# Central Processing Unit (CPU)

- CPU performs the data processing operations in a computer.
- Three major parts (Fig. 8-1)

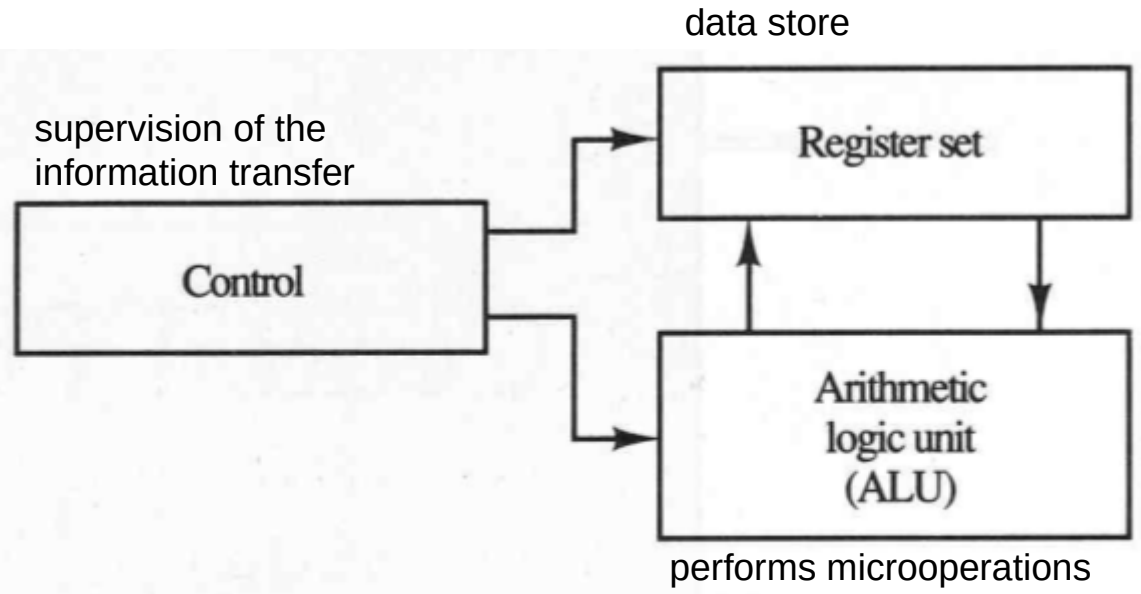
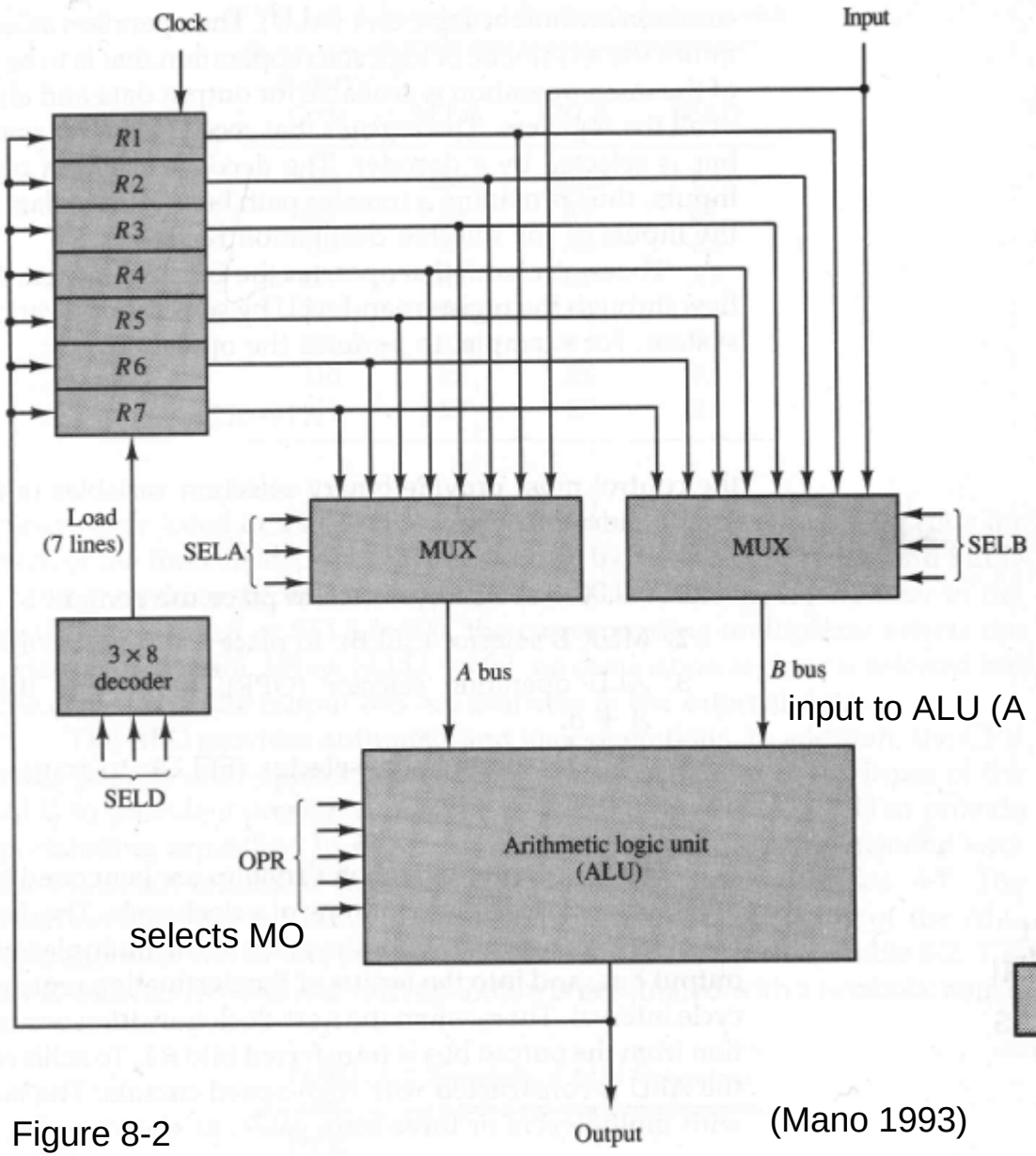



**Figure 8-1** Major components of CPU.

- Instruction set provides the specification for the design of the CPU.
- The design of CPU involves choosing the hardware for implementing the machine instructions.
- A programmer using assembly language  has to be aware of the register set, the memory structure, the type of data supported by the instructions, and the function of each instruction.

# General Register Organization

- Storing pointers, counters, return addresses, temporary results, and partial products into the memory is not efficient: referring to memory locations is time consuming: it is more efficient to use registers.
- If CPU has a large number of registers, a common bus is used to connect the registers.
- Arithmetic logic unit (ALU) is used to perform various microoperations.
- A bus organization of 7 CPU registers is shown in Fig. 8-2.




input to ALU (A bus and B bus)

selects MO

control word

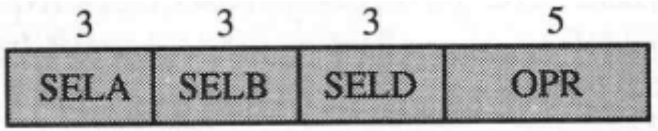



Figure 8-2

(Mano 1993)

- Control unit operates the CPU bus system. For example:

$$R1 \leftarrow R2 + R3$$

1. SELA is used to place R2 into bus A.
2. SELB is used to place R3 into bus B.
3. OPR selects the arithmetic addition.
4. SELD is used to transfer the result into R1.

