CIS 429/529
Influence of Technology and Software on Architecture
Prof. Michel A. Kinsky
And then there was IBM

- Users stopped building their own machines

- 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!

- 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
Computers in mid 50’s

• 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
Earliest Instruction Sets

- Burks, Goldstein & von Neumann ~1946
- Single Accumulator - A carry-over from calculators.
- Typically less than 2 dozen instructions!

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
</tr>
<tr>
<td>STORE</td>
</tr>
<tr>
<td>ADD</td>
</tr>
<tr>
<td>SUB</td>
</tr>
<tr>
<td>MUL</td>
</tr>
<tr>
<td>DIV</td>
</tr>
<tr>
<td>SHIFT LEFT</td>
</tr>
<tr>
<td>SHIFT RIGHT</td>
</tr>
<tr>
<td>JUMP</td>
</tr>
<tr>
<td>JGE</td>
</tr>
<tr>
<td>LOAD ADR</td>
</tr>
<tr>
<td>STORE ADR</td>
</tr>
</tbody>
</table>
Single Accumulator Machine

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

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP</td>
<td>LOAD N</td>
</tr>
<tr>
<td></td>
<td>JGE DONE</td>
</tr>
<tr>
<td></td>
<td>ADD ONE</td>
</tr>
<tr>
<td></td>
<td>STORE N</td>
</tr>
<tr>
<td>F1</td>
<td>LOAD A</td>
</tr>
<tr>
<td>F2</td>
<td>ADD B</td>
</tr>
<tr>
<td>F3</td>
<td>STORE C</td>
</tr>
<tr>
<td></td>
<td>JUMP LOOP</td>
</tr>
<tr>
<td>DONE</td>
<td>HLT</td>
</tr>
</tbody>
</table>

How to modify the addresses A, B and C?
Self-Modifying Code

- Modify the program for the next iteration

```
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
```

```
DONE  JUMP   LOOP
      HLT
```
Self-Modifying Code

- Most of the executed instructions are for bookkeeping!
- Each iteration involves total book-keeping
  - Instruction fetches: 17 14
  - Operand fetches: 10 8
  - Stores: 5 4

\[ C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n \]
Processor-Memory Bottleneck

• Early Solutions
  ‣ Fast local storage in the processor
    • 8-16 registers as opposed to one accumulator
    • to save on loads/stores
  ‣ Indexing capability
    • to reduce book keeping instructions
  ‣ Complex instructions
    • to reduce instruction fetches
Early Solutions

- Compact instructions
  - implicit address bits for operands
  - to reduce instruction fetch cost
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
Processor State

• 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 $IX \leftarrow (IX) + 1$

Index registers have accumulator-like characteristics
Using Index Registers

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

<table>
<thead>
<tr>
<th></th>
<th>with index regs</th>
<th>without index regs</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction fetch</td>
<td>(2)</td>
<td>17 (14)</td>
</tr>
<tr>
<td>operand fetch</td>
<td>2</td>
<td>10 (8)</td>
</tr>
<tr>
<td>store</td>
<td>2</td>
<td>5 (4)</td>
</tr>
</tbody>
</table>

- Costs:
  - Complex control
  - Need to operate on index registers (ALU-like circuitry)

\[ C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n \]
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 more 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
  ‣ $INCI \ k, IX \quad IX \leftarrow (IX) + k$

• More instructions to manipulate index register
  ‣ $STOREI \ x, IX \quad M[x] \leftarrow (IX)$ (extended to fit a word)

• IX begins to look like an accumulator
Support for Subroutine

- A special subroutine jump instruction
  - M: JSR F \( \rightarrow \) M + 1 and jump to F+1
Indirect Addressing

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

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

Events:
- Execute M
- Execute S1
- Execute S2
- Execute S3
Recursive Procedure Calls

- Indirect Addressing through a register
  \[ \text{LOAD } R_1, (R_2) \]
- 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
   \[\text{LOAD } x\]

2. Single accumulator, index registers
   \[\text{LOAD } x, \text{IX}\]

3. Indirection
   \[\text{LOAD } (x)\]

4. Multiple accumulators, index registers, indirection
   \[\text{LOAD } R, \text{IX}, x\]
   \[\text{or LOAD } R, \text{IX}, (x)\]
   the meaning?
   \[R \leftarrow M[M[x] + (IX)]\]
   \[\text{or } R \leftarrow M[M[x + (IX)]]\]

5. Indirect through registers
   \[\text{LOAD } R_I, (R_J)\]

6. The works
   \[\text{LOAD } R_I, R_J, (R_K)\]
   \[R_J = \text{index, } R_K = \text{base addr}\]
Variety of Instruction Formats

