MICROPROCESSOR SYSTEMS

Lecture 6

Instructions

Topics

- Instruction Types

Instruction Types

- **Data Transfer Instructions**
  - Operations that move data from one place to another.
  - These instructions don't actually modify the data,
    they just copy data from a source to the destination.

- **Data Operation Instructions**
  - Data operation instructions modify the values of data.
  - They perform some operation using one or two data values (operands)
    and store the result.

- **Program Control Instructions**
  - Jump or branch instructions are used to go to another part of the program.
  - They can be unconditional (always taken) or
    conditional (taken only if some condition is met).

EDU-CPU Instruction Set

Data Transfer	
MOV	Move
LDA	Load
STA	Store
EXC	Exchange
CHN	Change

Directives	
START	Start
ORG	Origin
EQU	Equal
RMB	Reserve memory bytes
DAT	Data
END	End

Branch - Compare	
CMP	Compare
BIT	Bit test
BRA	Branch (unconditional)
JMP	Jump (unconditional)
JMC	Jump conditionally
BEQ	Branch if equal
BNE	Branch if not equal
BGT	Branch if greater than
BGE	Branch if greater or equal
BLT	Branch if less than
BHI	Branch if higher
BHE	Branch if higher or equal
BLO	Branch if lower
BIO	Branch if overflow
BNO	Branch if not overflow
BIC	Branch if carry
BNC	Branch if not carry
BIH	Branch if half carry
BNH	Branch if not half carry
BSR	Branch to subroutine
JSR	Jump to subroutine
BSC	Branch to subroutine conditionally
JSC	Jump to subroutine conditionally
DBNZ	Decrease, branch if not zero

Arithmetic	
ADD	Add
ADC	Add with carry
SUB	Subtract
SUE	Subtract with carry
MUL	Multiply
DIV	Divide
INC	Incremenet
DEC	Decrement

Logic	
AND	And
OR	Or
XOR	Exclusive or
CLR	Clear
SET	Set
COM	Complement
NEG	Negate

Operational	
DAA	Decimal adjust accumulator
PSH	Push
PUL	Pull
EIN	Enable interrupt
DIN	Disable interrupt
NOP	No operation
INT	Interrupt
RTS	Return from subroutine
RTI	Return from interrupt

Shift/Rotate	
LSL	Logical shift left
LSR	Logical shift right
ASR	Arithmetic shift right
ROL	Rotate left
ROR	Rotate right

Total 64 instructions

Symbols used in EDU-CPU Instruction Catalog

Symbol	Meaning
K_i	8-bit register (first operand R_i)
K_j	8-bit register (second operand R_j)
K_{ii}	16-bit register (first operand R_{ii})
K_{jj}	16-bit register (second operand R_{jj})
V	8-bit immediate data
VV	16-bit immediate data
<ADR>	Memory address (16-bit always)
A_i	8-bit accumulator (A or B)
A_{ii}	16-bit accumulator (AB)

Number prefix symbols

Number System	Base	Prefix Symbol	Instruction Examples (All of the examples load decimal 27 to accumulator A)
Decimal (default)	10	None	**LDA A, 27**
Hexadecimal	16	**$**	**LDA A, $1B**
Binary	2	**%**	**LDA A, %00011011**

All numbers are converted to binary form by the Assembly compiler.

Directives

- The directives are not real instructions.
- The assembly compiler (Assembler) **does not generate machine codes** for the directives.
- They are processed only at compile-time.

START : Indicates beginning of program (Compulsory label).

END : Indicates end of program (Optional label).

EQU : Equal: Defines an equivalent constant symbol for a value.

Examples:

```
PI    EQU   314
NUM   EQU   $2000
```


Directives

ORG : Origin: Defines absolute address of a data segment or an instruction.

ORG $3000

DAT : Data: Initializes immediate data (each must be 1 byte) in memory.

DAT $10, $20, $30, $40, $50

Invalid DAT Example:
DAT $1234
(Only $34 is stored, $12 is lost)

RMB : Reserve Memory Bytes:

Defines a variable with specified number of bytes.

(Assembly compiler determines the memory address of variables.)

```
SAYI   RMB   1     ;Variable with 1 byte
DIZI   RMB   5     ;Array with 5 bytes
```

All symbolic identifier names (EQU constant names and RMB variable names) are converted to binary memory addresses by the Assembly compiler.

Compiler Rules of EDU-CPU Simulator

- Program instructions **must** begin with the **START label**.
- Labels such as START must be written in the first column.
- At least one space should be typed before an instruction, otherwise compiler gives error.
- Uppercase/lowercase is not important.
- Comments:
  - Lines beginning with (*) are comments.
  - Sections beginning with (;) are also comments.
  - At least one tab before the (;) symbol is required.
- **INT** (interrupt) instruction should be written at the end, to stop the program.

