Computer Organisation and Architecture

Priya Pankaj Kumar
Assistant Professor- CSE

Sershah Engineering College, Sasaram

Instruction Set Architecture (ISA)

- Serves as an interface between software and hardware.
- Provides a mechanism by which the software tells the hardware what should be done.

High level language code : C, C++, Java, Fortran,

compiler

Assembly language code: architecture specific statements

assembler

Machine language code: architecture specific bit patterns

software

hardware

What is an Instruction Set?

- The complete collection of instructions that are understood by a CPU
- Machine language: binary representation of operations and (addresses of) arguments
- Assembly language: mnemonic representation for humans, e.g.,

  OP A,B,C (meaning A <- OP(B,C))

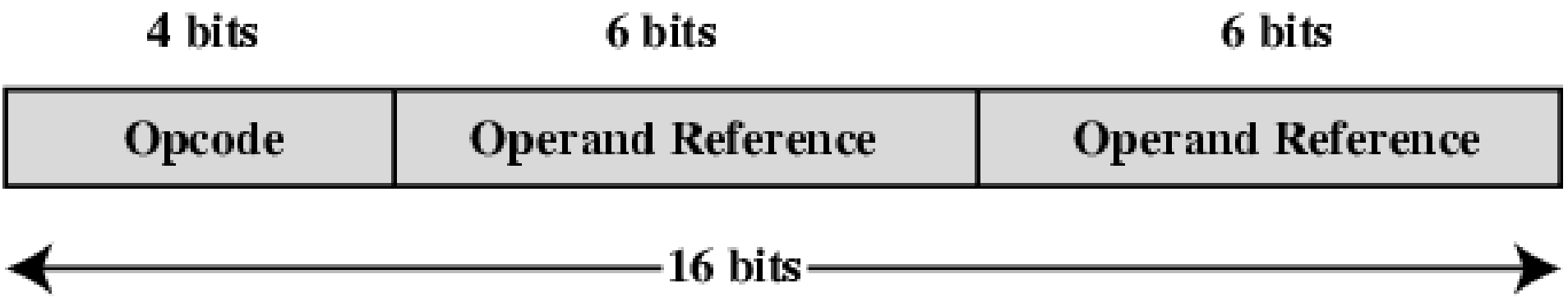
Design Decisions

- Registers
  - Number of CPU registers available
  - Which operations can be performed on which registers? General purpose and specific registers
- Addressing modes (see later)

Instruction Set Design Issues

- Instruction set design issues include:
  - Where are operands stored?
    - registers, memory, stack, accumulator
  - How many explicit operands are there?
    - 0, 1, 2, or 3
  - How is the operand location specified?
    - register, immediate, indirect, . . .
  - What type & size of operands are supported?
    - byte, int, float, double, string, vector. . .
  - What operations are supported?
    - add, sub, mul, move, compare . . .

Simple Instruction Format (using two addresses)



Classifying ISAs

Accumulator (before 1960, e.g. 68HC11):

1-address	add A	acc ¬ acc + mem[A]

Stack (1960s to 1970s):

0-address	add	tos ¬ tos + next

Memory-Memory (1970s to 1980s):

2-address	add A, B	mem[A] ¬ mem[A] + mem[B]
3-address	add A, B, C	mem[A] ¬ mem[B] + mem[C]

