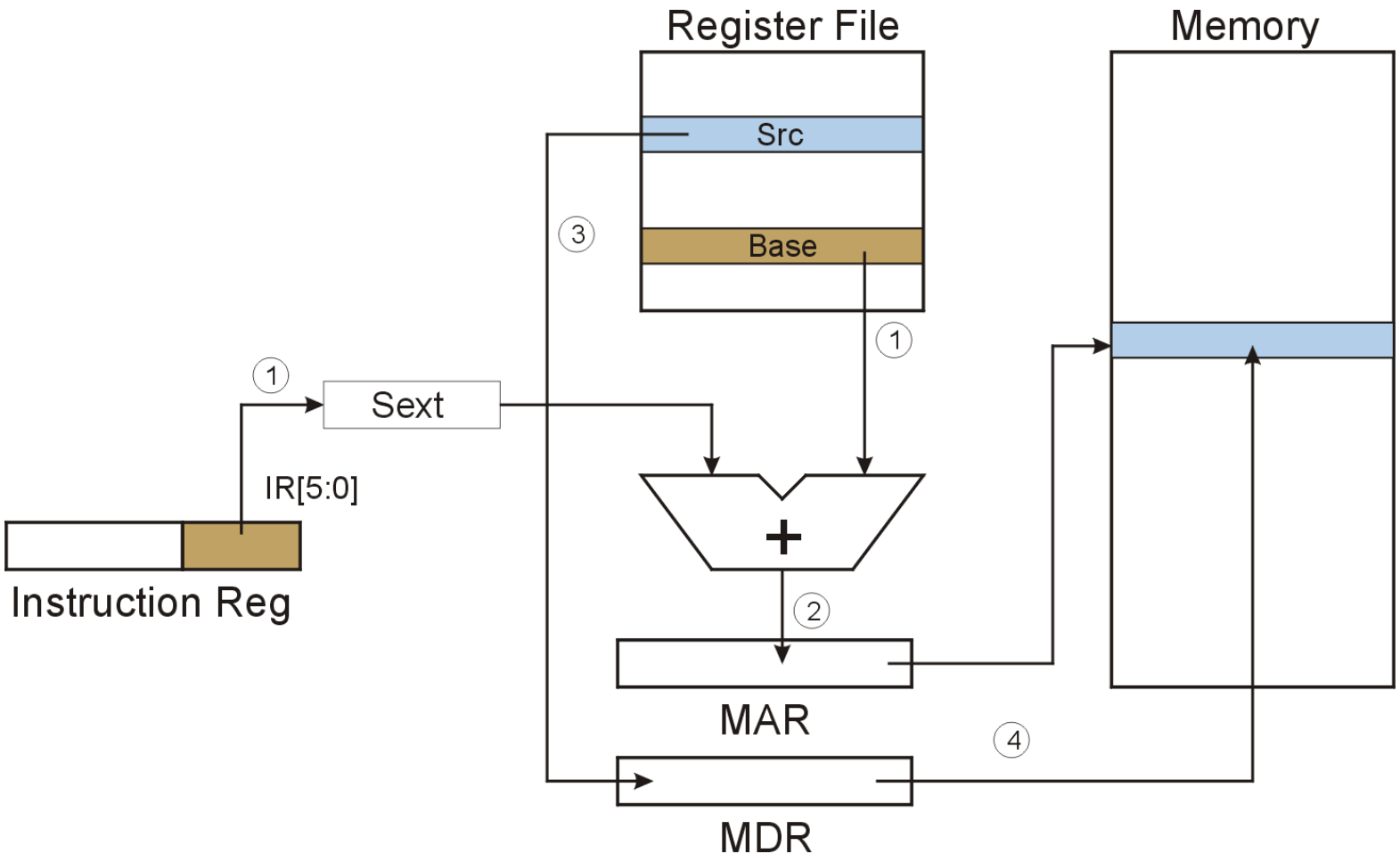
What is the contents of R1 after the following instruction is executed?

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LDR	0	1	1	0	Dst			Base			offset6					
LDR	0	1	1	0	0	0	1	0	1	0	0	1	1	1	0	1



STR (Base+Offset)



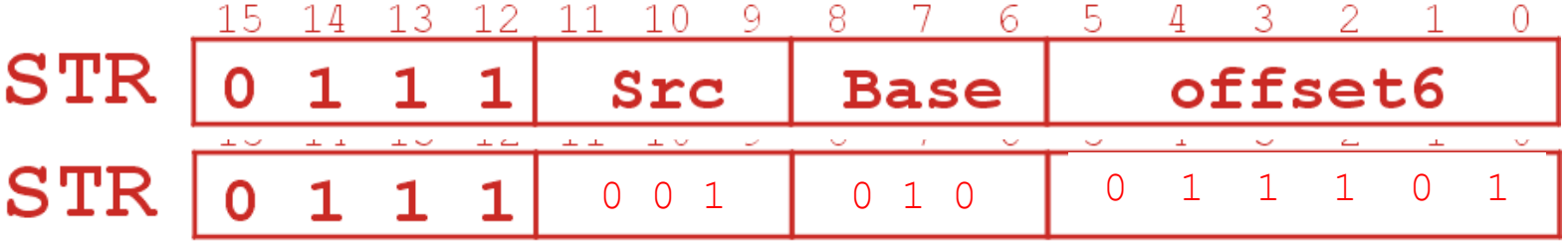


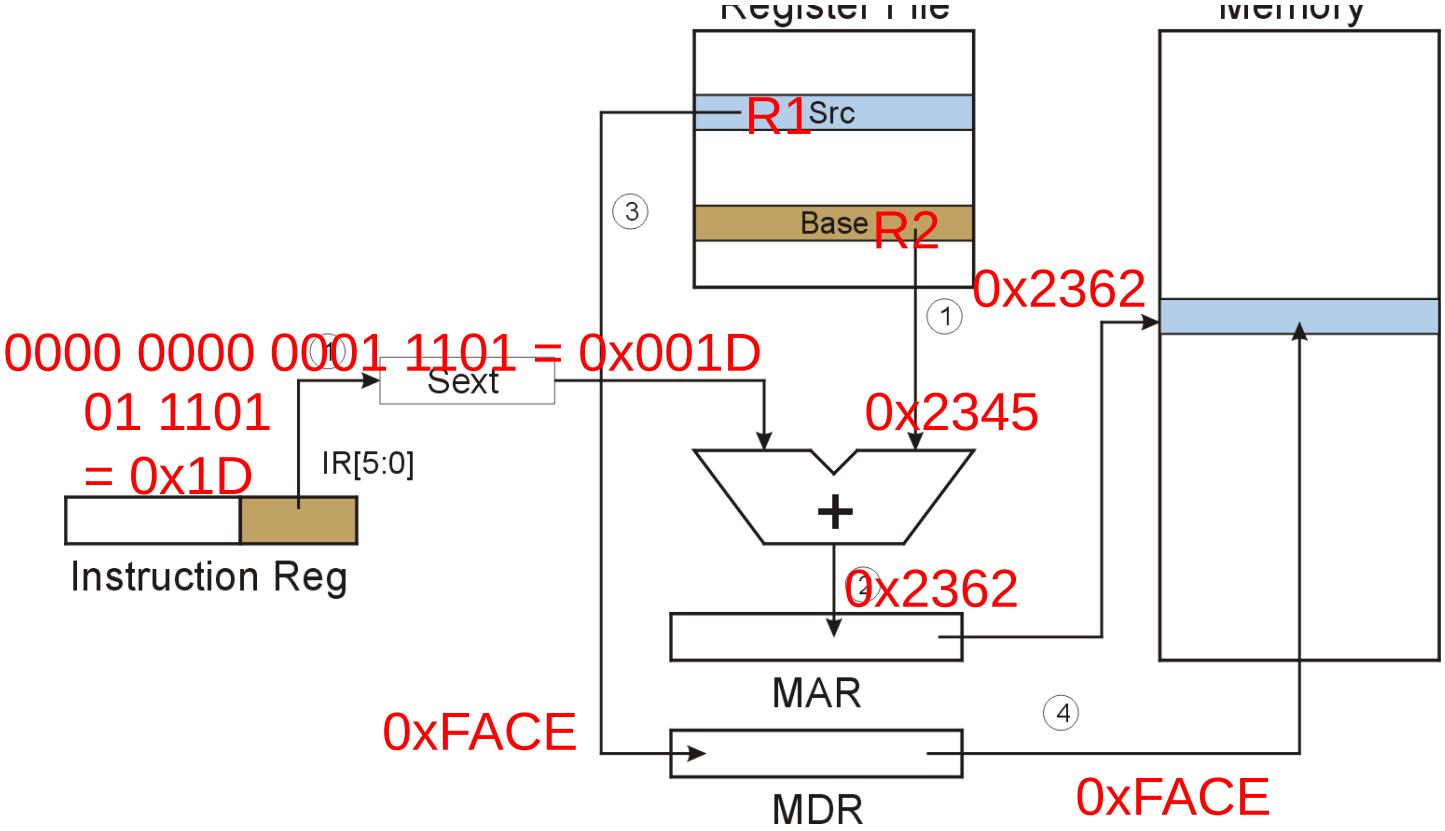
STR (Base+Offset) Example

R2 is currently 0x2345

R1 is currently 0xFACE

What is the contents of M[0x2362] after the following instruction is executed?