Program uses accumulator A to calculate total of two numbers.

```
* Example Assembly program.

START
   LDA  A, 5    ; Load 5 to accumulator A
   ADD  A, 3    ; Add 3 to A
   INT          ; Interrupt (stop)
```

Transfer Instructions

- **Data Transfer:** Moves (copies) the content of **one register** into **another register**.

MOV R_i, R_j	$R_i \leftarrow R_j$
MOV R_{ii}, R_{jj}	$R_{ii} \leftarrow R_{jj}$
MOV A, B	$A \leftarrow B$
MOV B, DK	$B \leftarrow CCR$ (Condition Code Register : Status flags)
MOV SK, YG	$IX \leftarrow SP$ (Stack Pointer to Index Register)
MOV CD, AB	$CD \leftarrow AB$

10

Transfer Instructions

- **Load:** The load instruction copies the content of a **memory location** or places an **immediate value**, into a **register**.
- Memory contents are not changed.
- The symbols <> are used as address syntax.

LDA R_i, <ADDRESS>	$R_i \longleftarrow$	<ADDRESS>
LDA R_{ij}, <ADDRESS>	$R_{ij} \longleftarrow$	<ADDRESS> & <ADDRESS+1>
LDA R_i, DATA	$R_i \longleftarrow$	DATA
LDA A, <$1000>	A $\longleftarrow$	<$1000>
LDA B, $25	B $\longleftarrow$	$25
LDA SK, $3000	IX $\longleftarrow$	$3000
LDA AB, <$2000>	A $\longleftarrow$	<$2000>
	B $\longleftarrow$	<$2001>

LDA (Load) Examples

LDA AB, $1234

LDA A, $12
LDA B, $34

→ Same result

LDA A, $1234 → A is loaded with only data $34 (low byte). Data $12 (high byte) is lost.

LDA AB, <$2000>

LDA A, <$2000>
LDA B, <$2001>

→ Same result

Transfer Instructions

- **Store:** The store instruction copies the content of a register (or an immediate value) into a **memory location**.
- The contents of the accumulator or register are not changed.
- Usage of <> symbols is optional in STA.

STA R_i, <ADDRESS>	<ADRESS> ← R_i
STA DATA, <ADDRESS>	<ADRESS> ← DATA
STA $35, $1000	$1000 ← $35
STA A, $1000	$1000 ← A
STA SK, $2000	$2000 & $2001 ← IX

13

STA (Store) Examples

```
STA    <$1234>, <$2000>
```

→ Compiler error : No direct data transfer from an address to another address.

```
STA    $1234, <$2000>
```

→
- Only data $34 (low byte) is stored in address $2000.
- Data $12 (high byte) is lost.

```
LDA    AB, $1234
STA    AB, <$2000>
```

→
- Data $12 is stored in address $2000.
- Data $34 is stored in address $2001.

14

Transfer Instructions

- **Exchange:** Exchange instruction exchanges the contents of pairs of registers or accumulators.

EXC R_i, R_j $R_i \longleftrightarrow R_j$
EXC A, B A $\longleftrightarrow$ B
EXC SK, YG IX $\longleftrightarrow$ SP

- **Change:** The bits are divided into two groups of **four bits**. Change insruction is used to swap the content of the first group with the second one.

CHN R_i New R_i : $[D_3,D_2,D_1,D_0 , D_7,D_6,D_5,D_4]$
CHN A

15

Transfer Instructions

- **Push:** The contents of the 8-bit accumulators (A or B) are added onto the stack. (Stack Pointer is updated automatically.)

PSH A Push the contents of A accumulator onto stack

- **Pull:** Pull instructions are used to read data into the specified accumulator from the stack. (SP is updated automatically.)

PUL A Data on top of the stack is retrieved to A accumulator

16

Arithmetic Instructions

- **Add:** Add instructions are used to add the content specified (either in an accumulator, in a register, or in memory location) into the accumulator (A, B, or AB). The result is stored in the accumulator.

ADD A, R_i	A ←	A + R_i
ADD A, B	A ←	A + B
ADD A, DATA	A ←	A + DATA
ADD A, <ADDRESS>	A ←	A + <ADDRESS>

Arithmetic Instructions

- **Add with Carry:** Add operation is performed by including the carry flag (from a previous operation instruction).