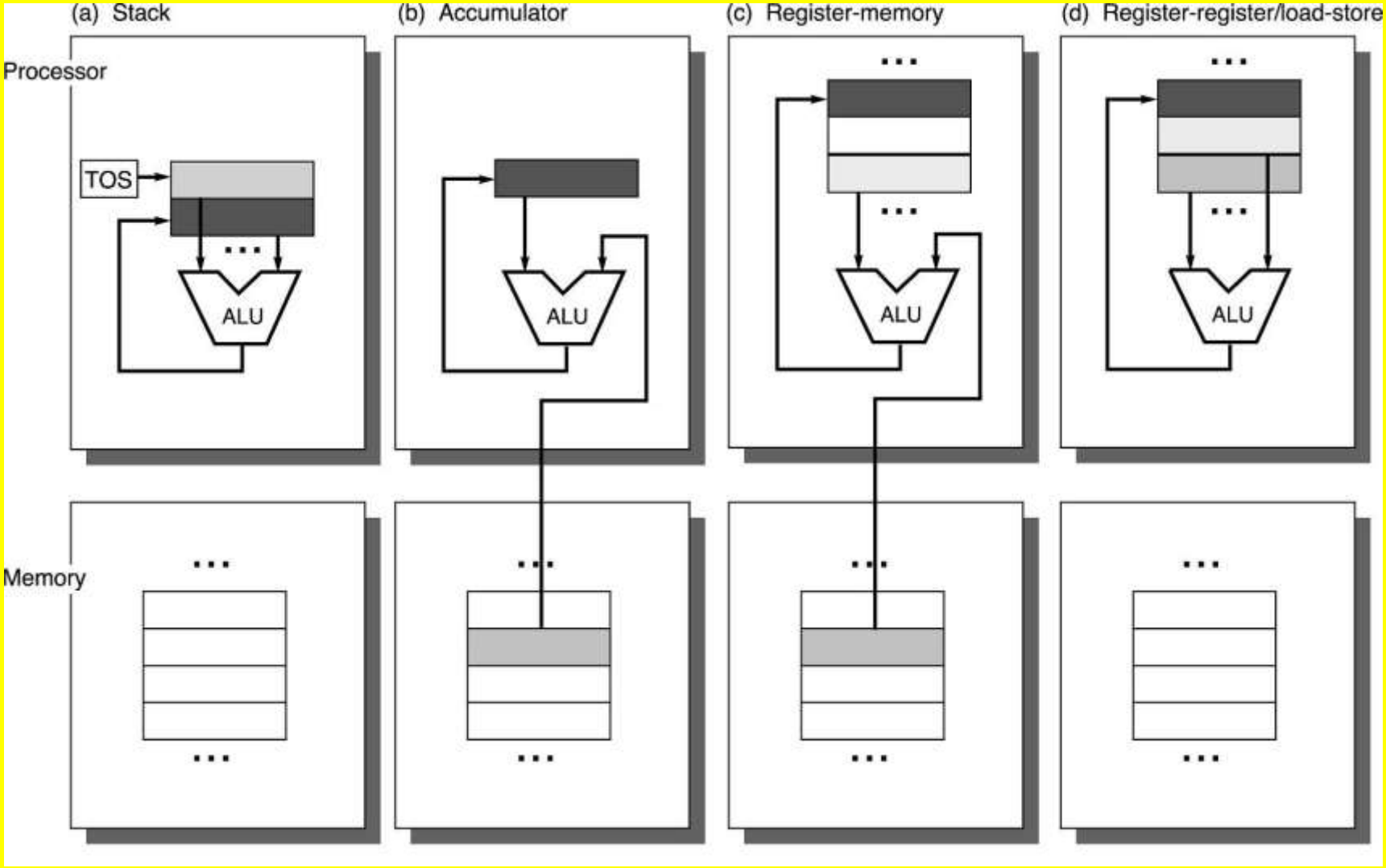
Register-Memory (1970s to present, e.g. 80x86):

2-address	add R1, A	R1 ¬ R1 + mem[A]
	load R1, A	R1 ¬ mem[A]

Register-Register (Load/Store) (1960s to present, e.g. MIPS):

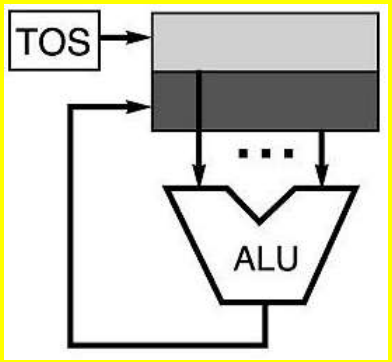
3-address	add R1, R2, R3	R1 ¬ R2 + R3
	load R1, R2	R1 ¬ mem[R2]
	store R1, R2	mem[R1] ¬ R2

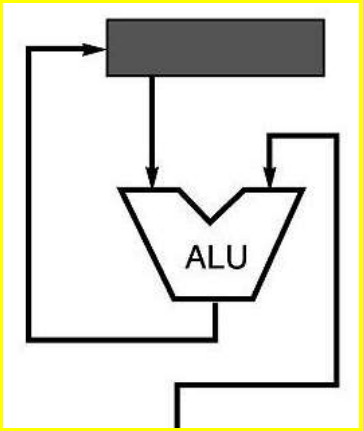
Operand Locations in Four ISA Classes



Code Sequence C = A + B

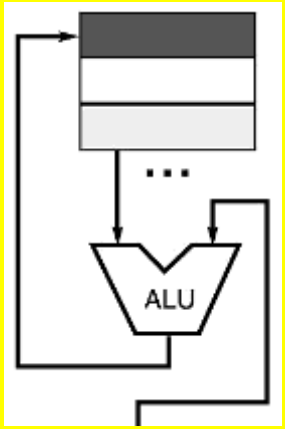
Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Add R1, B Store C, R1	Load R1,A Load R2, B Add R3, R1, R2 Store C, R3





