Unit 5
Central Processing Unit (CPU)

Introduction

Part of the computer that performs the bulk of data-processing operations is called the central processing unit (CPU). It consists of 3 major parts:

- **Register set**: stores intermediate data during execution of an instruction
- **ALU**: performs various microoperations required
- **Control unit**: supervises register transfers and instructs ALU

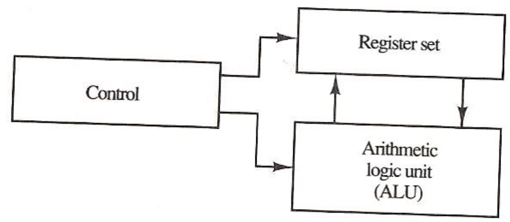
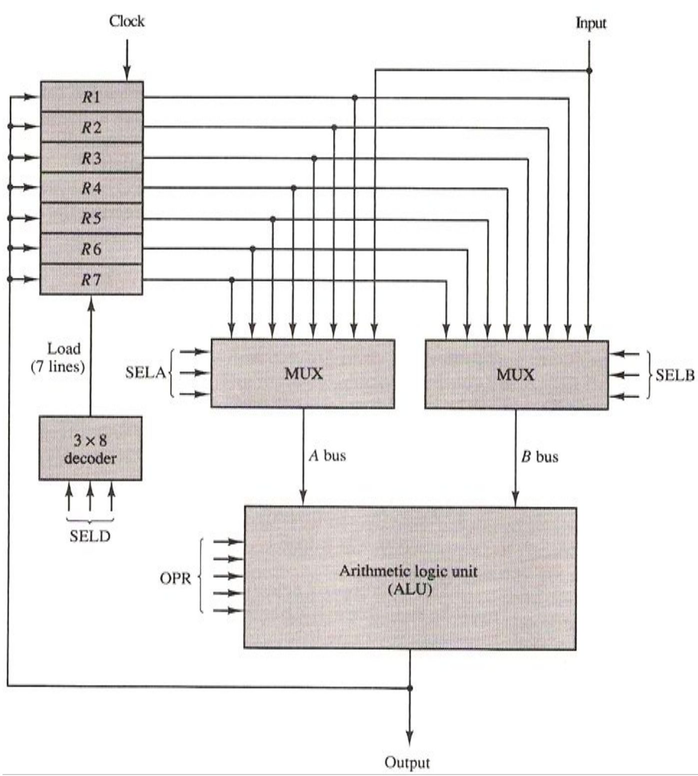


Fig: Major components of CPU

Here, we will proceed from programmer's point of view (as we know CA is the study of computer structure and behavior as seen by the programmer) which includes the instruction formats, addressing modes, instruction set and general organization of CPU registers.

General Register Organization

A bus organization of seven CPU registers is shown below:



(a) Block diagram (register organization)

Why we need CPU registers?

➔ During instruction execution, we could store pointers, counters, return addresses, temporary results and partial products in some locations in RAM, but having to refer memory locations for such applications is time consuming compared to instruction cycle. So for convenient and more efficient processing, we need processor registers (connected through common bus system) to store intermediate results.

All registers are connected to two multiplexers (MUX) that select the registers for bus A and bus B. Registers selected by multiplexers are sent to ALU. Another selector (OPR) connected to ALU selects the operation for the ALU. Output produced by ALU is stored in some register and this destination register for storing the result is activated by the destination decoder (SELD).

Example: R1 ← R2 + R3

- MUX selector (SELA): BUS A ← R2
- MUX selector (SELB): BUS B ← R3
- ALU operation selector (OPR): ALU to ADD
- Decoder destination selector (SELD): R1 ← Out Bus

Control word

Combination of all selection bits of a processing unit is called control word. Control Word for above CPU is as below:

3	3	3	5
SELA	SELB	SELD	OPR

The 14 bit control word when applied to the selection inputs specify a particular microoperation. Encoding of the register selection fields and ALU operations is given below:

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	$R7$	$R7$	$R7$

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

Example: R1 ← R2 - R3

This microoperation specifies R2 for A input of the ALU, R3 for the B input of the ALU, R1 for the destination register and ALU operation to subtract A-B. Binary control word for this microoperation statement is:

