



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

Arvind

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

*Based on the material prepared by
Arvind and Krste Asanovic*

Importance of Technology

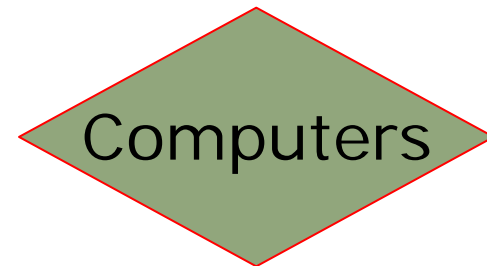
New technologies not only provide greater speed, size and reliability at lower cost, but more importantly these dictate the kinds of structures that can be considered and thus come to shape our whole view of what a computer is.

Bell & Newell

Technology is the dominant factor in computer design

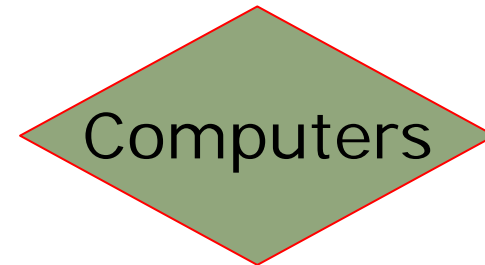
Technology

Transistors
Integrated circuits
VLSI (initially)
Laser disk, CD's



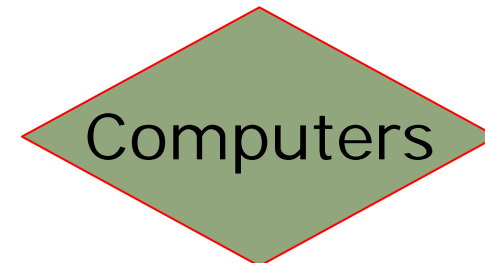
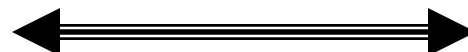
Technology

Core memories
Magnetic tapes
Disks



Technology

ROMs, RAMs
VLSI
Packaging
Low Power

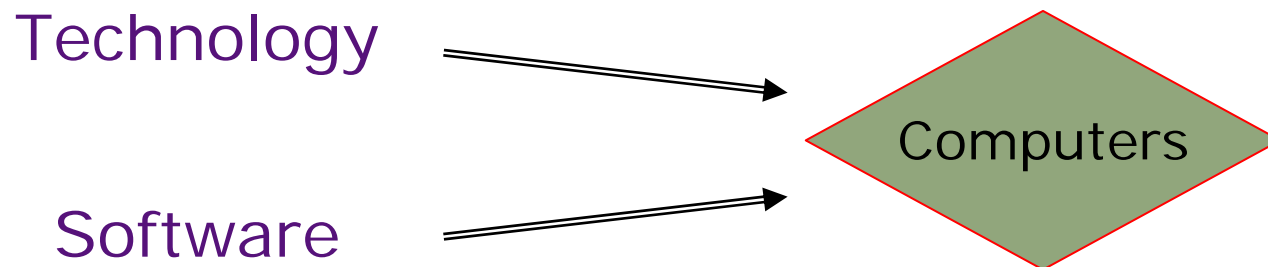


But Software...

As people write programs and use computers, our understanding of *programming* and *program behavior* improves.

This has profound though slower impact on computer architecture

Modern architects cannot avoid paying attention to software and compilation issues.



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

Programmer's view of the 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.”

Good programmers optimized the placement of instructions on the drum to reduce latency!

The Earliest Instruction Sets

Single Accumulator - A carry-over from the calculators.