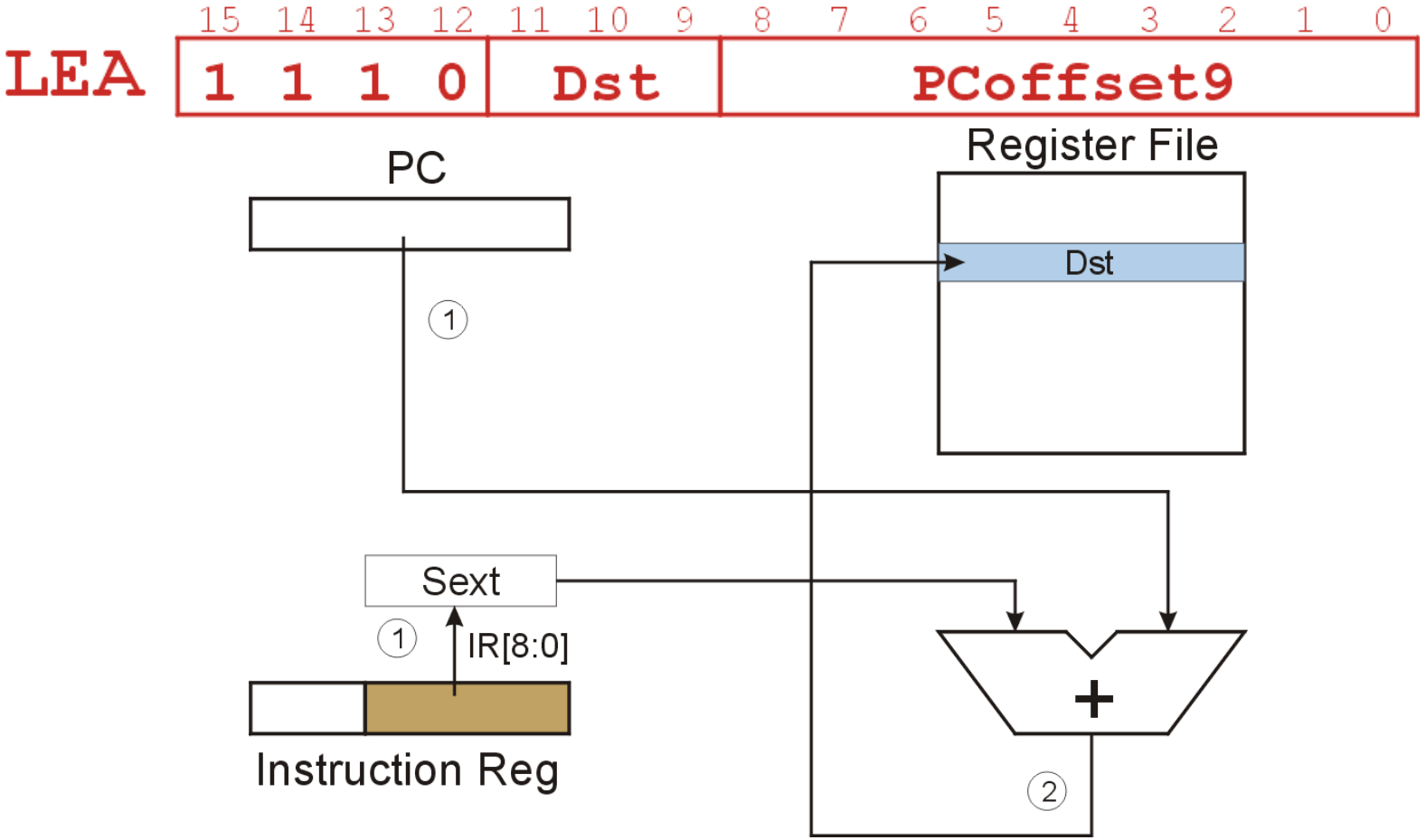




Load Effective Address

Computes address like PC-relative (PC plus signed offset) and stores the result into a register.

Note: The *address* is stored in the register, not the contents of the memory location.

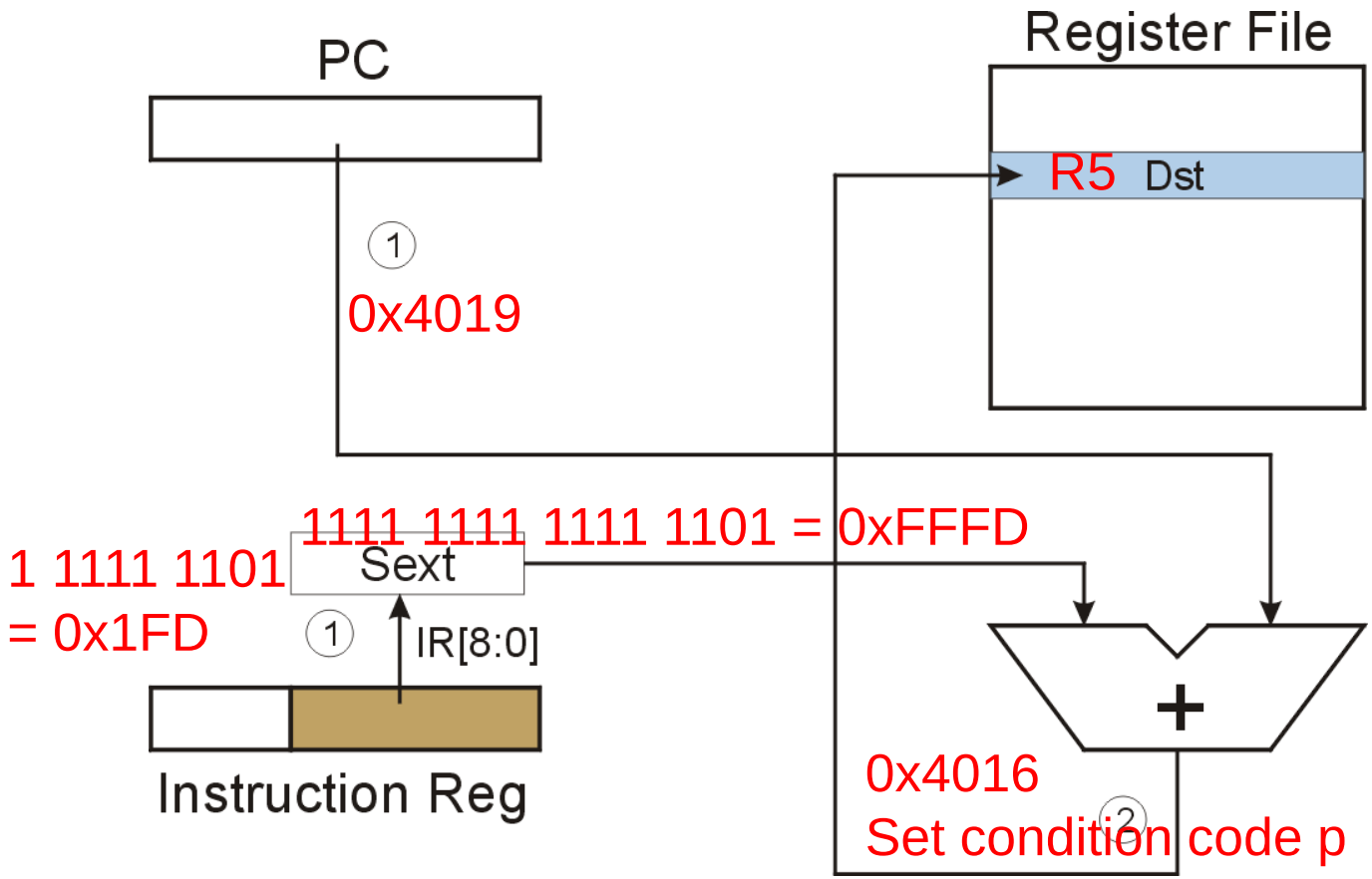


LEA (Immediate) Example

PC is currently 0x4018

What is the contents of R5 after the following instruction is executed?

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LEA	1	1	1	0	Dst			PCoffset9								
LEA	1	1	1	0	1	0	1	1	1	1	1	1	1	1	0	1



```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+ LEA, DR, label ; Load Effective Address
| 1   1   1   0 |    DR     |                PCoffset9              |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+ DR ← PC' + SEXT(PCoffset9) also setcc()
```

Address	Instruction	Comments
x30F6	1 1 1 0 0 0 1 1 1 1 1 1 1 1 0 1 LEA R1 -3	R1 ← PC – 3 = x30F4
x30F7	0 0 0 1 0 1 0 0 0 1 1 0 1 1 1 0 ADD R2 R1 imm 14=xE	R2 ← R1 + 14 = x3102
x30F8	0 0 1 1 0 1 0 1 1 1 1 1 1 1 0 1 1 ST R2 -5	M[PC - 5] ← R2 M[x30F4] ← x3102
x30F9	0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0 AND R2 R2 imm 0	R2 ← 0
x30FA	0 0 0 1 0 1 0 0 1 0 1 0 0 1 0 1 ADD R2 R2 imm 5	R2 ← R2 + 5 = 5
x30FB	0 1 1 1 0 1 0 0 0 1 0 0 1 1 1 0 STR R2 R1 14=xE	M[R1+14] ← R2 M[x3102] ← 5
x30FC	1 0 1 0 0 1 1 1 1 1 1 1 0 1 1 1 LDI R3 -9	R3 ← M[M[x30F4]] R3 ← M[x3102] R3 ← 5

opcode

Control Instructions

Used to alter the sequence of instructions (by changing the Program Counter)

Conditional Branch

- **branch is *taken* if a specified condition is true**
  - **signed offset is added to PC to yield new PC**
- **else, the branch is *not taken***
  - **PC is not changed, points to the next sequential instruction**

Unconditional Branch (or Jump)

- **always changes the PC**

TRAP

- **changes PC to the address of an OS "service routine"**
- **routine will return control to the next instruction (after TRAP)**

Condition Codes

LC-3 has three condition code registers:

- **N -- negative**
- **Z -- zero**
- **P -- positive (greater than zero)**

Set by any instruction that writes a value to a register (ADD, AND, NOT, LD, LDR, LDI, LEA)