Field:	SELA	SELB	SELD	OPR
Symbol:	R2	R3	R1	SUB
Control word:	010	011	001	00101

Examples of different microoperations are shown below:

	Symbolic Designation				
Microoperation	SELA	SELB	SELD	OPR	Control Word
$R1 \leftarrow R2 - R3$	R2	R3	R1	SUB	010 011 001 00101
$R4 \leftarrow R4 \vee R5$	R4	R5	R4	OR	100 101 100 01010
$R6 \leftarrow R6 + 1$	R6	—	R6	INCA	110 000 110 00001
$R7 \leftarrow R1$	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
$R4 \leftarrow$ sh1 $R4$	R4	—	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

Stack Organization

This is useful *last-in, first-out* (LIFO) list (actually storage device) included in most CPU's. Stack in digital computers is essentially a memory unit with a stack pointer (SP). SP is simply an address register that points stack top. Two operations of a stack are the insertion (push) and deletion (pop) of items. In a computer stack, nothing is pushed or popped; these operations are simulated by incrementing or decrementing the SP register.

Register stack

It is the collection of finite number of registers. Stack pointer (SP) points to the register that is currently at the top of stack.

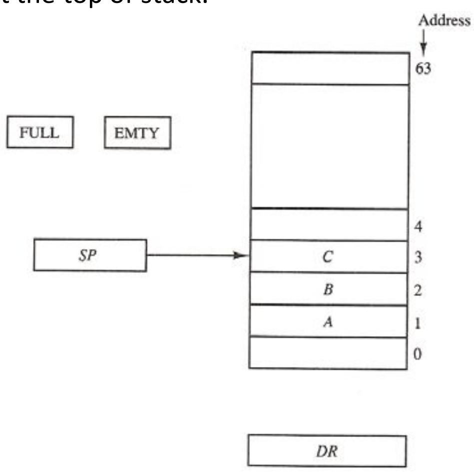


Diagram shows 64-word register stack. 6-bit address SP points stack top. Currently 3 items are placed in the stack: A, B and C do that content of SP is now 3 (actually 000011). 1-bit registers FULL and EMTY are set to 1 when the stack is full and empty respectively. DR is data register that holds the binary data to be written into or read out of the stack.

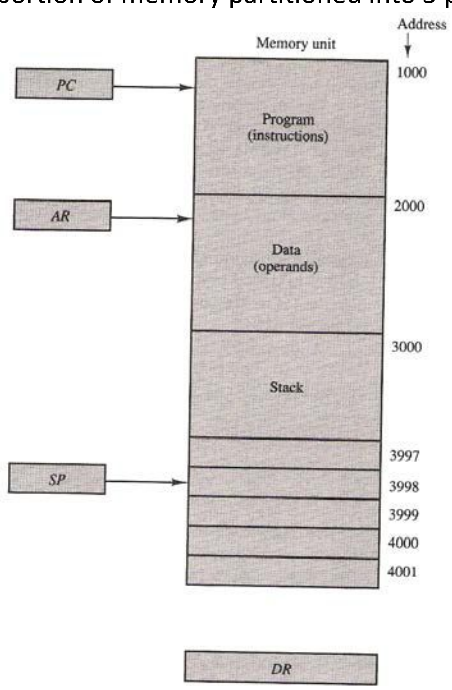
/* Initially, SP = 0, EMPTY = 1(true), FULL = 0(false) */

Push operation	**Pop operation**
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

Fig: Block diagram of a 64-word stack

Memory stack

A portion of memory can be used as a stack with a processor register as a SP. Figure below shows a portion of memory partitioned into 3 parts: program, data and stack.



PC: used during fetch phase to read an instruction.
AR: used during execute phase to read an operand.
SP: used to push or pop items into or from the stack.

Here, initial value of SP is 4001 and stack grows with *decreasing addresses*. First item is stored at 4000, second at 3999 and last address that can be used is 3000. No provisions are available for stack limit checks.