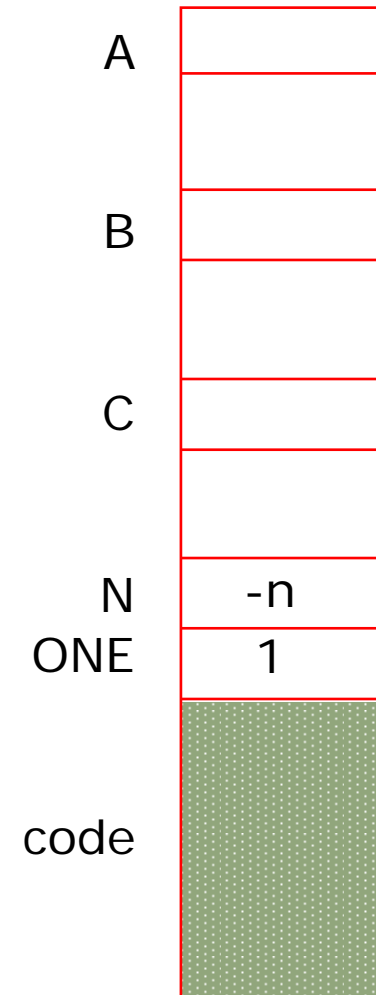
| | | |
|-------------|---|--|
| LOAD | X | $AC \leftarrow M[x]$ |
| STORE | X | $M[x] \leftarrow (AC)$ |
| ADD | X | $AC \leftarrow (AC) + M[x]$ |
| SUB | X | |
| MUL | X | Involved a quotient register |
| DIV | X | |
| SHIFT LEFT | | $AC \leftarrow 2 \times (AC)$ |
| SHIFT RIGHT | | |
| JUMP | X | $PC \leftarrow x$ |
| 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 | |



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

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

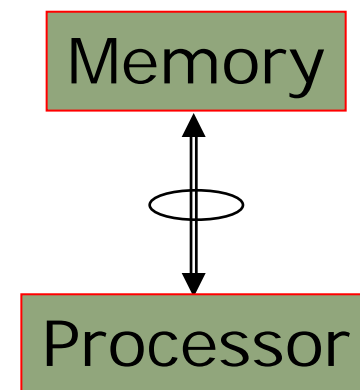
Each iteration involves

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



Processor-Memory Bottleneck: Early Solutions

- Fast local storage in the processor
 - 8-16 registers as opposed to one accumulator
- Indexing capability
 - to reduce book keeping instructions
- Complex instructions
 - to reduce instruction fetches
- Compact instructions
 - implicit address bits for operands, to reduce instruction fetches



Processor State

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

Program Counter, Accumulator, . . .

Programmer visible state of the processor (and memory) plays a central role in computer organization for both hardware and software:

- *Software* must make efficient use of it
- 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

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) |
| ... | | |

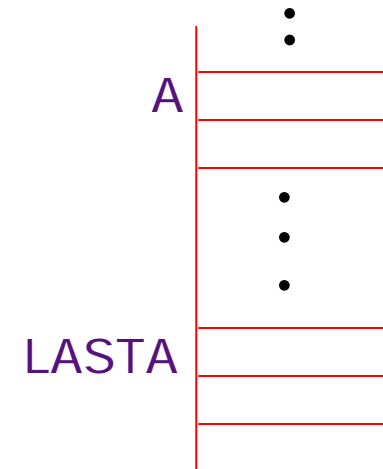
Index registers have accumulator-like characteristics