Exactly one will be set at all times

- **Based on the last instruction that altered a register**

Branch Instruction

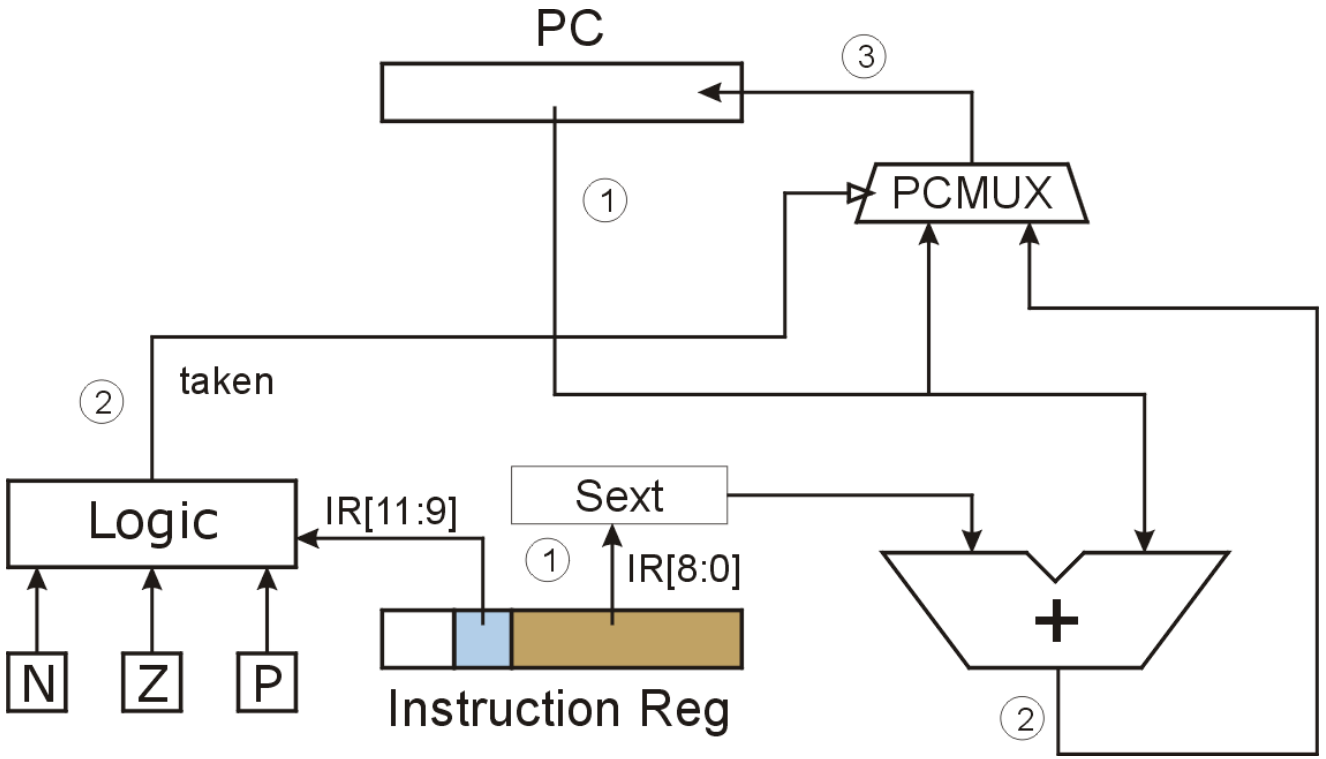
Branch specifies one or more condition codes.

If the set bit is specified, the branch is taken.

- **PC-relative addressing:**
  target address is made by adding signed offset (IR[8:0])
  to current PC.
- **Note: PC has already been incremented by FETCH stage.**
- **Note: Target must be within 256 words of BR instruction.**

If the branch is not taken,
the next sequential instruction is executed.

BR (PC-Relative)



If all zero, no CC is tested, so branch is never taken. (See Appendix B.)

What happens if bits [11:9] are all zero? All one?

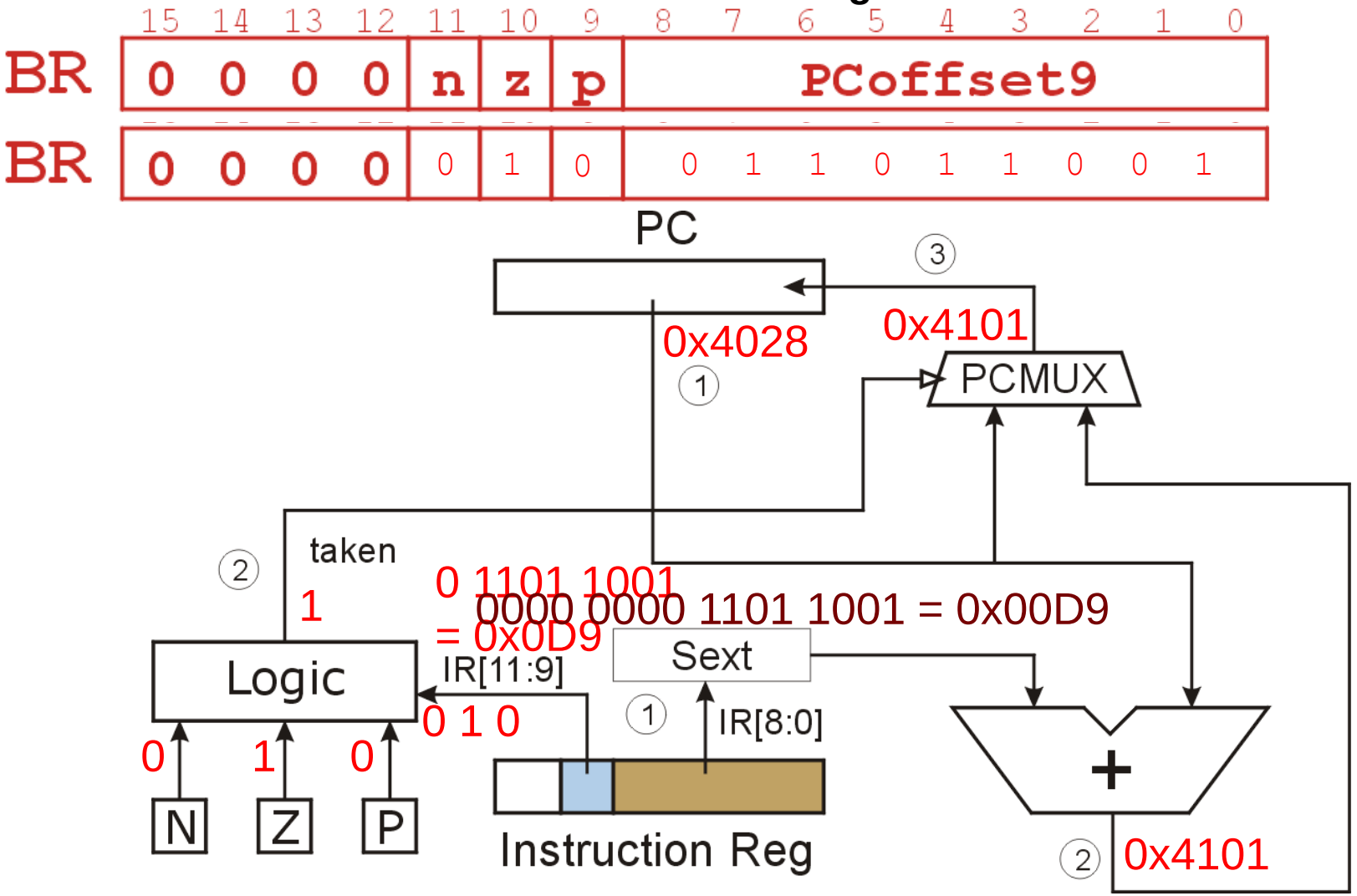
If all one, then all are tested. Since at least one of the CC bits is set to one after each operate/load instruction, then branch is always taken.

BR (PC-Relative) Example

PC is currently 0x4027

The last value loaded into a general purpose register was 0.

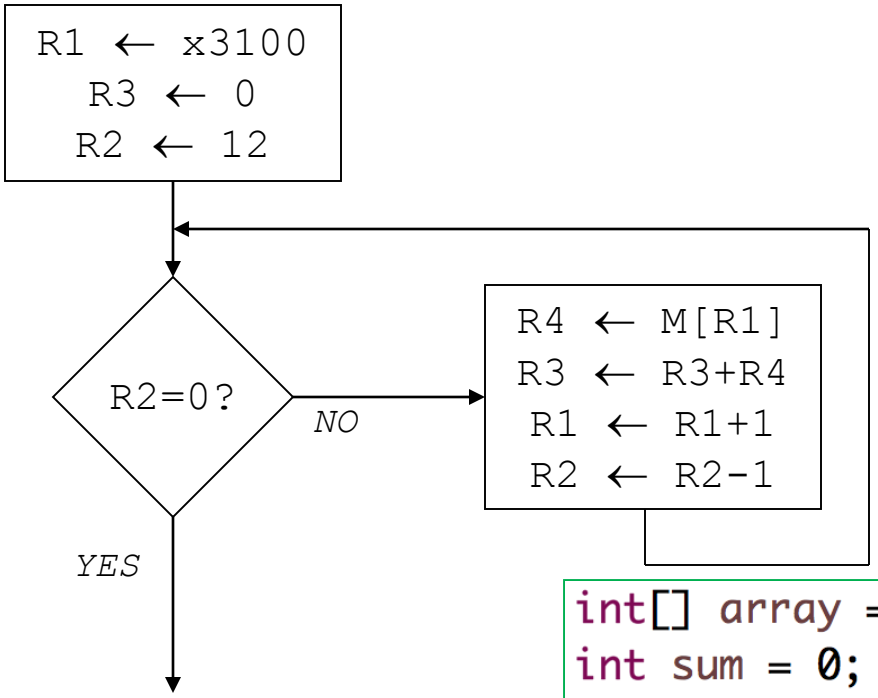
What is the contents of PC after the following instruction is executed?



Using Branch Instructions

Compute sum of 12 integers.

