# CENTRAL  PROCESSING  UNIT

- **Introduction**

- **General Register Organization**

- **Stack Organization**

- **Instruction Formats**

- **Addressing Modes**

- **Data Transfer and Manipulation**

- **Program Control**

- **Reduced Instruction Set Computer**

---

# MAJOR  COMPONENTS  OF  CPU

- **Storage Components**
  **Registers**
  **Flags**

- **Execution (Processing) Components**
  **Arithmetic Logic Unit(ALU)**
  **Arithmetic calculations, Logical computations, Shifts/Rotates**

- **Transfer Components**
  **Bus**

- **Control Components**
  **Control Unit**

---
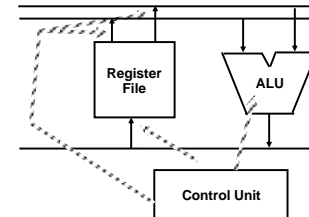
# REGISTERS

- **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
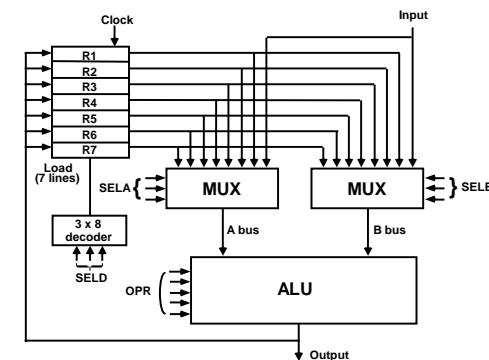
## OPERATION  OF  CONTROL  UNIT
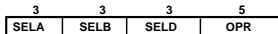
**The control unit**
   **Directs the information flow through ALU by**
      **- Selecting various *Components* in the system**
      **- Selecting the *Function* of ALU**

**Example:  R1 ← R2 + R3**
   **[1] MUX A selector (SELA):  BUS A ← R2**
   **[2] MUX B selector (SELB):  BUS B ← R3**
   **[3] ALU operation selector (OPR): ALU to ADD**
   **[4] Decoder destination selector (SELD): R1 ← Out Bus**

**Control Word**

| 3 | 3 | 3 | 5 |
|---|---|---|---|
| SELA | SELB | SELD | OPR |

**Encoding of register selection fields**

| Binary Code | SELA | SELB | SELD |
|---|---|---|---|
| 000 | Input | Input | None |
| 001 | R1 | R1 | R1 |
| 010 | R2 | R2 | R2 |
| 011 | R3 | R3 | R3 |
| 100 | R4 | R4 | R4 |
| 101 | R5 | R5 | R5 |
| 110 | R6 | R6 | R6 |
| 111 | R7 | R7 | R7 |

## ALU  CONTROL

**Encoding of ALU operations**

| OPR Select | Operation | Symbol |
|---|---|---|
| 00000 | Transfer A | TSFA |
| 00001 | Increment A | INCA |
| 00010 | ADD A + B | ADD |
| 00101 | Subtract A - B | SUB |
| 00110 | Decrement A | DECA |
| 01000 | AND A and B | AND |
| 01010 | OR A and B | OR |
| 01100 | XOR A and B | XOR |
| 01110 | Complement A | COMA |
| 10000 | Shift right A | SHRA |
| 11000 | Shift left A | SHLA |

**Examples of ALU Microoperations**

| | Symbolic Designation | | | | |
|---|---|---|---|---|---|
| Microoperation | SELA | SELB | SELD | OPR | Control Word |
| R1 ← R2 – R3 | R2 | R3 | R1 | SUB | 010 011 001 00101 |
| R4 ← R4 ∨ R5 | R4 | R5 | R4 | OR | 100 101 100 01010 |
| R6 ← R6 + 1 | R6 | - | R6 | INCA | 110 000 110 00001 |
| R7 ← R1 | R1 | - | R7 | TSFA | 001 000 111 00000 |
| Output ← R2 | R2 | - | None | TSFA | 010 000 000 00000 |
| Output ← Input | Input | - | None | TSFA | 000 000 000 00000 |
| R4 ← shl R4 | R4 | - | R4 | SHLA | 100 000 100 11000 |
| R5 ← 0 | R5 | R5 | R5 | XOR | 101 101 101 01100 |

