

# Influence of Technology and Software on Instruction Sets: Up to the dawn of IBM 360

*Joel Emer*

Computer Science and Artificial Intelligence Laboratory  
M.I.T.

# And then there was IBM 701

---

IBM 701 -- 30 machines were sold in 1953-54

IBM 650 -- a cheaper, drum based machine,  
more than 120 were sold in 1954  
and there were orders for 750 more!

*Users stopped building their own machines.*

Why was IBM late getting into computers?

*IBM was making too much money!*

Even without computers, IBM revenues were doubling every 4 to 5 years in 40's and 50's.

# Computers in mid 50's

---

- Hardware was expensive
- Stores were small (1000 words)
  - ⇒ No resident system-software!
- Memory access time was 10 to 50 times slower than the processor cycle
  - ⇒ Instruction execution time was totally dominated by the *memory reference time*.
- The *ability to design complex control circuits* to execute an instruction was the central design concern as opposed to *the speed* of decoding or an ALU operation
- Programmer's view of the machine was inseparable from the actual hardware implementation

# Accumulator-based computing

---



- *Single Accumulator*
  - Calculator design carried over to computers

*Why?*

*Registers expensive*

# The Earliest Instruction Sets

*Burks, Goldstein & von Neumann ~1946*

---

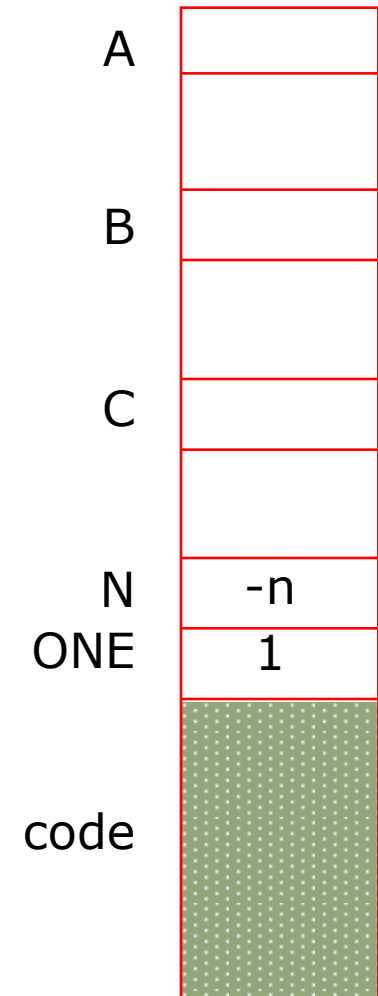
LOAD	x	$AC \leftarrow M[x]$
STORE	x	$M[x] \leftarrow (AC)$
ADD	x	$AC \leftarrow (AC) + M[x]$
SUB	x	
MUL	x	Involved a quotient register
DIV	x	
SHIFT LEFT		$AC \leftarrow 2 \times (AC)$
SHIFT RIGHT		
JUMP	x	$PC \leftarrow x$
JGE	x	if $(AC) \geq 0$ then $PC \leftarrow x$
LOAD ADR	x	$AC \leftarrow \text{Extract address field}(M[x])$
STORE ADR	x	

*Typically less than 2 dozen instructions!*

# Programming: Single Accumulator Machine

$$C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n$$

LOOP	LOAD	N
	JGE	DONE
	ADD	ONE
	STORE	N
F1	LOAD	A
F2	ADD	B
F3	STORE	C
	JUMP	LOOP
DONE	HLT	



Problem?

How to modify the addresses A, B and C ?

# Self-Modifying Code

LOOP	LOAD	N
	JGE	DONE
	ADD	ONE
	STORE	N
F1	LOAD	A
F2	ADD	B
F3	STORE	C
	LOAD ADR	F1
	ADD	ONE
	STORE ADR	F1
	LOAD ADR	F2
	ADD	ONE
	STORE ADR	F2
	LOAD ADR	F3
	ADD	ONE
	STORE ADR	F3
	JUMP	LOOP
DONE	HLT	

