EC 513
Computer Architecture

Influence of Technology and Software on Instruction Sets

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>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD x</td>
<td>AC ← M[x]</td>
</tr>
<tr>
<td>STORE x</td>
<td>M[x] ← (AC)</td>
</tr>
<tr>
<td>ADD x</td>
<td>AC ← (AC) + M[x]</td>
</tr>
<tr>
<td>SUB x</td>
<td></td>
</tr>
<tr>
<td>MUL x</td>
<td>Involved a quotient register</td>
</tr>
<tr>
<td>DIV x</td>
<td></td>
</tr>
<tr>
<td>SHIFT LEFT</td>
<td>AC ← 2 × (AC)</td>
</tr>
<tr>
<td>SHIFT RIGHT</td>
<td></td>
</tr>
<tr>
<td>JUMP x</td>
<td>PC ← x</td>
</tr>
<tr>
<td>JGE x</td>
<td>if (AC) ≥ 0 then PC ← x</td>
</tr>
<tr>
<td>LOAD ADR x</td>
<td>AC ← Extract address field(M[x])</td>
</tr>
<tr>
<td>STORE ADR x</td>
<td></td>
</tr>
</tbody>
</table>
Single Accumulator Machine

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

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

DONE HLT

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
JUMP
LOOP
DONE HLT
```

```
LOAD ADR F1
ADD ONE
STORE ADR F1
LOAD ADR F2
ADD ONE
STORE ADR F2
LOAD ADR F3
ADD ONE
STORE ADR F3
```
Self-Modifying Code

- Most of the executed instructions are for bookkeeping!
- Each iteration involves total bookkeeping
  - 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
Processor-Memory Bottleneck

- 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
- Programmer’s machine model is a contract between the hardware and software
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 ← M[x + (IX)]
    - ADD x, IX  AC ← (AC) + M[x + (IX)]
    - ...
  - Add new instructions to manipulate index registers
    - JZi x, IX  if (IX)=0 then  PC <-- x
      else  IX <-- (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)

- 1 to 2 bits longer Instructions
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 $k$
  - $AC < (IX)$ new instruction
  - $AC < (AC) + k$
  - $IX < (AC)$ new instruction

- Also the AC must be saved and restored

- It may be better to increment IX directly
  - $INCI \ k, IX \ IX \leftarrow (IX) + k$

- More instructions to manipulate index register
  - $STOREi x, IX \ M[x] \leftarrow (IX)$ (extended to fit a word)

- IX begins to look like an accumulator
Support for Subroutine

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

```
Main Program

| call F | a1 | a2 | \cdot | \cdot | call F | b1 | b2 |
```

```
Subroutine F

\cdot

return
```
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 ← M[M[x]]
Recursive Procedure Calls

- Indirect Addressing through a register
  LOAD  R₁, (R₂)
- Load register R₁ with the contents of the word whose address is contained in register R₂
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
   \[ \text{LOAD~} R, \text{IX, } x \]
   or \[ \text{LOAD~} R, \text{IX, } (x) \]
   The meaning?
   \[ R \leftarrow M[M[x] + (\text{IX})] \]
   or \[ R \leftarrow M[M[x + (\text{IX})]] \]

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
Next Class

- Intel Pin introduction