ECE 486/586
Computer Architecture

Prof. Mark G. Faust

Maseeh College of Engineering
and Computer Science

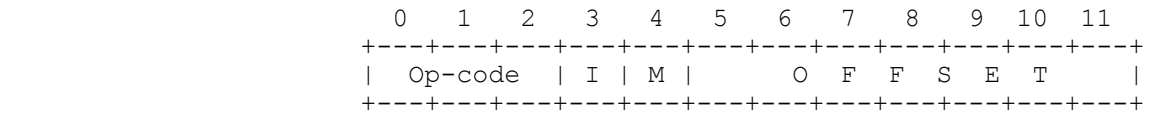
PORTLAND STATE UNIVERSITY

PDP-8

- DEC PDP-8 (1964)
  - First minicomputer
  - “desktop”
  - Omnibus
- Architecture
  - Accumulator, Link bit
  - 12-bit word (bit 0 is MSB)
  - 4K memory

PDP-8 Architecture

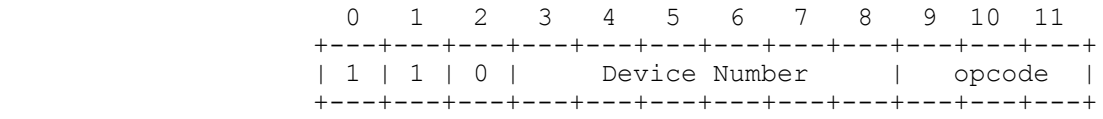
A.1 Memory Reference Instructions: Opcodes 0 - 5



Bits 0 - 2 : Operation Code
Bit 3 : Indirect Addressing Bit (0:Direct/1:Indirect)
Bit 4 : Memory Page (0:Zero Page/1 Current Page)
Bits 5 - 11 : Offset Address

Assembler Mnemonic	Machine Code	Effect
AND	0nnn	logical AND C(AC) <- C(AC) AND C(EAddr)
TAD	1nnn	Twos Complement Add C(AC) <- C(AC) + C(EAddr) If carry out then complement Link
ISZ	2nnn	Increment and Skip on Zero C(EAddr) <- C(EAddr) + 1 If C(EAddr) = 0 then C(PC) <- C(PC) + 1
DCA	3nnn	Deposit and Clear Accumulator C(EAddr) <- C(AC) C(AC) <- 0
JMS	4nnn	JuMp to Subroutine C(EAddr) <- C(PC) C(PC) <- EAddr + 1
JMP	5nnn	JuMP C(PC) <- EAddr

A.2 Input Output Transfer Instructions: Opcode 6



Bits 0 - 2 : Opcode 6
Bits 3 - 8 : Device Number
Bits 9 - 11 : Extended Opcode (operation specification bits)

A.2.1 Keyboard - Device #3

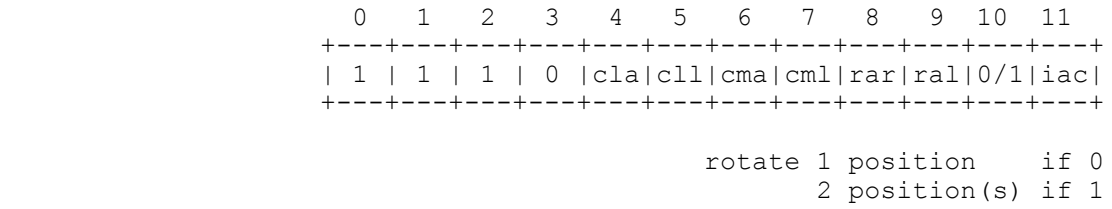
Assembler Mnemonic	Machine Code	Effect
KCF	6030	Clear Keyboard Flag
KSF	6031	Skip on Keyboard Flag set
KCC	6032	Clear Keyboard flag and aCcumulator
KRS	6034	Read Keyboard buffer Static AC4..AC11 <- AC4..AC11 OR Keyboard Buffer
KRB	6036	Read Keyboard Buffer dynamic C(AC) <- 0; Keyboard Flag <- 0; AC4..AC11 <- AC4..AC11 OR Keyboard Buffer

A.2.2 Printer (CRT) - Device #4

Assembler Mnemonic	Machine Code	Effect
TFL	6040	set prinTer FLag
TSF	6041	Skip on prinTer Flag set
TCF	6042	Clear prinTer Flag
TPC	6044	load prinTer buffer with aCcumulator and Print Printer Buffer <- AC4-11
TLS	6046	Load prinTer Sequence Printer Flag <- 0; Printer Buffer <- AC4-11

