# CENTRAL PROCESSING UNIT

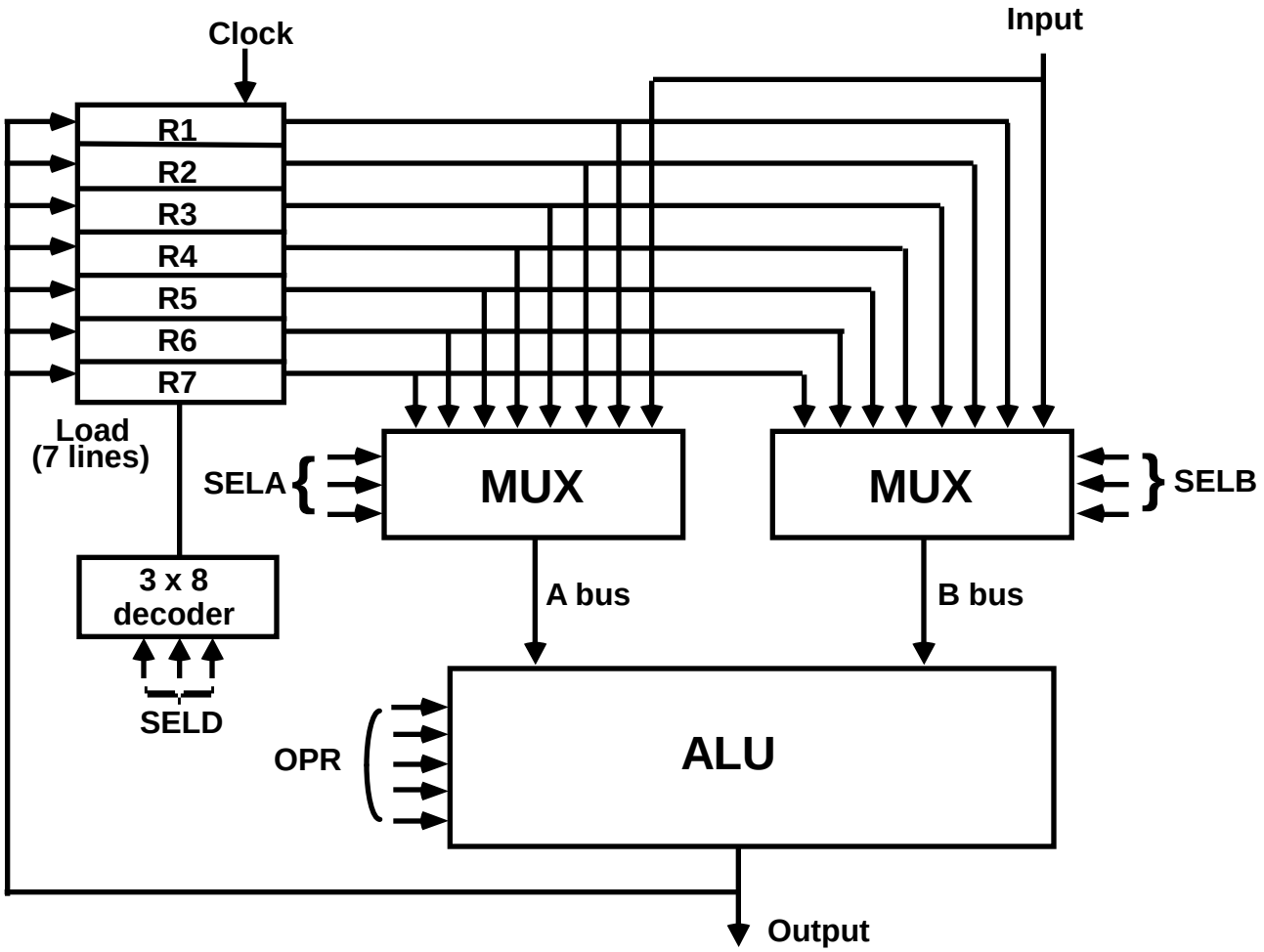
- Stack Organization
- Instruction Formats
- Addressing Modes
- Data Transfer and Manipulation
- Program Control
- Reduced Instruction Set Computer

# REGISTERS

- In Basic Computer, there is only one general purpose register, the Accumulator (AC)
- In modern CPUs, there are many general purpose registers
- It is advantageous to have many registers
  - Transfer between registers within the processor are relatively fast
  - Going “off the processor” to access memory is much slower

- How many registers will be the best ?