- The buses are implemented with multiplexers or 3-state gates.
- The state of 14 binary selection inputs specifies a control word.
- The 14-bit control word (when applied to the selection inputs specify a microoperation).
- The encoding of register selections in specified in Table 8-1.

TABLE 8-1 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 |

Input = external input

if SELD = 000 then the content of the output bus is available in the external output

- ALU provides arithmetic and logic operations.
- Encoding of ALU operations (function table for this ALU is listed in Table 4-8):

**TABLE 8-2** 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 microoperations and corresponding control words:

**TABLE 8-3** Examples of Microoperations for the CPU

| Microoperation | Symbolic Designation | | | | Control Word |
|---|---|---|---|---|---|
| | SELA | SELB | SELD | OPR | |
| $R1 \leftarrow R2 - R3$ | 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 $\leftarrow R2$ | R2 | — | None | TSFA | 010 000 000 00000 |
| Output $\leftarrow$ 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 |

000 (not used)

XOR(x,x) = 0

- The most efficient way to generate control words with a large number of bits is to store them in a memory unit, which is referred to as control memory (control store).
- By reading consecutive control words from memory, it is possible to initiate the desired sequence of microoperations for the CPU => microprogrammed control.

# Stack Organization

- A stack is a storage device for storing information in such a manner that the item stored last is the first item retrieved (LIFO – last-in, first-out).
- The stack is a memory unit with an address register called a stack pointer (SP), which always points at the top item in the stack.
- The two operations of a stack are the insertion (push) and deletion (pull) of items.
- push-operation increments the SP and pull-operation decrements the SP.

- Stack can reside in a portion of a large memory unit or it can be organized as a collection of a finite number of (fast) registers.
- Fig. 8-3: organization of 64-word register stack.

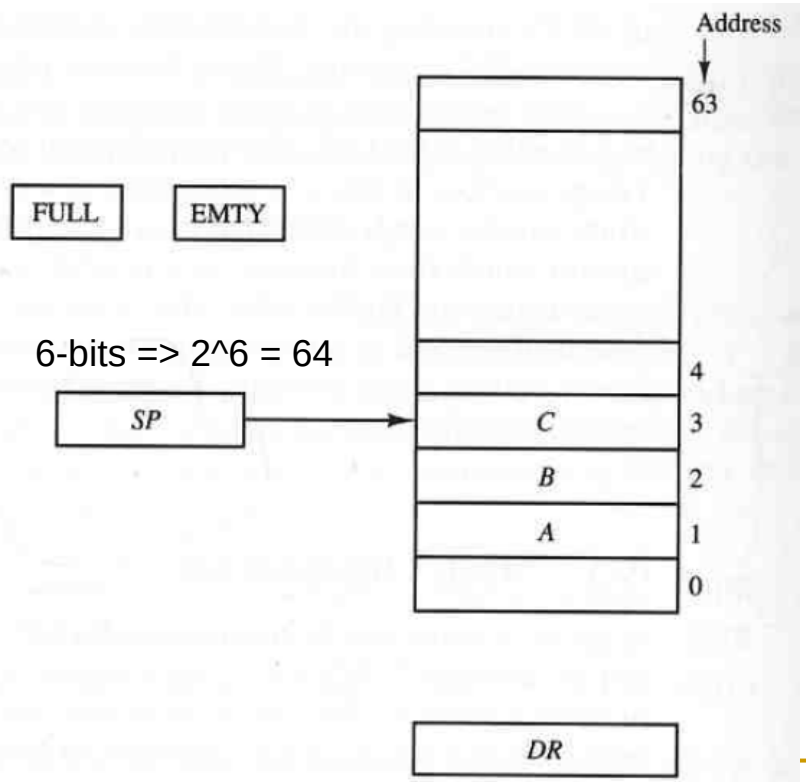



Figure 8-3 Block diagram of a 64-word stack.

push A
push B
push C
=> SP = 3

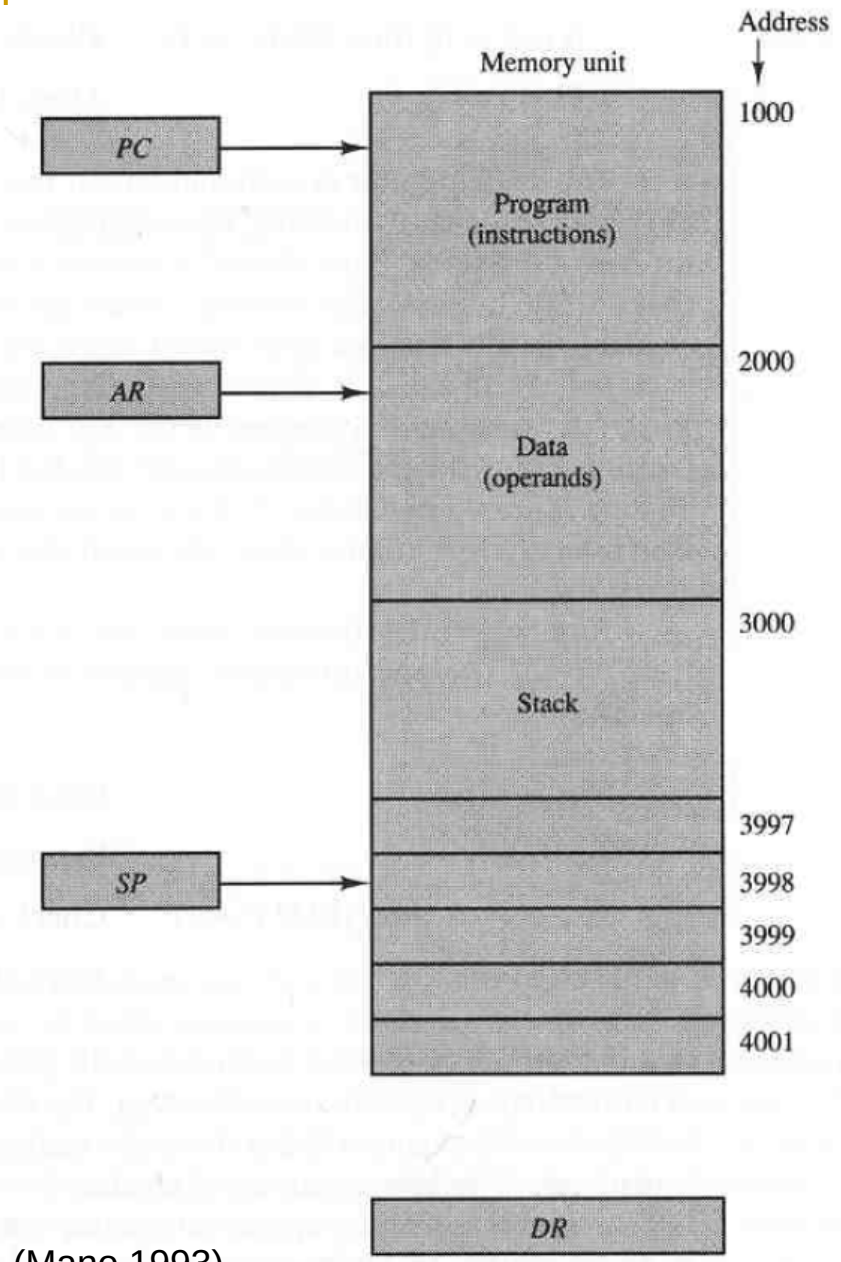
push (performed if stack is not full *i.e.* if FULL = 0):

| | |
|---|---|
| $SP \leftarrow SP + 1$ | Increment stack pointer |
| $M[SP] \leftarrow DR$ | Write item on top of the stack |
| If $(SP = 0)$ then $(FULL \leftarrow 1)$ | Check if stack is full |
| $EMTY \leftarrow 0$ | Mark the stack not empty |

pull (performed if stack is not empty *i.e.* if EMTY = 0):

| | |
|---|---|
| $DR \leftarrow M[SP]$ | Read item from the top of stack |
| $SP \leftarrow SP - 1$ | Decrement stack pointer |
| If $(SP = 0)$ then $(EMTY \leftarrow 1)$ | Check if stack is empty |
| $FULL \leftarrow 0$ | Mark the stack not full |