## 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**

**Flags**

| FULL | EMPTY |

**stack    Address**

63

**Stack pointer**

| SP |
6 bits

| | |
| C | 3 |
| B | 2 |
| A | 1 |
| | 0 |

| DR |
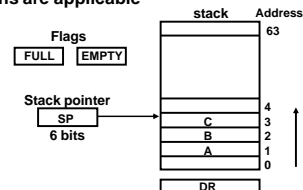
**Push, Pop operations**

/*  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**

| PC |
| AR |
| SP |

| | 1000 |
| Program (instructions) | |
| Data (operands) | |
| stack | 3000 |
| | 3997 |
| | 3998 |
| | 3999 |
| | 4000 |
| | 4001 |

**Stack grows In this direction**
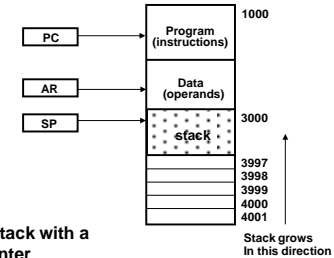
- **A portion of memory is used as a stack with a processor register as a stack pointer**

- **PUSH:  SP ← SP - 1**
             **M[SP] ← DR**
- **POP:    DR ← M[SP]**
             **SP ← 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**
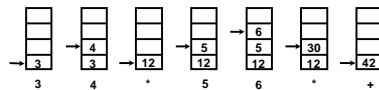  - **A B +**    **Postfix or reverse Polish notation**

    **- 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**

  $$(3 * 4) + (5 * 6) \;\; \Rightarrow \;\; 3\ 4 * 5\ 6 * +$$

| 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**

    - **EA = AC, or EA = Stack[SP]**

    - **Examples from Basic Computer**

        **CLA, CME, INP**

• **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**

    - **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 | | | |

| Address | Memory |
|---|---|
| 200 | Load to AC \| Mode |
| 201 | Address = 500 |
| 202 | Next instruction |
| 399 | 450 |
| 400 | 700 |
| 500 | 800 |
| 600 | 900 |
| 702 | 325 |
| 800 | 300 |

| 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 ← (RX+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**

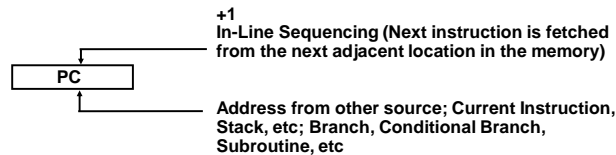| Name | Mnemonic |
|------|----------|
| Logical shift right | SHR |
| Logical shift left | SHL |
| Arithmetic shift right | SHRA |
| Arithmetic shift left | SHLA |
| Rotate right | ROR |
| Rotate left | ROL |
| Rotate right thru carry | RORC |
| Rotate left thru carry | ROLC |

---

## 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**

$A$   8   $B$   8

$c_7$

$c_8$

**8-bit ALU** $F_7 - F_0$

| V | Z | S | C |

$F_7$

Check for zero output   8

$F$

---

## PROGRAM  CONTROL  INSTRUCTIONS

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

| PC |

**Address from other source; Current Instruction, Stack, etc; Branch, Conditional Branch, Subroutine, etc**
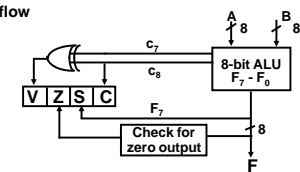
- **Program Control Instructions**

| 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 | $A \geq B$ |
| BLO | Branch if lower | $A < B$ |
| BLOE | Branch if lower or equal | $A \leq B$ |
| BE | Branch if equal | $A = B$ |
| BNE | Branch if not equal | $A \neq B$ |
| *Signed* compare conditions (A - B) | | |
| BGT | Branch if greater than | $A > B$ |
| BGE | Branch if greater or equal | $A \geq B$ |
| BLT | Branch if less than | $A < B$ |
| BLE | Branch if less or equal | $A \leq B$ |
| BE | Branch if equal | $A = B$ |
| BNE | Branch if not equal | $A \neq B$ |