# GENERAL REGISTER ORGANIZATION

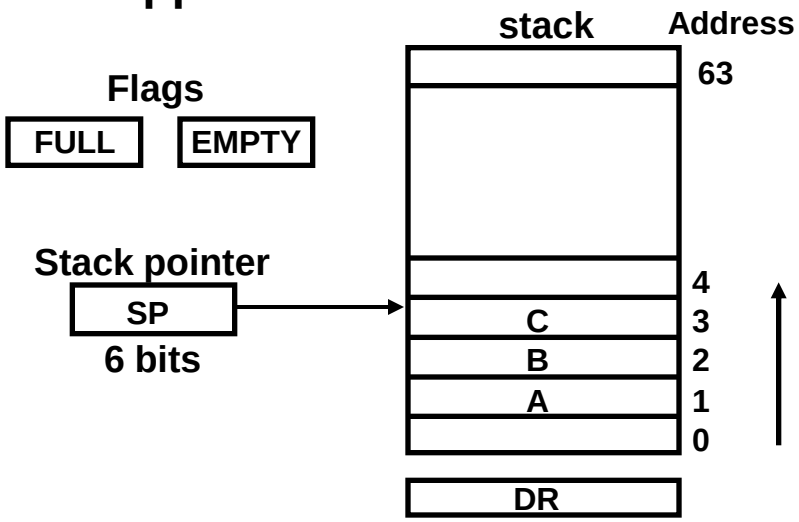


# REGISTER STACK ORGANIZATION

**Stack**

- Very useful feature for nested subroutines, nested interrupt services
- Also efficient for arithmetic expression evaluation
- Storage which can be accessed in LIFO
- Pointer: SP
- Only PUSH and POP operations are applicable

**Register Stack**



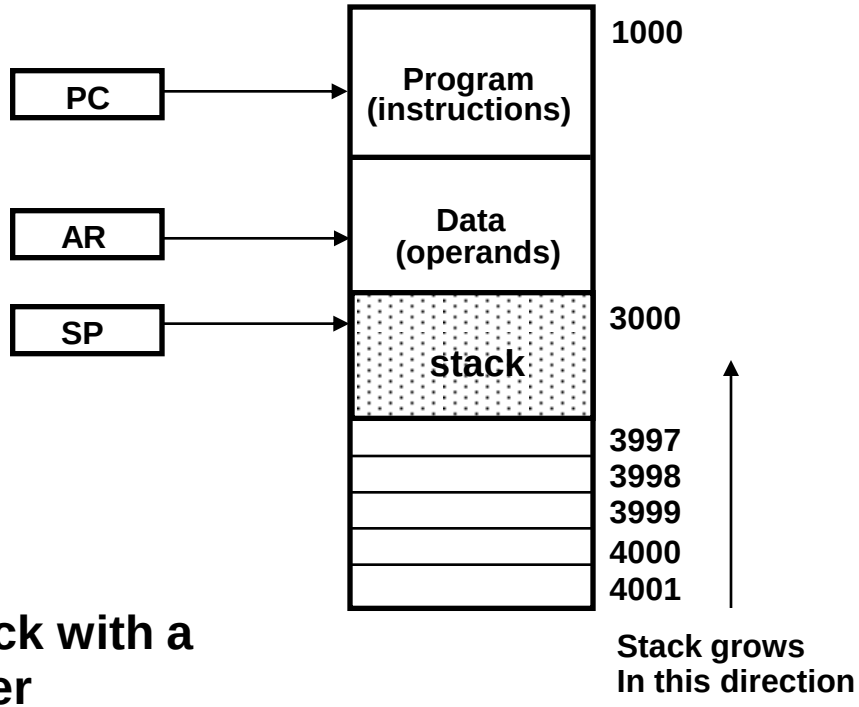

**Push, Pop operations**

/* Initially, SP = 0, EMPTY = 1, FULL = 0 */

| PUSH | POP |
|---|---|
| SP ← SP + 1 | DR ← M[SP] |
| M[SP] ← DR | SP ← SP – 1 |
| If (SP = 0) then (FULL ← 1) | If (SP = 0) then (EMPTY ← 1) |
| EMPTY ← 0 | FULL ← 0 |