- Stack can also be implemented with a RAM attached to a CPU:
  - a portion of memory is assigned to a stack operation
  - a processor register is used as a stack pointer
- Fig. 8-4 shows how a portion of memory partitioned into three segments: program, data, and stack.
- Most computer do not provide hardware for checking stack overflow or underflow
  - if registers are used to store the upper limit (e.g. 3000) and the lower limit (e.g. 4000), then after push SP can be compared against the upper limit register, and after pull against the lower limit register.
- The advantage of the memory stack is that CPU can refer it without having to specify an address: the address in always in SP and automatically updated during a push or pop instruction.




(Mano 1993)

**Figure 8-4** Computer memory with program, data, and stack segments.

PC, AR, and SP provide addresses for the memory.

- PC is used in fetch phase to read instruction from the memory
- AR is used to read an operand during the execution phase of an instruction.
- SP is used to push or pop items into or from the stack.

- In this example, stack grows with decreasing addresses.
- First item stored is at address 4000.
- the last address that can be used is 3000.

push:

$$SP \leftarrow SP - 1$$

$$M[SP] \leftarrow DR$$

pull:

$$DR \leftarrow M[SP]$$

$$SP \leftarrow SP + 1$$

- A stack is effective for evaluating arithmetic expressions.
- Arithmetic operations are usually written in infix notation: each operator resides between the operands, *e.g.*:
  (A * B) + (C * D), where x denotes multiplication:
  - A * B and C * D has to be computed and stored.
  - after the two products, sum (A * B) + (C * D) is computed

  => there is no straight forward way to determine the next operation that is performed.

- Arithmetic expressions can be presented in prefix notation (also referred to as Polish notation by Polish mathematician Lukasiewicz): operators are placed before the operands.
- The postfix notation (reverse Polish notation (RPN) ) places the operator after the operand.
- *E.g.*:

A + B , infix notation
+AB , prefix notation
AB+ , postfix notation (RPN)

- The reverse Polish notation suits of stack manipulation.
- E.g. the expression

  A * B + C * D

  is written in RPN as

  AB*CD*+

  and is evaluated by scanning from left to right: when operator is found, the operation is performed by using operands on the left side of the operator. The operator and operands are replaced by the result of operation. The scan is continued and the procedure is repeated for every operator:

1. * is found
2. take the two operands from left: A and B
3. compute P = A * B
4. replace operands and operator with the result => PCD*+
5. continue scan
6. * is found
7. take the two operands from left: C and D
8. compute Q = C * D
9. replace operands and operator with the result => PQ+
10. continue scan
11. + is found
12. take the two operands from left: P and Q
13. compute R = P + Q
14. replace operands and operator with the result: R
15. continue scan: no more operators => stop; R is the result of evaluation.

- The conversion from infix to RPN must take into consideration the operational hierarchy of infix notation:
  1. first perform arithmetic inside inner parentheses
  2. ..then inside outer parentheses
  3. perform multiplication and division before addition and subtraction.
- E .g.: (A + B)*[C*(D + E) + F] becomes AB+DE+C*F+* which is computed:
  1. P = A+B => PDE+C*F+*
  2. Q = D+E => PQC*F+*
  3. R = Q * C => PRF+*
  4. S = R+F => PS*
  5. T = P*S
  - T represents the result: T = AB+DE+C*F+*

- RPN is the most efficient method known for evaluating arithmetic expressions.
- Used *e.g.* in electronic calculators
  - RPN is an useful way to represent arithmetic expression for a generic arithmetic evaluator.
- Stack is useful for evaluating arithmetic expressions in RPN
  - operands are pushed into the stack in the order of appearance (in RPN)
  - the topmost operands are popped from the stack and used for the operation
  - The result is pushed to replace the popped operands
- Most compilers convert all arithmetic expressions into Polish notation: efficient translation of arithmetic expressions into machine language instructions.

- E.g.: (3*4)+(5*6) => 34*56*+

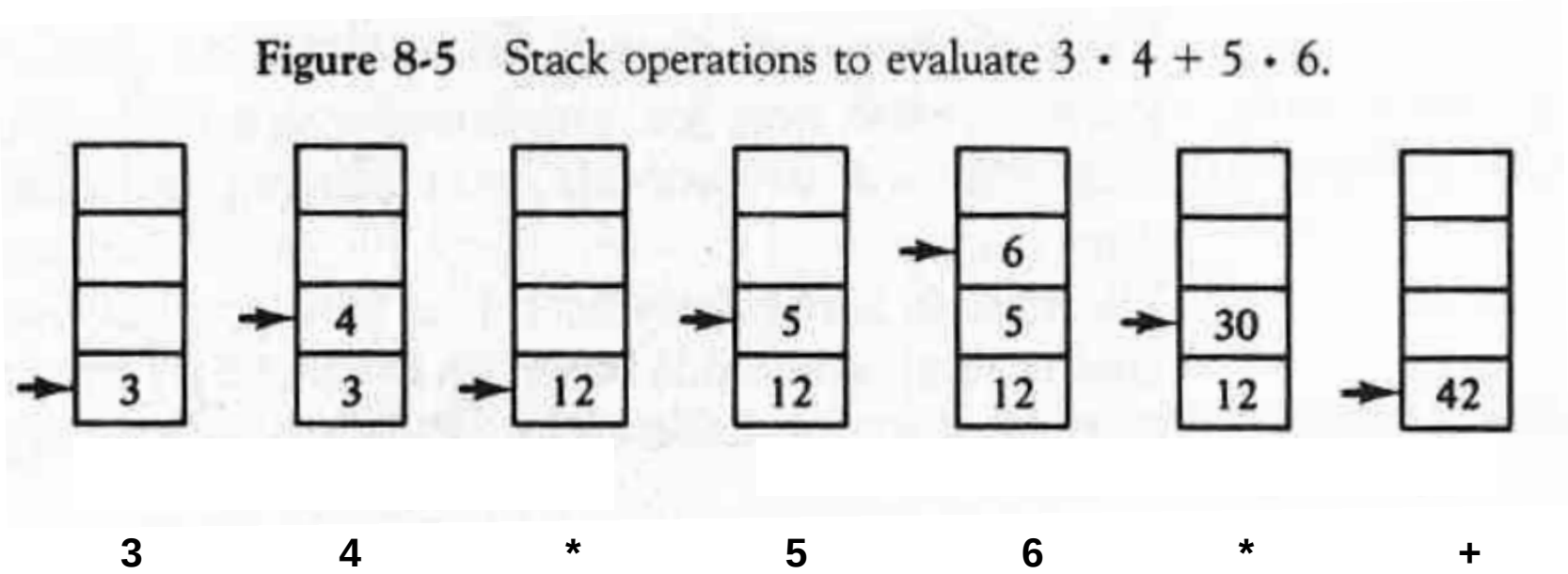


Figure 8-5 Stack operations to evaluate 3 • 4 + 5 • 6.
3
4
3
12
5
12
6
5
12
30
12
42
3
4
*
5
6
*
+