ADC A, R_i	A ←	A + R_i + carry
ADC A, B	A ←	A + B + carry
ADC A, DATA	A ←	A + DATA + carry
ADC A, <ADDRESS>	A ←	A + <ADDRESS> + carry

Examples: Add Instruction

1-byte arithmetic examples:

ADD A, B	A ←	A + B
ADC A, B	A ←	A + B + carry
ADD A, $25	A ←	A + $25
ADD A, <$1000>	A ←	A + <$1000>
ADC A, <SK+10>	A ←	A + <IX+10> + carry

Examples: Add Instruction

2-byte arithmetic examples:

ADD AB, $1234	AB ←	AB + $1234
ADD AB, <$1000>	AB ←	AB + <$1000> & <$1001>
ADD AB, CD	AB ←	AB + CD

Invalid Examples (Compiler errors) :

```
ADD   C, 5                ; C register is not an accumulator
ADC   AB, 5               ; Carry is for only 1-byte accumulators
ADD  <num1>, <num2>       ; Target must be an accumulator
```

Example: Adding 1-byte numbers

- The following program adds two numbers, each is 1 byte.
- DAT directive places data ($28, $25, $00) starting at address $0000.
- Program instructions begin at memory location $0003.
- Program reads two numbers from addresses $0000 and $0001.
- Result is stored at address $0002.

- **Version1 :** Program uses direct addresses for datas.

```
DAT $28, $25, $00
*Data are initialized at absolute addresses $0000, $0001, and $0002

START
   LDA  A, <$0000>     ;Load first number ($28) to A accumulator
   LDA  B, <$0001>     ;Load second number ($25) to B accumulator
   ADD  A, B           ;Add B to A
   STA  A, <$0002>     ;Store A result to the memory address
   INT                 ;Interrupt (stop)
```

```
  $28   number1
  $25   number2
+ ____________
  $4D  Result
```

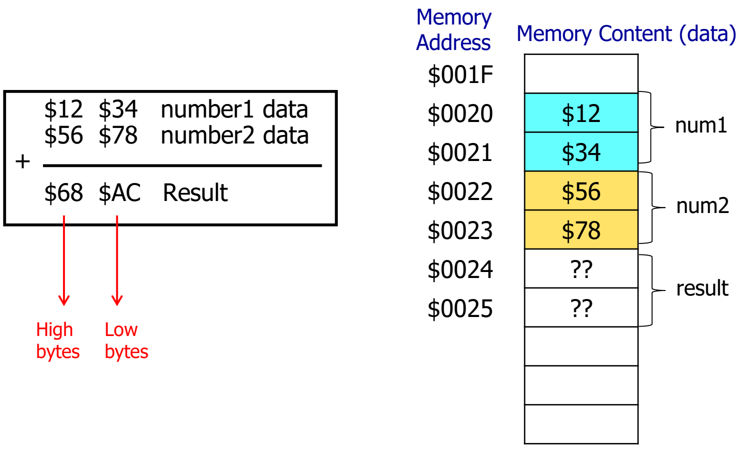
- **Version2 :** Program uses variable names with **RMB** directives (Reserve Memory Bytes).
- Compiler determines the memory addresses of all variables, thru ORG instruction.

```
Num1  RMB  1
Num2  RMB  1
Result  RMB  1

   ORG  Num1
   DAT $28, $25, $00

* Data stored at addresses
* beginning at $0000.

START
    LDA    A , <Num1>
    ADD    A , <Num2>
    STA    A , <Result>
    INT
```

- Compiler places Num1 variable at address $0000.
- Address of Num2 ($0001) and Result ($0002) follow sequentially after address of Num1.

Example: Adding 2-byte numbers (By using 1-byte arithmetic)

- Program reads two numbers (num1= $1234, num2= $5678) from memory, each is 2-byte.
- Suppose addresses of two numbers are between locations $0020 and $0023.
- In 1-byte arithmetic, low bytes and high bytes will be added separately.



- Program uses direct memory addresses.
- Absolute addresses of variables are specified thru ORG directives.
- Firstly, low bytes are added in B accumulator.
- Then, high bytes are **added** in A accumulator, with **carry** bit from the previous add operation.

```
   ORG $0020
   DAT $12, $34

   ORG $0021
   DAT $56, $78

START
   LDA  B, <$0021>    ; Load low byte of num1
   ADD  B, <$0023>    ; Add low byte of num2

   LDA  A, <$0020>    ; Load high byte of num1
   ADC  A, <$0022>    ; Add with carry high byte of num2

   STA  A, $0024      ; Store high byte of result
   STA  B, $0025      ; Store low byte of result
   INT                ; Stop
```

Example: Adding 2-byte numbers (By using 2-byte arithmetic)

- For 2-byte arithmetic add operations, the AB accumulator (2-byte) should be used.

```
  $1234   number1 data
  $5678   number2 data
+ ---------------------
  $68AC   Result
```

- **Version1 :** Program uses direct addresses.

```
ORG $0020
 DAT $12, $34

 ORG $0021
 DAT $56, $78

START
    LDA AB, <$0020>    ; Address
    ADD AB, <$0022>    ; Address
    STA AB, <$0024>    ; Address
    INT
```