Numbers start at location x3100. Program starts at location x3000.




```
int[] array = {2, 4, 7, 4, 9, 0, -2, 3, 5, 7, 1, 7};
int sum = 0;                        // R3 = 0
int pointer = 0;                    // R1 = index into array
int counter = 12;                   // R2 = counter = 12
while (counter > 0) {               // BRz
    sum = sum + array[pointer]; // R4 = array[pointer]
                                    // R3 = R3 + R4
    pointer = pointer + 1;      // R1 = R1 + 1
    counter = counter - 1;      // R2 = R2 + 1
}
```

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+ ADD DR, SR1, imm5 ; Addition with Immediate
| 0   0   0   1 |    DR     |    SR1    | 1 |       imm5        |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+ DR ← SR1 + SEXT(imm5) also setcc()
```

In-class Exercise: Write comments for each instruction

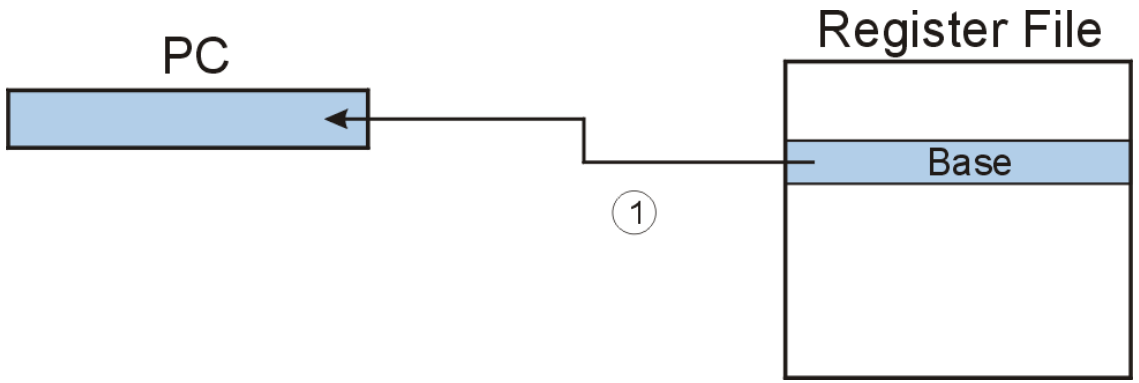
Address	Instruction	Comments
x3000	1 1 1 0 0 0 1 0 1 1 1 1 1 1 1 1 (LEA)	*R1 ← x3100 (PC+0xFF)*
x3001	0 1 0 1 0 1 1 0 1 1 1 0 0 0 0 0 (AND imm)	*R3 ← 0*
x3002	0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0 (AND imm)	*R2 ← 0*
x3003	0 0 0 1 0 1 0 0 1 0 1 0 1 1 0 0 (ADD imm)	*R2 ← 12*
x3004	0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1 (BRz)	*If Z, goto x300A (PC+5)*
x3005	0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0 (LDR)	*Load next value to R4*
x3006	0 0 0 1 0 1 1 0 1 1 0 0 0 1 0 0 (ADD reg)	*Add to R3*
x3007	0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1 (ADD imm)	*Increment R1 (pointer)*
X3008	0 0 0 1 0 1 0 0 1 0 1 1 1 1 1 1 (ADD imm)	*Decrement R2 (counter)*
x3009	0 0 0 0 1 1 1 1 1 1 1 1 1 0 1 0 (BRnzp)	*Goto x3004 (PC-6)*

opcode

JMP (Register)

Jump is an unconditional branch -- *always* taken.

- **Target address is the contents of a register.**
- **Allows any target address.**



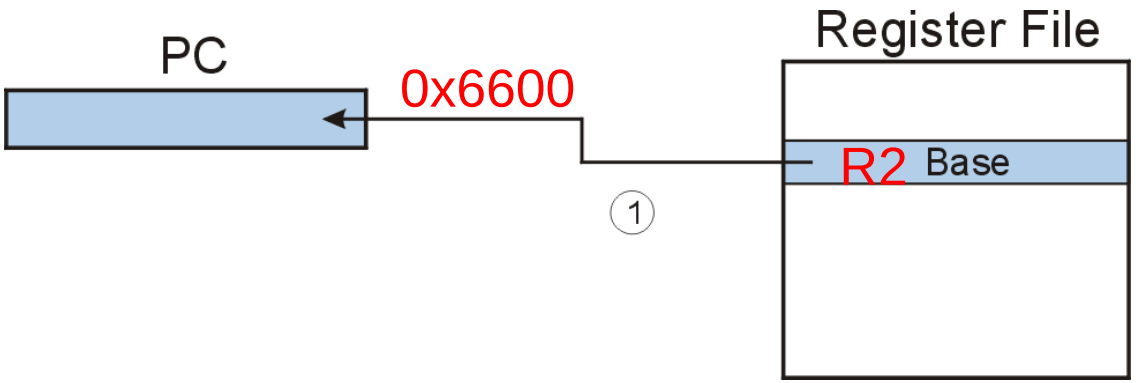
JMP (Register) Example

PC is currently 0x4000

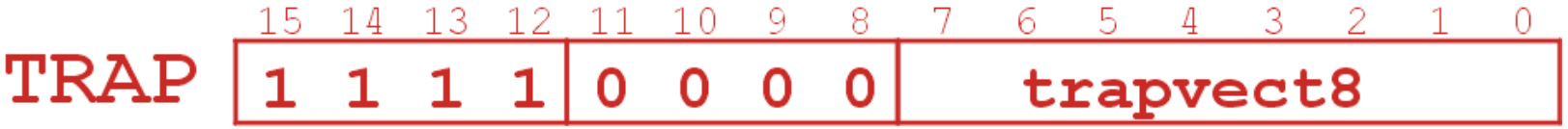
R2 is currently 0x6600

What is the contents of PC after the following instruction is executed?

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP	1	1	0	0	0	0	0	Base			0	0	0	0	0	0
JMP	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0



TRAP



Calls a service routine, identified by 8-bit "trap vector."

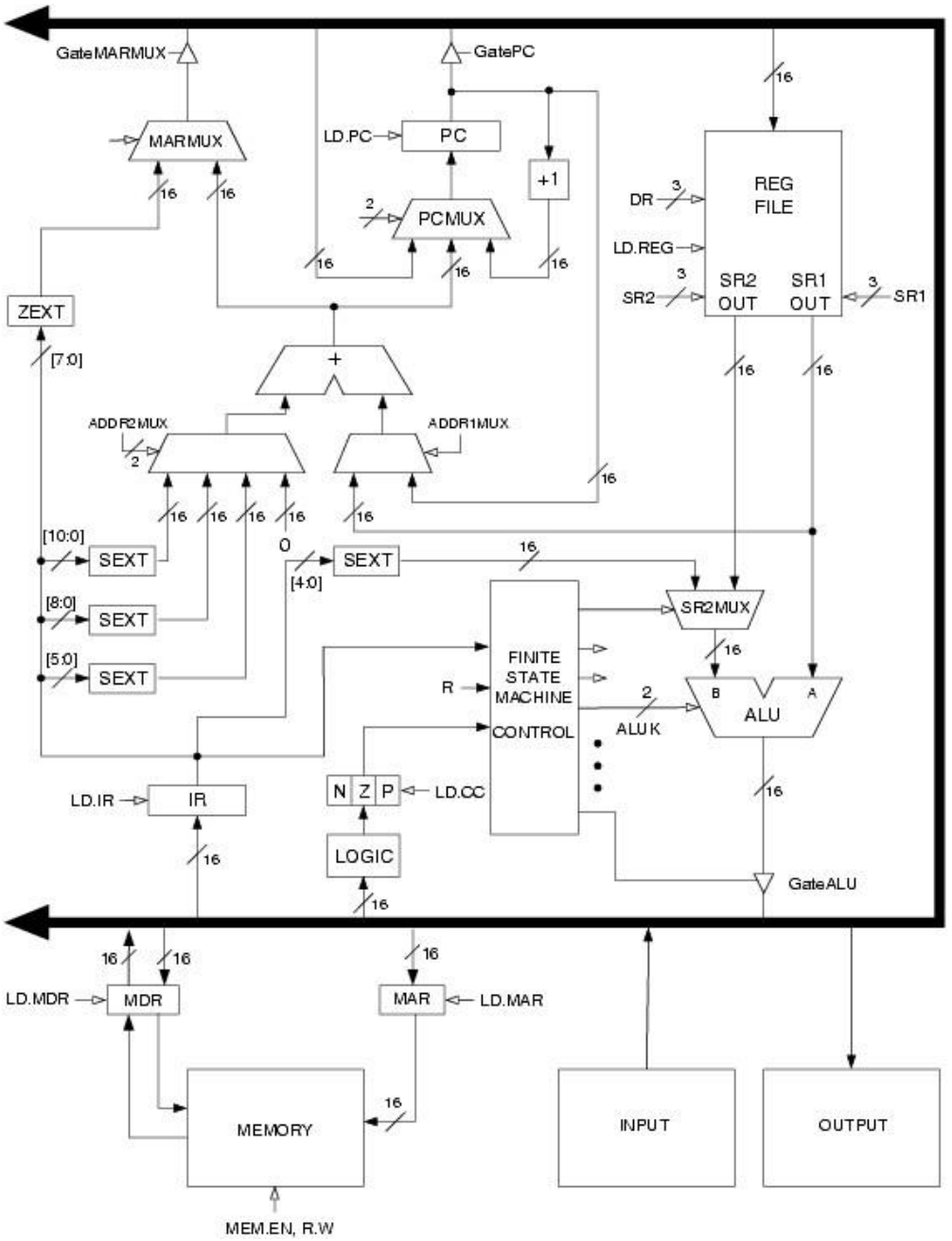
vector	*routine*
x23	input a character from the keyboard
x21	output a character to the monitor
x25	halt the program

When routine is done,
PC is set to the instruction following TRAP.

(We'll talk about how this works later.)

LC-3 Data Path Revisited

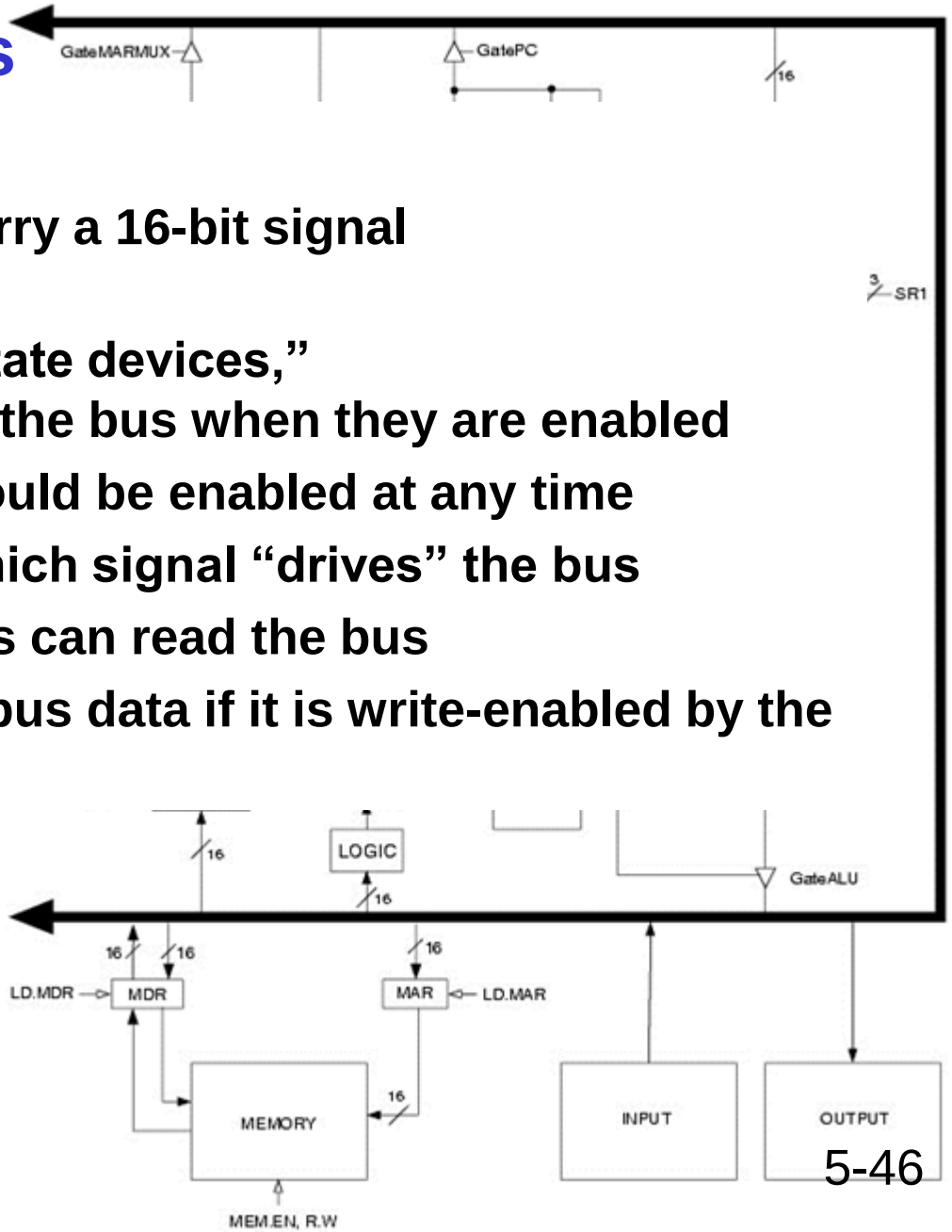
Filled arrow
= info to be processed.
Unfilled arrow
= control signal.



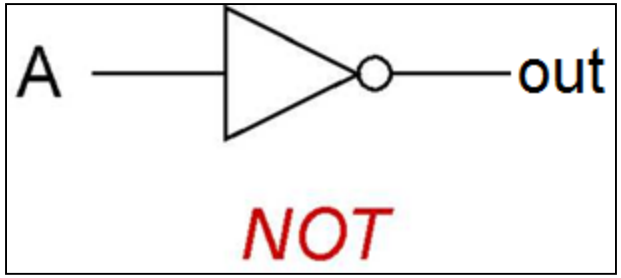
Data Path Components

Global bus

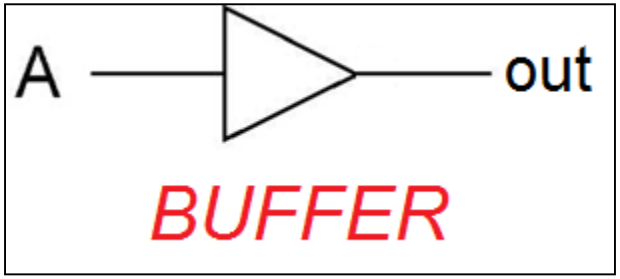
- special set of wires that carry a 16-bit signal to many components
- inputs to the bus are “tri-state devices,” that only place a signal on the bus when they are enabled
- only one (16-bit) signal should be enabled at any time