# Instruction Formats

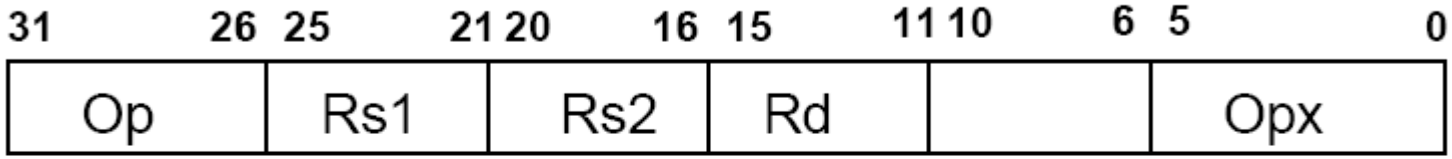
- The typical fields found in instruction formats are:
  1. Operation code specifying the operation: add, subtract, complement, etc.
  2. Address field designating a memory address or a register
  3. Mode field for specifying the way for determining the effective address of an operand.
- The number of address fields in the instruction format of a computer depends on the internal organization of its registers.

- E.g.: MIPS (a RISC microprocessor architecture developed by MIPS Computer Systems Inc.)

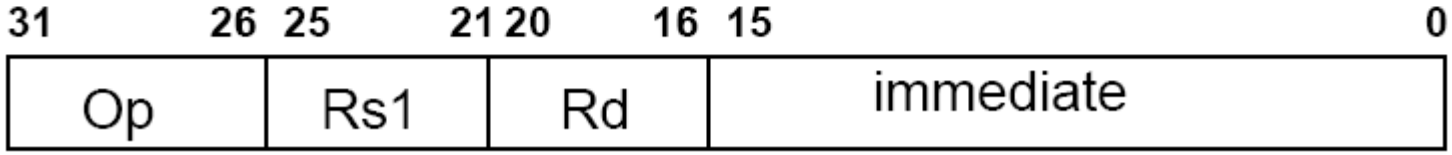
rs = source register

rd = destination register

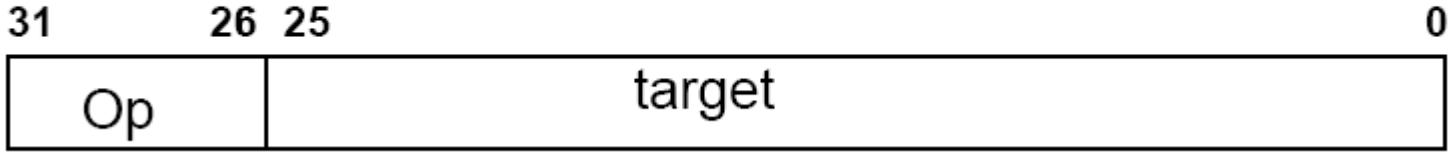
**Register-Register**




**Register-Immediate**




**Jump / Call**



- Computers may have instructions of several different lengths containing varying number of addresses.
- E.g. three-address instructions (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
```

- E.g. two-address instructions:

```
MOV   R1, A     R1←M[A]
ADD   R1, B     R1←R1 + M[B]
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
```

- E.g. RISC instructions:

```
LOAD    R1, A          R1←M[A]
LOAD    R2, B          R2←M[B]
LOAD    R3, C          R3←M[C]
LOAD    R4, D          R4←M[D]
ADD     R1, R1, R2     R1←R1 + R2
ADD     R3, R3, R4     R3←R3 + R4
MUL     R1, R1, R3     R1←R1*R3
STORE   X, R1          M[X]←R1
```

# Addressing Modes

- The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.
- Addressing modes are used:
  1. To provide programming versatility for the user: pointers to memory, counters for loop control, indexing data, etc.
  2. To reduce the number of bits in the addressing field of the instruction.
- The decoding phase of an instruction cycle determines the addressing mode(s) and the locations (registers and/or memory locations) of operands.
- Depending on the CPU, an instruction can have more than one address field, and each address field may be associated with its own particular addressing mode.

- Different addressing modes:
  - Implied mode: the operands are implicitly specified by the instruction, e.g.: “complement accumulator”.
  - Immediate mode: the operand is specified in the instruction (in operand field). Can be used e.g. to initialize register to constant value (=immediate operand).
  - Register mode: operands are in registers that reside within the CPU. The particular register is selected with the register field of the instruction.
  - Register indirect mode: the content of a register specifies the address of the operand in memory.
  - Autoincrement or autodecrement mode: similar to register indirect mode but the content of the register is automatically incremented/decremented after/prior data access.

- Direct address mode: the effective address is equal to the address part of the instruction. The operand resides in this address.
- Indirect address mode: the address field gives the address where the effective address is stored in memory.
- Relative address mode: the content of the program counter (PC) is added to the address part of the instruction in order to obtain the effective address *i.e.* the effective address is relative to the address of the next instruction. This address mode can be used in branch-type instructions when the branch address is in the area surrounding the instruction word.

- Indexed addressing mode: content of an index register (special CPU register) is added to the address part of the instruction to obtain the effective address. The index register can be incremented for accessing consecutive operands.
- Base register addressing mode: the content of a base register is added to the address part of the instruction to obtain the effective address. The address part of the instruction gives the displacement relative to the base address.

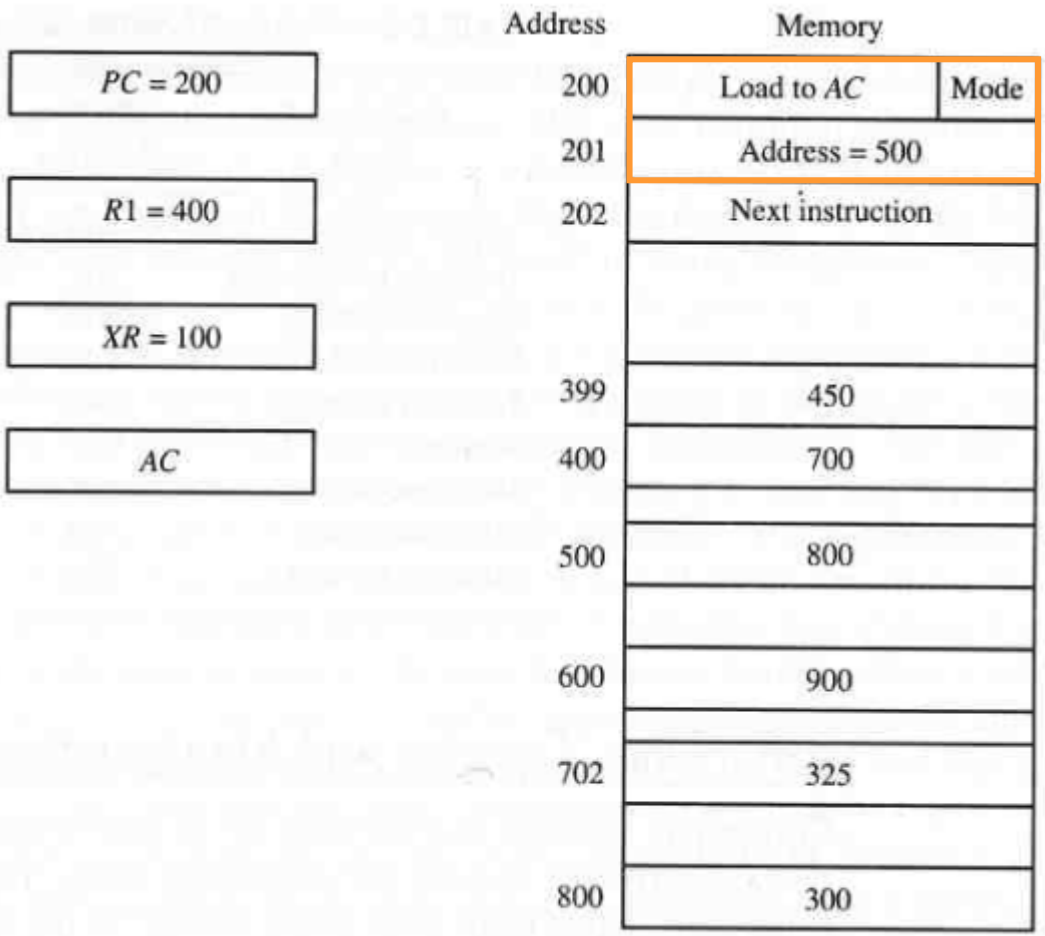



Figure 8-7 Numerical example for addressing modes.

(Mano 1993)

instruction with an address and addressing mode

decrements R1 prior the execution

**TABLE 8-4** Tabular List of Numerical Example

| Addressing Mode | Effective Address | Content of *AC* |
|---|---|---|
| Direct address | 500 | 800 |
| Immediate operand | 201 | 500 |
| Indirect address | 800 | 300 |
| Relative address | 702 | 325 |
| Indexed address | 600 | 900 |
| Register | — | 400 |
| Register indirect | 400 | 700 |
| Autoincrement | 400 | 700 |
| Autodecrement | 399 | 450 |

80x86 example