# MEMORY STACK ORGANIZATION

**Memory with Program, Data, and Stack Segments**




- A portion of memory is used as a stack with a processor register as a stack pointer

- PUSH: $SP \leftarrow SP - 1$
  $M[SP] \leftarrow DR$
- POP: $DR \leftarrow M[SP]$
  $SP \leftarrow SP + 1$

- Most computers do not provide hardware to check stack overflow (full stack) or underflow (empty stack) → must be done in software

# REVERSE POLISH NOTATION

- **Arithmetic Expressions: A + B**

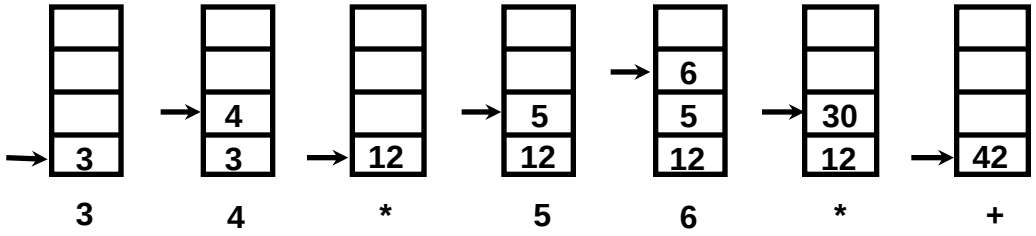
  A + B  Infix notation
  + A B  Prefix or Polish notation
  A B +  Postfix or reverse Polish notation

  - 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 * +



# PROCESSOR ORGANIZATION

- In general, most processors are organized in one of 3 ways
  - Single register (Accumulator) organization
    - » Basic Computer is a good example
    - » Accumulator is the only general purpose register
  - General register organization
    - » Used by most modern computer processors
    - » Any of the registers can be used as the source or destination for computer operations
  - Stack organization
    - » All operations are done using the hardware stack
    - » For example, an OR instruction will pop the two top elements from the stack, do a logical OR on them, and push the result on the stack

# INSTRUCTION FORMAT

- **Instruction Fields**

  **OP-code field - specifies the operation to be performed**
  **Address field - designates memory address(es) or a processor register(s)**
  **Mode field - determines how the address field is to be interpreted (to get effective address or the operand)**

- **The number of address fields in the instruction format depends on the internal organization of CPU**

- **The three most common CPU organizations:**

  **Single accumulator organization:**

```
ADD     X              /* AC ← AC + M[X]  */
```

  **General register organization:**

```
ADD     R1, R2, R3     /* R1 ← R2 + R3  */
ADD     R1, R2         /* R1 ← R1 + R2  */
MOV     R1, R2         /* R1 ← R2  */
ADD     R1, X          /* R1 ← R1 + M[X]  */
```

  **Stack organization:**

```
PUSH    X              /* TOS ← M[X]  */
ADD
```

# THREE, AND TWO-ADDRESS INSTRUCTIONS

- **Three-Address Instructions**

**Program to evaluate X = (A + B) * (C + D) :**

```
ADD   R1, A, B      /*  R1 ← M[A] + M[B]    */
ADD   R2, C, D      /*  R2 ← M[C] + M[D]    */
MUL   X, R1, R2     /*  M[X] ← R1 * R2      */
```

- Results in short programs
- Instruction becomes long (many bits)

- **Two-Address Instructions**

**Program to evaluate X = (A + B) * (C + D) :**

```
MOV   R1, A         /* R1 ← M[A]          */
ADD   R1, B         /* R1 ← R1 + M[A]     */
MOV   R2, C         /* R2 ← M[C]          */
ADD   R2, D         /* R2 ← R2 + M[D]     */
MUL   R1, R2        /* R1 ← R1 * R2       */
MOV   X, R1         /* M[X] ← R1          */
```