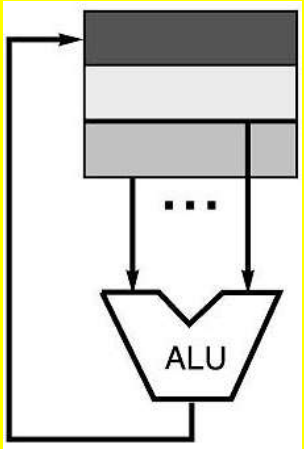
memory

acc = acc + mem[C]



memory

R1 = R1 + mem[C]



R3 = R1 + R2

REVERSE POLISH NOTATION

Arithmetic Expressions: A + B

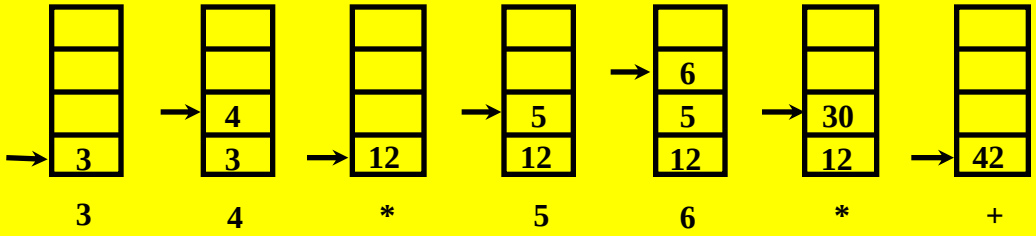
A + B	Infix notation
+ A B	Prefix or Polish notation
A B +	Postfix or reverse Polish notation

- The reverse Polish notation is very suitable for stack manipulation

Evaluation of Arithmetic Expressions

Any arithmetic expression can be expressed in parenthesis-free Polish notation, including reverse Polish notation

(3 * 4) + (5 * 6) ⇒ 3 4 * 5 6 * +



Stack Architectures

- Instruction set:

  add, sub, mult, div, . . .

  push A, pop A

- Example: A*B - (A+C*B)

  push A

  push B

  mul

  push A

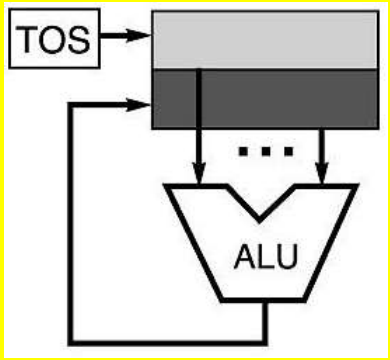
  push C

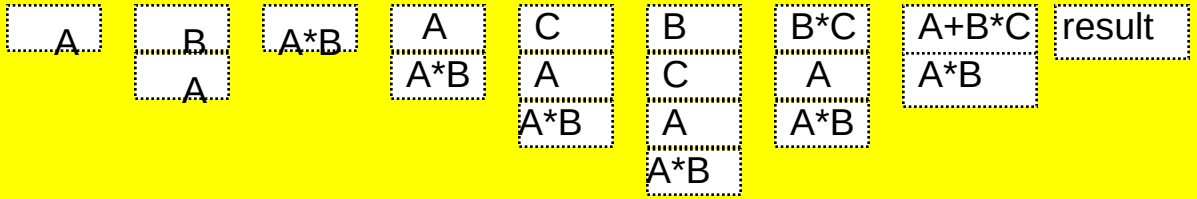
  push B

  mul

  add

  sub





Stacks: Pros and Cons

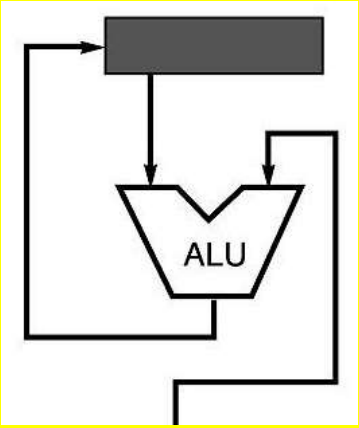
- Pros
  - Good code density (implicit top of stack)
  - Low hardware requirements
  - Easy to write a simpler compiler for stack architectures
- Cons
  - Stack becomes the bottleneck
  - Little ability for parallelism or pipelining
  - Data is not always at the top of stack when need, so additional instructions like TOP and SWAP are needed
  - Difficult to write an optimizing compiler for stack architectures