A.3 Microinstructions: Opcode 7

A.3.1 Group 1 Microinstructions (Bit 3 = 0)

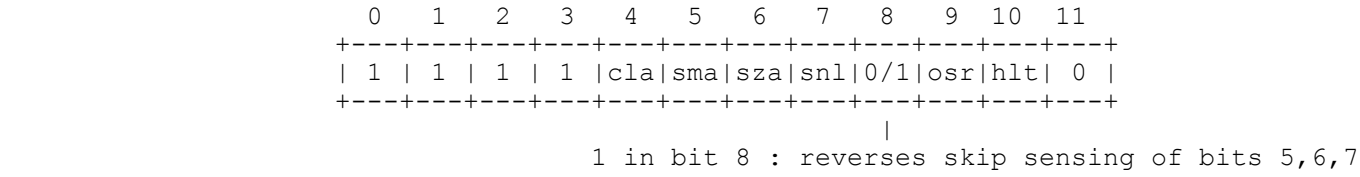


Sequence Numbers are in ()

Assembler Mnemonic	Machine Code	Effect
NOP	7000	No Operation
CLA	7200	CLear Accumulator (1)
CLL	7100	CLear Link (1)
CMA	7040	CoMplement Accumulator (2)
CML	7020	CoMplement Link (2)
IAC	7001	Increment ACumulator (3)
RAR	7010	Rotate Accumulator and link Right (4)
RTR	7012	Rotate accumulator and link Right Twice (4)
RAL	7004	Rotate Accumulator and link Left (4)
RTL	7006	Rotate Accumulator and link left Twice (4)

Group One microinstructions may be freely combined as long as the combination makes sense. Order of operations is determined by the sequence number.

A.3.2 Group 2 Microinstructions (Bit 3 = 1, Bit 11 = 0)



Sequence Numbers are in ()

Assembler Mnemonic	Machine Code	Effect
SMA	7500	Skip on Minus Accumulator (1)
SZA	7440	Skip on Zero Accumulator (1)
SNL	7420	Skip on Nonzero Link (1)

Combinations of SMA, SZA, and/or SNL will skip if at least one condition is true (OR Subgroup)

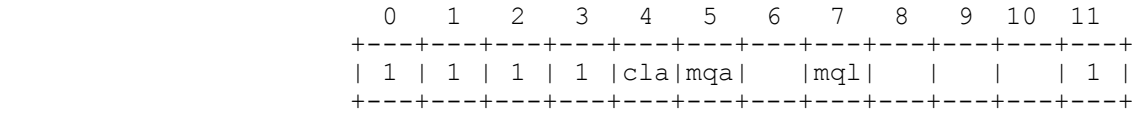
SPA	7510	Skip on Positive Accumulator (1)
SNA	7450	Skip on Nonzero Accumulator (1)
SZL	7430	Skip on Zero Link (1)

Combinations of SPA, SNA and/or SZL will skip when all conditions are true (AND Subgroup)

SKP	7410	SKiP always (1)
CLA	7600	CLear Accumulator (2)
OSR	7404	Or Switch Register with accumulator (3)
HLT	7402	HaLT (3)

Group Two microinstructions many be combined as long as instructions from the OR and the AND subgroups are not mixed and the combination makes sense. Order of operations is determined by sequence number.

A.3.3 Group 3 Microinstructions (Bit 3 = 1, Bit 11 = 1)



Sequence Numbers are in ()

Assembler Mnemonic	Machine Code	Effect
CLA	7601	CLear Accumulator (1)
MQL	7421	Load MQ register from AC and Clear AC (2) C(MQ) <- C(AC); C(AC) <- 0;
MQA	7501	Or AC with MQ register (2) C(AC) <- C(AC) Or C(MQ)
SWP	7521	SWap AC and MQ registers (3)
CAM	7621	Clear AC and MQ registers (3)

Appendix B: PDP-8 Addressing Modes

B.1 Zero Page Addressing (Bit 4 = 0)

Effective Address <- 00000 + Offset where '+' is the concatenate operation

B.2 Current Page Addressing (Bit 4 = 1)

Effective Address <− Old_PC0..Old_PC4 + Offset

where '+' is the concatenate operation. Old_PC0..Old_PC4 are bits 0 thru 4 of the PC *before* it is incremented (i.e. address of the current instruction).