# ONE, AND ZERO-ADDRESS INSTRUCTIONS

- **One-Address Instructions**
  - Use an implied AC register for all data manipulation
  - Program to evaluate X = (A + B) * (C + D) :

```
LOAD    A    /*  AC ← M[A]          */
ADD     B    /*  AC ← AC + M[B]     */
STORE   T    /*  M[T] ← AC          */
LOAD    C    /*  AC ← M[C]          */
ADD     D    /*  AC ← AC + M[D]     */
MUL     T    /*  AC ← AC * M[T]     */
STORE   X    /*  M[X] ← AC          */
```

- **Zero-Address Instructions**
  - Can be found in a stack-organized computer
  - Program to evaluate X = (A + B) * (C + D) :

```
PUSH    A    /*  TOS ← A                    */
PUSH    B    /*  TOS ← B                    */
ADD          /*  TOS ← (A + B)              */
PUSH    C    /*  TOS ← C                    */
PUSH    D    /*  TOS ← D                    */
ADD          /*  TOS ← (C + D)              */
MUL          /*  TOS ← (C + D) * (A + B)    */
POP     X    /*  M[X] ← TOS                 */
```

# ADDRESSING MODES

- **Addressing Modes**

  *** Specifies a rule for interpreting or modifying the address field of the instruction (before the operand is actually referenced)**

  *** Variety of addressing modes**

  **- to give programming flexibility to the user**
  **- to use the bits in the address field of the instruction efficiently**

# TYPES OF ADDRESSING MODES

- **Implied Mode**

  **Address of the operands are specified implicitly in the definition of the instruction**

  **- No need to specify address in the instruction**

  **- Examples from Basic Computer**

  **CMA,CLA, CME, INP**

**Zero address instructions in the stack-organized computers are implied mode instructions since the operands are implied to be on top of the stack.**

- **Immediate Mode**

  **Instead of specifying the address of the operand, operand itself is specified**

  **- No need to specify address in the instruction**

  **- However, operand itself needs to be specified**

  **- Sometimes, require more bits than the address**

  **- Fast to acquire an operand**

# TYPES OF ADDRESSING MODES

- **Register Mode**

  Address specified in the instruction is the register address
  - Designated operand need to be in a register
  - Shorter address than the memory address
  - Saving address field in the instruction
  - Faster to acquire an operand than the memory addressing
  - EA = IR(R) (IR(R): Register field of IR)

- **Register Indirect Mode**

  Instruction specifies a register which contains the memory address of the operand
  - Saving instruction bits since register address is shorter than the memory address
  - Slower to acquire an operand than both the register addressing or memory addressing
  - EA = [IR(R)] ([x]: Content of x)

- **. Autoincrement or Autodecrement Mode**

  - This is similar to register inditect mode except that, When the address in the register is used to access memory, the value in the register is incremented or decremented by 1 automatically

# TYPES OF ADDRESSING MODES

- **Direct Address Mode**

  **Instruction specifies the memory address which can be used directly to access the memory**

  **- Faster than the other memory addressing modes**

  **- Too many bits are needed to specify the address for a large physical memory space**

  **- EA = IR(addr) (IR(addr): address field of IR)**

- **Indirect Addressing Mode**

  **The address field of an instruction specifies the address of a memory location that contains the address of the operand**

  **- When the abbreviated address is used large physical memory can be addressed with a relatively small number of bits**

  **- Slow to acquire an operand because of an additional memory access**

  **- EA = M[IR(address)]**

# TYPES OF ADDRESSING MODES

- **Relative Addressing Modes**

  **The Address fields of an instruction specifies the part of the address (abbreviated address) which can be used along with a designated register to calculate the address of the operand**