PUSH:	**POP:**
SP ← SP - 1	DR ← M[SP]
M[SP] ← DR	SP ← SP + 1

Processor Organization

In general, most processors are organized in one of 3 ways:

1. **Single register (Accumulator) organization**
   - Basic Computer is a good example
   - Accumulator is the only general purpose register
   - Uses implied accumulator register for all operations

Example:	
ADD X	// AC ← AC + M[X]
LDA Y	// AC ← M[Y]

2. **General register organization**
   - Used by most modern processors
   - Any of the registers can be used as the source or destination for computer operations.

Example:	
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]

3. **Stack organization**
   - All operations are done with the 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.

Example:	
PUSH X	// TOS ← M[X]
ADD	// TOS = TOP(S) + TOP(S)

Types of instruction

Instruction format of a computer instruction usually contains 3 fields: operation code field (opcode), address field and mode field. The number of address fields in the instruction format depends on the internal organization of CPU. On the basis of no. of address field we can categorize the instruction as below:

- **Three-Address Instructions**

  Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand.

  Assembly language 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**

  These instructions are most common in commercial computers.

  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
```

- Tries to minimize the size of instruction
- Size of program is relatively larger.

- **One-Address Instructions**

  One-address instruction uses an implied accumulator (AC) register for all data manipulation. All operations are done between AC and memory operand.

  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
```

- Memory access is only limited to load and store
- Large program size

- **Zero-Address Instructions**

  A stack-organized computer uses this type of instructions.