## 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
```

---

## PROGRAM  INTERRUPT

**Types of Interrupts**

   **External interrupts**
     **External Interrupts initiated from the outside of CPU and Memory**
     - **I/O Device → Data transfer request or Data transfer complete**
     - **Timing Device → Timeout**
     - **Power Failure**
     - **Operator**

   **Internal interrupts (traps)**
     **Internal Interrupts are caused by the currently running program**
     - **Register, Stack Overflow**
     - **Divide by zero**
     - **OP-code Violation**
     - **Protection Violation**

   **Software Interrupts**
     **Both External and Internal Interrupts are initiated by the computer HW.**
     **Software Interrupts are initiated by the executing an instruction.**
     - **Supervisor Call → Switching from a user mode to the supervisor mode**
                       **→ Allows to execute a certain class of operations**
                          **which are not allowed in the user mode**

---

## INTERRUPT  PROCEDURE

**Interrupt Procedure and Subroutine Call**

   - **The interrupt is usually initiated by an internal or**
   **an external signal rather than from the execution of**
   **an instruction (except for the software interrupt)**

   - **The address of the interrupt service program is**
   **determined by the hardware rather than from the**
   **address field of an instruction**

   - **An interrupt procedure usually stores all the**
   **information necessary to define the state of CPU**
   **rather than storing only the PC.**

     **The state of the CPU is determined from;**
       **Content of the PC**
       **Content of all processor registers**
       **Content of status bits**
     **Many ways of saving the CPU state**
       **depending on the CPU architectures**

---

## RISC: Historical Background

**IBM System/360, 1964**

• **The real beginning of modern computer architecture**

• **Distinction between** *Architecture* **and** *Implementation*

• **Architecture: The abstract structure of a computer**
                    **seen by an assembly-language programmer**
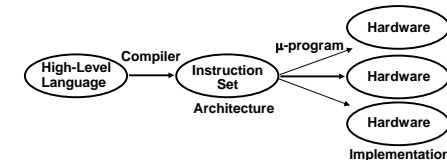


• **Continuing growth in semiconductor memory and**
**microprogramming**

     ⇒ **A much richer and complicated instruction sets**

     ⇒ **CISC(Complex Instruction Set Computer)**

## ARGUMENTS ADVANCED AT THAT TIME

- **Richer instruction sets would simplify compilers**
- **Richer instruction sets would alleviate the software crisis**
  - **– move as much functions to the hardware as possible**
- **Richer instruction sets would improve *architecture quality***

## ARCHITECTURE  DESIGN  PRINCIPLES  - IN  70's -

- **Large microprograms would add little or nothing to the cost of the machine**
  - **← Rapid growth of memory technology**
  - **⇒ Large General Purpose Instruction Set**

- **Microprogram is much faster than the machine instructions**
  - **← Microprogram memory is much faster than main memory**
  - **⇒ Moving the software functions into microprogram for the high performance machines**

- **Execution speed is proportional to the program size**
  - **← Architectural techniques that led to small program**
  - **⇒ High performance instruction set**

- **Number of registers in CPU has limitations**
  - **← Very costly**
  - **⇒ Difficult to utilize them efficiently**

## COMPARISONS  OF  EXECUTION  MODELS

**A ← B + C     Data: 32-bit**

- **Register-to-register**

| 8 | 4 | 16 |
|------|----|----|
| Load | rB | B |
| Load | rC | C |
| Add | rA | rB | rC |
| Store | rA | A |

**I = 104b; D = 96b; M = 200b**

- **Memory-to-register**

| 8 | 16 |
|-------|----|
| Load | B |
| Add | C |
| Store | A |

**I = 72b; D = 96b; M = 168b**

- **Memory-to-memory**

| 8 | 16 | 16 | 16 |
|-----|---|---|---|
| Add | B | C | A |

**I = 56b; D = 96b; M = 152b**

## FOUR  MODERN  ARCHITECTURES  IN  70's

| | IBM 370/168 | DEC VAX-11/780 | Xerox Dorado | Intel iAPX-432 |
|---|---|---|---|---|
| Year | 1973 | 1978 | 1978 | 1982 |
| # of instrs. | 208 | 303 | 270 | 222 |
| Control mem. size | 420 Kb | 480 Kb | 136 Kb | 420 Kb |
| Instr. size (bits) | 16-48 | 16-456 | 8-24 | 6-321 |
| Technology | ECL MSI | TTL MSI | ECL MSI | NMOS VLSI |
| Execution model | reg-mem mem-mem reg-reg | reg-mem mem-mem reg-reg | stack | stack mem-mem |
| Cache size | 64 Kb | 64 Kb | 64 Kb | 64 Kb |