  - control unit decides which signal “drives” the bus
- any number of components can read the bus
  - register only captures bus data if it is write-enabled by the control unit



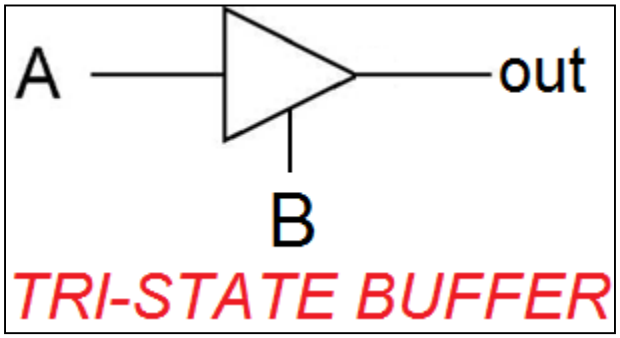
NOT Gate, Buffer, and Tri-state Buffer



A	out
0	1
1	0



A	out
0	0
1	1

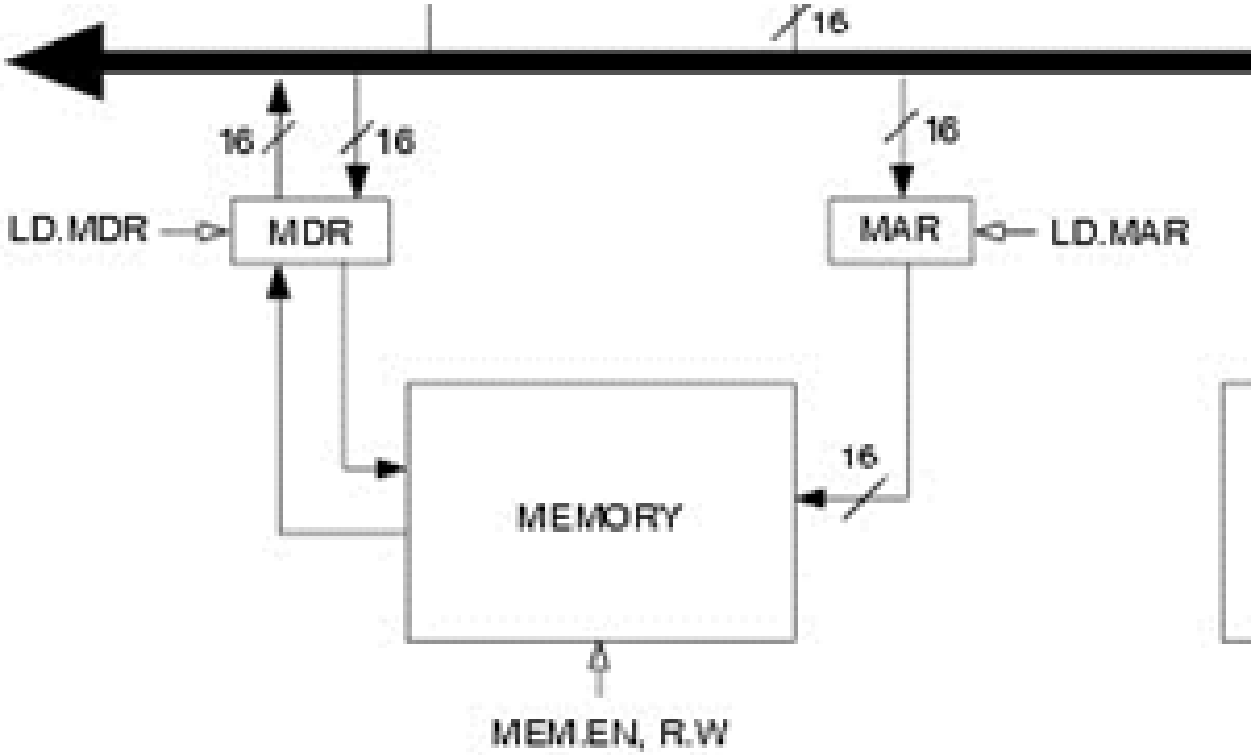


A	B	out
0	0	Z
0	1	0
1	0	Z
1	1	1

Data Path Components

Memory

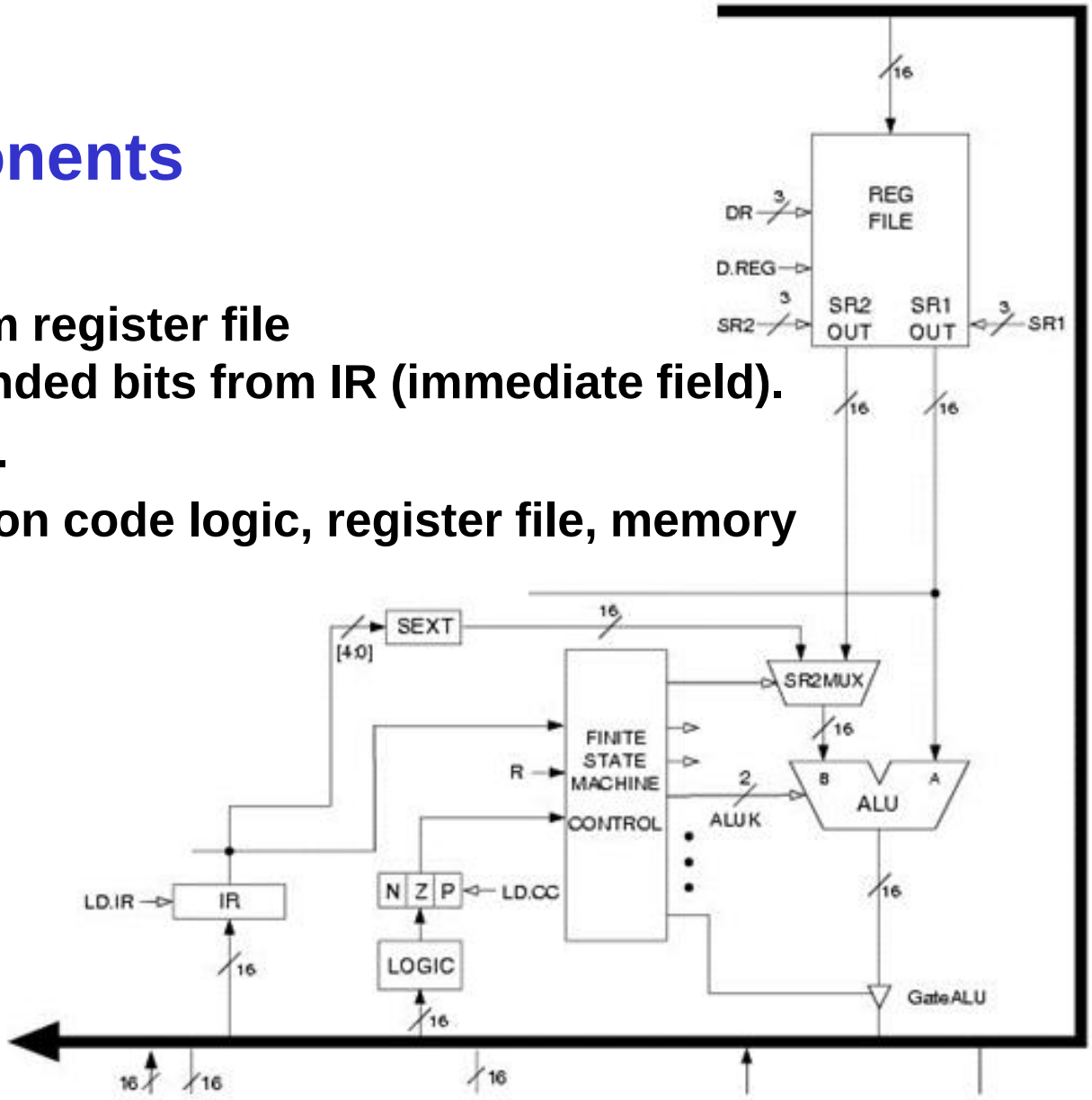
- Control and data registers for memory and I/O devices
- memory: MAR, MDR (also control signal for read/write)



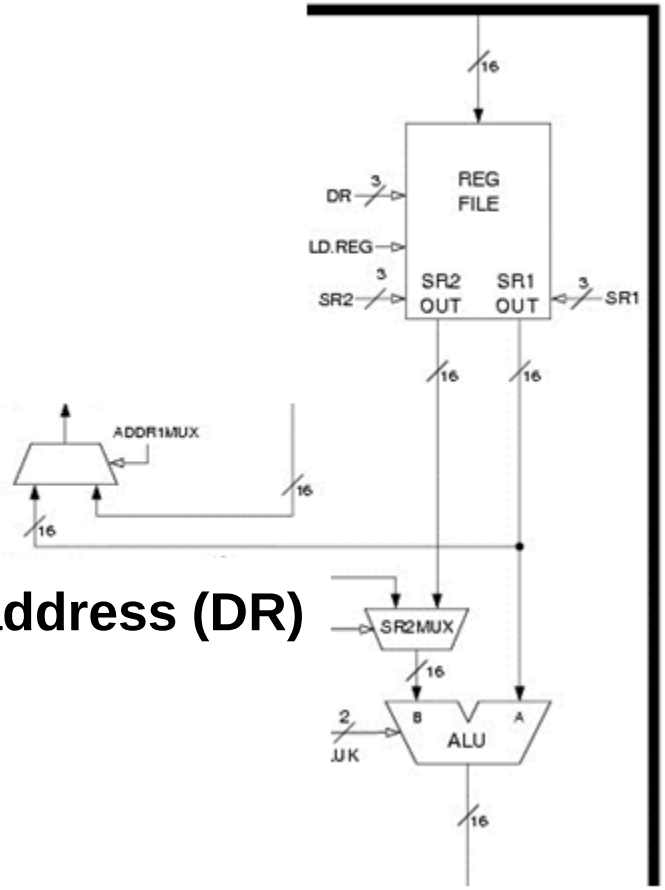
Data Path Components

ALU

- **Accepts inputs from register file and from sign-extended bits from IR (immediate field).**
- **Output goes to bus.**
  - **used by condition code logic, register file, memory**



Data Path Components



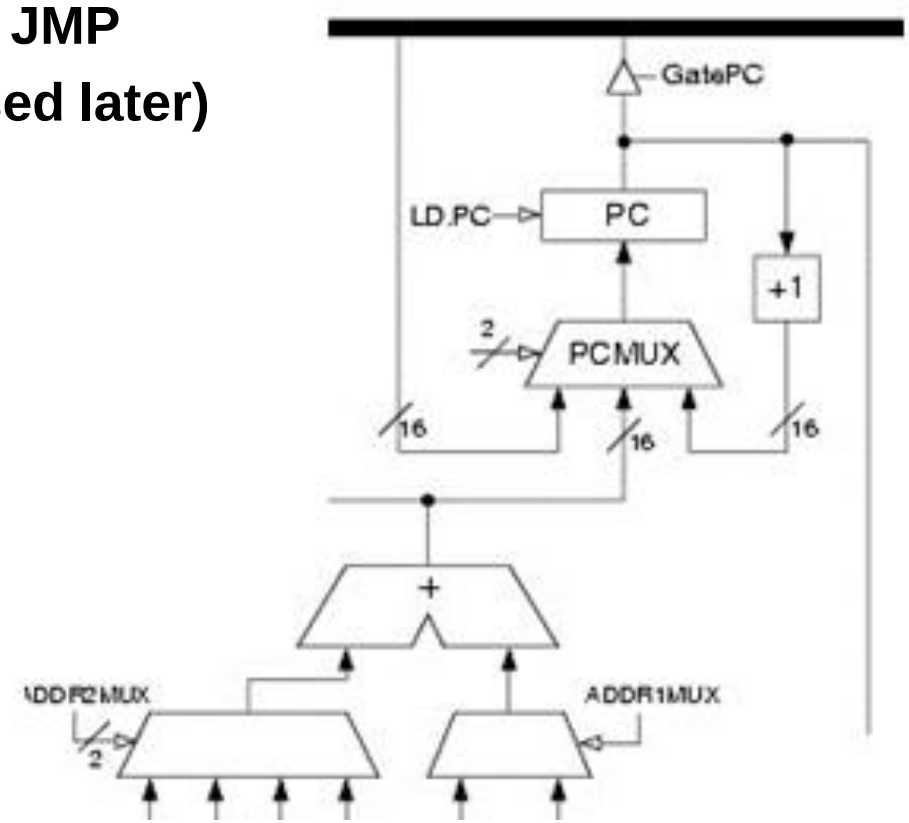
Register File

- Two read addresses (SR1, SR2), one write address (DR)
- Input from bus
  - result of ALU operation or memory read
- Two 16-bit outputs
  - used by ALU, PC, memory address
  - data for store instructions passes through ALU

Data Path Components

PC and PCMUX

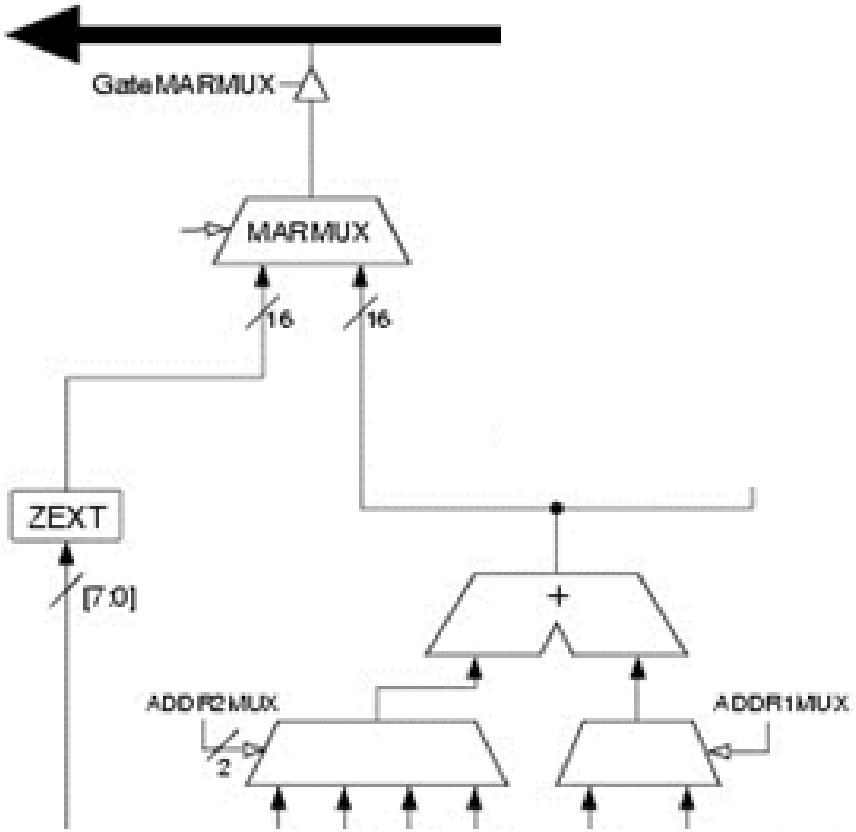
- **Three inputs to PC, controlled by PCMUX**
  1. **PC+1 – FETCH stage**
  2. **Address adder – BR, JMP**
  3. **bus – TRAP (discussed later)**



Data Path Components

MAR and MARMUX

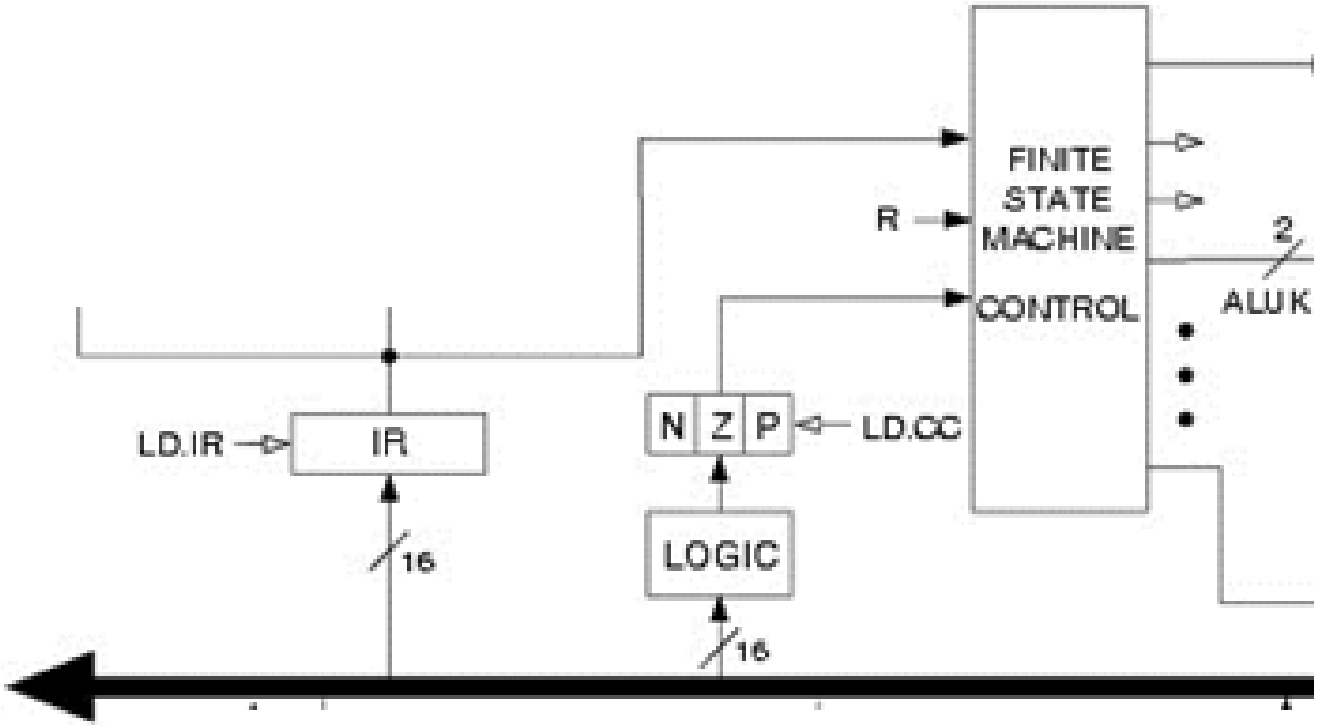
- **Two inputs to MAR, controlled by MARMUX**
  1. **Address adder – LD/ST, LDR/STR**
  2. **Zero-extended IR[7:0] -- TRAP (discussed later)**



Data Path Components

Condition Code Logic

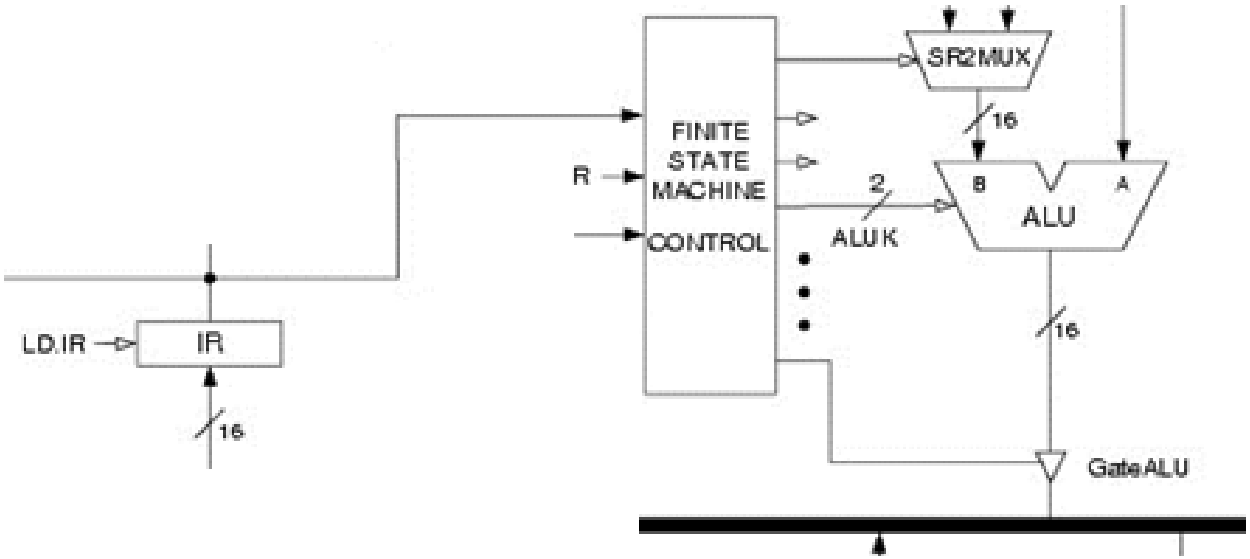
- **Looks at value on bus and generates N, Z, P signals**
- **Registers set only when control unit enables them (LD.CC)**
  - **only certain instructions set the codes**
    (ADD, AND, NOT, LD, LDI, LDR, LEA)



Data Path Components

Control Unit – Finite State Machine

- **On each machine cycle, changes control signals for next phase of instruction processing**
  - **who drives the bus?** (GatePC, GateALU, …)
  - **which registers are write enabled?** (LD.IR, LD.REG, …)
  - **which operation should ALU perform?** (ALUK)
  - **…**
- **Logic includes decoder for opcode, etc.**



Another Example

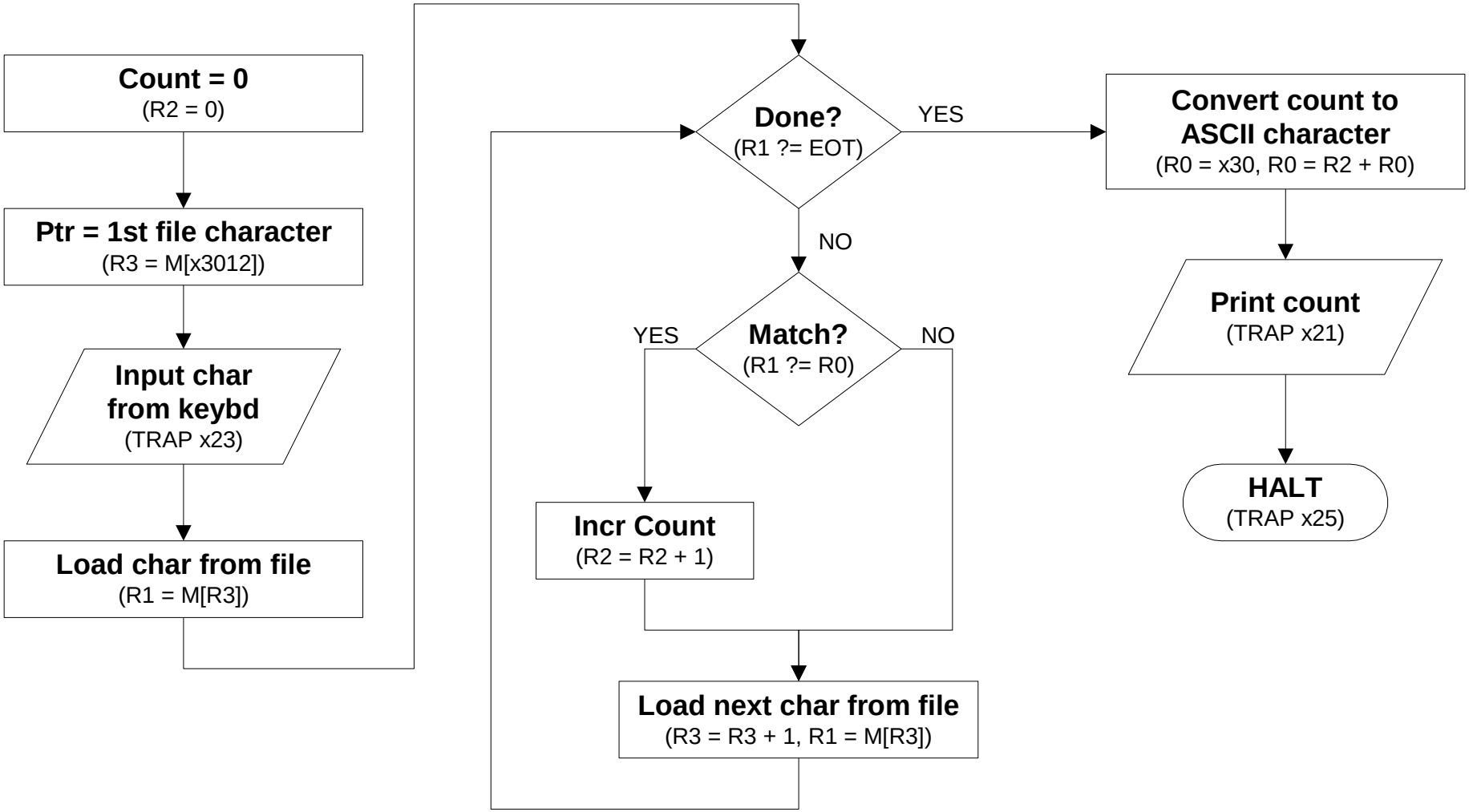
Count the occurrences of a character in a file

- **Program begins at location x3000**
- **Read character from keyboard**
- **Load each character from a "file"**
  - File is a sequence of memory locations
  - Starting address of file is stored in the memory location immediately after the program
- **If file character equals input character, increment counter**
- **End of file is indicated by a special ASCII value: EOT (x04)**
- **At the end, print the number of characters and halt**
  (assume there will be less than 10 occurrences of the character)

A special character used to indicate the end of a sequence is often called a sentinel.

- Useful when you don't know ahead of time how many times to execute a loop.

Flow Chart



High Level Programming Langauge