```
; Network Assembler example
;; $RCSfile: example2.asm,v $ $Revision: 1.1 $ $Date: 2001/09/14 09:54:06 $
BITS 16

segment stack stack
        resb 100h

segment data data

TimePrompt              DB 'Is it after 12 noon (Y/N)?$'
GoodMorningMessage      DB  13,10,'Good morning, world!',13,10,'$'
GoodAfternoonMessage    DB  13,10,'Good afternoon, world!',13,10,'$'
DefaultMessage          DB  13,10,'Good day, world!',10,13,'$'

segment code code

..start:
  mov     ax,data
  mov     ds,ax                   ;set DS to point to the data segment
  mov     dx,TimePrompt           ;point to the time prompt
  mov     ah,9                    ;DOS: print string
  int     21h                     ;display the time prompt
  mov     ah,1                    ;DOS: get character
  int     21h                     ;get a single-character response
  or      al,20h                  ;force character to lower case

  cmp     al,'y'                  ;typed Y for afternoon?
  je      IsAfternoon
  cmp     al,'n'                  ;typed N for morning?
  je      IsMorning

  mov     dx,DefaultMessage               ;default greeting
  jmp     DisplayGreeting

IsAfternoon:
  mov     dx,GoodAfternoonMessage         ;afternoon greeting
  jmp     DisplayGreeting

IsMorning:
  mov     dx,GoodMorningMessage           ;before noon greeting

DisplayGreeting:
  mov     ah,9                    ;DOS: print string
  int     21h                     ;display the appropriate greeting
  mov     ah,4ch                  ;DOS: terminate program
  mov     al,0                    ;return code will be 0
  int     21h                     ;terminate the program

.end
```

base register

immediate

relative to base register

relative address

**compile:**
**> nasm -f obj example2.asm**
**> val example2**

# Program Control

- Program flow can be altered by instructions that modify the value of the program counter: important feature of a digital computer – provides a control over the program flow and capability for branching to different program segments (blocks of memory).
- Typical program control instructions:

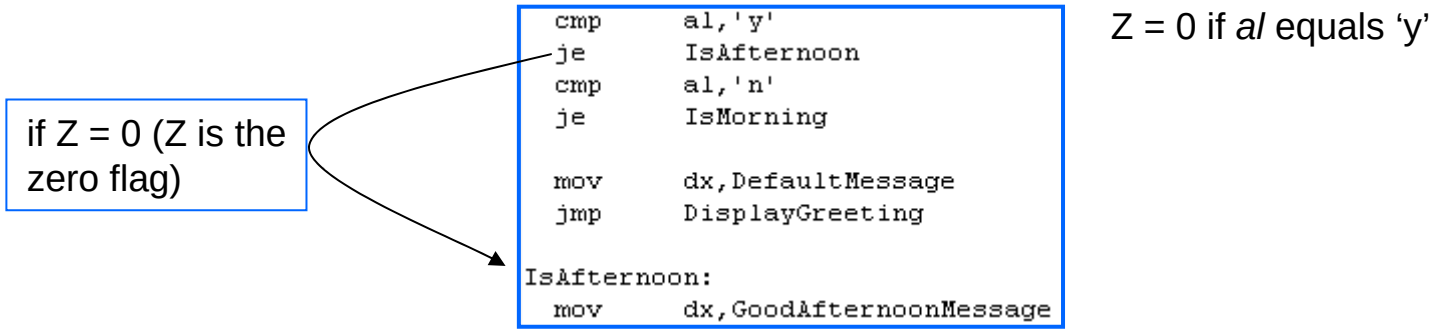
TABLE 8-10 Typical Program Control Instructions

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

- Branch and jump instructions may be conditional or unconditional
  - An unconditional branch instruction causes a branch to the specific address without any conditions, *e.g.*:

```
jmp     DisplayGreeting
```

  - The conditional branch specifies a condition, e.g. branch if zero: only when the condition is met, the program counter is loaded with the branch address, *e.g.*:




Z = 0 if *al* equals ‘y’

- Compare and test instructions can be used in setting conditions for subsequent conditional branch instructions
  - Compare performs an arithmetic subtraction: result is not saved – only status bit conditions are set as a result of operation.
  - Similarly test performs logical AND of two operands and updates certain status bits.

- The status register stores the values of the status bits (status register is composed of the status bits).
- Bits of the status register are modified as a result of an operation performed in the ALU.
- *E.g*. (8-bit ALU with a 4-bit status register):

C (carry) is set to 1 if end carry C8 is 1. C is cleared to 0 if C8 is 0.

S (sign) is set to 1 the highest-order bit F7 is 1. S is set to 0 if F7 is 0.

Z (zero) is set to 1 if the output of ALU is 0 (all 0's). Z is set to 0 otherwise.

V (overflow) is set to 1 is XOR of the last two carries is equal to 1, and cleared to 0 otherwise.

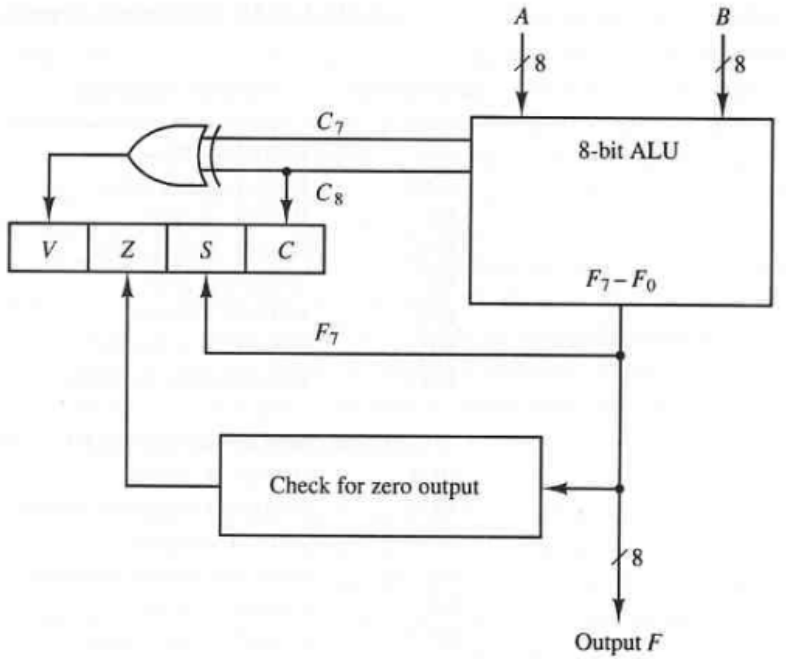



**Figure 8-8** Status register bits.

(Mano 1993)

- Status bits can be checked after ALU operation to determine certain relationships that exist between the values of A and B
  - V indicates overflow i.e. for 8-bit ALU the result is greater than 127 or less than -127.
  - If Z is set, the result is zero:
    - we can use e.g. XOR operation to compare two numbers (the result is zero iff *A* = *B*) and *Z* indicates the result of comparison.
    - A single bit in *A* can be checked with a mask that contains 1 in that particular bit position (others being 0's) and by using AND operation.

- Conditional branch instructions use the status bits for checking conditions for branching:

**TABLE 8-11** 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 | $A \neq B$ |

80x86

Z = 0 if *al* equals 'y'

```
  cmp     al,'y'
  je      IsAfternoon
  cmp     al,'n'
  je      IsMorning

  mov     dx,DefaultMessage
  jmp     DisplayGreeting

IsAfternoon:
  mov     dx,GoodAfternoonMessage
```

- For subroutine calls, different computers can use a different temporary location for storing the return address
  - some computers use the first memory location of the subroutine (like the Basic Computer).
  - some store the return address in a fixed memory location.
  - some computers use a processor register.
  - stack memory is yet another possibility (the most efficient way): when a succession of subroutines is called (nested calls), the sequential return addresses can be pushed into the stack. The “return from subroutine” instruction pops the return address (and assigns to program counter) from the top of the stack: we always have the return address for the last called subroutine.

- Subroutine call (stack based) microoperations:

| | |
|---|---|
| $SP \leftarrow SP - 1$ | Decrement stack pointer |
| $M[SP] \leftarrow PC$ | Push content of $PC$ onto the stack |
| $PC \leftarrow$ effective address | Transfer control to the subroutine |

- .. and return:

| | |
|---|---|
| $PC \leftarrow M[SP]$ | Pop stack and transfer to $PC$ |
| $SP \leftarrow SP + 1$ | Increment stack pointer |

- By using subroutine stack each return address (in nested calls) can be pushed into the stack without destroying any previous values
  - *e.g*. in basic computer a recursive subroutine call would destroy the previous return address stored in the first memory location of the subroutine.

- Program interrupt refers to the transfer of program control from a currently running program to another service program as a result of an external or internal generated request
  - otherwise similar to subroutine call, except:
    1. the interrupt is (usually) initiated by an internal of external signal rather than an execution of an instruction (software interrupts are exceptions).
    2. the address of the interrupt service program (routine) is determined by hardware rather than the address field of an instruction: the CPU must possess some form of HW procedure for selecting a branch address servicing the interrupt.
    3. Interrupt routine stores all the information (not just PC) necessary to recover the state of the CPU prior the return from the interrupt routine.

- After the interrupt routine the CPU must return exactly the same state that is was when the interrupt occurred.
- The state of the CPU at the end of the execute cycle (the interrupt is recognized in this phase) is determined from:
  1. The content of PC
  2. The content of all processor registers
  3. The content of status conditions
     - status bits (program status word PSW) stored in a separate status register.
     - contains status information about the state of the CPU: bits from ALU operation, interrupt enable bits, and CPU operation mode (system mode, user mode), for example.

- Some computer store only program counter (and PSW) prior entering to an interrupt routine
  - the interrupt routine must take care of storing and restoring the CPU status.
- CPU does not respond to an interrupt until the end of an instruction execution
  - in an interrupt is pending control goes to a interrupt cycle.
  - contents of PC and PSW are pushed onto stack.
  - the branch address is transferred to PC and new PSW is loaded into the status register.
  - the interrupt routine can now be executed starting from the branch address (which may contain a branch instruction to a user defined service routine).
  - the last instruction of the interrupt routine is a “return from interrupt”: the stack is popped to retrieve PWS to status register and return address to PC => CPU state is restored and the interrupted program can proceed like nothing had happen.

- Interrupt types:
  1. External interrupts
     - from I/O, timing, or any other external source.
     - e.g.: I/O device requesting new data, elapsed time of an event, power failure, etc.
  2. Internal interrupts (traps)
     - from illegal or erroneous use of an instruction or data.
     - e.g.: overflow, division by zero, invalid operation code, stack overflow, and protection violation.
     - usually occur as a result of a premature termination of the instruction execution: the service program determines the corrective measure to be taken (*e.g*. terminates the program).
  3. Software interrupts
     - initiated by an instruction (rather than HW signals)
     - a special call instruction that behaves like an interrupt.
     - can be used by a programmer to initiate an interrupt routine at any desired point in the program.
     - can be used for accessing operating system services, for example.

- *E.g.*: using INT-instruction (i.e. INT 21h) for invoking a DOS interrupt (see 80x86 example code shown earlier):

```
TimePrompt              DB 'Is it after 12 noon (Y/N)?$'
        .
        .
mov     dx,TimePrompt
mov     ah,9
int     21h
```

pass information to the operating system in order to specify the particular task requested

INT 21,9 documentation (http://members.tripod.com/~oldboard/assembly/int_21-9.html):

## INT 21,9 - Print String