## 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**

## VARIABLE LENGTH INSTRUCTIONS

- **The large number of instructions and addressing modes led CISC machines to have variable length instruction formats**
- **The large number of instructions means a greater number of bits to specify them**
- **In order to manage this large number of opcodes efficiently, they were encoded with different lengths:**
  - **More frequently used instructions were encoded using short opcodes.**
  - **Less frequently used ones were assigned longer opcodes.**
- **Also, multiple operand instructions could specify different addressing modes for each operand**
  - **For example,**
    - » **Operand 1 could be a directly addressed register,**
    - » **Operand 2 could be an indirectly addressed memory location,**
    - » **Operand 3 (the destination) could be an indirectly addressed register.**
- **All of this led to the need to have different length instructions in different situations, depending on the opcode and operands used**

## VARIABLE LENGTH INSTRUCTIONS

- **For example, an instruction that only specifies register operands may only be two bytes in length**
  - **One byte to specify the instruction and addressing mode**
  - **One byte to specify the source and destination registers.**

- **An instruction that specifies memory addresses for operands may need five bytes**
  - **One byte to specify the instruction and addressing mode**
  - **Two bytes to specify each memory address**
    - » **Maybe more if there's a large amount of memory.**

- **Variable length instructions greatly complicate the fetch and decode problem for a processor**

- **The circuitry to recognize the various instructions and to properly fetch the required number of bytes for operands is very complex**

## 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, looking much more like Mano's Basic Computer than 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**

---

## ARCHITECTURAL METRIC

$$A \leftarrow B + C$$
$$B \leftarrow A + C$$
$$D \leftarrow D - B$$

- **Register-to-register (Reuse of operands)**

| 8 | 4 | 16 | |
|------|------|------|------|
| Load | rB | B | |
| Load | rC | C | |
| Add | rA | rB | rC |
| Store | rA | A | |
| Add | rB | rA | rC |
| Store | rB | B | |
| Load | rD | D | |
| Sub | rD | rD | rB |
| Store | rD | D | |

I = 228b
D = 192b
M = 420b

- **Register-to-register (Compiler allocates operands in registers)**

| 8 | 4 | 4 | 4 |
|-----|-----|-----|-----|
| Add | rA | rB | rC |
| Add | rB | rA | rC |
| Sub | rD | rD | rB |

I = 60b
D = 0b
M = 60b

- **Memory-to-memory**

| 8 | 16 | 16 | 16 |
|-----|-----|-----|-----|
| Add | B | C | A |
| Add | A | C | B |
| Sub | B | D | D |

I = 168b
D = 288b
M = 456b

## CHARACTERISTICS  OF  INITIAL RISC MACHINES

|  | IBM 801 | RISC I | MIPS |
|---|---|---|---|
| Year | 1980 | 1982 | 1983 |
| Number of instructions | 120 | 39 | 55 |
| Control memory size | 0 | 0 | 0 |
| Instruction size (bits) | 32 | 32 | 32 |
| Technology | ECL MSI | NMOS VLSI | NMOS VLSI |
| Execution model | reg-reg | reg-reg | reg-reg |

## COMPARISON  OF  INSTRUCTION  SEQUENCE

A ← B + C
A ← A + 1
D ← D - B

## REGISTERS

- **By simplifying the instructions and addressing modes, there is space available on the chip or board of a RISC CPU for more circuits than with a CISC processor**
- **This extra capacity is used to**
  - Pipeline instruction execution to speed up instruction execution
  - Add a large number of registers to the CPU

## PIPELINING

- **A very important feature of many RISC processors is the ability to execute an instruction each clock cycle**
- **This may seem nonsensical, since it takes at least once clock cycle each to fetch, decode and execute an instruction.**
- **It is however possible, because of a technique known as pipelining**
  - We'll study this in detail later
- **Pipelining is the use of the processor to work on different phases of multiple instructions in parallel**

# PIPELINING

- **For instance, at one time, a pipelined processor may be**
  - **Executing instruction $i_t$**
  - **Decoding instruction $i_{t+1}$**
  - **Fetching instruction $i_{t+2}$ from memory**

- **So, if we're running three instructions at once, and it takes an average instruction three cycles to run, the CPU is executing an average of an instruction a clock cycle**

- **As we'll see when we cover it in depth, there are complications**
  - **For example, what happens to the pipeline when the processor branches**

- **However, pipelined execution is an integral part of all modern processors, and plays an important role**