25

- **Version2 :** Program uses variable names (RMB).
- Memory addresses of all variables are determined by compiler thru ORG directives, beginning at address $0000.

```
Num1  RMB  2
Num2  RMB  2
Result  RMB  2

    ORG Num1
    DAT  $12, $34

    ORG  Num2
    DAT  $56, $78

    ORG  Result
    DAT  $00, $00

START
     LDA    AB , <Num1>
     ADD    AB , <Num2>
     STA    AB , <Result>
     INT
```

Alternative1:

```
ORG Num1
DAT  $12, $34
DAT  $56, $78
DAT  $00, $00
```

Alternative2:

```
ORG Num1
DAT  $12, $34, $56, $78, $00, $00
```

26

Subtract Instruction

- **Subtract:** Substract instructions are used to subtract the specified contents, in an accumulator,in a register,or in the memory location, from the contents of the accumulator.
- The result is stored in the accumulator.

SUB A, R_i	A $\longleftarrow$	A - R_i
SUB A, B	A $\longleftarrow$	A - B
SUB A, DATA	A $\longleftarrow$	A - DATA
SUB A, <ADDRESS>	A $\longleftarrow$	A - <ADDRESS>

27

Subtract Instruction

- **Subtract with Borrow**

SUE A, R_i	A $\longleftarrow$	A - R_i - carry
SUE A, B	A $\longleftarrow$	A - B - carry
SUE A, DATA	A $\longleftarrow$	A - DATA - carry
SUE A, <ADDRESS>	A $\longleftarrow$	A - <ADDRESS> - carry

28

Multiply Instruction

- **Multiplication:**
  - **The multiplicand should always be in A**.
  - The multiplier can be in B, in another **8-bit register**,in a memory location, or an immediate data.
  - **The result is always stored in AB.**
  - The multiplicand and multiplier are considered as unsigned.

MUL A, R_i AB ← A * R_i
MUL A, B AB ← A * B
MUL A, DATA AB ← A * DATA
MUL A, <ADDRESS> AB ← A * <ADDRESS>

Example : Multiplication

- Two 8-bit numbers ($12 and $34) are in memory locations $0000 and $0001.
- Multiply them.
- Store 16-bit result at memory locations $0002 and $0003.

```
  DAT  $12
  DAT  $34
  DAT  $00, $00

START
   LDA  A, <$0000>

   MUL  A, <$0001>
   STA  AB, $0002

   INT
```

Memory Address (Hex)	Memory Content (Hex)	
$0000	$12	Multiplicand
$0001	$34	Multiplier
$0002	$03	Result
$0003	$A8	

Divide Instruction

- **Division:**
  - **The dividend should always be in AB.**
  - The divisor can be an **8-bit register**, or a memory location, or an immediate data.
  - **The quotient (result) is in AB.**
  - **The remainder is in C.**
  - The dividend and the divisor are considered as unsigned numbers.
  - If the divisor is 0, the Overflow flag in CCR is set automatically.

DIV AB, R_i	AB ← <AB> / R_i
DIV AB, DATA	AB ← <AB> / DATA
DIV AB, <ADDRESS>	AB ← <AB>/<ADDRESS>

31

Example : Division

- An 16-bit number ($1234) is in memory locations $0000 and $0001.
- Divide the number to a 8-bit number ($56) that is in memory location $0002.
- Store 16-bit result at memory locations $0003 and $0004.
- Store 8-bit remainder at memory location $0005.

```
 DAT  $12, $34
 DAT  $56
 DAT  $00, $00
 DAT  $00

START
  LDA  AB, <$0000>
  DIV  AB, <$0002>

  STA  AB, $0003
  STA  C, $005

  INT
```

Memory Address (Hex)	Memory Content (Hex)	
$0000	$12	Dividend
$0001	$34	
$0002	$56	Divisor
$0003	$00	Result
$0004	$36	
$0005	$10	Remainder

32

Logical Operations

- AND , OR, exclusive OR: instructions are used to perform the boolean logical operations AND, OR, exclusive OR.
- **First operand** should be Accumulator **A or B**.
- Second operand can be a register, memory or immediate value.

AND A, $25	A ← A AND $25
OR A, B	A ← A OR B
XOR A, <$1000>	A ← A XOR <$1000>

Logical Operations

- One's complement: Accumulators, registers and specified memory locations can be logically complemented (bit-wise).

COM A	A ← one's complement of A
COM <$1000>	<$1000> ← one's complement of <$1000>

- Two's complement (Negate) : Contents of an accumulator, register or memory location are converted to two's complement.

NEG A	A ← two's complement of A
NEG <$1000>	<$1000> ← two's complement of <$1000>

Logical Operations

- Clear: Clear writes zeros into the destination operand (8 bit).

CLR A	A ←	0
CLR <$1000>	<$1000> ←	0
CLR DK	CCR ←	0

- CLR can also be used to clear specific **status flags** (bits) in the CCR.