  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
```

The name “zero-address” is given to this type of computer because of the absence of an address field in the computational instructions.

Addressing Modes

I am repeating it again guys:"Operation field of an instruction specifies the operation that must be executed on some data stored in computer register or memory words". The way operands (data) are chosen during program execution depends on the addressing mode of the instruction. So, *addressing mode* specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.

We use variety of addressing modes to accommodate one or both of following provisions:

- To give programming versatility to the user (by providing facilities as: pointers to memory, counters for loop control, indexing of data and program relocation)
- To use the bits in the address field of the instruction efficiently

Types of addressing modes

- **Implied Mode**

  Address of the operands is specified implicitly in the definition of the instruction.
  - No need to specify address in the instruction
  - Examples from Basic Computer CLA, CME, INP

  ADD X;
  PUSH Y;
- **Immediate Mode**

  Instead of specifying the address of the operand, operand itself is specified in the instruction.
  - 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
- **Register Mode**

  Address specified in the instruction is the address of a register
  - Designated operand need to be in a register
  - Shorter address than the memory address
  - A k-bit address field can specify one of 2^k registers.
  - Faster to acquire an operand than the memory addressing
- **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 (effective address) = content of R.
- **Autoincrement or Autodecrement Mode**

  It is similar to register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory. When address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table.
- **Direct Addressing 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(address)

- **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)]

- **Relative Addressing Modes**

  The Address field of an instruction specifies the part of the address which can be used along with a designated register (e.g. PC) 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

3 different Relative Addressing Modes:

* ***PC Relative Addressing Mode***:
  - EA = PC + IR(address)
* ***Indexed Addressing Mode***
  - EA = IX + IR(address) { IX is index register }
* ***Base Register Addressing Mode***
  - EA = BAR + IR(address)

Numerical Example (Addressing modes)

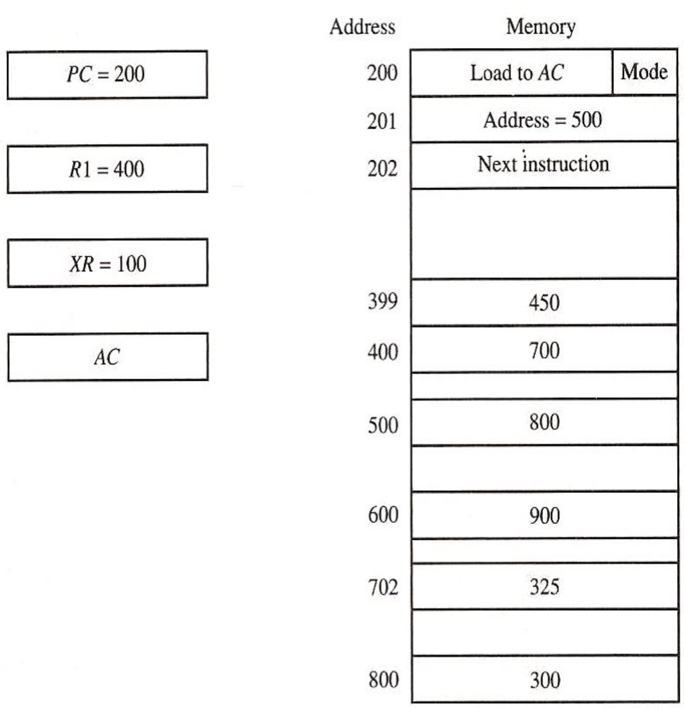


Fig: numerical example of addressing modes

→We have 2-word instruction "load to AC" occupying addresses 200 and 201. First word specifies an operation code and mode and second part specifies an address part (500 here).

→Mode field specify any one of a number of modes. For each possible mode we calculate effective address (EA) and operand that must be loaded into AC.

→**Direct addressing mode**: EA = address field 500 and AC contains 800 at that time.

→**Immediate mode**: Address part is taken as the operand itself. So AC = 500. (Obviously EA = 201 in this case)

→**Indirect mode**: EA is stored at memory address 500. So EA=800. And operand in AC is 300.

→**Relative mode:**

- PC relative: EA = PC + 500=702 and operand is 325. (since after fetch phase PC is incremented)
- Indexed addressing: EA=XR+500=600 and operand is 900.

→**Register mode**: Operand is in R1, AC = 400

→**Register indirect mode**: EA = 400, so AC=700

→**Autoincrement mode**: same as register indirect except R1 is incremented to 401 after execution of the instruction.

→ **Autodecrement mode:** decrements R1 to 399, so AC is now 450.

Following listing shows the vale of effective address and operand loaded into AC for 9 addressing modes.

Direct address	EA = 500 AC content = 800	// AC ← M[500]
Immediate operand	EA = 201 AC content = 500	// AC ← 500
Indirect address	EA = 500 AC content = 300	// AC ← M[M[500]]
Relative address	EA = 500 AC content = 325	// AC ← M[PC+500]
Indexed address	EA = 500 AC content = 900	// AC ← (IX+500)
Register	EA = 500 AC content = 400	// AC ← R1
Register indirect	EA = 400 AC content = 700	// AC ← M[R1]
Autoincrement	EA = 500 AC content = 700	// AC ← (R1)
Autodecrement	EA = 399 AC content = 450	//AC ← -(R)

Data Transfer and Manipulation

Computers give extensive set of instructions to give the user the flexibility to carryout various computational tasks. The actual operations in the instruction set are not very different from one computer to another although binary encodings and symbol name (operation) may vary. So, most computer instructions can be classified into 3 categories:

1. Data transfer instructions
2. Data manipulation instructions
3. Program control instructions

Data transfer Instructions

Data transfer instructions causes transfer of data from one location to another without modifying the binary information content. The most common transfers are:

- between memory and processor registers
- between processor registers and I/O
- between processor register themselves

Table below lists 8 data transfer instructions used in many computers.

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

Load: denotes transfer from memory to registers (usually AC)
Store: denotes transfer from a processor registers into memory
Move: denotes transfer between registers, between memory words or memory & registers.
Exchange: swaps information between two registers or register and a memory word.
Input & Output: transfer data among registers and I/O terminals.
Push & Pop: transfer data among registers and memory stack.

HEY!, different computer use different mnemonics for the same instruction name.

Instructions described above are often associated with the variety of addressing modes. Assembly language uses special character to designate the addressing mode. E.g. # sign placed before the operand to recognize the immediate mode. (Some other assembly languages modify the mnemonics symbol to denote various addressing modes, e.g. for load immediate: LDI). Example: consider *load to accumulator* instruction when used with 8 different addressing modes:

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

Table: Recommended assembly language conventions for load instruction in different addressing modes

Data manipulation Instructions

Data manipulation instructions provide computational capabilities for the computer. These are divided into 3 parts:

4. Arithmetic instructions
5. Logical and bit manipulation instructions
6. Shift instructions

These instructions are similar to the microoperations in unit3. But actually; each instruction when executed must go through the *fetch phase* to read its binary code value from memory. The operands must also be brought into registers according to the rules of different addressing mode. And the last step of executing instruction is implemented by means of microoperations listed in unit 3.

Arithmetic instructions

Typical arithmetic instructions are listed below:

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

- Increment (decrement) instr. adds 1 to (subtracts 1 from) the register or memory word value.
- Add, subtract, multiply and divide instructions may operate on different data types (fixed-point or floating-point, binary or decimal).

Logical and bit manipulation instructions

Logical instructions perform binary operations on strings of bits stored in registers and are useful for manipulating individual or group of bits representing binary coded information. Logical instructions each bit of the operand separately and treat it as a Boolean variable.

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

- Clear instr. causes specified operand to be replaced by 0's.
- Complement instr. produces the 1's complement.
- AND, OR and XOR instructions produce the corresponding logical operations on individual bits of the operands.

Shift instructions

Instructions to shift the content of an operand are quite useful and are often provided in several variations (bit shifted at the end of word determine the variation of shift). Shift instructions may specify 3 different shifts:

- Logical shifts
- Arithmetic shifts
- Rotate-type operations

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 through carry	RORC
Rotate left through carry	ROLC

- Table lists 4 types of shift instructions.
- Logical shift inserts 0 at the end position
- Arithmetic shift left inserts 0 at the end (identical to logical left shift) and arithmetic shift right leave the sign bit unchanged (should preserve the sign).
- Rotate instructions produce a circular shift.
- Rotate left through carry instruction transfers carry bit to right and so is for rotate shift right.

Program control instructions

Instructions are always stored in successive memory locations and are executed accordingly. But sometimes it is necessary to condition the data processing instructions which change the PC value accidently causing a break in the instruction execution and branching to different program segments.

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

- Branch (usually one address instruction) and jump instructions can be changed interchangeably.