*modify the program for the next iteration*

$$C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n$$

*Each iteration involves*

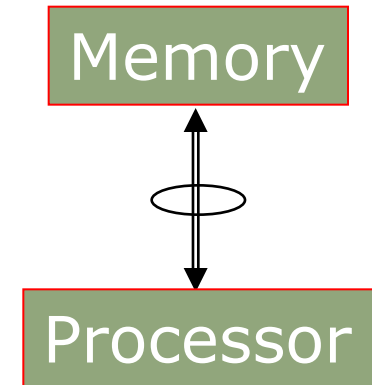
	<i>total</i>	<i>book-keeping</i>
<i>instruction fetches</i>	<b>17</b>	<b>14</b>
<i>operand fetches</i>	<b>10</b>	<b>8</b>
<i>stores</i>	<b>5</b>	<b>4</b>

*Most of the executed instructions are for book keeping!*

# Processor-Memory Bottleneck: Early Solutions

---

- Indexing capability
  - to reduce book keeping instructions
- Fast local storage in the processor
  - 8-16 registers as opposed to one accumulator
  - to save on loads/stores
- Complex instructions
  - to reduce instruction fetches
- Compact instructions
  - implicit address bits for operands
  - to reduce instruction fetch cost





# Index Registers

*Tom Kilburn, Manchester University, mid 50's*

*One or more specialized registers to simplify address calculation*

## Modify existing instructions

LOAD	x, IX	$AC \leftarrow M[x + (IX)]$
ADD	x, IX	$AC \leftarrow (AC) + M[x + (IX)]$
...		

## Add new instructions to manipulate *index registers*

JZi	x, IX	if (IX)=0 then $PC \leftarrow x$ else $IX \leftarrow (IX) + 1$
LOADi	x, IX	$IX \leftarrow M[x]$ (truncated to fit IX)
...		

*Index registers have accumulator-like characteristics*

# Using Index Registers

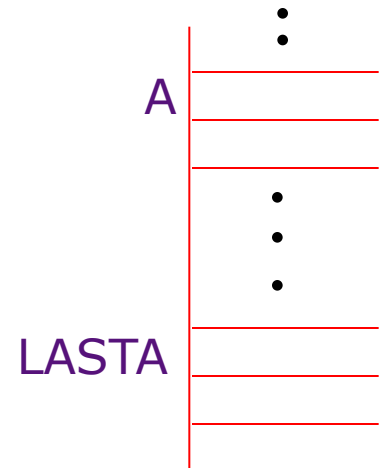
$$C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n$$

```

LOADi  N, IX
LOOP   JZi   DONE, IX
      LOAD  LASTA, IX
      ADD   LASTB, IX
      STORE LASTC, IX
      JUMP  LOOP
DONE   HALT

```

N starts with -n



- *Program does not modify itself*
- *Efficiency has improved dramatically (ops / iter)*

	with index regs	without index regs
instruction fetch	5(2)	17 (14)
operand fetch	2	10 (8)
store	1	5 (4)

- *Costs?*
  - *Complex control*
  - *Index register computations (ALU-like circuitry)*
  - *1 to 2 bits longer Instructions*

# Operations on Index Registers

---

To increment index register by  $k$

$AC \leftarrow (IX)$  *new instruction*

$AC \leftarrow (AC) + k$

$IX \leftarrow (AC)$  *new instruction*

also the AC must be saved and restored

It may be better to increment IX directly

$INCi \quad k, IX \quad IX \leftarrow (IX) + k$

More instructions to manipulate index register

$STOREi \quad x, IX \quad M[x] \leftarrow (IX)$  (extended to fit a word)

...