```
AH = 09
DS:DX = pointer to string ending in "$"


returns nothing


- outputs character string to STDOUT up to "$"
- backspace is treated as non-destructive
- if Ctrl-Break is detected, INT 23 is executed
```

# Reduced Instruction Set Computer (RISC)

- Instruction set determines the way that machine language programs are constructed.
- Early computers had small and simple instruction sets in order to minimize the (expensive) hardware needed for their implementation.
- Today many computers have instructions that include 100 to 200 instructions
  - variety of data types
  - large number of addressing modes

- Complex instruction set computer (CISC) has complex hardware and large instruction set: functions from software to hardware.
- In contrast, reduced instruction set computer (RISC) uses fewer and simpler instructions which can be executed faster within the CPU.
- RISC chips require fewer transistors (than CISC), which makes them cheaper to design and produce.
- There is still considerable controversy among experts about the ultimate value of RISC architectures
  - Its proponents argue that RISC machines are both cheaper and faster, and are therefore the machines of the future.
  - Skeptics note that by making the hardware simpler, RISC architectures put a greater burden on the software. They argue that this is not worth the trouble because conventional microprocessors are becoming increasingly fast and cheap anyway.

- However, CISC and RISC implementations are becoming more and more alike
  - Many of today's RISC chips support as many instructions as yesterday's CISC chips
  - Today's CISC chips use many techniques formerly associated with RISC chips.
- One reason for the trend to provide a complex instruction set is to simplify the translation from high-level to machine language programs.
- Characteristics of CISC architecture:
  1. A large instruction set.
  2. Instructions that perform special tasks and are used infrequently.
  3. A large variety of addressing modes (5-20 different modes).
  4. Variable-length instruction formats.
  5. Instructions that manipulate operands in memory.

- Characteristics of RISC architecture:
  1. Relatively few instructions
     - mostly register-to-register operations
  2. Relatively few addressing modes (because of 1)
  3. Memory access limited to load and store instructions.
  4. All operations done within the register of the CPU.
  5. Fixed-length, easily decoded instruction format
     - aligned to word boundaries
     - simplifies control logic
  6. Single-cycle instruction execution
     - fetch, decode, and execute phases for two to three instructions overlap: pipelining.
     - Memory references may take more clock cycles.
  7. Hardwired rather than microprogrammed control
     - faster

- Other RISC characteristics:
  1. A large number of register
     - useful for storing intermediate results and for optimizing operand references: much faster than memory references.
     - most frequent accessed operands are kept in registers
  2. Use of overlapped register windows to speed-up procedure call and return.
  3. Efficient instruction pipeline
  4. Compiler support for efficient translation of high-level language programs into machine language programs.
- A characteristic of some RISC processors is their use of overlapped register windows to provide the passing of parameters and avoid need for saving and restoring register values: speeds up procedure calls and returns.

- Each procedure call results in the allocation of a new window consisting of a set of registers
  - current window pointer (CWP) is decremented: corresponds ‘save’ in Fig. 8-11.
- Each return statement increments the CWP
  - corresponds ‘restore’ in Fig. 8-11.
- Windows for adjacent procedures (nested calls) have overlapping registers that are shared to provide the passing of parameters and results.
- Local register can be used for local variables
  - by using local registers there is no risk of corrupting data of another procedure (e.g. caller).

- Overlapped registers are used to pass parameters (in) and store results (out).
- Only one register windows is activated at any given time with a CWP.
- Each procedure call activates new register window by updating (decrementing) the CWP.