CLR E	clear carry (Elde) flag to zero
CLR Y	clear half carry (Yarım elde) flag to zero
CLR S	clear zero (Sıfır) flag to zero
CLR T	clear overflow (Taşma) flag to zero
CLR N	clear negative flag to zero

Logical Operations

Set: Specific **status flags** (bits) in the CCR can be set to one.

SET E	set carry (Elde) flag to one
SET Y	set half carry (Yarım elde) flag to one
SET S	set zero (Sıfır) flag to one
SET T	set overflow (Taşma) flag to one
SET N	set negative flag to one

Logical Operations

- Decimal Adjustment of accumulator: Translate a binary number in an accumulator to BCD.

DAA A
DAA B

Example :

```
START
   LDA    B, $0F
   DAA    B      ; B now contains 15
   INT
```


Logical Operations

- Increment: Increment instructions are used to increment operand (register or memory) by one.

INC A — A ← A + 1
INC <$1000> — <$1000> ← <$1000> + 1

- Decrement: Decrement instructions are used to decrement operand (register or memory) by one.

DEC A — A ← A - 1
DEC <$1000> — <$1000> ← <$1000> - 1

Shift and Rotate Instructions

- All shift and rotate instructions involve the **carry bit** in the CCR.
- Shift/rotate instructions are used with 8 bit **operands** (register or memory).
- Also, by setting or clearing the carry bit before a shift or rotate instruction, program can control the initial value of carry bit.
- Used only for **unsigned** numbers.

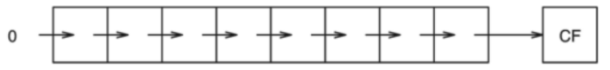
39

Shift Instructions

- Logical Shift Right:
  1) Least Significant Bit (LSB) (rightmost) is copied to Carry Flag (CF).
  2) All bits of operand are shifted to right.
  3) A zero bit is placed in Most Sigificant Bit (MSB) (leftmost).
- The shift instructions can operate on accumulators, registers, or on a memory location.
- They are used for **unsigned numbers**.

LSR A

LSR <$3000>

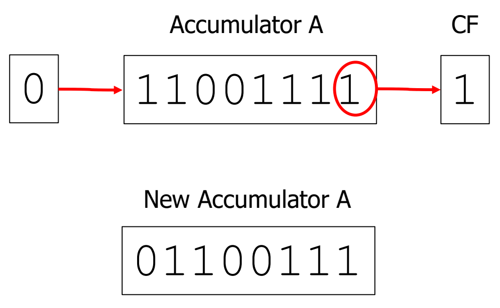


One LSR call corresponds to division by 2.

40

Example: LSR

```
START
   LDA A,%11001111
   LSR A
   INT
```

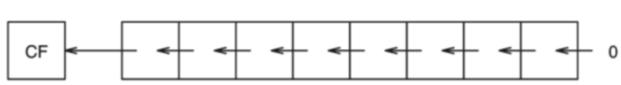


Shift Instructions

- Logical Shift left :
  1) MSB is copied to CF.
  2) All bits of operand are shifted to left.
  3) A zero is placed in LSB.

LSL A
LSL B
LSL <$3000>

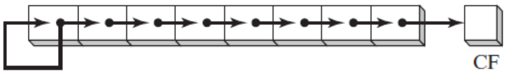
One left shift corresponds to multiplication by 2.



Shift Instructions

- Arithmetic shift right**:** The arithmetic shift right instruction **maintains the original value of the MSB** (Most significant bit = leftmost sign bit) of the operand.

ASR A
ASR B
ASR <$3000>

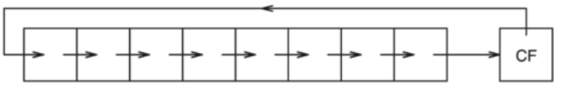


Rotate Instructions

- Rotate Right: It rotates content of an accumulator or a memory location to right by 1-bit position.
  1) All bits are shifted to right.
  2) CF is copied to MSB bit.
  3) LSB bit is copied to CF.

- Initial value of Carry bit in CCR must be assigned (1 or 0) before using the ROR instruction.

ROR A
ROR B
ROR <$3000>



Example: ROR

```
START
   LDA  A,%01001100
   SET  E     ; Set Carry Flag to 1
   ROR  A
   INT
```

Accumulator A　　CF

01001100	1

New Accumulator A　　New CF

10100110	0

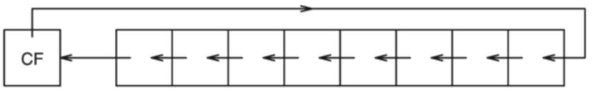
Rotate Instructions

- Rotate Left
  1) All bits are shifted to left.
  2) CF is copied to LSB bit.
  3) MSB bit is copied to CF.