```
1  // "file" is a sequence of characters stored in memory
2  char[] file = {'i', 'n', 's', 'p', 'i', 'r', 'a', 't', 'i', 'o', 'n',
3  System.out.println(file);                    // debug
4  int count = 0;                               // R2 = 0
5  int pointer = 0;                             // R3 = points to 1st character
6  char userInput = new Scanner(System.in).next().charAt(0); // TRAP 0x23
7  char charFromFile = file[pointer];           // R1 = M[R3]
8  while (charFromFile != '\4') {               // R1 ?= EOT
9      if (charFromFile == userInput) {         // R1 ?= R0
10         count = count + 1;                   // R2 = R2 + 1
11     }
12     pointer = pointer + 1;                   // R3 = R3 + 1
13     charFromFile = file[pointer];            // R1 = M[R3]
14 }
15 char charToPrint = 0x30;                     // R0 = 0x30
16 charToPrint += count;                       // R0 = R2 + R0
17 System.out.println(charToPrint);            // TRAP x21
```

Text: ASCII Characters

ASCII: Maps 128 characters to 7-bit code.

- both printable and non-printable (ESC, DEL, …) characters

00	nul	10	dle	20	sp	30	0	40	@	50	P	60	`	70	p
01	soh	11	dc1	21	!	31	1	41	A	51	Q	61	a	71	q
02	stx	12	dc2	22	"	32	2	42	B	52	R	62	b	72	r
03	etx	13	dc3	23	#	33	3	43	C	53	S	63	c	73	s
04	eot	14	dc4	24	$	34	4	44	D	54	T	64	d	74	t
05	enq	15	nak	25	%	35	5	45	E	55	U	65	e	75	u
06	ack	16	syn	26	&	36	6	46	F	56	V	66	f	76	v
07	bel	17	etb	27	'	37	7	47	G	57	W	67	g	77	w
08	bs	18	can	28	(	38	8	48	H	58	X	68	h	78	x
09	ht	19	em	29	)	39	9	49	I	59	Y	69	i	79	y
0a	nl	1a	sub	2a	*	3a	:	4a	J	5a	Z	6a	j	7a	z
0b	vt	1b	esc	2b	+	3b	;	4b	K	5b	[	6b	k	7b	{
0c	np	1c	fs	2c	,	3c	<	4c	L	5c	\	6c	l	7c	|
0d	cr	1d	gs	2d	-	3d	=	4d	M	5d	]	6d	m	7d	}
0e	so	1e	rs	2e	.	3e	>	4e	N	5e	^	6e	n	7e	~
0f	si	1f	us	2f	/	3f	?	4f	O	5f	_	6f	o	7f	del

Program (1 of 2)

Address	*Instruction*	*Comments*
x3000		*R2 ← 0 (counter)*
x3001		*R3 ← M[x3012] (ptr)*
x3002	1 1 1 1 0 0 0 0 0 0 1 0 0 0 1 1	*Input to R0 (TRAP x23)*
x3003		*R1 ← M[R3]*
x3004		*R4 ← R1 – 4 (EOT)*
x3005		*If Z, goto x300E*
x3006		*R1 ← NOT R1*
x3007		*R1 ← R1 + 1*
X3008		*R1 ← R1 + R0*
x3009		*If N or P, goto x300B*

Program (2 of 2)

Address	*Instruction*	*Comments*
x300A		*R2 ← R2 + 1*
x300B		*R3 ← R3 + 1*
x300C		*R1 ← M[R3]*
x300D		*Goto x3004*
x300E		*R0 ← M[x3013]*
x300F		*R0 ← R0 + R2*
x3010	1 1 1 1 0 0 0 0 0 0 1 0 0 0 0 1	*Print R0 (TRAP x21)*
x3011	1 1 1 1 0 0 0 0 0 0 1 0 0 1 0 1	*HALT (TRAP x25)*
X3012	Starting Address of File	
x3013	0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0	*ASCII x30 ('0')*

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+ LD DR, label ; Load PC-Relative
| 0   0   1   0 |     DR    |                PCoffset9              |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+ DR ← mem[PC' + SEXT(PCoffset9)] also setcc()
```