  **- Address field of the instruction is short**

  **- Large physical memory can be accessed with a small number of address bits**

  **- EA = f(IR(address), R), R is sometimes implied**

**3 different Relative Addressing Modes depending on R;**

*** PC Relative Addressing Mode (R = PC)**

**- EA = PC + IR(address)**

*** Indexed Addressing Mode (R = IX, where IX: Index Register)**

**- EA = IX + IR(address)**

*** Base Register Addressing Mode (R = BAR, where BAR: Base Address Register)**

**- EA = BAR + IR(address)**

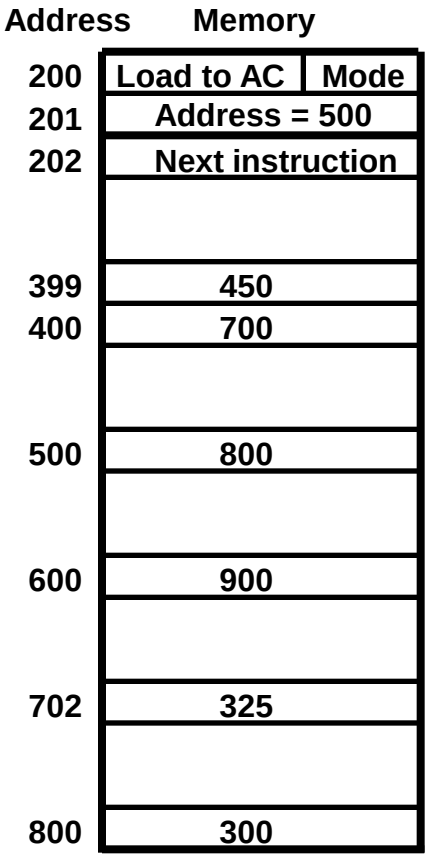
# ADDRESSING MODES - EXAMPLES -

PC = 200

R1 = 400

XR = 100

AC




| Addressing Mode | Effective Address | | Content of AC |
|---|---|---|---|
| Direct address | 500 | /* AC ← (500) */ | 800 |
| Immediate operand | - | /* AC ← 500 */ | 500 |
| Indirect address | 800 | /* AC ← ((500)) */ | 300 |
| Relative address | 702 | /* AC ← (PC+500) */ | 325 |
| Indexed address | 600 | /* AC ← (XR+500) */ | 900 |
| Register | - | /* AC ← R1 */ | 400 |
| Register indirect | 400 | /* AC ← (R1) */ | 700 |
| Autoincrement | 400 | /* AC ← (R1)+ */ | 700 |
| Autodecrement | 399 | /* AC ← -(R) */ | 450 |

# 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 |

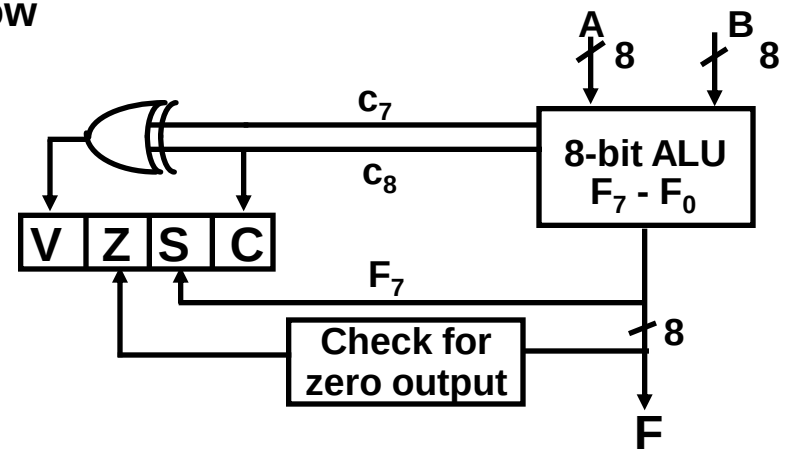
- **Shift Instructions**

| 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 |

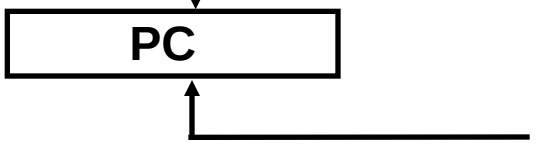
# FLAG, PROCESSOR STATUS WORD

- In Basic Computer, the processor had several (status) flags – 1 bit value that indicated various information about the processor's state – E, FGI, FGO, I, IEN, R
- In some processors, flags like these are often combined into a register – the processor status register (PSR); sometimes called a processor status word (PSW)
- Common flags in PSW are
  - C (Carry): Set to 1 if the carry out of the ALU is 1
  - S (Sign): The MSB bit of the ALU's output
  - Z (Zero): Set to 1 if the ALU's output is all 0's
  - V (Overflow): Set to 1 if there is an overflow