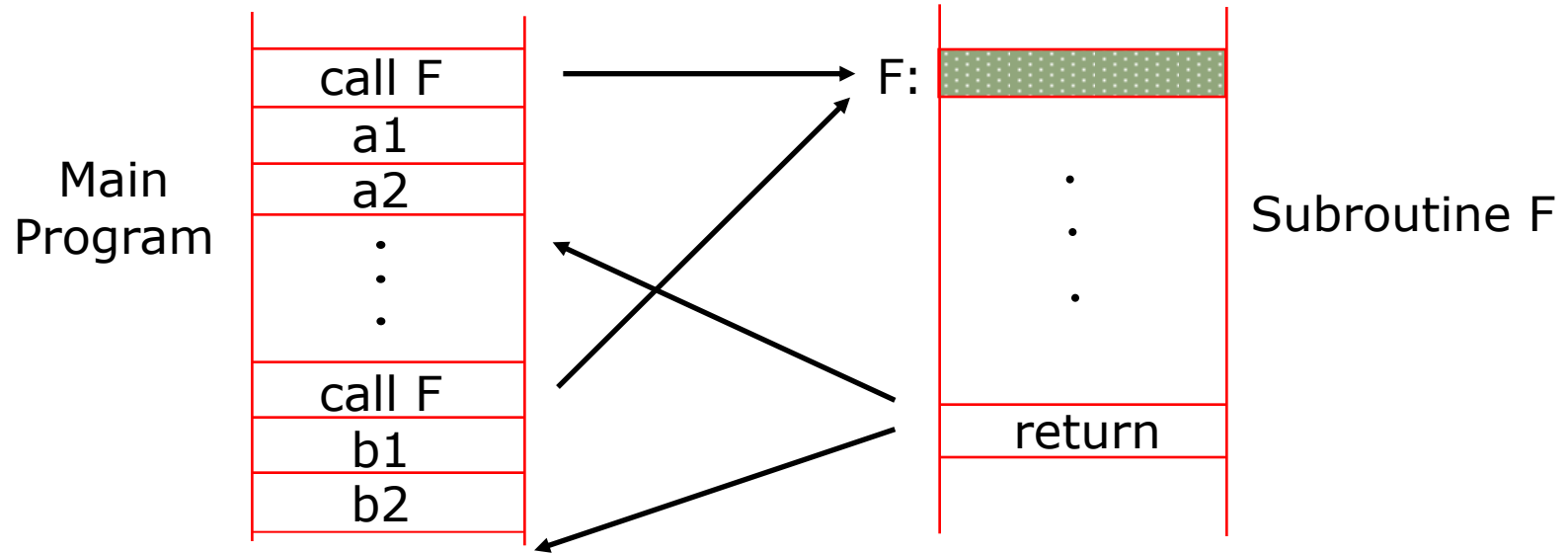
*IX begins to look like an accumulator*

⇒ several index registers

several accumulators

⇒ *General Purpose Registers*

# Support for Subroutine Calls



A special *subroutine jump instruction*

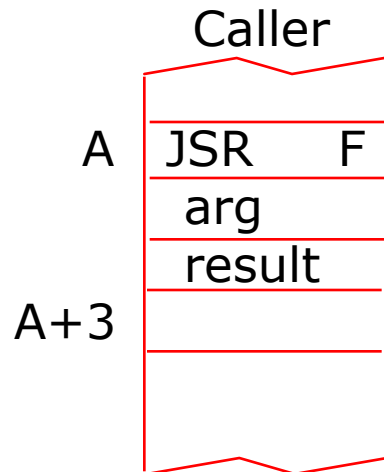
A: JSR F

$M[F] \leftarrow A + 1$  and  
jump to  $F + 1$

# Indirect Addressing and Subroutine Calls

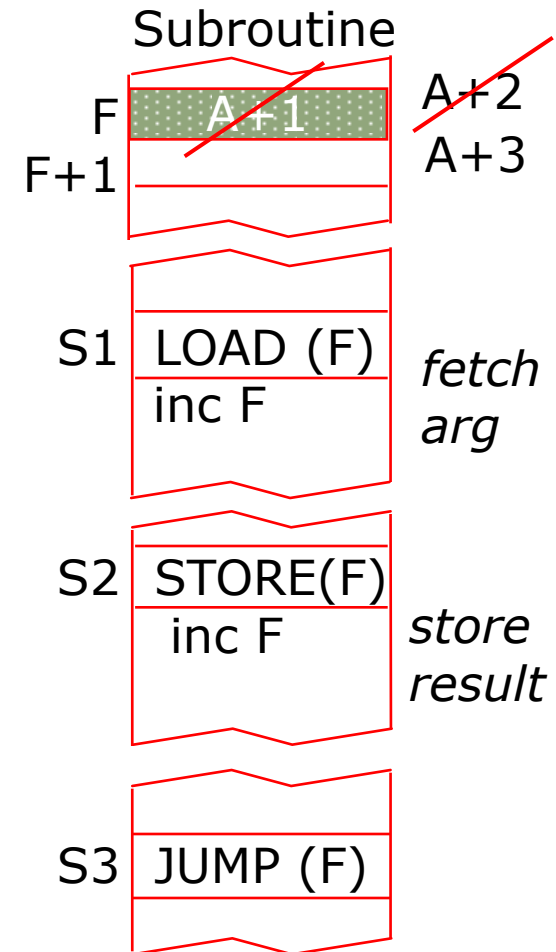
*Indirect addressing*

LOAD (x) means  $AC \leftarrow M[M[x]]$   
 ...