- To summarize:
  - Register windows provide easy access to a large collection of registers and can reduce the need to save registers in memory.
  - If you write a procedure with more parameters than common registers (reserved for input parameters), you will need to to use the stack for any parameters beyond the amount of reserved number of registers.
  - If your call sequence gets deeper than number of windows (as it probably will in most recursive procedures), you are again forced to use the stack.

NOTE: In Mano 1993, CWP is incremented in a call, and decremented in a return. The example in these slides refers to SPARC processor architecture.

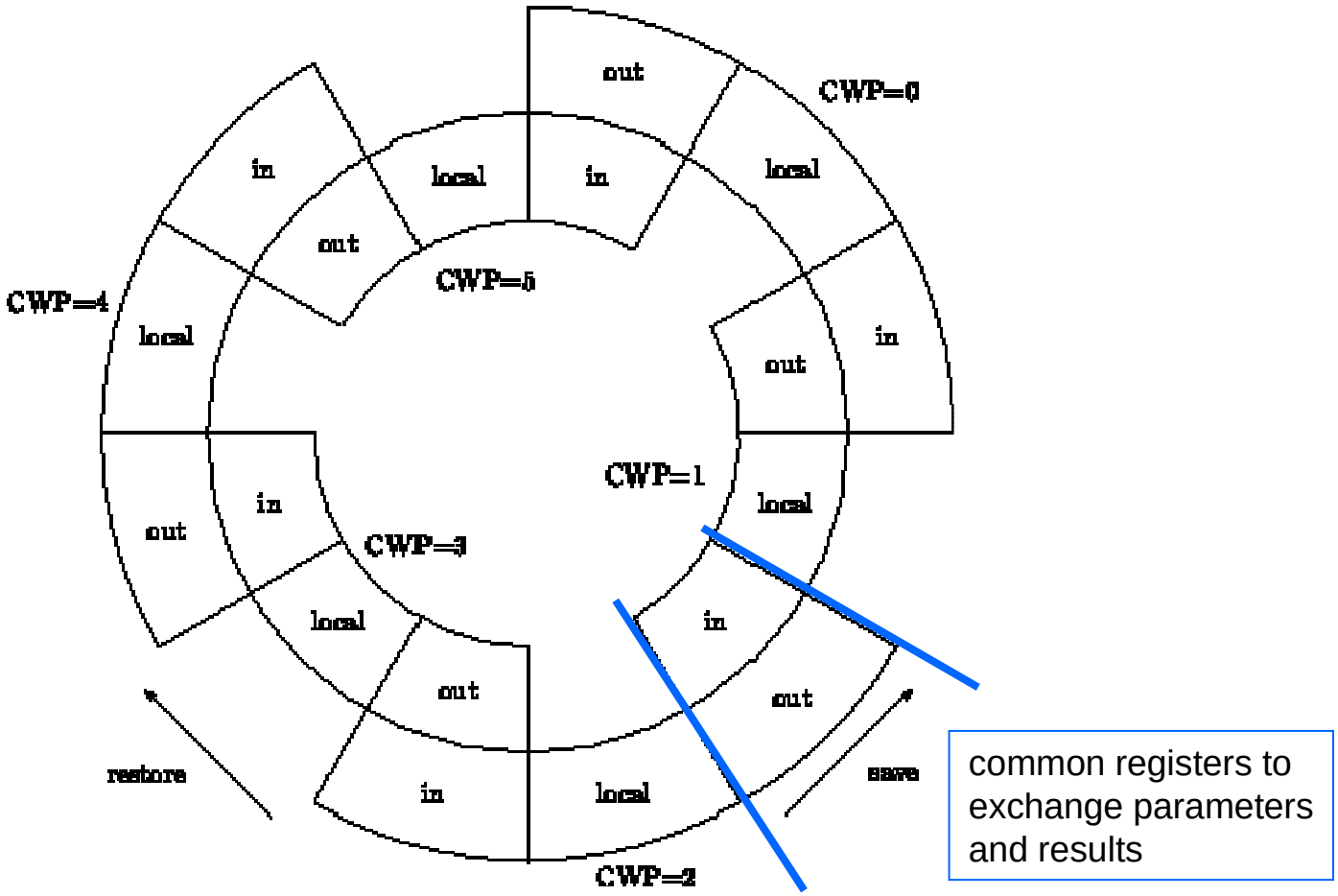
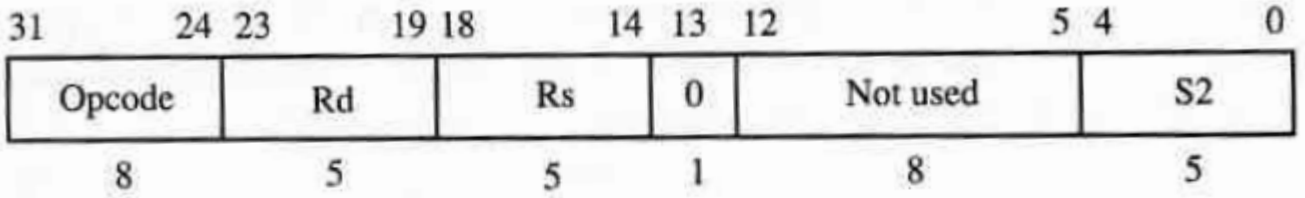

CWP=0
out
local
in
CWP=1
local
in
out
CWP=2
out
local
in
CWP=3
local
in
out
CWP=4
local
in
out
CWP=5
out
local
in
restore
save
common registers to exchange parameters and results

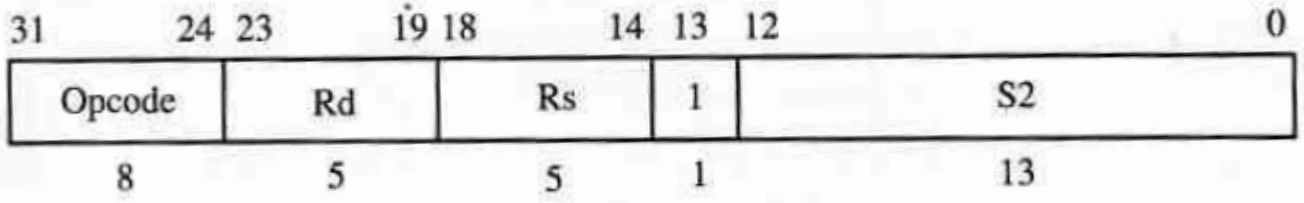
Fig. 8-11

- Berkeley RISC I
  - One of the first projects for showing the advantages of RISC architecture (in University of California, Berkeley).
    - 32-bit CPU
    - 32-bit addresses
    - 8-, 16-, or 32-bit data
    - 32-bit fixed length instruction format
      - suits well for pipelining
    - 32 instructions
    - three addressing modes: register, immediate operand, and relative to PC addressing for branch instructions.
    - 138 registers
      - 10 global and 8 windows of 32 registers.

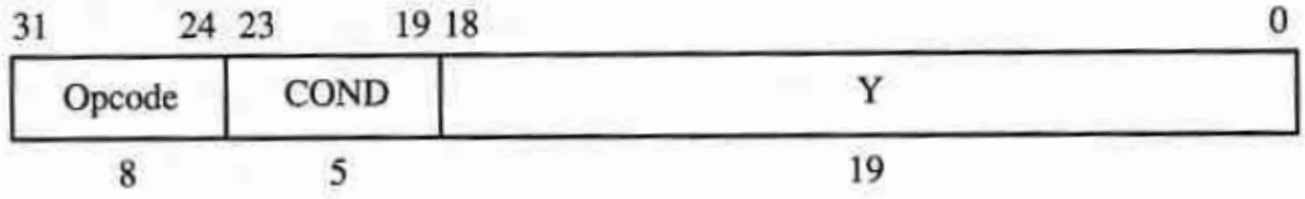
Figure 8-10 Berkeley RISC I instruction formats.




(a) Register mode: (S2 specifies a register)




(b) Register–immediate mode: (S2 specifies an operand)




(c) *PC* relative mode:

(Mano 1993)

**TABLE 8-12** Instruction Set of Berkeley RISC I

| Opcode | Operands | Register Transfer | Description |
|---|---|---|---|
| Data manipulation instructions | | | |
| ADD | Rs,S2,Rd | $Rd \leftarrow Rs + S2$ | Integer add |
| ADDC | Rs,S2,Rd | $Rd \leftarrow Rs + S2 + \text{carry}$ | Add with carry |
| SUB | Rs,S2,Rd | $Rd \leftarrow Rs - S2$ | Integer subtract |
| SUBC | Rs,S2,Rd | $Rd \leftarrow Rs - S2 - \text{carry}$ | Subtract with carry |
| SUBR | Rs,S2,Rd | $Rd \leftarrow S2 - Rs$ | Subtract reverse |
| SUBCR | Rs,S2,Rd | $Rd \leftarrow S2 - Rs - \text{carry}$ | Subtract with carry |
| AND | Rs,S2,Rd | $Rd \leftarrow Rs \wedge S2$ | AND |
| OR | Rs,S2,Rd | $Rd \leftarrow Rs \vee S2$ | OR |
| XOR | Rs,S2,Rd | $Rd \leftarrow Rs \oplus S2$ | Exclusive-OR |
| SLL | Rs,S2,Rd | $Rd \leftarrow Rs$ shifted by $S2$ | Shift-left |
| SRL | Rs,S2,Rd | $Rd \leftarrow Rs$ shifted by $S2$ | Shift-right logical |
| SRA | Rs,S2,Rd | $Rd \leftarrow Rs$ shifted by $S2$ | Shift-right arithmetic |
| Data transfer instructions | | | |
| LDL | (Rs)S2,Rd | $Rd \leftarrow M[Rs + S2]$ | Load long |
| LDSU | (Rs)S2,Rd | $Rd \leftarrow M[Rs + S2]$ | Load short unsigned |
| LDSS | (Rs)S2,Rd | $Rd \leftarrow M[Rs + S2]$ | Load short signed |
| LDBU | (Rs)S2,Rd | $Rd \leftarrow M[Rs + S2]$ | Load byte unsigned |
| LDBS | (Rs)S2,Rd | $Rd \leftarrow M[Rs + S2]$ | Load byte signed |
| LDHI | Rd,Y | $Rd \leftarrow Y$ | Load immediate high |
| STL | Rd,(Rs)S2 | $M[Rs + S2] \leftarrow Rd$ | Store long |
| STS | Rd,(Rs)S2 | $M[Rs + S2] \leftarrow Rd$ | Store short |
| STB | Rd,(Rs)S2 | $M[Rs + S2] \leftarrow Rd$ | Store byte |
| GETPSW | Rd | $Rd \leftarrow PSW$ | Load status word |
| PUTPSW | Rd | $PSW \leftarrow Rd$ | Set status word |
| Program control instructions | | | |
| JMP | COND, S2(Rs) | $PC \leftarrow Rs + S2$ | Conditional jump |
| JMPR | COND,Y | $PC \leftarrow PC + Y$ | Jump relative |
| CALL | Rd,S2(Rs) | $Rd \leftarrow PC$<br>$PC \leftarrow Rs + S2$<br>$CWP \leftarrow CWP - 1$ | Call subroutine and change window |
| CALLR | Rd,Y | $Rd \leftarrow PC$<br>$PC \leftarrow PC + Y$<br>$CWP \leftarrow \text{CWP} - 1$ | Call relative and change window |
| RET | Rd,S2 | $PC \leftarrow Rd + S2$<br>$CWP - CWP + 1$ | Return and change window |
| CALLINT | Rd | $Rd \leftarrow PC$<br>$CWP \leftarrow CWP - 1$ | Disable interrupts |
| RETINT | Rd,S2 | $PC \leftarrow Rd + S2$<br>$CWP \leftarrow CWP + 1$ | Enable interrupts |
| GTLPC | Rd | $Rd \leftarrow PC$ | Get last $PC$ |

(Mano 1993)