ROL　A

ROL　B

ROL　<$3000>



Controlling the Flow of Execution

- Most programs must sometimes **jump** or **branch** to execute something other than the next instruction.
- The jump or branch can be **unconditional** or **conditional** to go from a location to another location.

47

Unconditional Branches

- Branch Always: Branch-always instructions transfer control unconditionally to a location specified using **relative addressing**.

  BRA ADDRESS (location label)

- Jump: The program branches to the **memory location specified in the instruction**. (Direct addressing)

  JMP ADDRESS (location label)

48

Difference between BRA and JMP

- **BRA** instruction uses **Relative** addressing method.
- **JMP** instruction uses **Direct** addressing method.

- Compiler replaces the address label with a relative number in BRA instruction.
- Operand $05 (computed by compiler) in BRA instruction machine code means goto $05 locations (bytes) ahead from Program Counter address.
- Operand $0007 in JMP instruction means goto absolute memory address $0007.

BRA

JMP

Address (Program Counter)	Machine Codes (Generated by compiler)	Instructions
0000	80 **05**	BRA DEVAM
0002	1E 28 **00 07**	JMP DEVAM
0006	C2	NOP
0007	C2	DEVAM NOP
0008	C3	INT

49

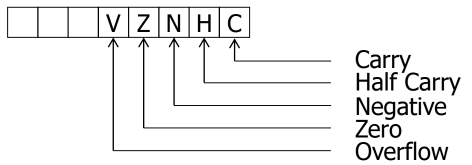
Conditional Branches

- Conditional branching is especially important, as it implements high-level language constructs as
  - if-then-else
  - while loop
  - for loop

- There are two groups, which operate differently.
  - **Simple conditional** branches can base their decisions on a single bit in the condition code register (CCR).
  - **Comparison** branches can base their decisions on more than one CCR bit.

50

Condition Code Register
(Status Flags Register)



Status Flag Bit	Indication
C	indicates a carry (or borrow) out of the most significant bit (MSB) when an addition (or subtraction) is performed
H	indicates a half carry out of bit 3. (Useful in BCD arithmetic)
N	indicates a negative result (it's the MSB of the result)
Z	indicates a result of exactly zero
V	indicates overflow

How CCR bits are affected

- As a general rule, all CCR (status flags) bits are affected by **arithmetic** instructions and by **data transfer** instructions.
- Branch instructions affect none of them.

- The effect of each instruction on each CCR bit is described using the notation below.
- These symbols are used in Instruction Set Catalog tables.

Symbol	Operation
–	Bit unaffected
0	Bit always cleared (to 0)
1	Bit always set (to 1)
↕	Bit depends on result

Example1 : Add two numbers

- Decimal 100 and 50 are two positive numbers (their MSB bits are 0).
- The result of add operation can be interpreted as negative (because its MSB bit is 1), which also indicates an overflow.
- Since MSB bit of result is 1, this also indicates a negative.

```
START
  LDA  A, 100
  ADD  A, 50
  INT
```

	0110 0100	$(100)_{10}$	
+	0011 0010	$(50)_{10}$	
	1001 0110	$(150)_{10}$	(Unsigned interpretation)
		$(-106)_{10}$	(Signed interpretation)

Overflow Negative

- 8-bit unsigned interpretation of result is 150 (%10010110).
- Signed interpretation of result is -106 (%10010110).

			1	0	1	0	0

Carry
Half Carry
Negative
Zero
Overflow

Example2 : Add two numbers

- Two negative signed numbers are added (their MSB bits are 1).
- The result is a positive number (its MSB bit is 0), so there is an overflow.
- There are also half carry and end carry.

```
START
  LDA  A, 170
  ADD  A, 188
  INT
```

Half carry

	1		
	1 0 1 0 1 0 1 0	$(170)_{10}$	
+	1 0 1 1 1 1 0 0	$(188)_{10}$	
	1 0 1 1 0 0 1 1 0	$(358)_{10}$	(Expected result)
		$(102)_{10}$	(Actual result)

Carry (discarded) Overflow

- 8-bit unsigned interpretation of result is 102 (%01100110), because carry bit will be discarded.
- If AB register pair was used, then 16-bit unsigned interpretation of result would be 358 (%00000001 01100110).

			1	0	0	1	1

Carry
Half Carry
Negative
Zero
Overflow

Comparison Instructions

- **Compare :** The CMP instruction performs a **<u>subtraction</u>** that sets the CCR bits according to the results. (Result is not saved anywhere).
- In general, a Branch or Jump instruction is written after a CMP instruction.

- **CMP A, B** Compare B to A.
  (Calculates A-B, and compares result to zero)

CMP A, DATA
CMP R_i, DATA
CMP A, B
CMP A, R_i
CMP R_i, R_j
CMP A, <ADDRESS>

CMP R_{ii}, R_{jj}
CMP R_{ii}, DATA
CMP R_{ii}, <ADDRESS>

55

Comparison Instructions

- **Test :** The BIT (Bit Test) instruction is a special form of the **AND instruction** that produces no result except for a change of the flags.