Address	Instruction	Comments
x3000	0 1 0 1 | 0 1 0 | 0 1 0 | 1 | 0 0 0 0 0 ANDi R2 R2 imm 0	R2 ← 0 (counter)
x3001	0 0 1 0 | 0 1 1 | 0 0 0 0 1 0 0 0 0 LD R3 0x10	R3 ← M[x3012] (ptr)
x3002	1 1 1 1 | 0 0 0 0 | 0 0 1 0 0 0 1 1 TRAP 0x23	Input to R0 (TRAP x23)
x3003	0 1 1 0 | 0 0 1 | 0 1 1 | 0 0 0 0 0 0 LDR R1 R3 0	R1 ← M[R3]
x3004	0 0 0 1 | 1 0 0 | 0 0 1 | 1 1 1 1 0 0 ADD R4 R1 imm 0xFFFC=-4	R4 ← R1 – 4 (EOT)
x3005	0 0 0 0 | 0 1 0 | 0 0 0 0 0 0 1 0 0 0 BRz N Z P 0x0008=8	If Z, goto x300E
x3006	1 0 0 1 | 0 0 1 | 0 0 1 | 1 1 1 1 1 1 NOT R1 R1	R1 ← NOT R1
x3007	0 0 0 1 | 0 0 1 | 0 0 1 | 1 | 0 0 0 0 1 ADDi R1 R1 imm 0x0001=1	R1 ← R1 + 1
x3008	0 0 0 1 | 0 0 1 | 0 0 1 | 0 | 0 0 0 0 0 ADDr R1 R1 reg R0	R1 ← R1 + R0
x3009	0 0 0 0 | 1 0 1 | 0 0 0 0 0 0 0 0 1 BRnp N Z P 0x0001=1	If N or P, goto x300B