Status Flag Circuit



# 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.**

# 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 |

# SUBROUTINE CALL AND RETURN

- **Subroutine Call**
  - Call subroutine
  - Jump to subroutine
  - Branch to subroutine
  - Branch and save return address

- **Two Most Important Operations are Implied;**

  * Branch to the beginning of the Subroutine
    - Same as the Branch or Conditional Branch

  * Save the Return Address to get the address of the location in the Calling Program upon exit from the Subroutine

- **Locations for storing Return Address**
  - Fixed Location in the subroutine (Memory)
  - Fixed Location in memory
  - In a processor Register
  - In memory *stack*
    - most efficient way

```
CALL
  SP ← SP - 1
  M[SP] ← PC
  PC ← EA

RTN
  PC ← M[SP]
  SP ← SP + 1
```

# COMPLEX INSTRUCTION SET COMPUTER

- These computers with many instructions and addressing modes came to be known as Complex Instruction Set Computers (CISC)

- One goal for CISC machines was to have a machine language instruction to match each high-level language statement type

# COMPLEX INSTRUCTION SET COMPUTER

- **Another characteristic of CISC computers is that they have instructions that act directly on memory addresses**
  - **For example,**
    **ADD L1, L2, L3**
    **that takes the contents of *M*[*L1*] adds it to the contents of *M*[*L2*] and stores the result in location *M*[*L3*]**
- **An instruction like this takes three memory access cycles to execute**
- **That makes for a potentially very long instruction execution cycle**
- **The problems with CISC computers are**
  - **The complexity of the design may slow down the processor,**
  - **The complexity of the design may result in costly errors in the processor design and implementation,**
  - **Many of the instructions and addressing modes are used rarely, if ever**

# SUMMARY: CRITICISMS ON CISC

**High Performance General Purpose Instructions**

- Complex Instruction
  - → Format, Length, Addressing Modes
  - → Complicated instruction cycle control due to the complex decoding HW and decoding process

- Multiple memory cycle instructions
  - → Operations on memory data
  - → Multiple memory accesses/instruction

- Microprogrammed control is necessity
  - → Microprogram control storage takes substantial portion of CPU chip area
  - → Semantic Gap is large between machine instruction and microinstruction

- General purpose instruction set includes all the features required by individually different applications
  - → When any one application is running, all the features required by the other applications are extra burden to the application

# REDUCED INSTRUCTION SET COMPUTERS

- In the late ‘70s and early ‘80s there was a reaction to the shortcomings of the CISC style of processors
- Reduced Instruction Set Computers (RISC) were proposed as an alternative
- The underlying idea behind RISC processors is to simplify the instruction set and reduce instruction execution time

- RISC processors often feature:
  - Few instructions
  - Few addressing modes
  - Only load and store instructions access memory
  - All other operations are done using on-processor registers
  - Fixed length instructions
  - Single cycle execution of instructions
  - The control unit is hardwired, not microprogrammed

# REDUCED INSTRUCTION SET COMPUTERS

- Since all but the load and store instructions use only registers for operands, only a few addressing modes are needed
- By having all instructions the same length, reading them in is easy and fast
- The fetch and decode stages are simple, compared to a CISC machine
- The instruction and address formats are designed to be easy to decode
- Unlike the variable length CISC instructions, the opcode and register fields of RISC instructions can be decoded simultaneously
- The control logic of a RISC processor is designed to be simple and fast
- The control logic is simple because of the small number of instructions and the simple addressing modes
- The control logic is hardwired, rather than microprogrammed, because hardwired control is faster

# CHARACTERISTICS OF RISC

- RISC Characteristics
  - Relatively few instructions
  - Relatively few addressing modes
  - Memory access limited to load and store instructions
  - All operations done within the registers of the CPU
  - Fixed-length, easily decoded instruction format
  - Single-cycle instruction format
  - Hardwired rather than microprogrammed control

- Advantages of RISC
  - VLSI Realization
  - Computing Speed
  - Design Costs and Reliability
  - High Level Language Support