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
    
```



- *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:*
 - Instructions are 1 to 2 bits longer
 - Index registers with ALU-like circuitry
 - Complex control*

Indexing vs. Index Registers

Suppose instead of registers, memory locations are used to implement index registers.

LOAD x, IX

Arithmetic operations on index registers can be performed by bringing the contents to the accumulator

Most bookkeeping instructions will be avoided but each instruction will implicitly cause several fetches and stores

⇒ *complex control circuitry*

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)

...

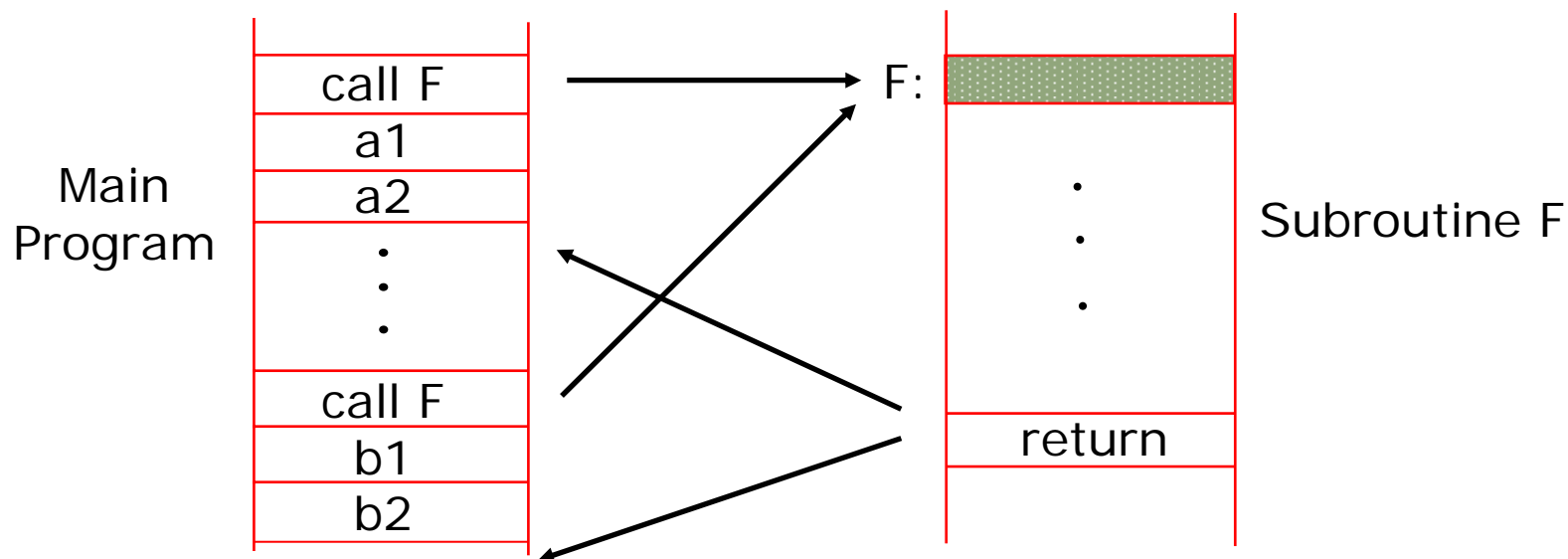
IX begins to look like an accumulator

⇒ several index registers

several accumulators

⇒ *General Purpose Registers*

Support for Subroutine Calls



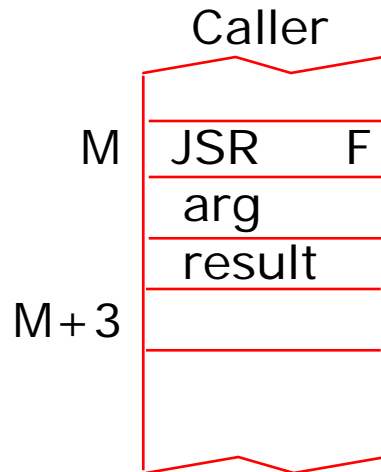
A special *subroutine jump instruction*

M: JSR F

$F \leftarrow M + 1$ and
jump to $F + 1$

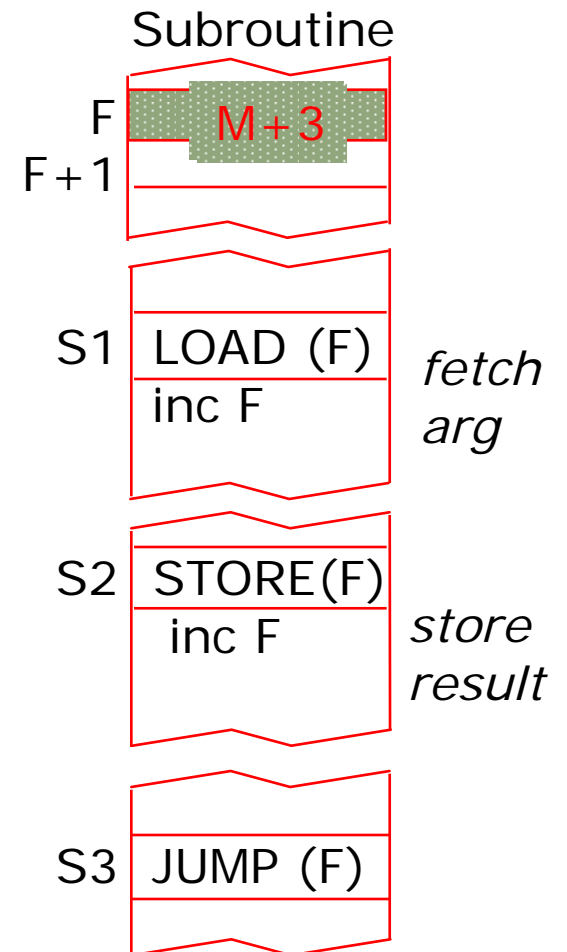
Indirect Addressing and Subroutine Calls

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



Events:

- Execute M
- Execute S1
- Execute S2
- Execute S3



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

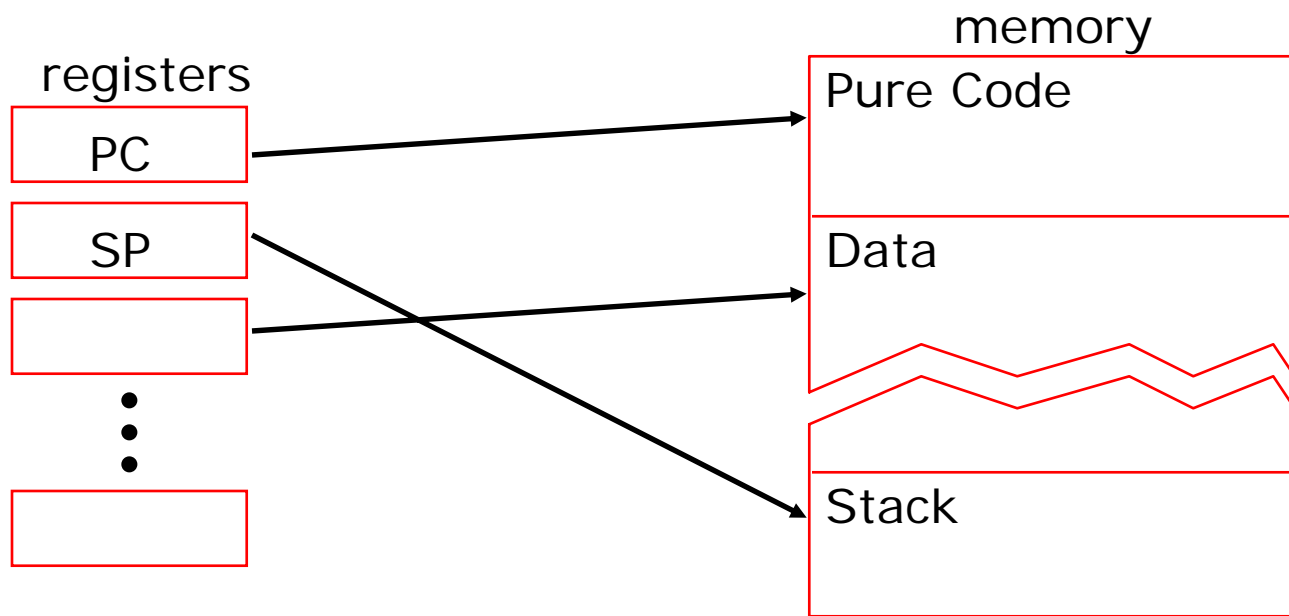
⇒ *Problems with recursive procedure calls*

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 = index, R_K = base address

Variety of Instruction Formats

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

(Reg × Reg) to Reg

$$R_i \leftarrow (R_j) + (R_k)$$

(Reg × Mem) to Reg

$$R_i \leftarrow (R_j) + M[x]$$

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