B.3 Indirect Addressing (Bit 3 = 1)

If Zero Page Addressing (Then

Effective Address <− C(00000 + Offset)

If Current Page Addressing (Then

Effective Address <− C(Old_PC0-4 + Offset)

where '+' is the concatenate operation.

B.4 Autoindexing (Bit 3 = 1, Bit 4 = 0, Offset = 010o - 017o)

Addresses 0010o - 0017o are special AutoIndex Registers.

Whenever one of these locations is addressed indirectly, its contents is first incremented then used as the address of the Effective Address.

C(AutoIndex_Register) <− C(AutoIndex_Register) + 1
Effective Address <− C(AutoIndex_Register)

where AutoIndex_Register is an address in the range 0010o - 0017o.

PDP-8 Subroutine Linkage

Subroutine instructions

The jump to subroutine (jms) instruction stores the return address in the first word of the subroutine and starts execution with the second word (subr address + 1).

```
jms      jump to subroutine     memory[effective address] <- pc
                                pc <- (effective address) + 1
```

Subroutine return is performed by an indirect jump using the address in the first word of the subroutine.

```
jmp i    jump indirect          pc <- memory[effective address]
```

Parameter passing

Parameters are typically passed in-line after the jms instruction. The subroutine accesses the parameters via the stored return address, which it then increments so that it will point to the next instruction after the in-line parameters. A variable-length parameter list should start with a parameter count as the first in-line parameter.

Calling program structure

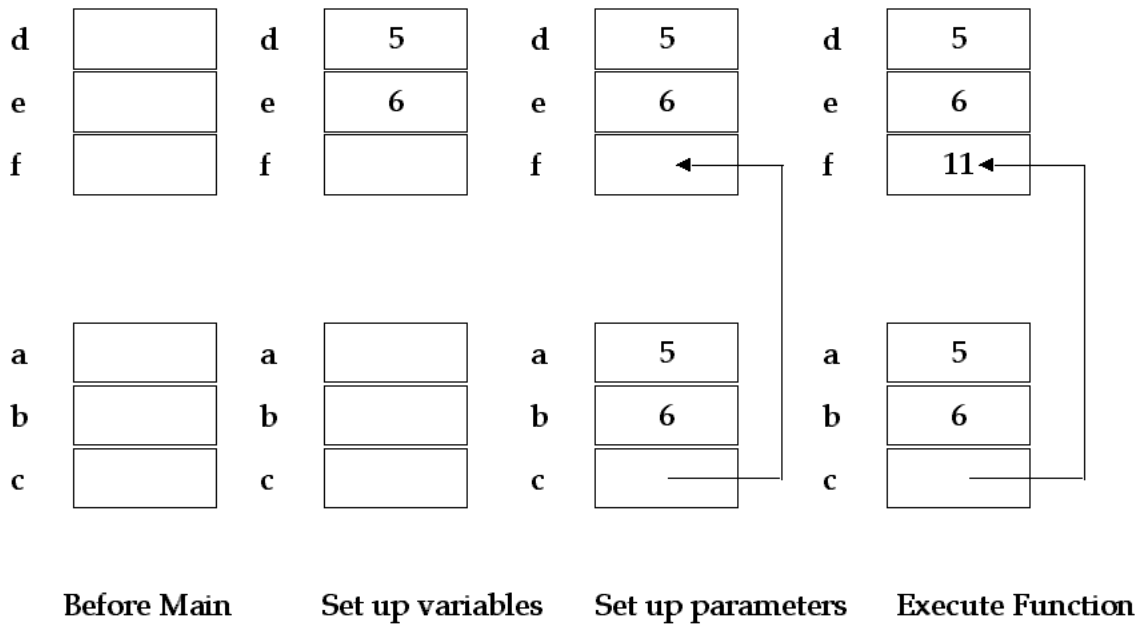
```
   ...
   jms   subr    ! call with fixed number of parameters (two)
parm1,   1       ! first in-line parameter
parm2,   2       ! second in-line parameter
   ...           ! subroutine will return here, after parms.
```