- Skip is zero address instruction and may be conditional & unconditional.
- Call and return instructions are used in conjunction with subroutine calls.

RISC and CISC

An important aspect of computer architecture is the design of the instruction set for the processor. Early computers had small and simple instruction sets, forced mainly by the need to minimize the hardware used to implement them. As digital hardware became cheaper with the advent of ICs, computer instructions tended to increase both in number and complexity. Many computers have instruction sets that include 100-200 instructions employing variety of data types and large number of addressing modes and are classified as **Complex Instruction Set Computer** (**CISC**). In early 1980s, a number of computer designers recommended that computers use fewer instructions with simple constructs so as to execute them faster with in CPU without using memory as often. This type of computer is classified as a **Reduced Instruction Set Computer** (**RISC**).

CISC

One reason to provide a complex instruction set is the desire to simplify the compilation (done by compilers to convert high level constructs to machine instructions) and improve the overall computer performance.
Essential goal: Provide a single machine instruction for each statement in high level language.
Examples: Digital Equipment Corporation VAX computer and IBM 370 computer.

Characteristics:

1. A large no of instructions - typically from 100 to 250 instructions.
2. A large variety of addressing modes – typically form 5 to 20.
3. Variable-length instruction formats
4. Instructions that manipulate operands in memory

RISC

Main Concept: Attempt to reduce execution time by simplifying the instruction set of the computer.

Characteristics:

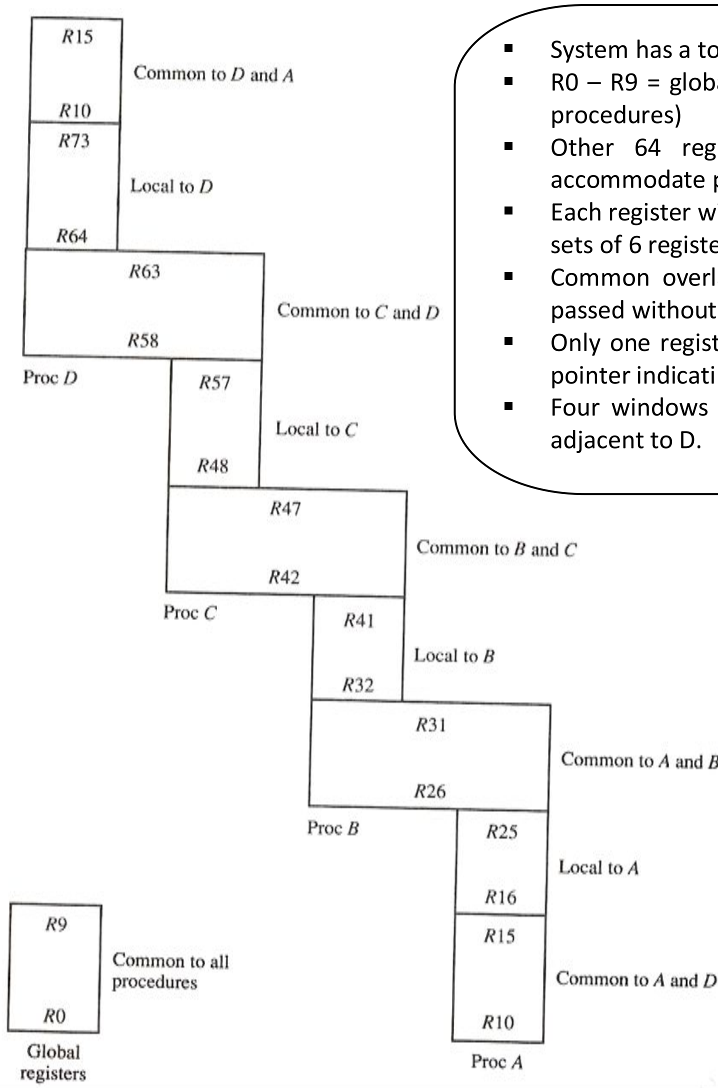
1. Relatively few instructions and addressing modes.
2. Memory access limited to load and store instructions
3. All operations done with in CPU registers (relatively large no of registers)
4. Fixed-length, easily decoded instruction format

5. Single cycle instruction execution
6. Hardwired rather than Microprogrammed control
7. Use of overlapped-register windows to speed procedure call and return
8. Efficient instruction pipeline

Overlapped Resister Windows

Procedure call and return occurs quite often in high-level programming languages. When translated into machine language, procedure call produces a sequence of instructions that **save register values**, **pass parameters** needed for the procedure and then **calls a subroutine** to execute the body of the procedure. After a procedure return, the program restores the old register values, passes results to the calling program and returns from the subroutine. Saving & restoring registers and passing of parameters & results involve time consuming operations.

A characteristic of some RISC processors is use of overlapped register windows to provide the passing of parameters and avoid need for saving & restoring register values. The concept of overlapped register windows is illustrated below:



- System has a total of 74 registers (Just an example)
- R0 – R9 = global registers (hold parameters shared by all procedures)
- Other 64 registers are divided into 4 windows to accommodate procedures A, B, C and D.
- Each register window consists of 10 local registers and two sets of 6 registers common to adjacent windows.
- Common overlapped registers permit parameters to be passed without the actual movement of data
- Only one register window is activated at any time with a pointer indicating the active window.
- Four windows have a circular organization with A being adjacent to D.

Example: Procedure A calls B

- Registers R26 to R31 are common to both procedures and therefore procedure A stores the parameters for procedure B in these registers.
- B uses local registers R32 through R41 for local variable storage.
- When B is ready to return at the end of its computation, programs stores results in registers R26-R31 and transfers back to the register window of procedure A.

Fig: Overlapped Resister Windows

In general, the organization of register windows will have following relationships:

- Number of global registers = G
- Number of local register in each window = L
- Number of registers common to windows = C
- Number of windows = W

Now,

- ➢ Window size = L + 2C +G
- ➢ Register file = (L+C)W + G (total number of register needed in the processor)

Example: In above fig, G = 10, L = 10, C = 6 and W = 4. Thus window size = 10+12+10 = 32 registers and register file consists of (10+6)*4+10 = 74 registers.

Exercises: textbook chapter 8 → 8.12 (do it yourself)