BIT A, V
BIT R_i, V
BIT A, B
BIT A, R_i
BIT R_i, R_j
BIT A, <ADDRESS>
BIT R_i, <ADDRESS>

56

Simple Branch Instructions

- Simple branch instructions uses the **Z, N, V, C, H** status flag bits.
- **BEQ** branch on zero
  - Branch if the Z bit is 1.
- **BNE** branch on not zero
  - Branch if the Z bit is 0.
- **BIO, BNO** branch on overflow set, branch on overflow cleared
  - Branch if the V bit is 1 or 0, respectively.
- **BIC, BNC** branch on carry, branch on no carry
  - Branch if the C bit is 1 or 0, respectively.
- **BIH, BNH** branch on half carry, branch on no half carry
  - Branch if the H bit is 1 or 0, respectively.

57

Comparision Branch Instructions used for Unsigned Data

- Intended for use with **unsigned data.**
- Absolute data values are used.
- Conditions are named as higher, lower, and equal.
  - **BLO** branch on lower than
    - Branch if the result is < 00
  - **BHI** branch on higher than
    - Branch if the result is > 00
  - **BHE** branch on higher than or equal
    - Branch if the result is ≥ 00

58

Comparision Branch Instructions used for Signed Data

- Intended for use with **signed data** (two's complement).
- Conditions are named as greater, less, and equal.

  - **BLT** branch on less than.
    - Checks whether **N flag** is 1
    - Branch if the result is < 00
  - **BGT** branch on greater than.
    - Checks whether **Z and N flags** are 0
    - Branch if the result is > 00
  - **BGE** branch on greater than or equal to.
    - Checks whether **Z flag** is 1, or **N flag** is 0
    - Branch if the result is ≥ 00

Example1: Convert a negative number to absolute number

- The CMP (compare) instruction is optional below, program can work also without CMP, because LDA instruction affects the Negative status flag.
- The NEG (negate) instruction is used to get absolute value of a negative number.

```
START
    LDA  A, -5
    CMP  A,  0          ; compare A to 0
    BGT  POSITIVE       ; branch if Negative flag=0
    NEG  A              ; negate the negative number
POSITIVE  NOP           ; No operation
    INT
```

-5   = %11111011 = $FB (8-bit signed interpretation)
251 = %11111011 = $FB (8-bit unsigned interpretation)

Result of Negate +5   = %00000101 = $05

Example2 : Sign bit testing

- Leftmost bit of a number indicates the sign.
  (1 means negative, 0 means positive.)
- The **BIT** (Bit Test) instruction performs AND operation.
  (But the result of the operation is not stored.)
- Leftmost bit in A determines the result.

```
START
     LDA  A, -$5               ; content of A is %11111011
     BIT  A, %10000000         ; filters the leftmost bit of A
     BEQ  POZITIF              ; branch if Zero flag=1
     BRA  NEGATIF              ; goto label
POZITIF  NOP                   ; No operation
      INT
NEGATIF  NOP                   ; No operation
      INT
```

- BIT instruction = %11111011 AND %10000000
- Result of AND operation = %10000000
- Zero status flag = 0

61

Example3: Determine if a number is odd or even

- Rightmost bit of a number indicates whether it is odd or even.
- 0 means number is even, 1 means number is odd.
- Program gets the rightmost bit of A into Carry Status Flag, by using LSR instruction.

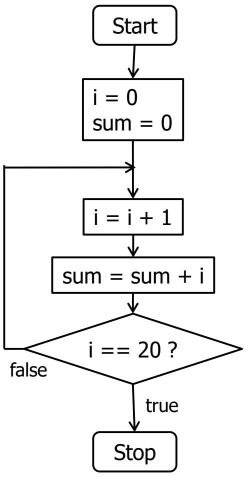
```
SONUC   RMB  1      ; Result code : 1 means odd, 2 means even

START
    LDA   A, 5      ; Initialize A with a test number
    LSR   A         ; Logical Shift Right     (Get rightmost bit into Carry status flag)
    BNC   EVEN      ; Branch if Not Carry     (If carry flag is 0, data is even)
    STA  1, <SONUC>        ; store odd code
    BRA  SON               ; goto end label
EVEN
    STA  2, <SONUC>        ; store even code
SON  INT
```

- Initial A accumulator = %00000101
- A accumulator after the LSR operation = %000000010
- Carry status flag = **1**

62

Example4 : Calculate sum of series

- Write a program to compute 1 + 2 + ... + 20 and save the sum result at address $1000.