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

(Reg × Reg) to Reg

$$R_i \leftarrow (R_j) + (R_k)$$

(Reg × Mem) to Reg

$$R_i \leftarrow (R_j) + M[x]$$

More Instruction Formats

- *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)

- *One address formats:* Accumulator machines
 - Accumulator is always other implicit operand

Many different formats are possible!

Data Formats and Memory Addresses

Data formats:

Bytes, Half words, words and double words

Some issues

- *Byte addressing*

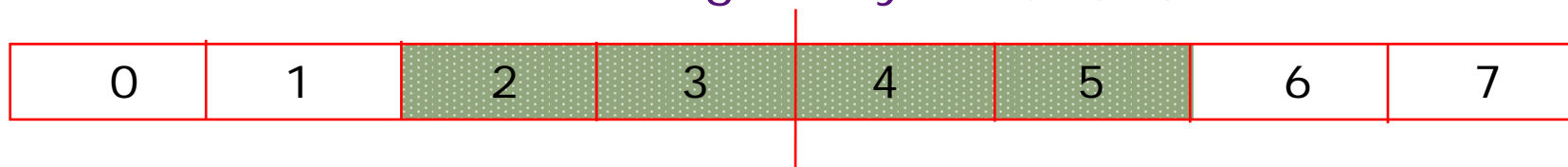
Big Endian
vs. Little Endian

| | | | |
|---|---|---|---|
| 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 Problems

- 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*

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

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*
- A 32-bit machine with 24-bit addresses
 - No instruction contains a 24-bit address !
- Data Formats
 - 8-bit bytes, 16-bit half-words, 32-bit words, 64-bit double-words

IBM 360: Implementation

| | <i>Model 30</i> | <i>. . .</i> | <i>Model 70</i> |
|----------------------|-----------------------|--------------|-----------------------|
| <i>Storage</i> | 8K - 64 KB | | 256K - 512 KB |
| <i>Datapath</i> | 8-bit | | 64-bit |
| <i>Circuit Delay</i> | 30 nsec/level | | 5 nsec/level |
| <i>Local Store</i> | Main Store | | Transistor Registers |
| <i>Control Store</i> | Read only 1 μ sec | | Conventional circuits |

IBM 360 instruction set architecture completely hid the underlying technological differences between various models.

With minor modifications it survives till today

IBM S/390 z900 Microprocessor