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

  \[(\text{Reg} \times \text{Reg}) \text{ to Reg} \quad \text{RI} \leftarrow (\text{RJ}) + (\text{RK})\]

  \[(\text{Reg} \times \text{Mem}) \text{ to Reg} \quad \text{RI} \leftarrow (\text{RJ}) + \text{M}[x]\]

  - \(x\) 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

  \[(\text{Reg} \times \text{Reg}) \text{ to Reg} \quad \text{RI} \leftarrow (\text{RI}) + (\text{RJ})\]

  \[(\text{Reg} \times \text{Mem}) \text{ to Reg} \quad \text{RI} \leftarrow (\text{RI}) + \text{M}[x]\]
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
Data Formats and Addresses

• Data formats:
  ‣ Bytes, Half words, words and double words

• Some issues
  ‣ Byte addressing
    • Big Endian vs. Little Endian
  ‣ 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
The IBM 650 (1953-4)

Magnetic Drum (1,000 or 2,000 10-digit decimal words)

Active instruction (including next program counter)

Digit-serial ALU

20-digit accumulator

[From 650 Manual, © IBM]
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:
Compatibility Problem at IBM

• By early 60’s, IBM had 4 incompatible lines of computers!

• 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 : 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 → programming aids
  ‣ Machine must be capable of supervising itself without manual intervention
IBM 360: Design Premises

• Amdahl, Blaauw and Brooks, 1964
  ‣ 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 GPR Machine

- 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
IBM 360: A GPR Machine

• A General-Purpose Register (GPR) Machine: Processor State
  ▸ 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: Addressing Modes

- **RR**

  \[
  R_1 \leftarrow (R_1) \text{ op } (R_2)
  \]

- **RD**

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

  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: String Operations

• SS format: store to store instructions

\[ \text{iterate “length” times} \]

\[ \text{SS format: store to store instructions} \]

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

• Most operations on decimal and character strings use this format

<table>
<thead>
<tr>
<th>Opcode</th>
<th>length</th>
<th>B1</th>
<th>D1</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>4</td>
<td>12</td>
<td>4</td>
<td>12</td>
</tr>
</tbody>
</table>

› MVC  move characters

› MP   multiply two packed decimal strings

› CLC  compare two character strings
IBM 360: Control Flow

• Arithmetic and logic instructions set condition codes
  ‣ equal to zero
  ‣ greater than zero
  ‣ overflow
  ‣ carry...

• I/O instructions also set condition codes
  ‣ channel busy
IBM 360: Control Flow

- 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 ← (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 next instruction began execution

- Exception/interrupt behavior constant across family of implementations
IBM 360: 1964 Implementations

- 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

<table>
<thead>
<tr>
<th></th>
<th>Model 30</th>
<th>Model 70</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Capacity</td>
<td>8K - 64 KB</td>
<td>256K - 512 KB</td>
</tr>
<tr>
<td>Memory Cycle</td>
<td>2.0µs</td>
<td>1.0µs</td>
</tr>
<tr>
<td>Datapath</td>
<td>8-bit</td>
<td>64-bit</td>
</tr>
<tr>
<td>Circuit Delay</td>
<td>30 nsec/level</td>
<td>5 nsec/level</td>
</tr>
<tr>
<td>Registers in</td>
<td>Main Store</td>
<td>in Transistor</td>
</tr>
<tr>
<td>Control Store</td>
<td>Read only 1µsec</td>
<td>Dedicated circuits</td>
</tr>
</tbody>
</table>
IBM 360: Forty years later...

- The zSeries z990 Microprocessor
  - 64-bit virtual addressing
    - original 360 was 24-bit; 370 was a 31-bit extension
  - Dual core design
  - Dual-issue in-order superscalar
  - 10-stage CISC pipeline
  - Out-of-order memory accesses
  - Redundant datapaths
    - every instruction performed in two parallel datapaths and results compared

[ IBM Journal R&D, 48(3/4), May/July 2004 ]
The zSeries z990 Microprocessor

- 256KB L1 I-cache, 256KB L1 D-cache on-chip
- 32MB shared L2 unified cache, off-chip
- 512-entry L1 TLB + 4K-entry L2 TLB
  - very large TLB, to support multiple virtual machines
- 8K-entry Branch Target Buffer
  - Very large buffer to support commercial workloads
- Up to 64 processors (48 visible to customer) in one machine
- 1.2 GHz in IBM 130nm SOI CMOS technology, 55W for both cores
Next Class

• Single-cycle ISA Implementation