---

# REGISTERS

- **By having a large number of general purpose registers, a processor can minimize the number of times it needs to access memory to load or store a value**
- **This results in a significant speed up, since memory accesses are *much* slower than register accesses**
- **Register accesses are fast, since they just use the bus on the CPU itself, and any transfer can be done in one clock cycle**
- **To go off-processor to memory requires using the much slower memory (or system) bus**
- **It may take many clock cycles to read or write to memory across the memory bus**
  - **The memory bus hardware is usually slower than the processor**
  - **There may even be competition for access to the memory bus by other devices in the computer (*e.g.* disk drives)**
- **So, for this reason alone, a RISC processor may have an advantage over a comparable CISC processor, since it only needs to access memory**
  - **for its instructions, and**
  - **occasionally to load or store a memory value**

---

# UTILIZING RISC REGISTERS – REGISTER WINDOW

**<Weighted Relative Dynamic Frequency of HLL Operations>**

|  | Dynamic Occurrence | | Machine-Instruction Weighted | | Memory Reference Weighted | |
|---|---|---|---|---|---|---|
|  | Pascal | C | Pascal | C | Pascal | C |
| **ASSIGN** | 45 | 38 | 13 | 13 | 14 | 15 |
| **LOOP** | 5 | 3 | 42 | 32 | 33 | 26 |
| **CALL** | 15 | 12 | 31 | 33 | *44* | *45* |
| **IF** | 29 | 43 | 11 | 21 | 7 | 13 |
| **GOTO** |  | 3 |  |  |  |  |
| **Other** | 6 | 1 | 3 | 1 | 2 | 1 |

⇒**The procedure (function) call/return is the most time-consuming operations in typical HLL programs**

---

# CALL-RETURN BEHAVIOR

**Call-return behavior as a function of nesting depth and time**
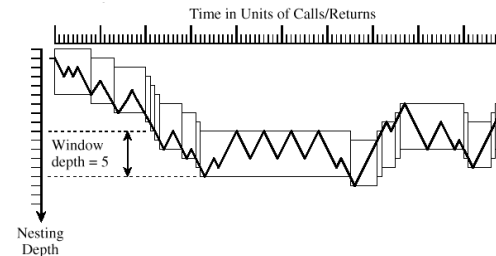
## REGISTER WINDOW APPROACH

- **Observations**
  - **Weighted Dynamic Frequency of HLL Operations**
    ⇒ **Procedure call/return is the most time consuming operations**

  - **Locality of Procedure Nesting**
    ⇒ **The depth of procedure activation fluctuates**
       **within a relatively narrow range**

  - **A typical procedure employs only a few passed**
        **parameters and local variables**
- **Solution**
  - **Use multiple small sets of registers (windows),**
        **each assigned to a different procedure**

  - **A procedure call automatically switches the CPU to use a different**
        **window of registers, rather than saving registers in memory**

  - **Windows for adjacent procedures are overlapped**
        **to allow parameter passing**

## CIRCULAR OVERLAPPED REGISTER WINDOWS
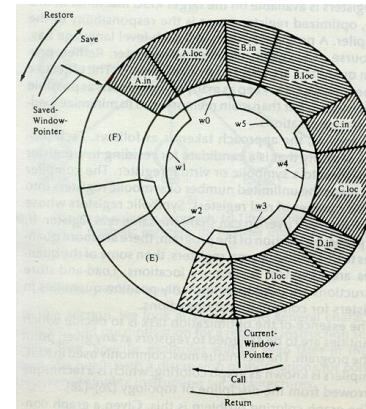
## OVERLAPPED REGISTER WINDOWS
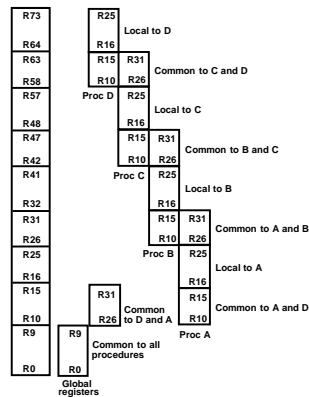
## OVERLAPPED REGISTER WINDOWS

- **There are three classes of registers:**
  - **Global Registers**
    - » **Available to all functions**
  - **Window local registers**
    - » **Variables local to the function**
  - **Window shared registers**
    - » **Permit data to be shared without actually needing to copy it**
- **Only one register window is active at a time**
  - **The active register window is indicated by a pointer**
- **When a function is called, a new register window is activated**
  - **This is done by incrementing the pointer**
- **When a function calls a new function, the high numbered registers of the calling function window are shared with the called function as the low numbered registers in its register window**
- **This way the caller's high and the called function's low registers overlap and can be used to pass parameters and results**
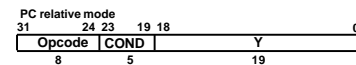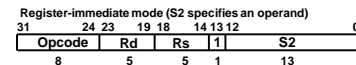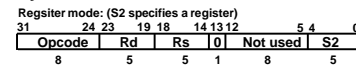
## OVERLAPPED REGISTER WINDOWS

- **In addition to the overlapped register windows, the processor has some number of registers, *G*, that are global registers**
  - **This is, all functions can access the global registers.**

- **The advantage of overlapped register windows is that the processor does not have to push registers on a stack to save values and to pass parameters when there is a function call**
  - **Conversely, pop the stack on a function return**
- **This saves**
  - **Accesses to memory to access the stack.**
  - **The cost of copying the register contents at all**
- **And, since function calls and returns are so common, this results in a significant savings relative to a stack-based approach**

---

## BERKELEY  RISC I