- 64-bit virtual addressing
 - first 64-bit S/390 design (original S/360 was 24-bit, and S/370 was 31-bit extension)
- 1.1 GHz clock rate (announced ISSCC 2001)
 - 0.18 μ m CMOS, 7 layers copper wiring
 - 770MHz systems shipped in 2000
- Single-issue 7-stage CISC pipeline
- Redundant datapaths
 - every instruction performed in two parallel datapaths and results compared
- 256KB L1 I-cache, 256KB L1 D-cache on-chip
- 20 CPUs + 32MB L2 cache per Multi-Chip Module
- Water cooled to 10°C junction temp

What makes a good instruction set?

One that provides a simple software interface yet allows simple, fast, efficient hardware implementations

... but across 25+ year time frame

Example of difficulties:

- Current machines have register files with more storage than entire main memory of early machines!
- On-chip test circuitry in current machines has hundreds of times more transistors than entire early computers!

Full Employment for Architects

- Good news: “Ideal” instruction set changes continually
 - Technology allows larger CPUs over time
 - Technology constraints change (e.g., now it is power)
 - Compiler technology improves (e.g., register allocation)
 - Programming styles change (assembly, HLL, object-oriented, ...)
 - Applications change (e.g., multimedia,)
- Bad news: Software compatibility imposes huge damping coefficient on instruction set innovation
 - Software investment dwarfs hardware investment
 - Innovate at microarchitecture level, below the ISA level (this is what most computer architects do)
- New instruction set can only be justified by new large market and technological advantage
 - Network processors
 - Multimedia processors
 - DSP's