Subroutine structure

```
subr, 0          ! first word is reserved for return address
   cla
   tad i subr    ! load indirect first word after jms inst.
   dca   local1  !    and store in local variable
   isz   subr    ! increment address in subr
   tad i subr    ! load indirect second word after jms inst.
   dca   local2  !    and store in local variable
   isz   subr    ! increment address in subr

   ... body of subroutine ...

   jmp i subr    ! return via first word

local1,  0       ! local copy of first parameter
local2,  0       ! local copy of second parameter
```

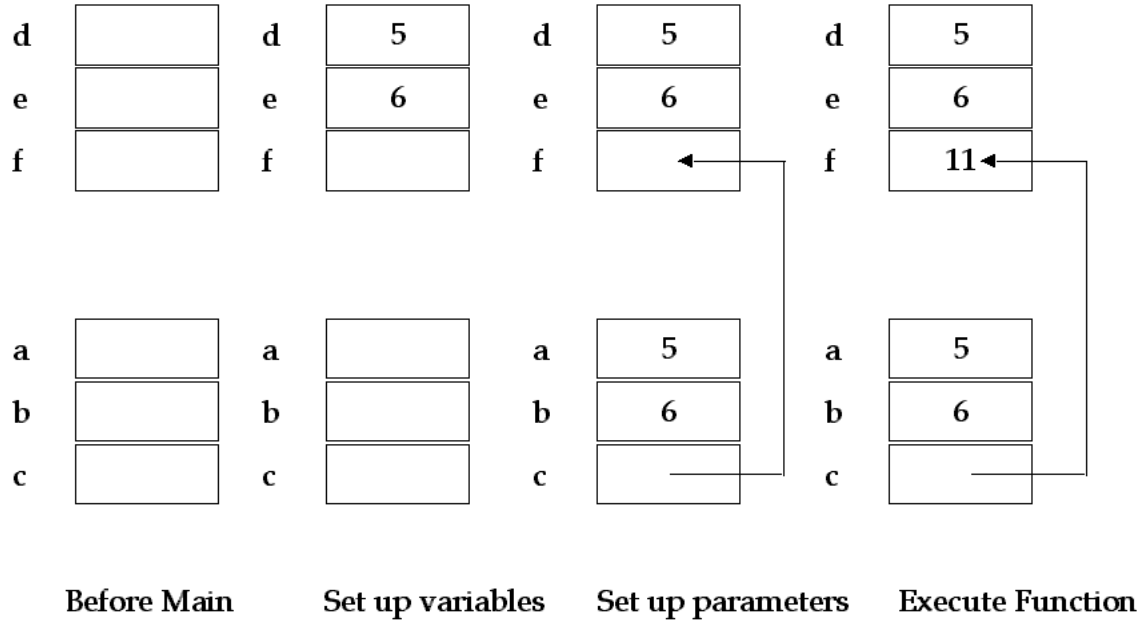
PDP-8 Subroutine: Pass by Reference

```
void sum (int a, int b, int *c)
{
   *c = a + b;
}

int d, e, f;
d = 5;
e = 6;
sum (d, e, &f);
```



```
            *20
d,          0
e,          0
f,          0
addr_f,     f
five,       5
six,        6
            *200
            cla
            tad five
            dca d
            tad six
            dca e
            tad d
            dca a
            tad e
            dca b
            tad addr_f
            dca c
            jms sum
a,          0
b,          0
c,          0
            <next instruction>
```

```
            *400
sum,        0           / start of subroutine sum
            cla
            tad i sum   / pick up a
            isz sum     / point to b
            tad i sum   / pick up b
            dca temp    / save partial sum for return
            isz sum     / point to c
            tad i sum   / get address of f
            dca addr    / save it for use
            tad temp    / retrieve partial sum
            dca i addr  / return to f
            isz sum     / point to return point
            jmp i sum
addr,       0           / holding area for address
temp,       0           / temporary variable
```



ISA Simulators

- Simulate architecture at ISA level
  - Simulates execution of binary machine object code (some will interpret assembly code)
  - Requires an assembler and/or compiler
  - Simulates effect of each instruction on ISA state (registers, memory, flags)
  - Doesn't simulate microarchitecture (pipeline, execution units, ROBs, etc)
  - May be cycle accurate

ISA Simulators

- Numerous uses
  - Try out architectural concepts
  - Permit concurrent engineering
    - Compilers/assemblers, Software tools
    - Hardware
  - Baseline against which to compare detailed design
  - Creation of trace information
    - Instruction traces
    - Memory accesses (may be affected by organization/ microarchitecture)
    - Branches