Accumulator Architectures

- Instruction set:

  add A, sub A, mult A, div A, . . .

  load A, store A



acc = acc +,-,*,/ mem[A]

- Example: A*B - (A+C*B)

  load B

  mul C

  add A

  store D

  load A

  mul B

  sub D

Accumulators: Pros and Cons

- Pros
  - Very low hardware requirements
  - Easy to design and understand
- Cons
  - Accumulator becomes the bottleneck
  - Little ability for parallelism or pipelining
  - High memory traffic

Memory-Memory Architectures

- Instruction set:

(3 operands)	add A, B, C	sub A, B, C	mul A, B, C
(2 operands)	add A, B	sub A, B	mul A, B

- Example: A*B - (A+C*B)
  - 3 operands

```
mul D, A, B
mul E, C, B
add E, A, E
sub E, D, E
```

2 operands

```
mov D, A
mul D, B
mov E, C
mul E, B
add E, A
sub E, D
```

Memory-Memory: Pros and Cons

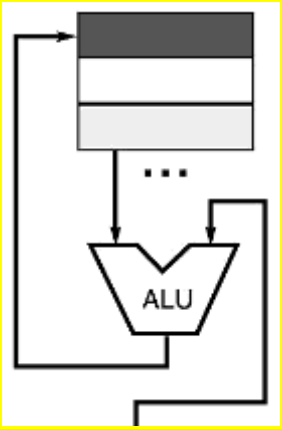
- Pros
  - Requires fewer instructions (especially if 3 operands)
  - Easy to write compilers for (especially if 3 operands)
- Cons
  - Very high memory traffic (especially if 3 operands)
  - Variable number of clocks per instruction
  - With two operands, more data movements are required

Register-Memory Architectures

- Instruction set:

  add R1, A sub R1, A mul R1, B

  load R1, A store R1, A



R1 = R1 +,-,*,/ mem[B]

- Example: A*B - (A+C*B)

```
load R1, A
mul R1, B       /*    A*B          */
store R1, D
load R2, C
mul R2, B       /*    C*B          */
add R2, A       /*    A + CB       */
sub R2, D       /*    AB - (A + C*B) */
```

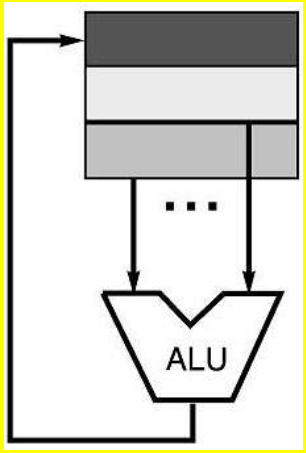
Memory-Register: Pros and Cons

- Pros
  - Some data can be accessed without loading first
  - Instruction format easy to encode
  - Good code density
- Cons
  - Operands are not equivalent (poor orthogonal)
  - Variable number of clocks per instruction
  - May limit number of registers

Load-Store Architectures

- Instruction set:

  add R1, R2, R3 sub R1, R2, R3 mul R1, R2, R3
  load R1, &Astore R1, &A move R1, R2



R3 = R1 +,-,*,/ R2

- Example: A*B - (A+C*B)

```
load R1, &A
load R2, &B
load R3, &C
mul R7, R3, R2          /*   C*B          */
add R8, R7, R1          /*   A + C*B      */
mul R9, R1, R2          /*   A*B          */
sub R10, R9, R8         /*   A*B - (A+C*B) */
```

Load-Store: Pros and Cons

- Pros
  - Simple, fixed length instruction encodings
  - Instructions take similar number of cycles
  - Relatively easy to pipeline and make superscalar
- Cons
  - Higher instruction count
  - Not all instructions need three operands
  - Dependent on good compiler

Registers: Advantages and Disadvantages

- Advantages
  - Faster than cache or main memory (no addressing mode or tags)
  - Deterministic (no misses)
  - Can replicate (multiple read ports)
  - Short identifier (typically 3 to 8 bits)
  - Reduce memory traffic
- Disadvantages
  - Need to save and restore on procedure calls and context switch
  - Can't take the address of a register (for pointers)
  - Fixed size (can't store strings or structures efficiently)
  - Compiler must manage
  - Limited number

Every ISA designed after 1980 uses a load-store ISA (i.e RISC, to simplify CPU design).