*Events:*

Execute A  
 Execute S1  
 Execute S2  
 Execute S3



Indirect addressing almost eliminates the need to write self-modifying code (location F still needs to be modified)

*Problems? ⇒ recursive procedure calls*

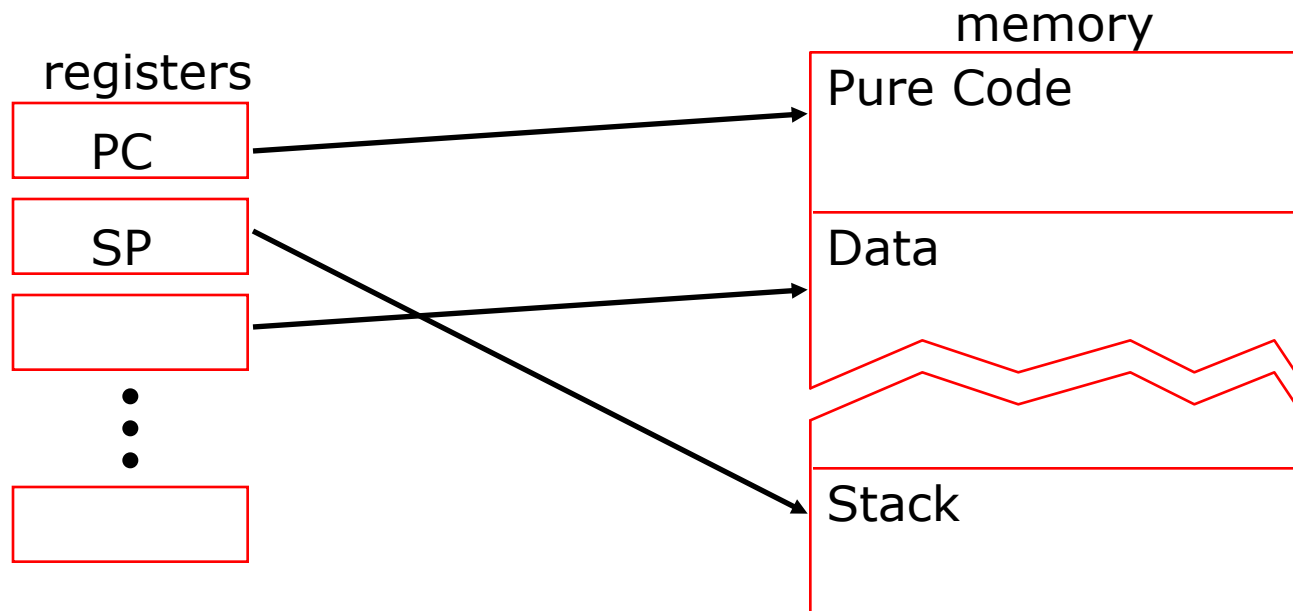
# Recursive Procedure Calls and Reentrant Codes

---

*Indirect Addressing through a register*

LOAD  $R_1, (R_2)$

Load register  $R_1$  with the contents of the word whose address is contained in register  $R_2$



# Evolution of Addressing Modes

---

1. Single accumulator, absolute address

LOAD x

2. Single accumulator, index registers

LOAD x, IX

3. Indirection

LOAD (x)

4. Multiple accumulators, index registers, indirection

LOAD R, IX, x

or

LOAD R, IX, (x)

the meaning?

$R \leftarrow M[M[x] + (IX)]$

or  $R \leftarrow M[M[x + (IX)]]$

5. Indirect through registers

LOAD  $R_I, (R_J)$

6. The works

LOAD  $R_I, R_J, (R_K)$

$R_J = \text{index}, R_K = \text{base addr}$

# Variety of Instruction Formats

---

- *Three address formats:* One destination and up to two operand sources per instruction

(Reg op Reg) to Reg  
(Reg op Mem) to Reg

$$R_I \leftarrow (R_J) \text{ op } (R_K)$$

$$R_I \leftarrow (R_J) \text{ op } M[x]$$

- x can be specified directly or via a register
- effective address calculation for x could include indexing, indirection, ...