- A accumulator is used as sum.
- B accumulator is used as loop counter.

```
START   CLR     A        ; Clear sum
        CLR     B        ; Clear loop counter
DONGU
        INC     B        ; Increment counter
        ADD     A, B     ; Add B to A
        CMP     B, 20    ; Compare B to 20
        BNE     DONGU    ; Branch if Zero flag=0
        STA     A, $1000 ;Store the sum
        INT
```

B-20 = 0?

63

Example5 : Calculate array total

- Suppose there are 5 data numbers in memory starting from address $0000.
- Write a program to find sum of numbers in the array.
- Store the result at absolute memory address $2000.

Version1 : Program uses SK as index on array, B as loop counter, A as total.

```
* Data values are at adress $0000.
DAT  10,20,30,40,50     ; Decimal datas

START
     CLR   A             ; Clear total
     LDA   B, 5          ; Loop counter (down)
     LDA   SK, $0000     ; Beginning index on array
DONGU
     ADD   A, <SK+0>     ; Add data to Accumulator
     INC   SK            ; Increment SK
     DEC   B             ; Decrement SK
     BNE   DONGU         ; Branch if Zero flag is not 1
     STA   A, $2000      ; Store total to memory
     INT                 ; Stop
```

64

Version2 : Program uses SK register as an index on ARRAY, and also as a loop counter.

- The following symbolic identifier names (constants and variables) are defined.
  - **SIZE** is a constant name. It represents array length.
  - **ARRAY** is a variable name. It represents beginning address of array.
  - **TOTAL** is defined as a variable name for sum.

```
SIZE EQU 5
ARRAY  RMB  SIZE
   ORG  ARRAY
   DAT  10,20,30,40,50
TOTAL   RMB   1

START
  LDA  A, 0              ; Initialize the total
  LDA  SK, 0             ; SK is used as index on ARRAY, and also as loop counter
DONGU
  ADD  A, <SK+ARRAY> ; Add next data from SK+ARRAY address to accumulator
  INC  SK                ; Increment SK index
  CMP  SK, SIZE          ; Compare SK to SIZE
  BLT  DONGU             ; If less than, go to loop
  STA  A, TOTAL          ; Store accumulator result to memory
  INT                    ; Stop
```

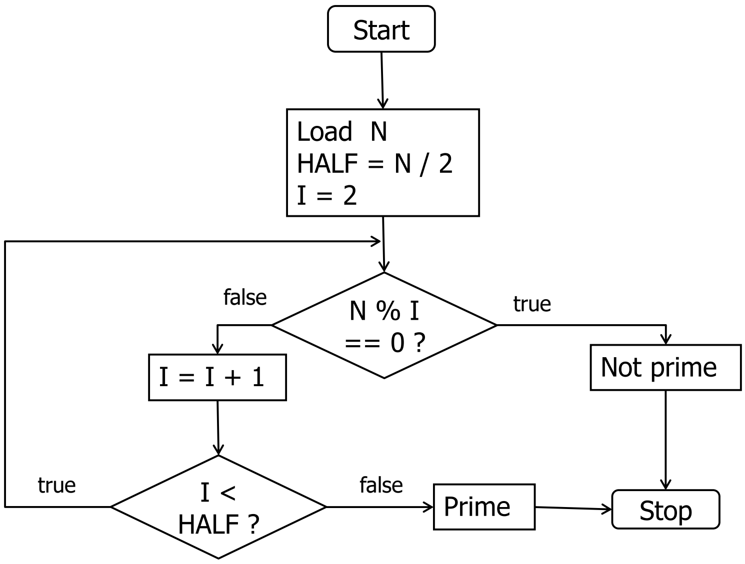
65

Example6: Determine if a number is prime or not

- Write a program to determine whether an unsigned 1 byte number is prime or not.
- Prime numbers : **2, 3, 5, 7, 11, 13, 17, ….**
- RULE : If a number can not be divided by any other number, except 1 and itself, it is a prime number.
- Program should try to divide the given number (N) with all sequential numbers **2, 3, 4, 5, … , up to N/2.**

66

Algorithm Flowchart



67

Program

```
NUMBER   RMB  1
HALF     RMB  1
SONUC    RMB  1
* Result codes :
* 1 means prime (true)
* 0 means not prime (false)

START
    STA  8, <NUMBER>  ; Initialize test number
    LDA  A, <NUMBER>
    LSR  A            ; Means A = A / 2
    STA  A, <HALF>
    LDA  D, 2         ; D register is loop counter
DONGU
    LDA  A, 0
    LDA  B, <NUMBER>
    DIV  AB, D        ; Remainder is in C register
    CMP  C, 0         ; Check if remainder is zero
    BEQ  PRIME_DEGIL
```

```
    INC  D            ; Increment loop counter
    CMP  D, <HALF>
    BLT  DONGU        ; Branch to label

    STA 1, <SONUC>    ; Prime
    BRA SON

PRIME_DEGIL
    STA 0, <SONUC>    ; Not prime

SON  INT              ; Stop
```

68