Types of Operations

- Arithmetic and Logic: AND, ADD
- Data Transfer: MOVE, LOAD, STORE
- Control BRANCH, JUMP, CALL
- System OS CALL, VM
- Floating Point ADDF, MULF, DIVF
- Decimal ADDD, CONVERT
- String MOVE, COMPARE
- Graphics (DE)COMPRESS

80x86 Instruction Frequency

Rank	*Instruction*	*Frequency*
1	load	22%
2	branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	register move	4%
9	call	1%
10	return	1%
Total		96%

DATA TRANSFER INSTRUCTIONS

Typical Data Transfer Instructions

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

Data Transfer Instructions with Different Addressing Modes

Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	AC ← M[ADR]
Indirect address	LD @ADR	AC ← M[M[ADR]]
Relative address	LD $ADR	AC ← M[PC + ADR]
Immediate operand	LD #NBR	AC ← NBR
Index addressing	LD ADR(X)	AC ← M[ADR + XR]
Register	LD R1	AC ← R1
Register indirect	LD (R1)	AC ← M[R1]
Autoincrement	LD (R1)+	AC ← M[R1], R1 ← R1 + 1
Autodecrement	LD -(R1)	R1 ← R1 - 1, AC ← M[R1]

DATA MANIPULATION INSTRUCTIONS

Three Basic Types:

- **Arithmetic instructions**
- **Logical and bit manipulation instructions**
- **Shift instructions**

Arithmetic Instructions

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with Carry	ADDC
Subtract with Borrow	SUBB
Negate(2's Complement)	NEG

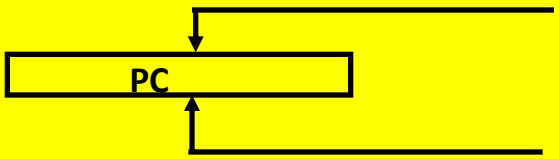
Logical and Bit Manipulation Instructions

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

Shift Instructions

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right thru carry	RORC
Rotate left thru carry	ROLC

PROGRAM CONTROL INSTRUCTIONS



+1
In-Line Sequencing
(Next instruction is fetched from the next adjacent location in the memory)

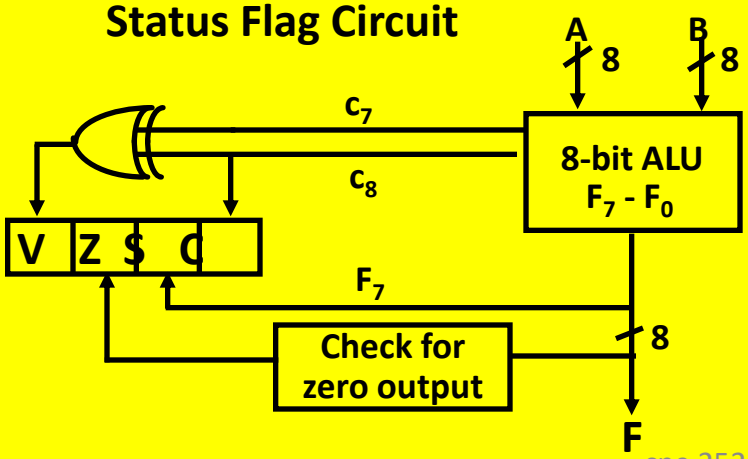
Address from other source; Current Instruction, Stack, etc
Branch, Conditional Branch, Subroutine, etc

Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RTN
Compare(by -)	CMP
Test (by AND)	TST

*** CMP and TST instructions do not retain their results of operations(- and AND, respectively). They only set or clear certain Flags.**

Status Flag Circuit



CONDITIONAL BRANCH INSTRUCTIONS

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	Z = 1
BNZ	Branch if not zero	Z = 0
BC	Branch if carry	C = 1
BNC	Branch if no carry	C = 0
BP	Branch if plus	S = 0
BM	Branch if minus	S = 1
BV	Branch if overflow	V = 1
BNV	Branch if no overflow	V = 0
	Unsigned **compare conditions (A - B)**	
BHI	Branch if higher	A > B
BHE	Branch if higher or equal	A ≥ B
BLO	Branch if lower	A < B
BLOE	Branch if lower or equal	A ≤ B
BE	Branch if equal	A = B
BNE	Branch if not equal	A ≠ B
	Signed **compare conditions (A - B)**	
BGT	Branch if greater than	A > B
BGE	Branch if greater or equal	A ≥ B
BLT	Branch if less than	A < B
BLE	Branch if less or equal	A ≤ B
BE	Branch if equal	A = B
BNE	Branch if not equal	A ≠ B

Thank You