- *Two address formats:* the destination is same as one of the operand sources

(Reg op Reg) to Reg  
(Reg op Mem) to Reg

$$R_I \leftarrow (R_I) \text{ op } (R_J)$$

$$R_I \leftarrow (R_I) \text{ op } M[x]$$

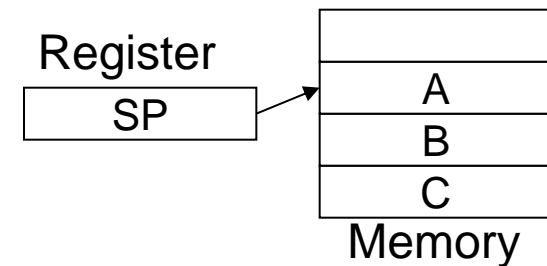


# More Instruction Formats

---

- *One address formats: Accumulator machines*
  - Accumulator is always other implicit operand
- *Zero address formats: operands on a stack*

add      $M[sp-1] \leftarrow M[sp] + M[sp-1]$   
 load     $M[sp] \leftarrow M[M[sp]]$



- Stack can be in registers or in memory
  - usually top of stack cached in registers

*Many different formats are possible!*

# Data Formats and Memory Addresses

---

Data formats:

Bytes, Half words, words and double words

Some issues

- *Byte addressing*

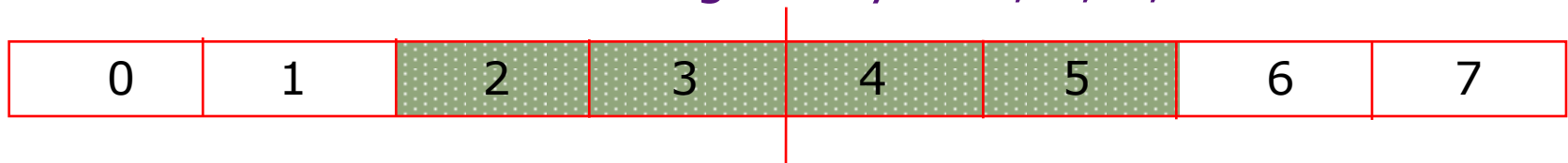
Big Endian  
vs. Little Endian

0	1	2	3
3	2	1	0

- *Word alignment*

Suppose the memory is organized in 32-bit words.

Can a word address begin only at 0, 4, 8, .... ?