```
- 32-bit integrated circuit CPU
- 32-bit address, 8-, 16-, 32-bit data
- 32-bit instruction format
- total 31 instructions
- three addressing modes:
     register; immediate; PC relative addressing
- 138 registers
     10 global registers
     8 windows of 32 registers each
```

**Berkeley  RISC I  Instruction Formats**

**Regsiter mode: (S2 specifies a register)**

| 31 | 24 23 | 19 18 | 14 13 12 | 5 4 | 0 |
|----|----|----|----|----|----|
| Opcode | Rd | Rs | 0 | Not used | S2 |
| 8 | 5 | 5 | 1 | | 5 |

**Register-immediate mode (S2 specifies an operand)**

| 31 | 24 23 | 19 18 | 14 13 12 | 0 |
|----|----|----|----|----|
| Opcode | Rd | Rs | 1 | S2 |
| 8 | 5 | 5 | 1 | 13 |

**PC relative mode**

| 31 | 24 23 | 19 18 | 0 |
|----|----|----|----|
| Opcode | COND | | Y |
| 8 | 5 | | 19 |

---

## BERKELEY RISC I

- **Register 0 was hard-wired to a value of 0.**
- **There are eight memory access instructions**
  - **Five load-from-memory instructions**
  - **Three store-to-memory instructions.**

- **The load instructions:**
  - LDL          load long
  - LDSU        load short unsigned
  - LDSS        load short signed
  - LDBU        load byte unsigned
  - LDBS        load byte signed
  - **Where long is 32 bits, short is 16 bits and a byte is 8 bits**
- **The store instructions:**
  - STL          store long
  - STS          store short
  - STB          store byte

---

## Berkeley RISC I

| | | |
|---|---|---|
| LDL | $R_d \leftarrow M[(R_s) + S_2]$ | load long |
| LDSU | $R_d \leftarrow M[(R_s) + S_2]$ | load short unsigned |
| LDSS | $R_d \leftarrow M[(R_s) + S_2]$ | load short signed |
| LDBU | $R_d \leftarrow M[(R_s) + S_2]$ | load byte unsigned |
| LDBS | $R_d \leftarrow M[(R_s) + S_2]$ | load byte signed |
| STL | $M[(R_s) + S_2] \leftarrow R_d$ | store long |
| STS | $M[(R_s) + S_2] \leftarrow R_d$ | store short |
| STB | $M[(R_s) + S_2] \leftarrow R_d$ | store byte |

- **Here the difference between the lengths is**
  - **A long is simply loaded, since it is the same size as the register  (32 bits).**
  - **A short or a byte can be loaded into a register**
    - » **Unsigned - in which case the upper bits of the register are loaded with 0's.**
    - » **Signed - in which case the upper bits of the register are loaded with the sign bit of the short/byte loaded.**

## INSTRUCTION SET OF BERKELEY RISC I

| Opcode | Operands | Register Transfer | Description |
|--------|----------|-------------------|-------------|
| **Data manipulation instructions** | | | |
| ADD | Rs,S2,Rd | Rd ← Rs + S2 | Integer add |
| ADDC | Rs,S2,Rd | Rd ← Rs + S2 + carry | Add with carry |
| SUB | Rs,S2,Rd | Rd ← Rs - S2 | Integer subtract |
| SUBC | Rs,S2,Rd | Rd ← Rs - S2 - carry | Subtract with carry |
| SUBR | Rs,S2,Rd | Rd ← S2 - Rs | Subtract reverse |
| SUBCR | Rs,S2,Rd | Rd ← S2 - Rs - carry | Subtract with carry |
| AND | Rs,S2,Rd | Rd ← Rs ∧ S2 | AND |
| OR | Rs,S2,Rd | Rd ← Rs ∨ S2 | OR |
| XOR | Rs,S2,Rd | Rd ← Rs ⊕ S2 | Exclusive-OR |