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+ ADD DR, SR1, imm5 ; Addition with Immediate
| 0   0   0   1 |     DR    |    SR1    | 1 |        imm5       |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+ DR ← SR1 + SEXT(imm5) also setcc()
```

Address	Instruction	Comments
x300A	0 0 0 1 0 1 0 0 1 0 1 0 0 0 0 1 (ADD R2 R2 imm 0x0001=1)	R2 ← R2 + 1
x300B	0 0 0 1 0 1 1 0 1 1 1 0 0 0 0 1 (ADD R3 R3 imm 0x0001=1)	R3 ← R3 + 1
x300C	0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0 (LDR R1 R3 0x0000=0)	R1 ← M[R3]
x300D	0 0 0 0 1 1 1 1 1 1 1 1 0 1 1 0 (BRnzp N Z P 0xFFF6=-0x000A=-10)	Goto x3004
x300E	0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 (LD R0 0x0004=4)	R0 ← M[x3013]
x300F	0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 (ADD R0 R0 reg R2)	R0 ← R0 + R2
x3010	1 1 1 1 0 0 0 0 0 0 1 0 0 0 0 1 (TRAP 0x21)	Print R0 (TRAP x21)
x3011	1 1 1 1 0 0 0 0 0 0 1 0 0 1 0 1 (TRAP 0x25)	HALT (TRAP x25)
x3012	Starting Address of File	
x3013	0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0	ASCII x30 ('0')

Backup Slides

Sample Program

Address	Instruction	Comments
x3000	1 1 1 0 0 0 1 0 1 1 1 1 1 1 1 1	*R1 ← x3100 (PC+0xFF)*
x3001	0 1 0 1 0 1 1 0 1 1 1 0 0 0 0 0	*R3 ← 0*
x3002	0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0	*R2 ← 0*
x3003	0 0 0 1 0 1 0 0 1 0 1 0 1 1 0 0	*R2 ← 12*
x3004	0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1	*If Z, goto x300A (PC+5)*
x3005	0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0	*Load next value to R4*
x3006	0 0 0 1 0 1 1 0 1 1 0 0 0 1 0 0	*Add to R3*
x3007	0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1	*Increment R1 (pointer)*
X3008	0 0 0 1 0 1 0 0 1 0 1 1 1 1 1 1	*Decrement R2 (counter)*
x3009	0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 1 0	*Goto x3004 (PC-6)*

Sample Program

Address	Instruction	Comments
x3000	1 1 1 0 0 0 1 0 1 1 1 1 1 1 1 1 (LEA)	R1 ← x3100 (PC+0xFF)
x3001	0 1 0 1 0 1 1 0 1 1 1 0 0 0 0 0 (AND)	R3 ← 0
x3002	0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0 (AND)	R2 ← 0
x3003	0 0 0 1 0 1 0 0 1 0 1 0 1 1 0 0 (ADD)	R2 ← 12
x3004	0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1 (BRz)	If Z, goto x300A (PC+5)
x3005	0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0 (LDR)	Load next value to R4
x3006	0 0 0 1 0 1 1 0 1 1 0 0 0 1 0 0 (ADD)	Add to R3
x3007	0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1 (ADD)	Increment R1 (pointer)
X3008	0 0 0 1 0 1 0 0 1 0 1 1 1 1 1 1 (ADD)	Decrement R2 (counter)
x3009	0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 1 0 (BRnzp)	Goto x3004 (PC-6)

Program (1 of 2)

Address	Instruction	Comments
x3000	0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0	*R2 ← 0 (counter)*
x3001	0 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0	*R3 ← M[x3012] (ptr)*
x3002	1 1 1 1 0 0 0 0 0 0 1 0 0 0 1 1	*Input to R0 (TRAP x23)*
x3003	0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0	*R1 ← M[R3]*
x3004	0 0 0 1 1 0 0 0 0 1 1 1 1 1 0 0	*R4 ← R1 – 4 (EOT)*
x3005	0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0	*If Z, goto x300E*
x3006	1 0 0 1 0 0 1 0 0 1 1 1 1 1 1 1	*R1 ← NOT R1*
x3007	0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1	*R1 ← R1 + 1*
X3008	0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0	*R1 ← R1 + R0*
x3009	0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1	*If N or P, goto x300B*

Program (2 of 2)

Address	Instruction	Comments
x300A	0 0 0 1 0 1 0 0 1 0 1 0 0 0 0 1	R2 ← R2 + 1
x300B	0 0 0 1 0 1 1 0 1 1 1 0 0 0 0 1	R3 ← R3 + 1
x300C	0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0	R1 ← M[R3]
x300D	0 0 0 0 1 1 1 1 1 1 1 1 1 0 1 1 0	Goto x3004
x300E	0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0	R0 ← M[x3013]
x300F	0 0 0 1 0 0 0 0 0 0 0 0 0 1 0	R0 ← R0 + R2
x3010	1 1 1 1 0 0 0 0 0 0 1 0 0 0 0 1	Print R0 (TRAP x21)
x3011	1 1 1 1 0 0 0 0 0 0 1 0 0 1 0 1	HALT (TRAP x25)
X3012	Starting Address of File	
x3013	0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0	ASCII x30 ('0')

```
SR ; Bit-wise Complement

T(SR) also setcc()

eturn from Subroutine



eturn from Interrupt

tbook (2nd Ed. page 537).

label ; Store PC-Relative

+ SEXT(PCoffset9)] ← SR

 label ; Store Indirect
```

```
  1   1 |
----+---+ DR ← NOT(SR) also setcc()

----+---+ RET ; Return from Subroutine
  0   0 |
----+---+ PC ← R7

----+---+ RTI ; Return from Interrupt
  0   0 |
```

```
   | 1 |        imm5
---+---+---+---+---+--

---+---+---+---+---+--

   | 0 | 0   0 |     SI
---+---+---+---+---+--

---+---+---+---+---+--

   | 1 |        imm5
---+---+---+---+---+--
```