# Some Tradeoffs

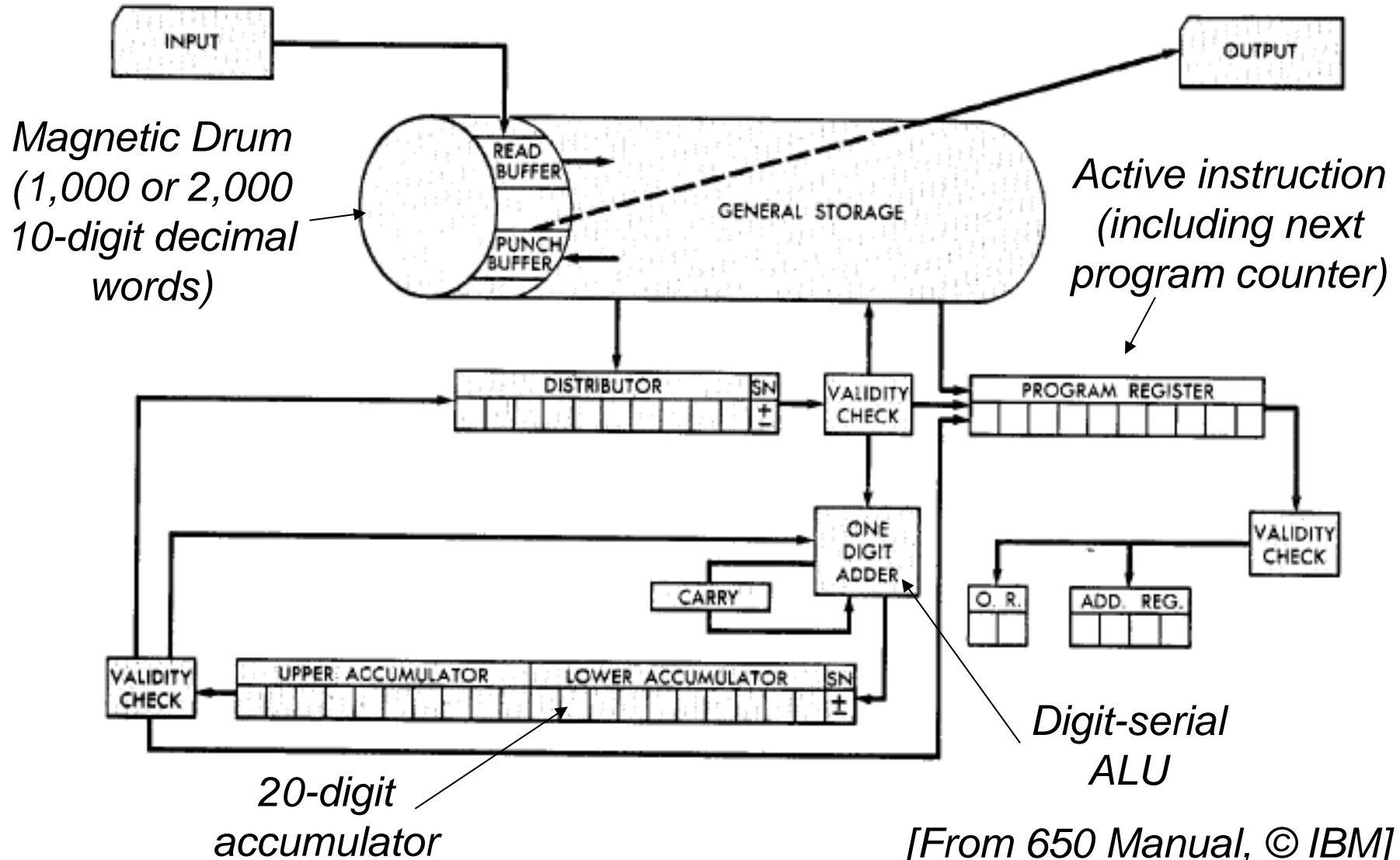
---

- Should all addressing modes be provided for every operand?
  - ⇒ *regular vs. irregular instruction formats*
- Separate instructions to manipulate Accumulators, Index registers, Base registers
  - ⇒ *large number of instructions*
- Instructions contained implicit memory references -- several contained more than one
  - ⇒ *very complex control*

Great variety of instruction sets

# The first definition of the Instruction Set Abstraction: IBM 360

# The IBM 650 (1953-4)



# Programmer's view of a machine: IBM 650

---

A drum machine with 44 instructions

Instruction:      60 1234 1009

- “Load the contents of location 1234 into the *distribution*; put it also into the *upper accumulator*; set *lower accumulator* to zero; and then go to location 1009 for the next instruction.”

- Programmer's view of the machine was inseparable from the actual hardware implementation
- Good programmers optimized the placement of instructions on the drum to reduce latency!

# Compatibility Problem at IBM

---

By early 60's, *IBM had 4 incompatible lines of computers!*

701	→	7094
650	→	7074
702	→	7080
1401	→	7010

Each system had its own

- Instruction set
- I/O system and Secondary Storage:  
magnetic tapes, drums and disks
- assemblers, compilers, libraries,...
- market niche  
business, scientific, real time, ...

⇒ *IBM 360*

# IBM 360 : Design Premises

*Amdahl, Blaauw and Brooks, 1964*

---

The design must lend itself to *growth and successor machines*

- General method for connecting I/O devices
- Total performance - answers per month rather than bits per microsecond  $\Rightarrow$  *programming aids*
- Machine must be capable of *supervising itself* without manual intervention
- Built-in *hardware fault checking* and locating aids to reduce down time
- Simple to assemble systems with redundant I/O devices, memories etc. for *fault tolerance*
- Some problems required floating point words larger than 36 bits



# Processor State and Data Types

---

*The information held in the processor at the end of an instruction to provide the processing context for the next instruction.*

Program Counter, Accumulator, . . .