| SLL | Rs,S2,Rd | Rd ← Rs shifted by S2 | Shift-left |
| SRL | Rs,S2,Rd | Rd ← Rs shifted by S2 | Shift-right logical |
| SRA | Rs,S2,Rd | Rd ← Rs shifted by S2 | Shift-right arithmetic |
| **Data transfer instructions** | | | |
| LDL | (Rs)S2,Rd | Rd ← M[Rs + S2] | Load long |
| LDSU | (Rs)S2,Rd | Rd ← M[Rs + S2] | Load short unsigned |
| LDSS | (Rs)S2,Rd | Rd ← M[Rs + S2] | Load short signed |
| LDBU | (Rs)S2,Rd | Rd ← M[Rs + S2] | Load byte unsigned |
| LDBS | (Rs)S2,Rd | Rd ← M[Rs + S2] | Load byte signed |
| LDHI | Rd,Y | Rd ← Y | Load immediate high |
| STL | Rd,(Rs)S2 | M[Rs + S2] ← Rd | Store long |

## INSTRUCTION SET OF BERKELEY RISC I

| Opcode | Operands | Register Transfer | Description |
|--------|----------|-------------------|-------------|
| **Program control instructions** | | | |
| JMP | COND,S2(Rs) | PC ← Rs + S2 | Conditional jump |
| JMPR | COND,Y | PC ← PC + Y | Jump relative |
| CALL | Rd,S2(Rs) | Rd ← PC, PC ← Rs + S2 | Call subroutine and |
| | | CWP ← CWP - 1 | change window |
| CALLR | Rd,Y | Rd ← PC, PC ← PC + Y | Call relative and |
| | | CWP ← CWP - 1 | change window |
| RET | Rd,S2 | PC ← Rd + S2 | Return and |
| | | CWP ← CWP + 1 | change window |
| CALLINT | Rd | Rd ← PC,CWP ← CWP - 1 | |
| | | Call an interrupt pr. | |
| RETINT | Rd,S2 | PC ← Rd + S2 | Return from |
| | | CWP ← CWP + 1 | interrupt pr. |
| GTLPC | Rd | Rd ← PC | Get last PC |

## 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**

## ADVANTAGES OF RISC

- **VLSI Realization**

  **Control area is considerably reduced**

  **Example:**
  **RISC I: 6%**
  **RISC II: 10%**
  **MC68020: 68%**
  **general CISCs: ~50%**

  ⇒ **RISC chips allow a large number of registers on the chip**

  - **Enhancement of performance and HLL support**
  - **Higher regularization factor and lower VLSI design cost**

  **The GaAs VLSI chip realization is possible**

- **Computing Speed**
  - **Simpler, smaller control unit ⇒ faster**
  - **Simpler instruction set; addressing modes; instruction format ⇒ faster decoding**
  - **Register operation ⇒ faster than memory operation**
  - **Register window ⇒ enhances the overall speed of execution**
  - **Identical instruction length, One cycle instruction execution ⇒ suitable for pipelining ⇒ faster**

# ADVANTAGES  OF  RISC

• **Design Costs and Reliability**

  **- Shorter time to design**
   **⇒ reduction in the overall design cost and**
      **reduces the problem that the end product will**
      **be obsolete by the time the design is completed**

  **- Simpler, smaller control unit**
   **⇒  higher reliability**

  **- Simple instruction format (of fixed length)**
   **⇒ ease of virtual memory management**

• **High Level Language Support**

  **- A single choice of instruction**
   **⇒ shorter, simpler compiler**

  **- A large number of CPU registers**
   **⇒ more efficient code**

  **- Register window**
   **⇒ Direct support of HLL**

  **- Reduced burden on compiler writer**