- The information held in the processor will be interpreted as having data types manipulated by the instructions.
- If the processing of an instruction can be interrupted then the *hardware* must save and restore the state in a transparent manner

*Programmer's machine model* is a **contract** between the hardware and software

# Instruction set

---

*The control for changing the information held in the processor are specified by the instructions available in the instruction set architecture or ISA.*

Some things an ISA must specify:

- *A way to reference registers and memory*
- *The computational operations available*
- *How to control the sequence of instructions*
- *A binary representation for all of the above*

*ISA must satisfy the needs of the software:  
- assembler, compiler, OS, VM*

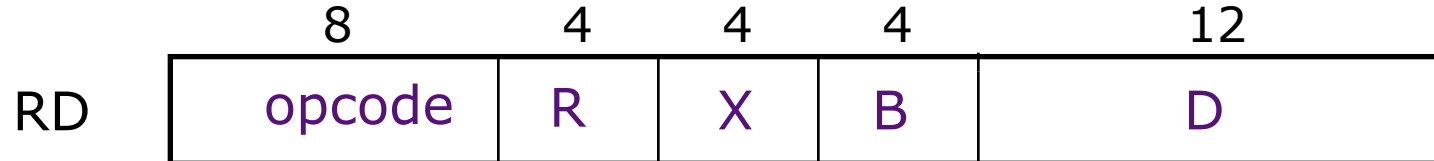
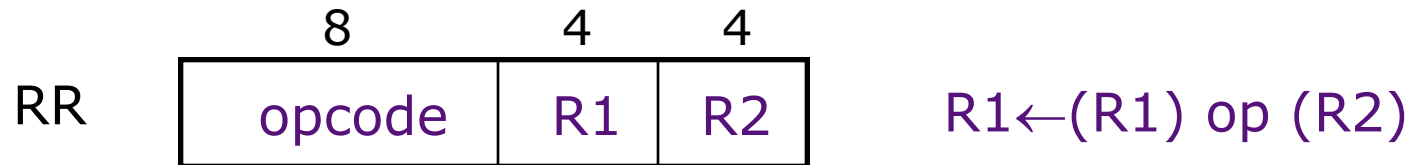
# IBM 360: *A General-Purpose Register (GPR) Machine*

---

- Processor State
  - 16 General-Purpose 32-bit Registers
    - *may be used as index and base register*
    - *Register 0 has some special properties*
  - 4 Floating Point 64-bit Registers
  - A Program Status Word (PSW)
    - *PC, Condition codes, Control flags*
- Data Formats
  - 8-bit bytes, 16-bit half-words, 32-bit words, 64-bit double-words
  - 24-bit addresses
- A 32-bit machine with 24-bit addresses
  - *No instruction contains a 24-bit address !*

# IBM 360: Some Addressing Modes

---



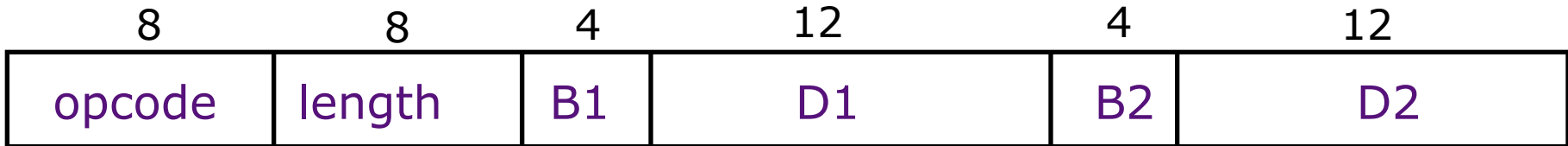
$R \leftarrow (R) \text{ op } M[(X) + (B) + D]$

a 24-bit address is formed by adding the 12-bit displacement (D) to a base register (B) and an Index register (X), if desired

*The most common formats for arithmetic & logic instructions, as well as Load and Store instructions*

# IBM 360: Character String Operations

---



SS format: store to store instructions

$$M[(B1) + D1] \leftarrow M[(B1) + D1] \text{ op } M[(B2) + D2]$$

iterate "length" times

*Most operations on decimal and character strings use this format*

MVC    move characters

MP     multiply two packed decimal strings

CLC    compare two character strings

...

*Multiple memory operations per instruction complicates exception & interrupt handling*

# IBM 360: Branches & Condition Codes

---

- Arithmetic and logic instructions set *condition codes*
  - equal to zero
  - greater than zero
  - overflow
  - carry...
- I/O instructions also set condition codes
  - channel busy
- Conditional branch instructions are based on testing condition code registers (CC's)
  - RX and RR formats
    - BC\_                    branch conditionally
    - BAL\_                    branch and link, i.e.,  $R15 \leftarrow (PC)+1$   
*for subroutine calls*
      - ⇒ CC's must be part of the PSW

# IBM 360: Precise Interrupts

---

- IBM 360 ISA (Instruction Set Architecture) preserves sequential execution model
- Programmers view of machine was that each instruction either completed or signaled a fault before the next instruction began execution
- Exception/interrupt behavior identical across family of implementations

# IBM 360: Initial Implementations (1964)

	<i>Model 30</i>	<i>. . .</i>	<i>Model 70</i>
<i>Memory Capacity</i>	8K - 64 KB		256K - 512 KB
<i>Memory Cycle</i>	2.0 $\mu$ s	<i>...</i>	1.0 $\mu$ s
<i>Datapath</i>	8-bit		64-bit
<i>Circuit Delay</i>	30 nsec/level		5 nsec/level
<i>Registers</i>	in Main Store		in Transistor
<i>Control Store</i>	Read only 1 $\mu$ sec		Dedicated circuits

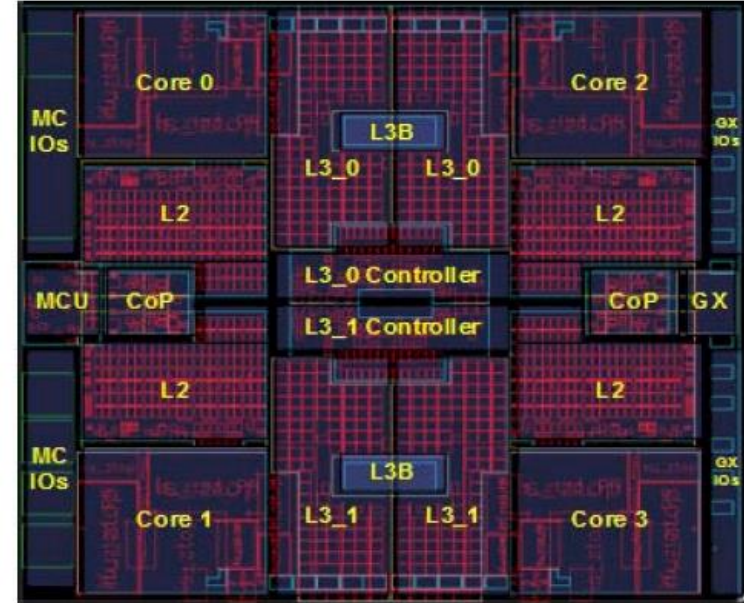
- Six implementations (Models, 30, 40, 50, 60, 62, 70)
- 50X performance difference cross models
- *ISA completely hid the underlying technological differences between various models.*

With minor modifications, IBM 360 ISA is still in use



# IBM 360: Forty-Six years later... zEnterprise196 Microprocessor

- 1.4 billion transistors, Quad core design
- Up to 96 cores (80 visible to OS) in one multichip module
- 5.2 GHz, IBM 45nm SOI CMOS technology
- 64-bit virtual addressing
  - original 360 was 24-bit; 370 was a 31-bit extension
- Superscalar, out-of-order
  - Up to 72 instructions in flight
- Variable length instruction pipeline: 15-17 stages
- Each core has 2 integer units, 2 load-store units and 2 floating point units
- 8K-entry Branch Target Buffer
  - Very large buffer to support commercial workload
- Four Levels of caches:
  - 64KB L1 I-cache, 128KB L1 D-cache
  - 1.5MB L2 cache per core
  - 24MB shared on-chip L3 cache
  - 192MB shared off-chip L4 cache



[ September 2010 ]

# Instruction Set Architecture (ISA) versus Implementation

---

- ISA is the hardware/software interface
  - Defines set of programmer visible state
  - Defines data types
  - Defines instruction semantics (operations, sequencing)
  - Defines instruction format (bit encoding)
  - Examples: *MIPS, Alpha, x86, IBM 360, VAX, ARM, JVM*
- Many possible implementations of one ISA
  - 360 implementations: *model 30 (c. 1964), zEnterprise196 (c. 2010)*
  - x86 implementations: *8086 (c. 1978), 80186, 286, 386, 486, Pentium, Pentium Pro, Pentium-4, Core i7, AMD Athlon, AMD Opteron, Transmeta Crusoe, SoftPC*
  - MIPS implementations: *R2000, R4000, R10000, ...*
  - JVM: *HotSpot, PicoJava, ARM Jazelle, ...*

*Next lecture:*

Implementing an ISA